

---

# **AiiDA documentation**

***Release 0.5.0***

**Giovanni Pizzi      Andrea Cepellotti**  
**Riccardo Sabatini    Nicola Marzari    Boris Kozinsky**

December 17, 2015



<b>1</b>	<b>User's guide</b>	<b>3</b>
1.1	User's guide . . . . .	3
<b>2</b>	<b>Other guide resources</b>	<b>111</b>
2.1	Other guide resources . . . . .	111
<b>3</b>	<b>Developer's guide</b>	<b>121</b>
3.1	Developer's guide . . . . .	121
<b>4</b>	<b>Modules provided with aiida</b>	<b>229</b>
4.1	Modules . . . . .	229
<b>5</b>	<b>Indices and tables</b>	<b>341</b>
	<b>Python Module Index</b>	<b>343</b>





Fig. 1: Automated Interactive Infrastructure and Database for Computational Science

AiiDA is a sophisticated framework designed from scratch to be a flexible and scalable infrastructure for computational science. Being able to store the full data provenance of each simulation, and based on a tailored database solution built for efficient data mining implementations, AiiDA gives the user the ability to interact seamlessly with any number of HPC machines and codes thanks to its flexible plugin interface, together with a powerful workflow engine for the automation of simulations.

The software is available at <http://www.aiida.net>.

If you use AiiDA for your research, please cite the following work:

Giovanni Pizzi, Andrea Cepellotti, Riccardo Sabatini, Nicola Marzari, and Boris Kozinsky, *AiiDA: automated interactive infrastructure and database for computational science*, Comp. Mat. Sci 111, 218-230 (2016); <http://dx.doi.org/10.1016/j.commatsci.2015.09.013>; <http://www.aiida.net>.

This is the documentation of the AiiDA framework. For the first setup, configuration and usage, refer to the [user's guide](#) below.

If, instead, you plan to add new plugins, or you simply want to understand AiiDA internals, refer to the [developer's guide](#).



---

## User's guide

---

### 1.1 User's guide

#### 1.1.1 Databases for AiiDA

AiiDA needs a database backend to store the nodes, node attributes and other information, allowing the end user to perform very fast queries of the results.

Before installing AiiDA, you have to choose (and possibly configure) a suitable supported backend.

#### Supported databases

---

**Note:** For those who do not want to read all this section, the short answer if you want to choose a database is SQLite if you just want to try out AiiDA without spending too much time in configuration (but SQLite is not suitable for production runs), while PostgreSQL for regular production use of AiiDA.

---

For those who are interested in the details, there are three supported database backends:

- **SQLite** The SQLite backend is the fastest to configure: in fact, it does not really use a “real” database, but stores everything in a file. This is great if you never configured a database before and you just want to give AiiDA a try. However, keep in mind that it has **many big shortcomings** for a real AiiDA usage!

In fact, since everything is stored on a single file, each access (especially when writing or doing a transaction) to the database locks it: this means that a second thread wanting to access the database has to wait that the lock is released. We set up a timeout of about 60 seconds for each thread to retry to connect to the database, but after that time you will get an exception, with the risk of storing corrupted data in the AiiDA repository.

Therefore, it is OK to use SQLite for testing, but as soon as you want to use AiiDA in production, with more than one calculation submitted at each given time, please switch to a real database backend, like PostgreSQL.

---

**Note:** note, however, that typically SQLite is pretty fast for queries, once the database is loaded into memory, so it could be an interesting solution if you do not want to launch new calculations, but only to import the results and then query them (in a single-user approach).

---

- **PostgreSQL** This is the database backend that we, the AiiDA developers, suggest to use, because it is the one with most features.
- **MySQL** This is another possible backend that you could use. However, we suggest that you use PostgreSQL instead of MySQL, due to some MySQL limitations (unless you have very strong reasons to prefer MySQL over PostgreSQL). In particular, some of the limitations of MySQL are:

- Only a precision of 1 second is possible for time objects, while PostgreSQL supports microsecond precision. This can be relevant for a proper sorting of calculations launched almost simultaneously.
- Indexed text columns can have an hardcoded maximum length. This can give issues with attributes, if you have very long key names or nested dictionaries/lists. These cannot be natively stored and therefore you either end up storing a JSON (therefore partially losing query capability) or you can even incur in problems.

## Setup instructions

For any database, you may need to install a specific python package using `pip`; this typically also requires to have the development libraries installed (the `.h` C header files). Refer to the [installation documentation](#) for more details.

### SQLite

To use SQLite as backend, please install:

```
sudo apt-get install libsqlite3-dev
```

SQLite requires almost no configuration. In the `verdi install` phase, just type `sqlite` when the Database engine is required, and then provide an absolute path for the AiiDA Database location field, that will be the file that will store the full database (if no file exists yet in that position, a fresh AiiDA database will be created).

---

**Note:** Do not forget to backup your database (instructions [here](#)).

---

### PostgreSQL

---

**Note:** We assume here that you already installed PostgreSQL on your computer and that you know the password for the `postgres` user (there are many tutorials online that explain how to do it, depending on your operating system and distribution). To install PostgreSQL under Ubuntu, you can do:

```
sudo apt-get install postgresql-9.1
sudo apt-get install postgresql-server-dev-9.1
sudo apt-get install postgresql-client-9.1
```

On Mac OS X, you can download binary packages to install PostgreSQL from the official website.

---

To properly configure a new database for AiiDA with PostgreSQL, you need to create a new `aiida` user and a new `aiidadb` table.

To create the new `aiida` user and the `aiidadb` database, first become the UNIX `postgres` user, typing as root:

```
su - postgres
```

(or equivalently type `sudo su - postgres`, depending on your distribution).

Then type the following command to enter in the PostgreSQL shell in the modality to create users:

```
psql template1
```

To create a new user for postgres (you can call it simply `aiida`, as in the example below), type in the `psql` shell:

```
CREATE USER aiida WITH PASSWORD 'the_aiida_password';
```



where of course you have to change `the_aiida_password` with a valid password.

**Note:** Remember, however, that since AiiDA needs to connect to this database, you will need to store this password in clear text in your home folder for each user that wants to have direct access to the database, therefore choose a strong password, but different from any that you already use!

**Note:** Did you just copy and paste the line above, therefore setting the password to `the_aiida_password`? Then, let's change it! Choose a good password this time, and then type the following command (this time replacing the string `new_aiida_password` with the password you chose!):

```
ALTER USER aida PASSWORD 'new_aiida_password';
```

Then create a new `aiidadb` database for AiiDA, and give ownership to user `aiida` created above:

```
CREATE DATABASE aiidadb OWNER aiida;
```

and grant all privileges on this DB to the previously-created `aiida` user:

```
GRANT ALL PRIVILEGES ON DATABASE aiidadb to aiida;
```

Finally, type `\q` to quit the `template1` shell, and `exit` to exit the PostgreSQL shell.

To test if this worked, type this on a bash terminal (as a normal user):

```
psql -h localhost -d aiidadb -U aiida -W
```

and type the password you inserted before, when prompted. If everything worked, you should get no error and the `psql` shell. Type `\q` to exit.

If you use the names suggested above, in the `verdi install` phase you should use the following parameters:

```
Database engine: postgresql
PostgreSQL host: localhost
PostgreSQL port: 5432
AiiDA Database name: aiidadb
AiiDA Database user: aiida
AiiDA Database password: the_aiida_password
```

**Note:** Do not forget to backup your database (instructions [here](#)).

**Note:** If you want to move the physical location of the data files on your hard drive AFTER it has been created and filled, look at the instructions [here](#).

**Note:** Due to the presence of a bug, PostgreSQL could refuse to restart after a crash. If this happens you should follow the instructions written [here](#).

## MySQL

To use properly configure a new database for AiiDA with MySQL, you need to create a new `aiida` user and a new `aiidadb` table.

We assume here that you already installed MySQL on your computer and that you know your MySQL root password (there are many tutorials online that explain how to do it, depending on your operating system and distribution). To install `mysql-client`, you can do:

```
sudo apt-get install libmysqlclient-dev
```

After MySQL is installed, connect to it as the MySQL root account to create a new account. This can be done typing in the shell:

```
mysql -h localhost mysql -u root -p
```

(we are assuming that you installed the database on `localhost`, even if this is not strictly required - if this is not the case, change `localhost` with the proper database host, but note that also some of the commands reported below need to be adapted) and then type the MySQL root password when prompted.

In the MySQL shell, type the following command to create a new user:

```
CREATE USER 'aiida'@'localhost' IDENTIFIED BY 'the_aiida_password';
```

where of course you have to change `the_aiida_password` with a valid password.

**Note:** Remember, however, that since AiiDA needs to connect to this database, you will need to store this password in clear text in your home folder for each user that wants to have direct access to the database, therefore choose a strong password, but different from any that you already use!

Then, still in the MySQL shell, create a new database named `aiidadb` using the command:

```
CREATE DATABASE aiidadb;
```

and give all privileges to the `aiida` user on this database:

```
GRANT ALL PRIVILEGES on aiidadb.* to aiida@localhost;
```

**Note:** “(only for developers)” If you are a developer and want to run the tests using the MySQL database (to do so, you also have to set the `tests.use_sqlite` AiiDA property to `False` using the `verdi devel setproperty tests.use_sqlite False` command), you also have to create a `test_aiidadb` database. In this case, run also the two following commands:

```
CREATE DATABASE test_aiidadb;  
GRANT ALL PRIVILEGES on test_aiidadb.* to aiida@localhost;
```

If you use the names suggested above, in the `verdi install` phase you should use the following parameters:

```
Database engine: mysql  
mySQL host: localhost  
mySQL port: 3306  
AiiDA Database name: aiidadb  
AiiDA Database user: aiida  
AiiDA Database password: the_aiida_passwd
```

**Note:** Do not forget to backup your database (instructions [here](#)).

## 1.1.2 AiiDA Backup

In this page you will find useful information on how to backup your database, how to move it to a different location and how to backup your repository.

## How to backup the databases

It is strongly advised to backup the content of your database daily. Below are instructions to set this up for the SQLite, PostgreSQL and MySQL databases, under Ubuntu (tested with version 12.04).

### SQLite backup

---

**Note:** Perform the following operation after having set up AiiDA. Only then the `~/.aiida` folder (and the files within) will be created.

---

Simply make sure your database folder (typically `/home/USERNAME/.aiida/` containing the file `aiida.db` and the `repository` directory) is properly backed up by your backup software (under Ubuntu, Backup -> check the “Folders” tab).

### PostgreSQL backup

---

**Note:** Perform the following operation after having set up AiiDA. Only then the `~/.aiida` folder (and the files within) will be created.

---

The database files are not put in the `.aiida` folder but in the system directories which typically are not backed up. Moreover, the database is spread over lots of files that, if backed up as they are at a given time, cannot be re-used to restore the database.

So you need to periodically (typically once a day) dump the database contents in a file that will be backed up. This can be done by the following bash script `backup_postgresql.sh`:

```
#!/bin/bash
AIIDAUSER=aiida
AIIDADB=aiidadb
AIIDAPORT=5432
## STORE THE PASSWORD, IN THE PROPER FORMAT, IN THE ~/.pgpass file
## see http://www.postgresql.org/docs/current/static/libpq-pgpass.html
AIIDALOCALTMPDUMPFIL=~/aiida/${AIIDADB}-backup.psql.gz

if [ -e ${AIIDALOCALTMPDUMPFIL} ]
then
    mv ${AIIDALOCALTMPDUMPFIL} ${AIIDALOCALTMPDUMPFIL}~
fi

# NOTE: password stored in ~/.pgpass, where pg_dump will read it automatically
pg_dump -h localhost -p $AIIDAPORT -U $AIIDAUSER $AIIDADB | gzip > $AIIDALOCALTMPDUMPFIL || rm $AIIDALOCALTMPDUMPFIL
```

Before launching the script you need to create the file `~/.pgpass` to avoid having to enter your database password each time you use the script. It should look like (`.pgpass`):

```
localhost:5432:aiidadb:aiida:YOUR_DATABASE_PASSWORD
```

where `YOUR_DATABASE_PASSWORD` is the password you set up for the database.

---

**Note:** Do not forget to put this file in `~/.pgpass` and to name it `.pgpass`. Remember also to give it the right permissions (read and write): `chmod u=rw .pgpass`.

---

To dump the database in a file automatically everyday, you can add the following script `backup-aiidadb-USERNAME` in `/etc/cron.daily/`, which will launch the previous script once per day:

```
#!/bin/bash
su USERNAME -c "/home/USERNAME/.aiida/backup_postgresql.sh"
```

where all instances of `USERNAME` are replaced by your actual user name. The `su USERNAME` makes the dumped file be owned by you rather than by `root`. Remember to give the script the right permissions:

```
sudo chmod +x /etc/cron.daily/backup-aiidadb-USERNAME
```

Finally make sure your database folder (`/home/USERNAME/.aiida/`) containing this dump file and the repository directory, is properly backed up by your backup software (under Ubuntu, Backup -> check the “Folders” tab).

**Note:** If your database is very large (more than a few hundreds of thousands of nodes and workflows), a standard backup of your repository folder will be very slow (up to days), thus slowing down your computer dramatically. To fix this problem you can set up an incremental backup of your repository by following the instructions [here](#).

---

## MySQL backup

---

### Todo

Back-up instructions for the MySQL database.

---

We do not have explicit instructions on how to back-up MySQL yet, but you can find plenty of information on Google.

## How to retrieve the database from a backup

### PostgreSQL backup

In order to retrieve the database from a backup, you have first to create a empty database following the instructions described above in “Setup instructions: PostgreSQL” except the `verdi install` phase. Once that you have created your empty database with the same names of the backuped one, type the following command:

```
psql -h localhost -U aida -d aiidadb -f aiidadb-backup.psql
```

## How to move the physical location of a database

It might happen that you need to move the physical location of the database files on your hard-drive (for instance, due to the lack of space in the partition where it is located). Below we explain how to do it.

### PostgreSQL move

First, make sure you have a backup of the full database (see instructions [here](#)), and that the AiiDA daemon is not running. Then, become the UNIX `postgres` user, typing as root:

```
su - postgres
```

(or, equivalently, type `sudo su - postgres`, depending on your distribution).

Stop the postgres database daemon:

```
service postgresql stop
```

Then enter the postgres shell:

```
psql
```

and look for the current location of the data directory:

```
SHOW data_directory;
```

Typically you should get something like `/var/lib/postgresql/9.1/main`.

**Note:** If you are experiencing memory problems and cannot enter the postgres shell, you can look directly into the file `/etc/postgresql/9.1/main/postgresql.conf` and check out the line defining the variable `data_directory`.

Then exit the shell with `\q`, go to this directory and copy all the files to the new directory:

```
cp -a SOURCE_DIRECTORY DESTINATION_DIRECTORY
```

where `SOURCE_DIRECTORY` is the directory you got from the `SHOW data_directory;` command, and `DESTINATION_DIRECTORY` is the new directory for the database files.

Make sure the permissions, owner and group are the same in the old and new directory (including all levels above the `DESTINATION_DIRECTORY`). The owner and group should be both `postgres`, at the notable exception of some symbolic links in `server.crt` and `server.key`.

**Note:** If the permissions of these links need to be changed, use the `-h` option of `chown` to avoid changing the permissions of the destination of the links. In case you have changed the permission of the links destination by mistake, they should typically be (beware that this might depend on your actual distribution!):

```
-rw-r--r-- 1 root root 989 Mar  1  2012 /etc/ssl/certs/ssl-cert-snakeoil.pem
-rw-r----- 1 root ssl-cert 1704 Mar  1  2012 /etc/ssl/private/ssl-cert-snakeoil.key
```

Then you can change the postgres configuration file, that should typically be located here:

```
/etc/postgresql/9.1/main/postgresql.conf
```

Make a backup version of this file, then look for the line defining `data_directory` and replace it with the new data directory path:

```
data_directory = 'NEW_DATA_DIRECTORY'
```

Then start again the database daemon:

```
service postgresql start
```

You can check that the data directory has indeed changed:

```
psql
SHOW data_directory;
\q
```

Before removing definitely the previous location of the database files, first rename it and test AiiDA with the new database location (e.g. do simple queries like `verdi code list` or create a node and store it). If everything went fine, you can delete the old database location.

## How to set up an incremental backup for the repository

Apart from the database backup, you should also backup the AiiDA repository. For small repositories, this can be easily done by a simple directory copy or, even better, with the use of the `rsync` command which can copy only the differences. However, both of the aforementioned approaches are not efficient in big repositories where even a partial recursive directory listing may take significant time, especially for filesystems where accessing a directory has a constant (and significant) latency time. Therefore, we provide scripts for making efficient backups of the AiiDA repository.

Before running the backup script, you will have to configure it. Therefore you should execute the `backup_setup.py` which is located under `MY_AIIIDA_FOLDER/aiida/common/additions/backup_script`. For example:

```
python MY_AIIIDA_FOLDER/aiida/common/additions/backup_script/backup_setup.py
```

This will ask a set of questions. More precisely, it will initially ask for:

- The backup folder. This is the destination of the backup *configuration file*. By default a folder named `backup` in your `.aiida` directory is proposed to be created.
- The destination folder of the backup. This is the destination folder of the files to be backed up. By default it is a folder inside the aforementioned backup folder (e.g. `/home/aiida_user/.aiida/backup/backup_dest`).

---

**Note:** You should backup the repository on a different disk than the one in which you have the AiiDA repository! If you just use the same disk, you don't have any security against the most common data loss cause: disk failure. The best option is to use a destination folder mounted over ssh. For this you need to install `sshfs` (under ubuntu: `sudo apt-get install sshfs`).

E.g. Imagine that you run your calculations on `server_1` and you would like to take regular repository backups to `server_2`. Then, you could mount a `server_2` directory via `sshfs` on `server_1` using the following command on `server_1`:

```
sshfs -o idmap=user -o rw backup_user@server_2:/home/backup_user/backup_destination_dir/ /home/aiida_user/remote_backup_dir/
```

You can put this line into the actions performed at start-up, so that the remote directory is mounted automatically after a reboot (but do not put it in your `.bashrc` file otherwise each time you open a new terminal, your computer will complain that the mount point is not empty...).

---

A template backup configuration file (`backup_info.json.tpl`) will be copied in the backup folder. You can set the backup variables by yourself after renaming the template file to `backup_info.json`, or you can answer the questions asked by the script, and then `backup_info.json` will be created based on you answers.

The main script backs up the AiiDA repository that is referenced by the current AiiDA database. The script will start from the `oldest_object_backedup` date or the date of the oldest node/workflow object found and it will periodically backup (in periods of `periodicity` days) until the ending date of the backup specified by `end_date_of_backup` or `days_to_backup`

The backup parameters to be set in the `backup_info.json` are:

- `periodicity` (in days): The backup runs periodically for a number of days defined in the `periodicity` variable. The purpose of this variable is to limit the backup to run only on a few number of days and therefore to limit the number of files that are backed up at every round. e.g. `"periodicity": 2` Example: if you have files in the AiiDA repositories created in the past 30 days, and `periodicity` is 15, the first run will backup the files of the first 15 days; a second run of the script will backup the next 15 days, completing the backup (if it is run within the same day). Further runs will only backup newer files, if they are created.
- `oldest_object_backedup` (timestamp or null): This is the timestamp of the oldest object that was backed up. If you are not aware of this value or if it is the first time that you start a backup up for this repository,

then set this value to `null`. Then the script will search the creation date of the oldest workflow or node object in the database and it will start the backup from that date. E.g. `"oldest_object_backedup": "2015-07-20 11:13:08.145804+02:00"`

- `end_date_of_backup`: If set, the backup script will backup files that have a modification date until the value specified by this variable. If not set, the ending of the backup will be set by the following variable (`days_to_backup`) which specifies how many days to backup from the start of the backup. If none of these variables are set (`end_date_of_backup` and `days_to_backup`), then the end date of backup is set to the current date. E.g. `"end_date_of_backup": null` or `"end_date_of_backup": "2015-07-20 11:13:08.145804+02:00"`
- `days_to_backup`: If set, you specify how many days you will backup from the starting date of your backup. If it set to `null` and also `end_date_of_backup` is set to `null`, then the end date of the backup is set to the current date. You can not set `days_to_backup` & `end_date_of_backup` at the same time (it will lead to an error). E.g. `"days_to_backup": null` or `"days_to_backup": 5`
- `backup_length_threshold` (in hours): The backup script runs in rounds and on every round it backs-up a number of days that are controlled primarily by `periodicity` and also by `end_date_of_backup` / `days_to_backup`, for the last backup round. The `backup_length_threshold` specifies the lowest acceptable round length. This is important for the end of the backup.
- `backup_dir`: The destination directory of the backup. e.g. `"backup_dir": "/home/aiida_user/.aiida/backup/backup_dest"`

To start the backup, run the `start_backup.py` script. Run as often as needed to complete a full backup, and then run it periodically (e.g. calling it from a cron script, for instance every day) to backup new changes.

---

**Note:** You can set up a cron job using the following command:

```
sudo crontab -u aiida_user -e
```

It will open an editor where you can add a line of the form:

```
00 03 * * * /home/aiida_user/.aiida/backup/start_backup.py 2>&1 | mail -s "Incremental backup of the
```

This will launch the backup of the database everyday at 3 AM, and send the output (or any error message) to the email address of the user (provided the `mail` command – from `mailutils` – is configured appropriately).

---

Finally, do not forget to exclude the repository folder from the normal backup of your home directory!

### 1.1.3 Installation and Deployment of AiiDA

If you are updating from a previous version and you don't want to reinstall everything from scratch, read the instructions [here](#).

#### Supported architecture

AiiDA has a few strict requirements, in its current version: first, it will run only on Unix-like systems - it is tested (and developed) in Mac OS X and Linux (Ubuntu), but other Unix flavours *should* work as well.

Moreover, on the clusters (computational resources) side, it expects to find a Unix system, and the default shell is **required** to be `bash`.

## Installing python

AiiDA requires python 2.7.x (only CPython has been tested). It is probable that you already have a version of python installed on your computer. To check, open a terminal and type:

```
python -V
```

that will print something like this:

```
Python 2.7.3
```

If you don't have python installed, or your version is outdated, please install a suitable version of python (either refer to the manual of your Linux distribution, or for instance you can download the ActiveState Python from [ActiveState](#). Choose the appropriate distribution corresponding to your architecture, and with version 2.7.x.x).

## Installation of the core dependencies

### Database

As a first thing, [choose and setup the database that you want to use](#).

### Other core dependencies

Before continuing, you still need to install a few more programs. Some of them are mandatory, while others are optional (but often strongly suggested), also depending for instance on the [type of database](#) that you plan to use.

Here is a list of packages/programs that you need to install (for each of them, there may be a specific/easier way to install them in your distribution, as for instance `apt-get` in Debian/Ubuntu -see below for the specific names of packages to install- or `yum` in RedHat/Fedora).

- [git](#) (required to download the code)
- [python-pip](#) (required to automatically download and install further python packages required by AiiDA)
- [ipython](#) (optional, but strongly recommended for interactive usage)
- python 2.7 development files (these may be needed; refer to your distribution to know how to locate and install them)
- To support SQLite:
  - [SQLite3 development files](#) (required later to compile the library, when configuring the python sqlite module; see below for the Ubuntu module required to install these files)
- To support PostgreSQL:
  - [PostgreSQL development files](#) (required later to compile the library, when configuring the python psycopg2 module; see below for the Ubuntu module required to install these files)

For Ubuntu, you can install the above packages using (tested on Ubuntu 12.04, names may change in different releases):

```
sudo apt-get install git python-pip ipython python2.7-dev
```

**Note:** For the latter line, please use the same version (in the example above is 9.1) of the postgresql server that you installed (in this case, to install the server of the same version, use the `sudo apt-get install postgresql-9.1` command).

If you want to use postgresQL, use a version greater than 9.1 (the greatest that your distribution supports).



For Mac OS X, you may either already have some of the dependencies above (e.g., git), or you can download binary packages to install (e.g., for PostgreSQL you can download and install the binary package from the official website).

## Downloading the code

Download the code using git in a directory of your choice (`~/git/aiida` in this tutorial), using the following command:

```
git clone https://USERNAME@bitbucket.org/aiida_team/aiida_core.git
```

(or use `git@bitbucket.org:aiida_team/aiida_core.git` if you are downloading through SSH; note that this requires your ssh key to be added on the Bitbucket account.)

## Python dependencies

Python dependencies are managed using `pip`, that you have installed in the previous steps.

As a first step, check that `pip` is at its most recent version.

One possible way of doing this is to update `pip` with itself, with a command similar to the following:

```
sudo pip install -U pip
```

Then, install the python dependencies is as simple as this:

```
cd ~/git/aiida # or the folder where you downloaded AiiDA
pip install --user -U -r requirements.txt
```

(this will download and install requirements that are listed in the `requirements.txt` file; the `--user` option allows to install the packages as a normal user, without the need of using `sudo` or becoming root). Check that every package is installed correctly.

There are some additional dependencies need to be installed if you are using PostgreSQL or MySQL as backend database. No additional dependency is required for SQLite.

For PostgreSQL:

```
pip install psycopg2==2.6
```

For MySQL:

```
pip install MySQL-python==1.2.5
```

**Note:** This step should work seamlessly, but there are a number of reasons for which problems may occur. Often googling for the error message helps in finding a solution. Some common pitfalls are described in the notes below.

**Note:** if the `pip install` command gives you this kind of error message:

```
OSError: [Errno 13] Permission denied: '/usr/local/bin/easy_install'
```

then try again as root:

```
sudo pip install -U -r requirements.txt
```

If everything went smoothly, congratulations! Now the code is installed! However, we need still a few steps to properly configure AiiDA for your user.

**Note:** if the `pip install` command gives you an error that resembles the one shown below, you might need to downgrade to an older version of pip:

```
Cannot fetch index base URL https://pypi.python.org/simple/
```

To downgrade pip, use the following command:

```
sudo easy_install pip==1.2.1
```

**Note:** Several users reported the need to install also `libqp-dev`:

```
apt-get install libqp-dev
```

But under Ubuntu 12.04 this is not needed.

**Note:** If the installation fails while installing the packages related to the database, you may have not installed or set up the database libraries as described in the section *Other core dependencies*.

In particular, on Mac OS X, if you installed the binary package of PostgreSQL, it is possible that the `PATH` environment variable is not set correctly, and you get a “Error: pg\_config executable not found.” error. In this case, discover where the binary is located, then add a line to your `~/.bashrc` file similar to the following:

```
export PATH=/the/path/to/the/pg_config/file:${PATH}
```

and then open a new bash shell. Some possible paths can be found at this [Stackoverflow link](#) and a non-exhaustive list of possible paths is the following (version number may change):

- `/Applications/Postgres93.app/Contents/MacOS/bin`
- `/Applications/Postgres.app/Contents/Versions/9.3/bin`
- `/Library/PostgreSQL/9.3/bin/pg_config`

Similarly, if the package installs but then errors occur during the first of AiiDA (with `Symbol not found` errors or similar), you may need to point to the path where the dynamical libraries are. A way to do it is to add a line similar to the following to the `~/.bashrc` and then open a new shell:

```
export DYLD_FALLBACK_LIBRARY_PATH=/Library/PostgreSQL/9.3/lib:$DYLD_FALLBACK_LIBRARY_PATH
```

(you should of course adapt the path to the PostgreSQL libraries).

---

## AiiDA configuration

### Path configuration

The main interface to AiiDA is through its command-line tool, called `verdi`. For it to work, it must be on the system path, and moreover the AiiDA python code must be found on the python path.

To do this, add the following to your `~/.bashrc` file (create it if not already present):

```
export PYTHONPATH=~/.git/aiida:${PYTHONPATH}
export PATH=~/.git/aiida/bin:${PATH}
```

and then source the `.bashrc` file with the command `source ~/.bashrc`, or login in a new window.

**Note:** replace `~/.git/aiida` with the path where you installed AiiDA. Note also that in the `PYTHONPATH` you

simply have to specify the AiiDA path, while in `PATH` you also have to append the `/bin` subfolder!

**Note:** if you installed the modules with the `--user` parameter during the `pip install` step, you will need to add one more directory to your `PATH` variable in the `~/.bashrc` file. For Linux systems, the path to add is usually `~/.local/bin`:

```
export PATH=~/.git/aiida/bin:~/.local/bin:${PATH}
```

For Mac OS X systems, the path to add is usually `~/Library/Python/2.7/bin`:

```
export PATH=~/.git/aiida/bin:~/Library/Python/2.7/bin:${PATH}
```

To verify if this is the correct path to add, navigate to this location and you should find the executable `supervisord` in the directory.

To verify if the path setup is OK:

- type `verdi` on your terminal, and check if the program starts (it should provide a list of valid commands). If it doesn't, check if you correctly set up the `PATH` environment variable above.
- go in your home folder or in another folder different from the AiiDA folder, run `python` or `ipython` and try to import a module, e.g. typing:

```
import aiida
```

If the setup is ok, you shouldn't get any error. If you do get an `ImportError` instead, check if you correctly set up the `PYTHONPATH` environment variable in the steps above.

**Bash completion** `verdi` fully supports bash completion (i.e., the possibility to press the `TAB` of your keyboard to get a list of sensible commands to type. We strongly suggest to enable bash completion by adding also the following line to your `.bashrc`, **after** the previous lines:

```
eval "$(verdi completioncommand)"
```

If you feel that the bash loading time is becoming too slow, you can instead run the:

```
verdi completioncommand
```

on a shell, and copy-paste the output directly inside your `.bashrc` file, **instead** of the `eval "$(verdi completioncommand)"` line.

Remember, after any modification to the `.bashrc` file, to source it, or to open a new shell window.

**Note:** remember to check that your `.bashrc` is sourced also from your `.profile` or `.bash_profile` script. E.g., if not already present, you can add to your `~/.bash_profile` the following lines:

```
if [ -f ~/.bashrc ]
then
    . ~/.bashrc
fi
```

## AiiDA first setup

Run the following command:

```
verdi install
```

to configure AiiDA. The command will guide you through a process to configure the database, the repository location, and it will finally (automatically) run a `django migrate` command, if needed, that creates the required tables in the database and installs the database triggers.

The first thing that will be asked to you is the timezone, extremely important to get correct dates and times for your calculations.

AiiDA will do its best to try and understand the local timezone (if properly configured on your machine), and will suggest a set of sensible values. Choose the timezone that fits best to you (that is, the nearest city in your timezone - for Lausanne, for instance, we choose `Europe/Zurich`) and type it at the prompt.

If the automatic zone detection did not work for you, type instead another valid string. A list of valid strings can be found at [http://en.wikipedia.org/wiki/List\\_of\\_tz\\_database\\_time\\_zones](http://en.wikipedia.org/wiki/List_of_tz_database_time_zones) but for the definitive list of timezones supported by your system, open a python shell and type:

```
import pytz
print pytz.all_timezones
```

as AiiDA will not accept a timezone string that is not in the above list.

As a second parameter to input during the `verdi install` phase, the “Default user email” is asked.

We suggest here to use your institution email, that will be used to associate the calculations to you.

---

**Note:** In AiiDA, the user email is used as username, and also as unique identifier when importing/exporting data from AiiDA.

---

---

**Note:** Even if you choose an email different from the default one (`aiida@localhost`), a user with email `aiida@localhost` will be set up, with its password set to `None` (disabling access via this user via API or Web interface).

The existence of a default user is internally useful for multi-user setups, where only one user runs the daemon, even if many users can simultaneously access the DB. See the page on *[setting up AiiDA in multi-user mode](#)* for more details (only for advanced users).

---

---

**Note:** The password, in the current version of AiiDA, is not used (it will be used only in the REST API and in the web interface). If you leave the field empty, no password will be set and no access will be granted to the user via the REST API and the web interface.

---

Then, the following prompts will help you configure the database. Typical settings are:

```
Insert your timezone: Europe/Zurich
Default user email: richard.wagner@leipzig.de
Database engine: sqlite3
AiiDA Database location: /home/wagner/.aiida/aiida.db
AiiDA repository directory: /home/wagner/.aiida/repository/
[...]
Configuring a new user with email 'richard.wagner@leipzig.de'
First name: Richard
Last name: Wagner
Institution: BRUHL, LEIPZIG
The user has no password, do you want to set one? [y/N] y
Insert the new password:
Insert the new password (again):
```

**Note:** When the “Database engine” is asked, use ‘sqlite3’ **only if** you want to try out AiiDA without setting up a database.

**However, keep in mind that for serious use, SQLite has serious limitations!!** For instance, when many calculations are managed at the same time, the database file is locked by SQLite to avoid corruption, but this can lead to timeouts that do not allow to AiiDA to properly store the calculations in the DB.

**Therefore, for production use of AiiDA, we strongly suggest to setup a “real” database** as PostgreSQL or MySQL. Then, in the “Database engine” field, type either ‘postgres’ or ‘mysql’ according to the database you chose to use. See [here](#) for the documentation to setup such databases (including info on how to proceed with `verdi install` in this case).

At the end, AiiDA will also ask to configure your user, if you set up a user different from `aiida@localhost`.

If something fails, there is a high chance that you may have misconfigured the database. Double-check your settings before reporting an error.

**Note:** The repository will contain the same number of folders as the number of nodes plus the number of workflows. For very large databases, some operations on the repository folder, such as `rsync` or scanning its content, might be very slow, and if they are performed regularly this will slow down the computer due to an intensive use of the hard drive. Check out our [tips](#) in the troubleshooting section in case this happens.

## Start the daemon

If you configured your user account with your personal email (or if in general there are more than just one user) you will not be able to start the daemon with the command `verdi daemon start` before its configuration.

*If you are working in a single-user mode, and you are sure that nobody else is going to run the daemon, you can configure your user as the (only) one who can run the daemon.*

To configure the daemon, run:

```
verdi daemon configureuser
```

and (after having read and understood the warning text that appears) insert the email that you used above during the `verdi install` phase.

To try AiiDA and start the daemon, run:

```
verdi daemon start
```

If everything was done correctly, the daemon should start. You can inquire the daemon status using:

```
verdi daemon status
```

and, if the daemon is running, you should see something like:

```
* aiida-daemon[0]      RUNNING    pid 12076, uptime 0:39:05
* aiida-daemon-beat[0]  RUNNING    pid 12075, uptime 0:39:05
```

To stop the daemon, use:

```
verdi daemon stop
```

A log of the warning/error messages of the daemon can be found in `in ~/.aiida/daemon/log/`, and can also be seen using the `verdi daemon logshow` command. The daemon is a fundamental component of AiiDA, and it is in charge of submitting new calculations, checking their status on the cluster, retrieving and parsing the results of finished calculations, and managing the workflow steps.

## Congratulations, your setup is complete!

Before going on, however, you will need to setup *at least one computer* (i.e., on computational resource as a cluster or a supercomputer, on which you want to run your calculations) *and one code*. The documentation for these steps can be found [here](#).

## Optional dependencies

### CIF manipulation

For the manipulation of [Crystallographic Information Framework \(CIF\)](#) files, following dependencies are required to be installed:

- [PyCifRW](#)
- [pymatgen](#)
- [pyspglib](#)
- [jmol](#)
- [Atomic Simulation Environment \(ASE\)](#)
- [cod-tools](#)

First four can be installed from the default repositories:

```
sudo pip install pycifrw==3.6.2.1
sudo pip install pymatgen==3.0.13
sudo pip install pyspglib
sudo apt-get install jmol
```

ASE has to be installed from source:

```
curl https://wiki.fysik.dtu.dk/ase-files/python-ase-3.8.1.3440.tar.gz > python-ase-3.8.1.3440.tar.gz
tar -zxvf python-ase-3.8.1.3440.tar.gz
cd python-ase-3.8.1.3440
setup.py build
setup.py install
export PYTHONPATH=$(pwd):$PYTHONPATH
```

For the setting up of cod-tools please refer to [installation of cod-tools](#).

## Further comments and troubleshooting

- For some reasons, on some machines (notably often on Mac OS X) there is no default locale defined, and when you run `verdi install` for the first time it fails (see also [this issue](#) of django). To solve the problem, first remove the sqlite database that was created.

Then, run in your terminal (or maybe even better, add to your `.bashrc`, but then remember to open a new shell window!):

```
export LANG="en_US.UTF-8"
export LC_ALL="en_US.UTF-8"
```

and then run `verdi install` again.

- *[Only for developers]* The developer tests of the *SSH* transport plugin are performed connecting to `localhost`. The tests will fail if a passwordless *ssh* connection is not set up. Therefore, if you want to run the tests:

- make sure to have a ssh server. On Ubuntu, for instance, you can install it using:

```
sudo apt-get install openssh-server
```

- Configure a ssh key for your user on your machine, and then add your public key to the authorized keys of localhost. The easiest way to achieve this is to run:

```
ssh-copy-id localhost
```

(it will ask your password, because it is connecting via ssh to localhost to install your public key inside ~/.ssh/authorized\_keys).

## Updating AiiDA from a previous version

### Updating from 0.4.1 to 0.5.0

- Stop your daemon (using `verdi daemon stop`)
- Store your aiiida source folder somewhere in case you did some modifications to some files
- Replace the aiiida folder with the new one (either from the tar.gz or, if you are using git, by doing a `git pull`). If you use the same folder name, you will not need to update the `PATH` and `PYTHONPATH` variables
- Run a `verdi` command, e.g., `verdi calculation list`. This should raise an exception, and in the exception message you will see the command to run to update the schema version of the DB (v.0.5.0 is using a newer version of the schema). The command will look like `python manage.py --aiida-profile=default migrate`, but please read the message for the correct command to run.
- If you run `verdi calculation list` again now, it should work without error messages.
- You can now restart your daemon and work as usual.

**Note:** If you modified or added files, you need to put them back in place. Note that if you were working on a plugin, the plugin interface changed: you need to change the `CalcInfo` returning also a `CodeInfo`, as specified [here](#) and also accept a `Code` object among the inputs (also described in the same page).

## 1.1.4 Setup of computers and codes

**Note:** The Ssh transport plugin referenced below is available in the EPFL version.

Before being able to run the first calculation, you need to setup at least one computer and one code, as described below.

### Remote computer requirements

A computer in AiiDA denotes any computational resource (with a batch job scheduler) on which you will run your calculations. Computers typically are clusters or supercomputers.

Requirements for a computer are:

- It must run a Unix-like operating system
- The default shell must be `bash`
- It should have a batch scheduler installed (see [here](#) for a list of supported batch schedulers)
- It must be accessible from the machine that runs AiiDA using one of the available transports (see below).

The first step is to choose the transport to connect to the computer. Typically, you will want to use the SSH transport, apart from a few special cases where SSH connection is not possible (e.g., because you cannot setup a password-less connection to the computer). In this case, you can install AiiDA directly on the remote cluster, and use the `local` transport (in this way, commands to submit the jobs are simply executed on the AiiDA machine, and files are simply copied on the disk instead of opening an SFTP connection).

If you plan to use the `local` transport, you can skip to the next section.

If you plan to use the SSH transport, you have to configure a password-less login from your user to the cluster. To do so type first (only if you do not already have some keys in your local `~/.ssh` directory - i.e. files like `id_rsa.pub`):

```
ssh-keygen -t rsa
```

Then copy your keys to the remote computer (in `~/.ssh/authorized_keys`) with:

```
ssh-copy-id YOURUSERNAME@YOURCLUSTERADDRESS
```

replacing `YOURUSERNAME` and `YOURCLUSTERADDRESS` by respectively your username and cluster address. Finally add the following lines to `~/.ssh/config` (leaving an empty line before and after):

```
Host YOURCLUSTERADDRESS
  User YOURUSERNAME
  HostKeyAlgorithms ssh-rsa
  IdentityFile YOURRSAKEY
```

replacing `YOURRSAKEY` by the path to the rsa private key you want to use (it should look like `~/.ssh/id_rsa`).

**Note:** In principle you don't have to put the `IdentityFile` line if you have only one rsa key in your `~/.ssh` folder.

Before proceeding to setup the computer, be sure that you are able to connect to your cluster using:

```
ssh YOURCLUSTERADDRESS
```

without the need to type a password. Moreover, make also sure you can connect via `sftp` (needed to copy files). The following command:

```
sftp YOURCLUSTERADDRESS
```

should show you a prompt without errors (possibly with a message saying `Connected to YOURCLUSTERADDRESS`).

**Warning:** Due to a current limitation of the current ssh transport module, we do not support ECDSA, but only RSA or DSA keys. In the present guide we've shown RSA only for simplicity. The first time you connect to the cluster, you should see something like this:

```
The authenticity of host 'YOURCLUSTERADDRESS (IP)' can't be established.
RSA key fingerprint is xx:xx:xx:xx:xx.
Are you sure you want to continue connecting (yes/no)?
```

Make sure you see RSA written. If you already installed the keys in the past, and you don't know which keys you are using, you could remove the cluster `YOURCLUSTERADDRESS` from the file `~/.ssh/known-hosts` (backup it first!) and try to ssh again. If you are not using a RSA or DSA key, you may see later on a submitted calculation going in the state `SUBMISSIONFAILED`.

**Note:** If the `ssh` command works, but the `sftp` command does not (e.g. it just prints `Connection closed`), a possible reason can be that there is a line in your `~/.bashrc` that either produces an output, or an error. Remove/comment it until no output or error is produced: this should make `sftp` working again.



Finally, try also:

```
ssh YOURCLUSTERADDRESS QUEUE_VISUALIZATION_COMMAND
```

replacing `QUEUE_VISUALIZATION_COMMAND` by the scheduler command that prints on screen the status of the queue on the cluster (i.e. `qstat` for PBSpro scheduler, `squeue` for SLURM, etc.). It should print a snapshot of the queue status, without any errors.

**Note:** If there are errors with the previous command, then edit your `~/.bashrc` file in the remote computer and add a line at the beginning that adds the path to the scheduler commands, typically (here for PBSpro):

```
export PATH=$PATH:/opt/pbs/default/bin
```

Or, alternatively, find the path to the executables (like using `which qsub`)

**Note:** If you need your remote `.bashrc` to be sourced before you execute the code (for instance to change the `PATH`), make sure the `.bashrc` file **does not** contain lines like:

```
[ -z "$PS1" ] && return
```

or:

```
case $- in
    *i*) ;;
    *) return;;
esac
```

in the beginning (these would prevent the `bashrc` to be executed when you `ssh` to the remote computer). You can check that e.g. the `PATH` variable is correctly set upon `ssh`, by typing (in your local computer):

```
ssh YOURCLUSTERADDRESS 'echo $PATH'
```

**Note:** If you need to `ssh` to a computer A first, from which you can then connect to computer B you wanted to connect to, you can use the `proxy_command` feature of `ssh`, that we also support in AiiDA. For more information, see [Using the proxy\\_command option with ssh](#).

## Computer setup and configuration

The configuration of computers happens in two steps.

**Note:** The commands use some `readline` extensions to provide default answers, that require an advanced terminal. Therefore, run the commands from a standard terminal, and not from embedded terminals as the ones included in text editors, unless you know what you are doing. For instance, the terminal embedded in `emacs` is known to give problems.

### 1. Setup of the computer, using the:

```
verdi computer setup
```

command. This command allows to create a new computer instance in the DB.

**Tip:** The code will ask you a few pieces of information. At every prompt, you can type the `?` character and press `<enter>` to get a more detailed explanation of what is being asked.

---

**Tip:** You can press <CTRL>+C at any moment to abort the setup process. Nothing will be stored in the DB.

---

**Note:** For multiline inputs (like the prepend text and the append text, see below) you have to press <CTRL>+D to complete the input, even if you do not want any text.

---

Here is a list of what is asked, together with an explanation.

- **Computer name:** the (user-friendly) name of the new computer instance which is about to be created in the DB (the name is used for instance when you have to pick up a computer to launch a calculation on it). Names must be unique. This command should be thought as a AiiDA-wise configuration of computer, independent of the AiiDA user that will actually use it.
- **Fully-qualified hostname:** the fully-qualified hostname of the computer to which you want to connect (i.e., with all the dots: `bellatrix.epfl.ch`, and not just `bellatrix`). Type `localhost` for the local transport.
- **Description:** A human-readable description of this computer; this is useful if you have a lot of computers and you want to add some text to distinguish them (e.g.: “cluster of computers at EPFL, installed in 2012, 2 GB of RAM per CPU”)
- **Enabled:** either True or False; if False, the computer is disabled and calculations associated with it will not be submitted. This allows to disable temporarily a computer if it is giving problems or it is down for maintenance, without the need to delete it from the DB.
- **Transport type:** The name of the transport to be used. A list of valid transport types can be obtained typing ?
- **Scheduler type:** The name of the plugin to be used to manage the job scheduler on the computer. A list of valid scheduler plugins can be obtained typing ?. See [here](#) for a documentation of scheduler plugins in AiiDA.
- **AiiDA work directory:** The absolute path of the directory on the remote computer where AiiDA will run the calculations (often, it is the scratch of the computer). You can (should) use the `{username}` replacement, that will be replaced by your username on the remote computer automatically: this allows the same computer to be used by different users, without the need to setup a different computer for each one. Example:

```
/scratch/{username}/aiida_work/
```

- **mpirun command:** The `mpirun` command needed on the cluster to run parallel MPI programs. You can (should) use the `{tot_num_mpi_procs}` replacement, that will be replaced by the total number of cpus, or the other scheduler-dependent fields (see the [scheduler docs](#) for more information). Some examples:

```
mpirun -np {tot_num_mpi_procs}
aprun -n {tot_num_mpi_procs}
poe
```

- **Text to prepend to each command execution:** This is a multiline string, whose content will be prepended inside the submission script before the real execution of the job. It is your responsibility to write proper `bash` code! This is intended for computer-dependent code, like for instance loading a module that should always be loaded on that specific computer. *Remember to end the input by pressing <CTRL>+D.* A practical example:

```
export NEWVAR=1
source some/file
```

A not-to-do example:

```
#PBS -l nodes=4:ppn=12
```

(it's the plugin that will do this!)

- **Text to append to each command execution:** This is a multiline string, whose content will be appended inside the submission script after the real execution of the job. It is your responsibility to write proper `bash` code! This is intended for computer-dependent code. *Remember to end the input by pressing <CTRL>+D.*

At the end, you will get a confirmation command, and also the ID in the database (pk, i.e. the principal key, and uuid).

## 2. Configuration of the computer, using the:

```
verdi computer configure COMPUTERNAME
```

command. This will allow to access more detailed configurations, that are often user-dependent and also depend on the specific transport (for instance, if the transport is SSH, it will ask for username, port, ...).

The command will try to provide automatically default answers, mainly reading the existing ssh configuration in `~/.ssh/config`, and in most cases one simply need to press enter a few times.

---

**Note:** At the moment, the in-line help (i.e., just typing `?` to get some help) is not yet supported in `verdi configure`, but only in `verdi setup`.

---

For `local` transport, you *need to run the command*, even if nothing will be asked to you. For `ssh` transport, the following will be asked:

- **username:** your username on the remote machine
- **port:** the port to connect to (the default SSH port is 22)
- **look\_for\_keys:** automatically look for the private key in `~/.ssh`. Default: `True`.
- **key\_filename:** the absolute path to your private SSH key. You can leave it empty to use the default SSH key, if you set `look_for_keys` to `True`.
- **timeout:** A timeout in seconds if there is no response (e.g., the machine is down. You can leave it empty to use the default value.
- **allow\_agent:** If `True`, it will try to use an SSH agent.
- **proxy\_command:** Leave empty if you do not need a proxy command (i.e., if you can directly connect to the machine). If you instead need to connect to an intermediate computer first, you need to provide here the command for the proxy: see documentation [here](#) for how to use this option, and in particular the notes [here](#) for the format of this field.
- **compress:** `True` to compress the traffic (recommended)
- **load\_system\_host\_keys:** `True` to load the known hosts keys from the default SSH location (recommended)
- **key\_policy:** What is the policy in case the host is not known. It is a string among the following:
  - `RejectPolicy` (default, recommended): reject the connection if the host is not known.
  - `WarningPolicy` (*not* recommended): issue a warning if the host is not known.
  - `AutoAddPolicy` (*not* recommended): automatically add the host key at the first connection to the host.

After these two steps have been completed, your computer is ready to go!

---

**Note:** To check if you set up the computer correctly, execute:

```
verdi computer test COMPUTERTNAME
```

that will run a few tests (file copy, file retrieval, check of the jobs in the scheduler queue) to verify that everything works as expected.

---

**Note:** If you are not sure if your computer is already set up, use the command:

```
verdi computer list
```

to get a list of existing computers, and:

```
verdi computer show COMPUTERTNAME
```

to get detailed information on the specific computer named COMPUTERTNAME. You have also the:

```
verdi computer rename OLDCOMPUTERTNAME NEWCOMPUTERTNAME
```

and:

```
verdi computer delete COMPUTERTNAME
```

commands, whose meaning should be self-explanatory.

---

**Note:** You can delete computers **only if** no entry in the database is using them (as for instance Calculations, or RemoteData objects). Otherwise, you will get an error message.

---

**Note:** It is possible to **disable** a computer.

Doing so will prevent AiiDA from connecting to the given computer to check the state of calculations or to submit new calculations. This is particularly useful if, for instance, the computer is under maintenance but you still want to use AiiDA with other computers, or submit the calculations in the AiiDA database anyway.

When the computer comes back online, you can re-enable it; at this point pending calculations in the TOSUBMIT state will be submitted, and calculations WITHSCHEDULER will be checked and possibly retrieved.

The relevant commands are:

```
verdi computer enable COMPUTERTNAME  
verdi computer disable COMPUTERTNAME
```

Note that the above commands will disable the computer for all AiiDA users. If instead, for some reason, you want to disable the computer only for a given user, you can use the following command:

```
verdi computer disable COMPUTERTNAME --only-for-user USER_EMAIL
```

(and the corresponding `verdi computer enable` command to re-enable it).

---

## Code setup and configuration

Once you have at least one computer configured, you can configure the codes.

In AiiDA, for full reproducibility of each calculation, we store each code in the database, and attach to each calculation a given code. This has the further advantage to make very easy to query for all calculations that were run with a given code (for instance because I am looking for phonon calculations, or because I discovered that a specific version had a bug and I want to rerun the calculations).

In AiiDA, we distinguish two types of codes: **remote** codes and **local** codes, where the distinction between the two is described here below.

### Remote codes

With remote codes we denote codes that are installed/compiled on the remote computer. Indeed, this is very often the case for codes installed in supercomputers for high-performance computing applications, because the code is typically installed and optimized on the supercomputer.

In AiiDA, a remote code is identified by two mandatory pieces of information:

- A computer on which the code is (that must be a previously configured computer);
- The absolute path of the code executable on the remote computer.

### Local codes

With local codes we denote codes for which the code is not already present on the remote machine, and must be copied for every submission. This is the case if you have for instance a small, machine-independent Python script that you did not copy previously in all your clusters.

In AiiDA, a local code can be set up by specifying:

- A folder, containing all files to be copied over at every submission
- The name of executable file among the files inside the folder specified above

### Setting up a code

The:

```
verdi code
```

command allows to manage codes in AiiDA.

To setup a new code, you execute:

```
verdi code setup
```

and you will be guided through a process to setup your code.

**Tip:** The code will ask you a few pieces of information. At every prompt, you can type the `?` character and press `<enter>` to get a more detailed explanation of what is being asked.

You will be asked for:

- **label:** A label to refer to this code. Note: this label is not enforced to be unique. However, if you try to keep it unique, at least within the same computer, you can use it later to refer and use to your code. Otherwise, you need to remember its ID or UUID.
- **description:** A human-readable description of this code (for instance “Quantum Espresso v.5.0.2 with 5.0.3 patches, pw.x code, compiled with openmpi”)
- **default input plugin:** A string that identifies the default input plugin to used to generate new calculations to use with this code. This string has to be a valid string recognized by the `CalculationFactory` function. To get the list of all available Calculation plugin strings, use the `verdi calculation plugins` command. Note: if you do not want to specify a default input plugin, you can write the string “None”, but this is strongly discouraged, because then you will not be able to use the `.new_calc` method of the `Code` object.

- **local:** either True (for local codes) or False (for remote codes). For the meaning of the distinction, see above. Depending on your choice, you will be asked for:
  - LOCAL CODES:
    - \* **Folder with the code:** The folder on your local computer in which there are the files to be stored in the AiiDA repository, and that will then be copied over to the remote computers for every submitted calculation. This must be an absolute path on your computer.
    - \* **Relative path of the executable:** The relative path of the executable file inside the folder entered in the previous step.
  - REMOTE CODES:
    - \* **Remote computer name:** The computer name as on which the code resides, as configured and stored in the AiiDA database
    - \* **Remote absolute path:** The (full) absolute path of the code executable on the remote machine

For any type of code, you will also be asked for:

- **Text to prepend to each command execution:** This is a multiline string, whose content will be prepended inside the submission script before the real execution of the job. It is your responsibility to write proper `bash` code! This is intended for code-dependent code, **like for instance loading the modules that are required for that specific executable to run**. Example:

```
module load intelmpi
```

*Remember to end the input by pressing <CTRL>+D.*

- **Text to append to each command execution:** This is a multiline string, whose content will be appended inside the submission script after the real execution of the job. It is your responsibility to write proper `bash` code! This is intended for code-dependent code. *Remember to end the input by pressing <CTRL>+D.*

At the end, you will get a confirmation command, and also the ID of the code in the database (the `pk`, i.e. the principal key, and the `uuid`).

---

**Note:** Codes are a subclass of the `Node` class, and as such you can attach any set of attributes to the code. These can be extremely useful for querying: for instance, you can attach the version of the code as an attribute, or the code family (for instance: “pw.x code of Quantum Espresso”) to later query for all runs done with a pw.x code and version more recent than 5.0.0, for instance. However, in the present AiiDA version you cannot add attributes from the command line using `verdi`, but you have to do it using Python code.

---

**Note:** You can change the label of a code by using the following command:

```
verdi code rename "ID"
```

(Without the quotation marks!) “ID” can either be the numeric ID (PK) of the code (preferentially), or possibly its label (or `label@computername`), if this string uniquely identifies a code.

You can also list all available codes (and their relative IDs) with:

```
verdi code list
```

The `verdi code list` accepts some flags to filter only codes on a given computer, only codes using a specific plugin, etc.; use the `-h` command line option to see the documentation of all possible options.

You can then get the information of a specific code with:

```
verdi code show "ID"
```

Finally, to delete a code use:

```
verdi code delete "ID"
```

(only if it wasn't used by any calculation, otherwise an exception is raised)

And now, you are ready to launch your calculations! You may want to follow to the examples of how you can submit a single calculation, as for instance the specific tutorial for [Quantum Espresso](#).

### 1.1.5 Plug-ins for AiiDA

AiiDA plug-ins are input generators and output parsers, enabling the integration of codes into AiiDA calculations and workflows.

#### Available plugins

##### Quantum Espresso

**Description** [Quantum Espresso](#) is a suite of open-source codes for electronic-structure calculations from first principles, based on density-functional theory, plane waves, and pseudopotentials, freely [available online](#). Documentation of the code and its internal details can be found in the distributed software, and in the [online forum](#) (and its [search engine](#)).

The plugins of `quantum-espresso` in AiiDA are not meant to completely automatize the calculation of the electronic properties. It is still required an underlying knowledge of how quantum espresso is working, which flags it requires, etc. A total automatization, if desired, has to be implemented at the level of a workflow.

Currently supported codes are:

- PW: Ground state properties, total energy, ionic relaxation, molecular dynamics, forces, etc...
- CP: Car-Parrinello molecular dynamics
- PH: Phonons from density functional perturbation theory
- Q2R: Fourier transform the dynamical matrices in the real space
- Matdyn: Fourier transform the dynamical matrices in the real space
- NEB: Energy barriers and reaction pathways using the Nudged Elastic Band (NEB) method

Moreover, support for further codes can be implemented adapting the **namelist** plugin.

#### Plugins

##### PW

**Description** Use the plugin to support inputs of Quantum Espresso `pw.x` executable.

##### Supported codes

- tested from `pw.x` v5.0 onwards. Back compatibility is not guaranteed (although versions 4.3x might work most of the times).

## Inputs

- **pseudo**, class *UpfData* One pseudopotential file per atomic species.

Alternatively, pseudo for every atomic species can be set with the **use\_pseudos\_from\_family** method, if a family of pseudopotentials has been installed..

- **kpoints**, class *KpointsData* Reciprocal space points on which to build the wavefunctions. Can either be a mesh or a list of points with/without weights
- **parameters**, class *ParameterData* Input parameters of pw.x, as a nested dictionary, mapping the input of QE. Example:

```
{ "CONTROL": { "calculation": "scf" },
  "ELECTRONS": { "ecutwfc": 30., "ecutrho": 100. },
}
```

See the QE documentation for the full list of variables and their meaning. Note: some keywords don't have to be specified or Calculation will enter the SUBMISSIONFAILED state, and are already taken care of by AiiDA (are related with the structure or with path to files):

```
'CONTROL', 'pseudo_dir': pseudopotential directory
'CONTROL', 'outdir': scratch directory
'CONTROL', 'prefix': file prefix
'SYSTEM', 'ibrav': cell shape
'SYSTEM', 'celldm': cell dm
'SYSTEM', 'nat': number of atoms
'SYSTEM', 'ntyp': number of species
'SYSTEM', 'a': cell parameters
'SYSTEM', 'b': cell parameters
'SYSTEM', 'c': cell parameters
'SYSTEM', 'cosab': cell parameters
'SYSTEM', 'cosac': cell parameters
'SYSTEM', 'cosbc': cell parameters
```

- **structure**, class *StructureData*
- **settings**, class *ParameterData* (optional) An optional dictionary that activates non-default operations. Possible values are:
  - **'FIXED\_COORDS'**: a list Nx3 booleans, with N the number of atoms. If True, the atomic position is fixed (in relaxations/md).
  - **'GAMMA\_ONLY'**: boolean. If True and the kpoint mesh is gamma, activate a speed up of the calculation.
  - **'NAMELISTS'**: list of strings. Specify all the list of Namelists to be printed in the input file.
  - **'PARENT\_FOLDER\_SYMLINK'**: boolean # If True, create a symlink to the scratch of the parent folder, otherwise the folder is copied (default: False)
  - **'CMDLINE'**: list of strings. parameters to be put after the executable and before the input file. Example: `["-npool", "4"]` will produce `pw.x -npool 4 < aiida.in`
  - **'ADDITIONAL\_RETRIEVE\_LIST'**: list of strings. Specify additional files to be retrieved. By default, the output file and the xml file are already retrieved.
  - **'ALSO\_BANDS'**: boolean. If True, retrieves the band structure (default: False)
  - **'FORCE\_KPOINTS\_LIST'**: If it is set to True and the KpointsData have a mesh set, it will pass the kpoints to QE as if they were a list of coordinates, generating a list of points. (at the moment used for wannier90)



- **parent\_folder**, class *RemoteData* (optional) If specified, the scratch folder coming from a previous QE calculation is copied in the scratch of the new calculation.
- **vdw\_table**, class *SinglefileData* (optional) If specified, it should be a file for the van der Waals kernel table. The file is copied in the pseudo subfolder, without changing its name, and without any check, so it is your responsibility to select the correct file that you want to use.

## Outputs

**Note:** The *output\_parameters* has more parsed values in the EPFL version and *output\_bands* is parsed only in the EPFL version.

There are several output nodes that can be created by the plugin, according to the calculation details. All output nodes can be accessed with the `calculation.out` method.

- *output\_parameters* *ParameterData* (accessed by `calculation.res`) Contains the scalar properties. Example: energy (in eV), total\_force (modulus of the sum of forces in eV/Angstrom), warnings (possible error messages generated in the run).
- *output\_array* *ArrayData* Produced in case of calculations which do not change the structure, otherwise, an *output\_trajectory* is produced. Contains vectorial properties, too big to be put in the dictionary. Example: forces (eV/Angstrom), stresses, ionic positions. Quantities are parsed at every step of the ionic-relaxation / molecular-dynamics run.
- *output\_trajectory* *ArrayData* Produced in case of calculations which change the structure, otherwise an *output\_array* is produced. Contains vectorial properties, too big to be put in the dictionary. Example: forces (eV/Angstrom), stresses, ionic positions. Quantities are parsed at every step of the ionic-relaxation / molecular-dynamics run.
- *output\_band* (non spin polarized calculations) or *output\_band1* + *output\_band2* (spin polarized calculations) *BandsData* Present only if parsing is activated with the ‘ALDO\_BANDS’ setting. Contains the list of electronic energies for every kpoint. If calculation is a molecular dynamics or a relaxation run, bands refer only to the last ionic configuration.
- *output\_structure* *StructureData* Present only if the calculation is moving the ions. Cell and ionic positions refer to the last configuration.
- *output\_kpoints* *KpointsData* Present only if the calculation changes the cell shape. Kpoints refer to the last structure.

**Errors** Errors of the parsing are reported in the log of the calculation (accessible with the `verdi calculation logshow` command). Moreover, they are stored in the *ParameterData* under the key `warnings`, and are accessible with `Calculation.res.warnings`.

## CP

**Description** Use the plugin to support inputs of Quantum Espresso pw.x executable.

## Supported codes

- tested from pw.x v5.0 onwards. Back compatibility is not guaranteed (although versions 4.3x might work most of the times).

## Inputs

- **pseudo**, class *UpfData* One pseudopotential file per atomic species.

Alternatively, pseudo for every atomic species can be set with the **use\_pseudos\_from\_family** method, if a family of pseudopotentials has been installed..

- **parameters**, class *ParameterData* Input parameters of cp.x, as a nested dictionary, mapping the input of QE. Example:

```
{ "ELECTRONS": { "ecutwfc": "30", "ecutrho": "100" },
}
```

See the QE documentation for the full list of variables and their meaning. Note: some keywords don't have to be specified or Calculation will enter the SUBMISSIONFAILED state, and are already taken care of by AiiDA (are related with the structure or with path to files):

```
'CONTROL', 'pseudo_dir': pseudopotential directory
'CONTROL', 'outdir': scratch directory
'CONTROL', 'prefix': file prefix
'SYSTEM', 'ibrav': cell shape
'SYSTEM', 'celldm': cell dm
'SYSTEM', 'nat': number of atoms
'SYSTEM', 'ntyp': number of species
'SYSTEM', 'a': cell parameters
'SYSTEM', 'b': cell parameters
'SYSTEM', 'c': cell parameters
'SYSTEM', 'cosab': cell parameters
'SYSTEM', 'cosac': cell parameters
'SYSTEM', 'cosbc': cell parameters
```

- **structure**, class *StructureData* The initial ionic configuration of the CP molecular dynamics.
- **settings**, class *ParameterData* (optional) An optional dictionary that activates non-default operations. Possible values are:
  - **'FIXED\_COORDS'**: a list Nx3 booleans, with N the number of atoms. If True, the atomic position is fixed (in relaxations/md).
  - **'NAMELISTS'**: list of strings. Specify all the list of Namelists to be printed in the input file.
  - **'PARENT\_FOLDER\_SYMLINK'**: boolean # If True, create a symlink to the scratch of the parent folder, otherwise the folder is copied (default: False)
  - **'CMDLINE'**: list of strings. parameters to be put after the executable and before the input file. Example: ["-npool", "4"] will produce *pw.x -npool 4 < aiida.in*
  - **'ADDITIONAL\_RETRIEVE\_LIST'**: list of strings. Specify additional files to be retrieved. By default, the output file and the xml file are already retrieved.
  - **'ALSO\_BANDS'**: boolean. If True, retrieves the band structure (default: False)
- **parent\_folder**, class *RemoteData* (optional) If specified, the scratch folder coming from a previous QE calculation is copied in the scratch of the new calculation.

**Outputs** There are several output nodes that can be created by the plugin, according to the calculation details. All output nodes can be accessed with the `calculation.out` method.

- **output\_parameters** *ParameterData* (accessed by `calculation.res`) Contains the scalar properties. Example: energies (in eV) of the last configuration, wall\_time, warnings (possible error messages generated in the run).

- `output_trajectory_array` *TrajectoryData* Contains vectorial properties, too big to be put in the dictionary, like energies, positions, velocities, cells, at every saved step.
- `output_structure` *StructureData* Structure of the last step.

**Errors** Errors of the parsing are reported in the log of the calculation (accessible with the `verdi calculation logshow` command). Moreover, they are stored in the `ParameterData` under the key `warnings`, and are accessible with `Calculation.res.warnings`.

## PH

**Note:** The PH plugin referenced below is available in the EPFL version.

**Description** Plugin for the Quantum Espresso `ph.x` executable.

### Supported codes

- tested from `ph.x` v5.0 onwards. Back compatibility is not guaranteed (although versions 4.3x might work most of the times).

### Inputs

- **parent\_calculation**, can either be a PW calculation to get the ground state on which to compute the phonons, or a PH calculation in case of restarts.

Note: There are no direct links between calculations. The `use_parent_calculation` will set a link to the Remote-Folder attached to that calculation. Alternatively, the method `use_parent_folder` can be used to set this link directly.

- **qpoints**, class *KpointsData* Reciprocal space points on which to build the dynamical matrices. Can either be a mesh or a list of points. Note: up to QE 5.1 only either an explicit list of 1 qpoint (1 point only) can be provided, or a mesh (containing gamma).
- **parameters**, class *ParameterData* Input parameters of `ph.x`, as a nested dictionary, mapping the input of QE. Example:

```
{ "INPUTPH": { "ethr-ph": 1e-16 },
}
```

See the QE documentation for the full list of variables and their meaning. Note: some keywords don't have to be specified or Calculation will enter the SUBMISSIONFAILED state, and are already taken care of by AiiDA (are related with the structure or with path to files):

```
'INPUTPH', 'outdir': scratch directory
'INPUTPH', 'prefix': file prefix
'INPUTPH', 'verbosity': file prefix
'INPUTPH', 'fildyn': file prefix
'INPUTPH', 'ldisp': logic displacement
'INPUTPH', 'nq1': q-mesh on b1
'INPUTPH', 'nq2': q-mesh on b2
'INPUTPH', 'nq3': q-mesh on b3
'INPUTPH', 'qplot': flag for list of qpoints
```

- **settings**, class *ParameterData* (optional) An optional dictionary that activates non-default operations. Possible values are:

- ‘**PARENT\_CALC\_OUT\_SUBFOLDER**’: string. The subfolder of the parent scratch to be copied in the new scratch.
- ‘**PREPARE\_FOR\_D3**’: boolean. If True, more files are created in preparation of the calculation of a D3 calculation.
- ‘**NAMELISTS**’: list of strings. Specify all the list of Namelists to be printed in the input file.
- ‘**PARENT\_FOLDER\_SYMLINK**’: boolean # If True, create a symlink to the scratch of the parent folder, otherwise the folder is copied (default: False)
- ‘**CMDLINE**’: list of strings. parameters to be put after the executable and before the input file. Example: [”-npool”,”4”] will produce *ph.x -npool 4 < aiida.in*
- ‘**ADDITIONAL\_RETRIEVE\_LIST**’: list of strings. Extra files to be retrieved. By default, dynamical matrices, text output and main xml files are retrieved.

**Outputs** There are several output nodes that can be created by the plugin, according to the calculation details. All output nodes can be accessed with the `calculation.out` method.

- `output_parameters` *ParameterData* (accessed by `calculation.res`) Contains small properties. Example: dielectric constant, warnings (possible error messages generated in the run). Furthermore, various `dynamical_matrix_*` keys are created, each is a dictionary containing the keys `q_point` and `frequencies`.

**Errors** Errors of the parsing are reported in the log of the calculation (accessible with the `verdi calculation logshow` command). Moreover, they are stored in the *ParameterData* under the key `warnings`, and are accessible with `Calculation.res.warnings`.

## Matdyn

---

**Note:** The Matdyn plugin referenced below is available in the EPFL version.

---

**Description** Use the plugin to support inputs of Quantum Espresso `matdyn.x` executable.

## Supported codes

- tested from `matdyn.x` v5.0 onwards. Back compatibility is not guaranteed (although versions 4.3x might work most of the times).

## Inputs

- **parameters**, class *ParameterData* Input parameters of `pw.x`, as a nested dictionary, mapping the input of QE. Example:

```
{"INPUT": {"ars": "simple"},
}
```

See the QE documentation for the full list of variables and their meaning. Note: some keywords don’t have to be specified or Calculation will enter the SUBMISSIONFAILED state, and are already taken care of by AiiDA (are related with the structure or with path to files):

```
'INPUT', 'flfrq': file with frequencies in output
'INPUT', 'flvec': file with eigenvectors
'INPUT', 'fldos': file with dos
```

```
'INPUT', 'q_in_cryst_coord': for qpoints
'INPUT', 'flfrc': input force constants
```

- **parent\_calculation**, pass the parent q2r calculation of its FolderData as the **parent\_folder** to pass the input force constants.
- **kpoints**, class *KpointsData* Points on which to compute the interpolated frequencies. Must contain a list of kpoints.

**Outputs** There are several output nodes that can be created by the plugin, according to the calculation details. All output nodes can be accessed with the `calculation.out` method.

- `output_parameters` *ParameterData* (accessed by `calculation.res`) Contains warnings
- `output_phonon_bands` *BandsData* Phonon frequencies as a function of qpoints.

**Errors** Errors of the parsing are reported in the log of the calculation (accessible with the `verdi calculation logshow` command). Moreover, they are stored in the *ParameterData* under the key `warnings`, and are accessible with `Calculation.res.warnings`.

## Q2R

**Note:** The Q2R plugin referenced below is available in the EPFL version.

**Description** Use the plugin to support inputs of Quantum Espresso q2r.x executable.

## Supported codes

- tested from q2r.x v5.0 onwards. Back compatibility is not guaranteed (although versions 4.3x might work most of the times).

## Inputs

- **parameters**, class *ParameterData* Input parameters of q2r.x, as a nested dictionary, mapping the input of QE. Example:

```
{ "INPUT": { "zasr": "simple" },
}
```

See the QE documentation for the full list of variables and their meaning. Note: some keywords don't have to be specified or Calculation will enter the SUBMISSIONFAILED state, and are already taken care of by AiiDA (are related with the structure or with path to files):

```
'INPUT', 'fildyn': name of input dynamical matrices
'INPUT', 'flfrc': name of output force constants
```

- **parent\_calculation**. Use the parent PH calculation, to take the dynamical matrices and convert them in real space. Alternatively, use the `parent_folder` to point explicitly to the retrieved FolderData of the parent PH calculation.

## Outputs

- `force_constants` *SinglefileData* A file containing the force constants in real space.

## Errors

### NEB

---

**Note:** The NEB plugin referenced below is available in the EPFL version.

---

**Description** Plugin for the Quantum Espresso `neb.x` executable.

### Supported codes

- tested from `neb.x` v5.2 onwards.

### Inputs

- **pseudo**, class *UpfData* One pseudopotential file per atomic species.  
Alternatively, pseudo for every atomic species can be set with the `use_pseudos_from_family` method, if a family of pseudopotentials has been installed..
- **kpoints**, class *KpointsData* Reciprocal space points on which to build the wavefunctions. Can either be a mesh or a list of points with/without weights
- **neb\_parameters**, class *ParameterData* Input parameters of `neb.x`, as a nested dictionary, mapping the input of QE. Example:

```
{ "PATH": { "num_of_images": 6, "string_method": "neb", "nstep_path": 50 },  
}
```

See the QE documentation for the full list of variables and their meaning.

- **pw\_parameters**, class *ParameterData* Nested dictionary containing the input parameters in PW format common to all images. Example:

```
{ "CONTROL": { "calculation": "scf" },  
  "ELECTRONS": { "ecutwfc": "30", "ecutrho": "100" },  
}
```

See the QE documentation for the full list of variables and their meaning. Note: some keywords don't have to be specified or Calculation will enter the SUBMISSIONFAILED state, and are already taken care of by AiiDA (are related with the structure or with path to files):

```
'CONTROL', 'pseudo_dir': pseudopotential directory  
'CONTROL', 'outdir': scratch directory  
'CONTROL', 'prefix': file prefix  
'SYSTEM', 'ibrav': cell shape  
'SYSTEM', 'celldm': cell dm  
'SYSTEM', 'nat': number of atoms  
'SYSTEM', 'ntyp': number of species  
'SYSTEM', 'a': cell parameters  
'SYSTEM', 'b': cell parameters  
'SYSTEM', 'c': cell parameters  
'SYSTEM', 'cosab': cell parameters  
'SYSTEM', 'cosac': cell parameters  
'SYSTEM', 'cosbc': cell parameters
```

- **first\_structure**, class *StructureData* Structure of the first image.
- **last\_structure**, class *StructureData* Structure of the last image.

- **settings**, class `ParameterData` (optional) An optional dictionary that activates non-default operations. Possible values are:
  - **‘CLIMBING\_IMAGES’**: list of integers. Specify the indices of the climbing images. Read only if the climbing image scheme is set to manual in `neb_parameters`.
  - **‘FIXED\_COORDS’**: a list Nx3 booleans, with N the number of atoms. If True, the atomic position is fixed.
  - **‘GAMMA\_ONLY’**: boolean. If True and the kpoint mesh is gamma, activate a speed up of the calculation.
  - **‘NAMELISTS’**: list of strings. Specify all the list of Namelists to be printed in the input file.
  - **‘PARENT\_FOLDER\_SYMLINK’**: boolean. If True, create a symlink to the scratch of the parent folder, otherwise the folder is copied (default: False)
  - **‘CMDLINE’**: list of strings. parameters to be put after the executable in addition to `-input_images 2`. Example: `['-npool','4']` will produce `neb.x -input_images 2 -npool 4 > aiida.out`
  - **‘ADDITIONAL\_RETRIEVE\_LIST’**: list of strings. Specify additional files to be retrieved. By default, the following files are already retrieved: \* NEB output file \* PATH output file containing the information on structures and gradients of each image at last iteration \* The calculated and interpolated energy profile as a function of the reaction coordinate (`.dat` and `.int` files) \* The PW output and xml file for each image
  - **‘ALL\_ITERATIONS’**: boolean. If true the energies and forces for each image at each intermediate iteration are also parsed and stored in the output node `iteration_array` (default: False)
- **parent\_folder**, class `RemoteData` (optional) If specified, the scratch folder coming from a previous NEB calculation is copied in the scratch of the new calculation.

**Outputs** There are several output nodes that can be created by the plugin, according to the calculation details. All output nodes can be accessed with the `calculation.out` method.

- `output_parameters` `ParameterData` (accessed by `calculation.res`) Contains the data obtained by parsing the NEB output file. Information on the last iteration are only reported. The parsed PW outputs of each image are also reported as a subdictionaries.
- `mep_array` `ArrayData` Contains the parsed data on the calculated and interpolated Minimum Energy Path (MEP), i.e. the energy profile as a function of the reaction coordinate.
- `output_trajectory` `ArrayData` Contains the structure of the images at the last iteration of the NEB calculation, too big to be put in the dictionary.
- `iteration_array` `ArrayData`, and other quantities at intermediate iterations.

**Errors** Errors of the parsing are reported in the log of the calculation (accessible with the `verdi calculation logshow` command). Moreover, they are stored in the `ParameterData` under the key `warnings`, and are accessible with `Calculation.res.warnings`.

## cod-tools

**Description** `cod-tools` is an open-source collection of command line scripts for handling of [Crystallographic Information Framework \(CIF\)](#) files. The package is developed by the team of [Crystallography Open Database \(COD\)](#) developers. Detailed information for the usage of each individual script from the package can be obtained by invoking commands with `--help` and `--usage` command line options. For example:

```
cif_filter --help
cif_filter --usage
```

- **cif\_cod\_check** Parse a CIF file, check if certain data values match COD requirements and IUCr data validation criteria (Version: 2000.06.09, <ftp://ftp.iucr.ac.uk/pub/dvntests> or <ftp://ftp.iucr.org/pub/dvntests>)
- **cif\_cod\_deposit** Deposit CIFs into COD database using CGI deposition interface.
- **cif\_cod\_numbers** Find COD numbers for the .cif files in given directories of file lists.
- **cif\_correct\_tags** Correct misspelled tags in a CIF file.
- **cif\_filter** Parse a CIF file and print out essential data values in the CIF format, the COD CIF style.

This script has also many capabilities – it can restore spacegroup symbols from symmetry operators (consulting pre-defined tables), parse and tidy-up `_chemical_formula_sum`, compute cell volume, exclude unknown or “empty” tags, and add specified bibliography data.

- **cif\_fix\_values** Correct temperature values which have units specified or convert between Celsius degrees and Kelvins. Changes ‘room/ambiente temperature’ to the appropriate numeric value. Fixes other undefined values (no, not measured, etc.) to ‘?’ symbol. Determine a report about changes made into standart I/O streams.

Fixes enumeration values in CIF file against CIF dictionaries.

- **cif\_mark\_disorder** Marks disorder in CIF files judging by distance and occupancy.
- **cif\_molecule** Restores molecules from a CIF file.
- **cif\_select** Read CIFs and print out selected tags with their values.
- **cif\_split** Split CIF files into separate files with one `data_` section each.

This script parses given CIF files to separate the datablocks, so is capable of splitting non-correctly formatted and nested CIF files.

- **cif\_split\_primitive** Split CIF files into separate files with one `data_` section each.

This is a very naive and primitive version of the splitter, which expects each `data_...` section to start on a new line. It may fail on some CIF files that do not follow such convention. For splitting of any correctly formatted CIF files, one must do full CIF parsing using CIF grammar and tokenisation of the file.

**Installation** Currently **cod-tools** package is distributed via source code only. To prepare the package for usage (as of source revision 2930) one has to follow these steps:

- Retrieve the source from the [Subversion](#) repository:

```
svn co svn://www.crystallography.net/cod-tools/trunk cod-tools
```

- Install the dependencies:

```
bash -e cod-tools/dependencies/Ubuntu-12.04/install.sh
```

---

**Note:** the dependency installer is written for Ubuntu 12.04, but works fine on some older or newer Ubuntu as well as Debian distributions.

---

- Build and test:

```
make -C cod-tools
```

- **Prepare the environment:** Described below are two methods of setting the environment for **cod-tools** as of source revision 3393:

- Using Bash:



```
CODTOOLS_SRC=~/.src/cod-tools

export PATH=${CODTOOLS_SRC}/scripts:${PATH}
export PERL5LIB=${CODTOOLS_SRC}/src/lib/perl5:${PERL5LIB}
```

These commands can be pasted to `~/.bashrc` file, which is sourced automatically by the AiiDA before each calculation.

**Note:** Be sure to restart the AiiDA daemon after modifying the `~/.bashrc`.

– Using `modulefile`:

```
##Module1.0#####
module-whatis      loads the cod-tools environment

set                CODTOOLS_SRC    ~/.src/cod-tools
prepend-path       PATH            ${CODTOOLS_SRC}/scripts
prepend-path       PERL5LIB        ${CODTOOLS_SRC}/src/lib/perl5
```

## Examples

- Fix a syntactically incorrect structure:

Some simple common CIF syntax errors can be fixed automatically using `cif_filter` with `--fix-syntax` option. In example, such structure:

```
data_broken
_publ_section_title "Runaway quote
loop_
_atom_site_label
_atom_site_fract_x
_atom_site_fract_y
_atom_site_fract_z
C 0 0 0
```

can be fixed (provided it's stored in `test.cif`):

```
cif_filter --fix test.cif
```

Obtained structure:

```
data_broken
_publ_section_title      'Runaway quote'
loop_
_atom_site_label
_atom_site_fract_x
_atom_site_fract_y
_atom_site_fract_z
C 0 0 0
```

A warning message tells what was done:

```
cif_filter: test.cif(2) data_broken: warning, double-quoted string is missing a closing quote --
```

where:

- `cif_filter` is the name of the used script;
- `test.cif` is the name of the CIF file;

- 2 is the number of a line in the file;
  - data\_broken is the CIF datablock name;
  - warning is the level of severity;
  - rest is the message text.
- Fetch a structure from Web, filter and fix it, restore the crystal contents and calculate summary formulae per each compound in a crystal:

```
curl --silent http://www.crystallography.net/cod/2231955.cif \  
| cif_filter \  
| cif_fix_values \  
| cif_molecule \  
| cif_cell_contents --use-attached-hydrogens
```

Obtained result:

```
C9 H14 N  
C10 H6 O6 S2  
H2 O
```

As well as a warning message:

```
cif_molecule: - data_2231955: WARNING, multiplicity ratios are given instead of multiplicities f
```

- Fetch a structure from Web and mark alternative atoms sharing same site:

```
curl --silent http://www.crystallography.net/2018107.cif \  
| cif_mark_disorder \  
| cif_select --cif --tag _atom_site_label
```

Obtained result:

```
data_2018107  
loop_  
_atom_site_type_symbol  
_atom_site_label  
_atom_site_fract_x  
_atom_site_fract_y  
_atom_site_fract_z  
_atom_site_u_iso_or_equiv  
_atom_site_adp_type  
_atom_site_calc_flag  
_atom_site_refinement_flags  
_atom_site_occupancy  
_atom_site_symmetry_multiplicity  
_atom_site_disorder_assembly  
_atom_site_disorder_group  
Pb Pb1 0.5000 0.0000 0.2500 0.0213(13) Uani d S 1 4 . .  
Mo Mo2 0.0000 0.0000 0.0000 0.022(4) Uani d S 1 4 . .  
Pb Pb3 0.5000 0.5000 0.0000 0.025(2) Uani d SP 0.881(8) 4 A 1  
Mo Mo3 0.5000 0.5000 0.0000 0.025(2) Uani d SP 0.119(8) 4 A 2  
Mo Mo1 0.0000 0.5000 0.2500 0.018(3) Uani d S 1 4 . .  
O O1 0.2344(13) -0.1372(14) 0.0806(6) 0.0302(17) Uani d . 1 1 . .  
O O2 0.2338(14) 0.3648(14) 0.1697(6) 0.0307(17) Uani d . 1 1 . .
```

As well as output messages:

```
cif_mark_disorder: - data_2018107: NOTE, atoms 'Mo3', 'Pb3' were marked as alternatives.
cif_mark_disorder: - data_2018107: NOTE, 1 site(s) were marked as disorder assemblies.
```

**Note:** atoms Mo3 and Pb3 share the same site, as can be found out by checking their coordinates. Moreover, sum of their occupancies are close to 1. In the original CIF file these sites have both `_atom_site_disorder_assembly` and `_atom_site_disorder_group` set to `‘.’`.

## Plugins

### `codtools.ciffilter`

**Description** This plugin is designed for filter-like codes from the **cod-tools** package, but can be adapted to any command line utilities, accepting CIF file as standard input and producing CIF file as standard output and messages/errors in the standard output (if any), without modifications.

### Supported codes

- `cif_adjust_journal_name_volume`
- `cif_CODify`
- `cif_correct_tags`
- `cif_create_AMCSD_pressure_temp_tags`
- `cif_estimate_spacegroup`
- `cif_eval_numbers`
- `cif_fillcell`
- `cif_filter`
- `cif_fix_values`
- `cif_hkl_check`
- `cif_mark_disorder`
- `cif_molecule`
- `cif_pl`
- `cif_reformat_AMCSD_author_names`
- `cif_reformat_pubmed_author_names`
- `cif_reformat_uppercase_author_names`
- `cif_select`<sup>1</sup>
- `cif_set_value`
- `cif_symop_apply`

<sup>1</sup> Only with the `--output-cif` command line option.

## Inputs

- *CifData* A CIF file.
- *ParameterData* (optional) Contains the command line parameters, specified in key-value fashion. Leading dashes (single or double) must be stripped from the keys. Values can be arrays with multiple items. Keys without values should point to boolean True value. In example:

```
calc = Code.get_from_string('cif_filter').new_calc()
calc.use_parameters(ParameterData(dict={
    's' : True,
    'exclude-empty-tags' : True,
    'dont-reformat-spacegroup' : True,
    'add-cif-header' : [ 'standard.txt', 'user.txt' ],
    'bibliography' : 'bibliography.cif',
}))
```

is equivalent to command line:

```
cif_filter \
-s \
--exclude-empty-tags \
--dont-reformat-spacegroup \
--add-cif-header standard.txt \
--add-cif-header user.txt \
--bibliography bibliography.cif
```

---

**Note:** it should be kept in mind that no escaping of Shell metacharacters are performed by the plugin. AiiDA encloses each command line argument with single quotes and that's being relied on.

---

## Outputs

- *CifData* A CIF file.
- *ParameterData* (optional) Contains lines of output messages and/or errors. For example:

```
print load_node(1, parent_class=ParameterData).get_dict()
```

would print:

```
{u'output_messages': [u'cif_cod_check: test.cif data_4000000: _publ_section_title is undefin
```

**Errors** Run-time errors are returned line-by-line in the *ParameterData* object.

## codtools.cifcellcontents

**Description** This plugin is used for chemical formula calculations from the CIF files, as being done by `cif_cell_contents` code from the **cod-tools** package.

## Supported codes

- `cif_cell_contents`

## Inputs

- **CifData** A CIF file.
- **ParameterData** (optional) Contains the command line parameters, specified in key-value fashion. For more information refer to *inputs for codtools.ciffilter plugin*.

## Outputs

- **ParameterData** Contains formulae in (CIF datablock name, 'formula') pairs. For example:

```
print load_node(1, parent_class=ParameterData).get_dict()
```

would print:

```
{u'formulae': {
    u'4000001': u'C24 H17 F5 Fe',
    u'4000002': u'C24 H17 F5 Fe',
    u'4000003': u'C24 H17 F5 Fe',
    u'4000004': u'C22 H8 F10 Fe'
}}
```

**Note:** data\_ is not prepended to the CIF datablock name – the CIF file, used for the example above, contains CIF datablocks data\_4000001, data\_4000002, data\_4000003 and data\_4000004.

- **ParameterData** Contains lines of output messages and/or errors. For more information refer to *outputs for codtools.ciffilter plugin*.

**Errors** Run-time errors are returned line-by-line in the *ParameterData* object.

## codtools.cifcodcheck

**Description** This plugin is specific for cif\_cod\_check script.

## Supported codes

- cif\_cod\_check

## Inputs

- **CifData** A CIF file.
- **ParameterData** (optional) Contains the command line parameters, specified in key-value fashion. For more information refer to *inputs for codtools.ciffilter plugin*.

## Outputs

- **ParameterData** Contains lines of output messages and/or errors. For more information refer to *outputs for codtools.ciffilter plugin*.

**Errors** Run-time errors are returned line-by-line in the *ParameterData* object.

## codtools.cifcoddeposit

**Description** This plugin is specific for `cif_cod_deposit` script.

### Supported codes

- `cif_cod_deposit`

### Inputs

- ***CifData*** A CIF file.
- ***ParameterData*** Contains deposition information, such as user name, password and deposition type:
  - username: depositor's user name to access the \*COD deposition interface;
  - password: depositor's password to access the \*COD deposition interface;
  - deposition-type: determines a type of the deposited CIF file, which can be one of the following:
    - \* published: CIF file is already published in a scientific paper;
    - \* prepublication: CIF file is a prepublication material and should not be revealed to the public until the publication of a scientific paper. In this case, a `hold_period` also has to be specified;
    - \* personal: CIF file is personal communication.
  - url: URL of \*COD deposition API (optional, default URL is <http://test.crystallography.net/cgi-bin/cif-deposit.pl>);
  - journal: name of the journal, where the CIF is/will be published;
  - user\_email: depositor's e-mail address;
  - author\_name: name of the CIF file author;
  - author\_email: e-mail of the CIF file author;
  - hold\_period: a period (in number months) for the structure to be kept on-hold (only for `deposition_type == 'prepublication'`).

### Outputs

- ***ParameterData*** Contains the result of deposition:
  - output\_messages: lines of output messages and/or errors. For more information refer to *[outputs for codtools.ciffilter plugin](#)*.
  - status: a string, one of the following:
    - \* SUCCESS: a deposition is successful, newly assigned \*COD number(s) is/are present in output\_messages field;
    - \* DUPLICATE: submitted data is already in the \*COD database thus is not deposited once more;
    - \* UNCHANGED: the redeposition of the data is unnecessary, as nothing was changed in the contents of file to be replaced;
    - \* INPUTERROR: an error, related to the input, has occurred, detailed reason may be present in output\_messages field;
    - \* SERVERERROR: an internal server error has occurred, detailed reason may be present in output\_messages field;
    - \* UNKNOWN: the result of the deposition is unknown.

**Errors** Run-time errors are returned line-by-line in the `output_messages` field of *ParameterData* object.

### codtools.cifcodnumbers

**Description** This plugin is specific for `cif_cod_numbers` script.

#### Supported codes

- `cif_cod_numbers`

#### Inputs

- *CifData* A CIF file.
- *ParameterData* (optional) Contains the command line parameters, specified in key-value fashion. For more information refer to *inputs for codtools.ciffilter plugin*.

#### Outputs

- *ParameterData* Contains two subdictionaries: `duplicates` and `errors`. In `duplicates` correspondence between the database and supplied file(s) is described. Example:

```
{
  "duplicates": [
    {
      "codid": "4000099",
      "count": 1,
      "formula": "C50_H44_N2_Ni_O4"
    }
  ],
  "errors": []
}
```

Here `codid` is numeric ID of a hit in the database, `count` is total number of hits for the particular datablock and `formula` is the summary formula of the described datablock.

**Errors** Run-time errors are returned line-by-line in the *ParameterData* object.

### codtools.cifsplitprimitive

**Description** This plugin is used by `cif_split` and `cif_split_primitive` codes from the **cod-tools** package.

#### Supported codes

- `cif_split`<sup>2</sup>
- `cif_split_primitive`

<sup>2</sup> Incompatible with `--output-prefixed` and `--output-tar` command line options.

### Inputs

- *CifData* A CIF file.
- *ParameterData* (optional) Contains the command line parameters, specified in key-value fashion. For more information, refer to *inputs for codtools.ciffilter plugin*.

### Outputs

- **List of *CifData*** One or more CIF files.
- *ParameterData* (optional) Contains lines of output messages and/or errors.

**Errors** Run-time errors are returned line-by-line in the *ParameterData* object.

### ASE

---

**Note:** The ASE plugin referenced below is available in the EPFL version.

---

**Description** *ASE* (Atomic Simulation Environment) is a set of tools and Python modules for setting up, manipulating, running, visualizing and analyzing atomistic simulations. The ASE code is freely available under the GNU LGPL license (the ASE installation guide and the source can be found [here](#)).

Besides the manipulation of structures (*Atoms* objects), one can attach *calculators* to a structure and run it to compute, as an example, energies or forces. Multiple calculators are currently supported by ASE, like GPAW, Vasp, Abinit and many others.

In AiiDA, we have developed a plugin which currently supports the use of ASE calculators for total energy calculations and structure optimizations.

### Plugins

#### ASE

---

**Note:** The ASE plugin referenced below is available in the EPFL version.

---

**Description** Use the plugin to support inputs of ASE structure optimizations and of total energy calculations. Requires the installation of ASE on the computer where AiiDA is running.

### Supported codes

- tested on ASE v3.8.1 and on GPAW v0.10.0. ASE back compatibility is not guaranteed. Calculators different from GPAW should work, if they follow the interface description of ASE calculators, but have not been tested. Usage requires the installation of both ASE and of the software used by the calculator.

### Inputs

- **kpoints**, class *KpointsData* (optional) Reciprocal space points on which to build the wavefunctions. Only kpoints meshes are currently supported.
- **parameters**, class *ParameterData* Input parameters that defines the calculations to be performed, and their parameters. See the ASE documentation for more details.



- **structure**, class *StructureData*
- **settings**, class *ParameterData* (optional) An optional dictionary that activates non-default operations. Possible values are:
  - ‘**CMDLINE**’: list of strings. parameters to be put after the executable and before the input file. Example: `["-npool","4"]` will produce `gpaw -npool 4 < aiiida_input`
  - ‘**ADDITIONAL\_RETRIEVE\_LIST**’: list of strings. Specify additional files to be retrieved. By default, the output file and the xml file are already retrieved.

**Outputs** Actual output production depends on the input provided.

- **output\_parameters** *ParameterData* (accessed by `calculation.res`) Contains the scalar properties. Example: energy (in eV) or warnings (possible error messages generated in the run).
- **output\_array** *ArrayData* Stores vectorial quantities (lists, tuples, arrays), if requested in output. Example: forces, stresses, positions. Units are those produced by the calculator.
- **output\_structure** *StructureData* Present only if the structure is optimized.

**Errors** Errors of the parsing are reported in the log of the calculation (accessible with the `verdi calculation logshow` command). Moreover, they are stored in the *ParameterData* under the key `warnings`, and are accessible with `Calculation.res.warnings`.

**Examples** The following example briefly describe the usage of GPAW within AiiDA, assuming that both ASE and GPAW have been installed on the remote machine. Note that ASE calculators, at times, require the definition of environment variables. Take your time to find them and make sure that they are loaded by the submission script of AiiDA (use the `prepend_text` fields of a *Code*, for example).

First of all install the AiiDA Code as usual, noting that, if you plan to use the serial version of GPAW (applies to all other calculators) the remote absolute path of the code has to point to the python executable (i.e. the output of `which python` on the remote machine, typically it might be `/usr/bin/python`). If the parallel version of GPAW is used, set instead the path to `gpaw-python`.

To understand the plugin, it is probably easier to try to run one test, to see the python script which is produced and executed on the remote machine. We describe in the following some example script, which can be called through the `verdi run` command (example: `verdi run test_script.py`). You should see a folder `submit_test` created in the location from which you run the command. Here there is the input script that is going to be executed in the remote machine, with the syntax of the ASE software.

In this first example script and execute it with the `verdi run` command. This is a minimal script that uses GPAW and a plane-wave basis to compute the total energy of a structure. Note that for a serial calculation, it is necessary to run the `calculation.set_with_mpi(False)` method. Note also, that by default, only the total energy of the structure is computed and retrieved.

This second example instead shows a demo of all possible options supported by the current plugin. By specifying an optimizer key in the dictionary, the ASE optimizers are run. In the example, the QuasiNewton algorithm is run to minimize the forces and find the equilibrium structures. By specifying the key “`calculator_getters`”, the code will get from the calculator, the properties which are specified in the value, using the `get` method of the calculator; similar applies for the `atoms_getters`, which will call the `atoms.get` method. `extra_lines` and `post_lines` are used to insert python commands that are executed before or after the call to the calculators. `extra_imports` is used to specify the import of more modules.

Lastly, this script is an example of how to run GPAW parallel. Essentially, nothing has to be changed in input, except that there is no need to call the method `calculation.set_with_mpi(False)`.

## NWChem

**Description** NWChem is an open-source high performance computational chemistry tool.

### Plugins

#### nwchem.basic

**Description** A very simple plugin for main NWChem's nwchem executable.

#### Inputs

- **StructureData** A structure.
- **ParameterData (optional)** A dictionary with control variables. An example (default values):

```
{
  "abbreviation": "aiida_calc",          # Short name for the computation
  "title":        "AiiDA NWChem calculation", # Long name for the computation
  "basis":        # Basis per chemical type
  {
    "Ba": "library 6-31g",
    "Ti": "library 6-31g",
    "O":  "library 6-31g",
  },
  "task":         "scf",                  # Computation task
  "add_cell":     True,                  # Include cell parameters?
}
```

#### Outputs

- **ParameterData** A dictionary with energy values. For example:

```
{
  "nuclear_repulsion_energy": "9.194980930276",
  "one_electron_energy":      "-122.979939235872",
  "total_scf_energy":          "-75.983997570474",
  "two_electron_energy":       "37.800960735123"
}
```

#### nwchem.nwcpymatgen

**Description** pymatgen-based input plugin for main NWChem's nwchem executable.

#### Inputs

- **StructureData (optional)** A structure.
- **ParameterData** A dictionary with control variables.

## Outputs

- **job\_info:** *ParameterData* A dictionary of general parameters of the computation, like details of compilation, used time and memory.

May also contain one or more of the following:

- **output:** *ParameterData* A dictionary describing the job. An example:

```
{
  "basis_set": {},
  "corrections": {},
  "energies": [],
  "errors": [],
  "frequencies": null,
  "has_error": false,
  "job_type": "NWChem Geometry Optimization"
}
```

- **trajectory:** *TrajectoryData* (optional) A trajectory, made of structures, produced in the steps of geometry optimization.

---

**Note:** Functionality to extract structures from *NWChem*'s output is not present in *pymatgen* 3.0.13 or earlier.

---

**Errors** Errors are reported in the `errors` field of output *ParameterData* dictionary. Additionally, there's a boolean flag `has_error` in the same dictionary.

## 1.1.6 Scripting with AiiDA

While many common functionalities are provided by either command-line tools (via `verdi`) or the web interface, for fine tuning (or automatization) it is useful to directly access the python objects and call their methods.

This is possible in two ways, either via an interactive shell, or writing and running a script. Both methods are described below.

### `verdi shell`

By running `verdi shell` on the terminal, a new interactive *IPython* shell will be opened (this requires that *IPython* is installed on your computer).

Note that simply opening *IPython* and loading the AiiDA modules will not work (unless you perform the operations described in the *following section*) because the database settings are not loaded by default and AiiDA does not know how to access the database.

Moreover, by calling `verdi shell`, you have the additional advantage that some classes and modules are automatically loaded. In particular the following modules/classes are already loaded and available:

```
from aiida.orm import (Node, Calculation, JobCalculation, Code, Data,
                      Computer, Group, DataFactory, CalculationFactory)
from aiida.djsite.db import models
```

---

**Note:** It is possible to customize the shell by adding modules to be loaded automatically, thanks to the `verdi devel setproperty verdishell.modules` command. See [here](#) for more information.

---

A further advantage is that bash completion is enabled, allowing to press the `TAB` key to see available submethods of a given object (see for instance the documentation of the [ResultManager](#)).

## Writing python scripts for AiiDA

Alternatively, if you do not need an interactive shell but you prefer to write a script and then launch it from the command line, you can just write a standard python `.py` file. The only modification that you need to do is to add, at the beginning of the file and before loading any other AiiDA module, the following two lines:

```
from aiida import load_dbenv
load_dbenv()
```

that will load the database settings and allow AiiDA to reach your database. Then, you can load as usual python and AiiDA modules and classes, and use them. If you want to have the same environment of the `verdi shell` interactive shell, you can also add (below the `load_dbenv` call) the following lines:

```
from aiida.orm import Calculation, Code, Computer, Data, Node
from aiida.orm import CalculationFactory, DataFactory
from aiida.djbsite.db import models
```

or simply import the only modules that you will need in the script.

While this method will work, we strongly suggest to use instead the `verdi run` command, described here below.

### The `verdi run` command and the `runaiida` executable

In order to simplify the procedure described above, it is possible to execute a python file using `verdi run`: this command will accept as parameter the name of a file, and will execute it after having loaded the modules described above.

The command `verdi run` has the additional advantage of adding all stored nodes to suitable special groups, of type `autogroup.run`, for later usage. You can get the list of all these groups with the command:

```
verdi group list -t autogroup.run
```

Some further command line options of `verdi run` allow the user to fine-tune the autogrouping behavior; for more details, refer to the output of `verdi run -h`. Note also that further command line parameters to `verdi run` are passed to the script as `sys.argv`.

**Note:** It is not possible to run multiple times the `load_dbenv()` command. Since calling `verdi run` will automatically call the `load_dbenv()` command, you cannot run a script that contains this call (this is instead needed if you want to run the script simply via `python scriptname.py`). If you want to allow for both options, use the following method to discover if the db environment was already loaded:

```
from aiida import load_dbenv, is_dbenv_loaded

if not is_dbenv_loaded():
    load_dbenv()
```

Finally, we also defined a `runaiida` command, that simply will pass all its parameters to `verdi run`. The reason for this is that one can define a new script to be run with `verdi run`, add as the first line the shebang command `#!/usr/bin/env runaiida`, and give to the file execution permissions, and the file will become an executable that is run using AiiDA. A simple example could be:

```
#!/usr/bin/env runaiida
import sys

pk = int(sys.argv[1])
node = load_node(pk)
print "Node {} is: {}".format(pk, repr(node))

import aiida
print "AiiDA version is: {}".format(aiida.get_version())
```

## 1.1.7 StructureData tutorial

### General comments

This section contains an example of how you can use the *StructureData* object to create complex crystals.

With the *StructureData* class we did not try to have a full set of features to manipulate crystal structures. Indeed, other libraries such as ASE exist, and we simply provide easy ways to convert between the ASE and the AiiDA formats. On the other hand, we tried to define a “standard” format for structures in AiiDA, that can be used across different codes.

### Tutorial

Take a look at the following example:

```
alat = 4. # angstrom
cell = [[alat, 0., 0.],
        [0., alat, 0.],
        [0., 0., alat],
        ]
s = StructureData(cell=cell)
s.append_atom(position=(0.,0.,0.), symbols='Fe')
s.append_atom(position=(alat/2.,alat/2.,alat/2.), symbols='O')
```

With the commands above, we have created a crystal structure *s* with a cubic unit cell and lattice parameter of 4 angstrom, and two atoms in the cell: one iron (Fe) atom in the origin, and one oxygen (O) at the center of the cube (this cell has been just chosen as an example and most probably does not exist).

---

**Note:** As you can see in the example above, both the cell coordinates and the atom coordinates are expressed in angstrom, and the position of the atoms are given in a global absolute reference frame.

---

In this way, any periodic structure can be defined. If you want to import from ASE in order to specify the coordinates, e.g., in terms of the crystal lattice vectors, see the guide on the conversion to/from ASE below.

When using the *append\_atom()* method, further parameters can be passed. In particular, one can specify the mass of the atom, particularly important if you want e.g. to run a phonon calculation. If no mass is specified, the mass provided by NIST (retrieved in October 2014) is going to be used. The list of masses is stored in the module *aiida.common.constants*, in the *elements* dictionary.

Moreover, in the *StructureData* class of AiiDA we also support the storage of crystal structures with alloys, vacancies or partial occupancies. In this case, the argument of the parameter *symbols* should be a list of symbols, if you want to consider an alloy; moreover, you must pass a *weights* list, with the same length as *symbols*, and with values between 0. (no occupancy) and 1. (full occupancy), to specify the fractional occupancy of that site for each of the symbols specified in the *symbols* list. The sum of all occupancies must be lower or equal to one; if the sum is lower than one, it means that there is a given probability of having a vacancy at that specific site position.

As an example, you could use:

```
s.append_atom(position=(0.,0.,0.),symbols=['Ba','Ca'],weights=[0.9,0.1])
```

to add a site at the origin of a structure *s* consisting of an alloy of 90% of Barium and 10% of Calcium (again, just an example).

The following line instead:

```
s.append_atom(position=(0.,0.,0.),symbols='Ca',weights=0.9)
```

would create a site with 90% probability of being occupied by Calcium, and 10% of being a vacancy.

Utility methods `s.is_alloy()` and `s.has_vacancies()` can be used to verify, respectively, if more than one element is given in the symbols list, and if the sum of all weights is smaller than one.

**Note:** if you pass more than one symbol, the method `s.is_alloy()` will always return `True`, even if only one symbol has occupancy 1. and all others have occupancy zero:

```
>>> s = StructureData(cell=[[4,0,0],[0,4,0],[0,0,4]])
>>> s.append_atom(position=(0.,0.,0.), symbols=['Fe','O'], weights=[1.,0.])
>>> s.is_alloy()
True
```

## Internals: Kinds and Sites

Internally, the `append_atom()` method works by manipulating the kinds and sites of the current structure. Kinds are instances of the *Kind* class and represent a chemical species, with given properties (composing element or elements, occupancies, mass, ...) and identified by a label (normally, simply the element chemical symbol).

Sites are instances of the *Site* class and represent instead each single site. Each site refers to a *Kind* to identify its properties (which element it is, the mass, ...) and to its three spatial coordinates.

The `append_atom()` works in the following way:

- It creates a new *Kind* class with the properties passed as parameters (i.e., all parameters except `position`).
- It tries to identify if an identical *Kind* already exists in the list of kinds of the structure (e.g., in the same atom with the same mass was already previously added). Comparison of kinds is performed using `aiida.orm.data.structure.Kind.compare_with()`, and in particular it returns `True` if the mass and the list of symbols and of weights are identical (within a threshold). If an identical kind *k* is found, it simply adds a new site referencing to kind *k* and with the provided `position`. Otherwise, it appends *k* to the list of kinds of the current structure and then creates the site referencing to *k*. The name of the kind is chosen, by default, equal to the name of the chemical symbol (e.g., “Fe” for iron).
- If you pass more than one species for the same chemical symbol, but e.g. with different masses, a new kind is created and the name is obtained postponing an integer to the chemical symbol name. For instance, the following lines:

```
s.append_atom(position = [0,0,0], symbols='Fe', mass = 55.8)
s.append_atom(position = [1,1,1], symbols='Fe', mass = 57)
s.append_atom(position = [1,1,1], symbols='Fe', mass = 59)
```

will automatically create three kinds, all for iron, with names `Fe`, `Fe1` and `Fe2`, and masses 55.8, 57. and 59. respectively.

- In case of alloys, the kind name is obtained concatenating all chemical symbols names (and a `X` is the sum of weights is less than one). The same rules as above are used to append a digit to the kind name, if needed.

- Finally, you can simply specify the `kind_name` to automatically generate a new kind with a specific name. This is the case if you want a name different from the automatically generated one, or for instance if you want to create two different species with the same properties (same mass, symbols, ...). This is for instance the case in Quantum ESPRESSO in order to describe an antiferromagnetic crystal, with different magnetizations on the different atoms in the unit cell.

In this case, you can for instance use:

```
s.append_atom(position = [0,0,0], symbols='Fe', mass = 55.845, name='Fe1')
s.append_atom(position = [2,2,2], symbols='Fe', mass = 55.845, name='Fe2')
```

To create two species Fe1 and Fe2 for iron, with the same mass.

**Note:** You do not need to specify explicitly the mass if the default one is ok for you. However, when you pass explicitly a name and it coincides with the name of an existing species, all properties that you specify must be identical to the ones of the existing species, or the method will raise an exception.

**Note:** If you prefer to work with the internal `Kind` and `Site` classes, you can obtain the same result of the two lines above with:

```
from aiida.orm.data.structure import Kind, Site
s.append_kind(Kind(symbols='Fe', mass=55.845, name='Fe1'))
s.append_kind(Kind(symbols='Fe', mass=55.845, name='Fe1'))
s.append_site(Site(kind_name='Fe1', position=[0.,0.,0.]))
s.append_site(Site(kind_name='Fe2', position=[2.,2.,2.]))
```

## Conversion to/from ASE

If you have an AiiDA structure, you can get an `ase.Atom` object by just calling the `get_ase` method:

```
ase_atoms = aiida_structure.get_ase()
```

**Note:** As we support alloys and vacancies in AiiDA, while `ase.Atom` does not, it is not possible to export to ASE a structure with vacancies or alloys.

If instead you have an ASE Atoms object and you want to load the structure from it, just pass it when initializing the class:

```
StructureData = DataFactory('structure')
# or:
# from aiida.orm.data.structure import StructureData
aiida_structure = StructureData(ase = ase_atoms)
```

## Creating multiple species

We implemented the possibility of specifying different Kinds (species) in the `ase.atoms` and then importing them.

In particular, if you specify atoms with different mass in ASE, during the import phase different kinds will be created:

```
>>> import ase
>>> StructureData = DataFactory("structure")
>>> asecell = ase.Atoms('Fe2')
>>> asecell[0].mass = 55.
>>> asecell[1].mass = 56.
```

```
>>> s = StructureData(ase=asecell)
>>> for kind in s.kinds:
>>>     print kind.name, kind.mass
Fe 55.0
Fe1 56.0
```

Moreover, even if the mass is the same, but you want to get different species, you can use the ASE tags to specify the number to append to the element symbol in order to get the species name:

```
>>> import ase
>>> StructureData = DataFactory("structure")
>>> asecell = ase.Atoms('Fe2')
>>> asecell[0].tag = 1
>>> asecell[1].tag = 2
>>> s = StructureData(ase=asecell)
>>> for kind in s.kinds:
>>>     print kind.name
Fe1
Fe2
```

---

**Note:** in complicated cases (multiple tags, masses, ...), it is possible that exporting a AiiDA structure to ASE and then importing it again will not perfectly preserve the kinds and kind names.

---

## Conversion to/from pymatgen

AiiDA structure can be converted to pymatgen's `Molecule` and `Structure` objects by using, accordingly, `get_pymatgen_molecule` and `get_pymatgen_structure` methods:

```
pymatgen_molecule = aiiда_structure.get_pymatgen_molecule()
pymatgen_structure = aiiда_structure.get_pymatgen_structure()
```

A single method `get_pymatgen` can be used for both tasks: converting periodic structures (periodic boundary conditions are met in all three directions) to pymatgen's `Structure` and other structures to pymatgen's `Molecule`:

```
pymatgen_object = aiiда_structure.get_pymatgen()
```

It is also possible to convert pymatgen's `Molecule` and `Structure` objects to AiiDA structures:

```
StructureData = DataFactory("structure")
from_mol       = StructureData(pymatgen_molecule=mol)
from_struct    = StructureData(pymatgen_structure=struct)
```

Also in this case, a generic converter is provided:

```
StructureData = DataFactory("structure")
from_mol      = StructureData(pymatgen=mol)
from_struct   = StructureData(pymatgen=struct)
```

---

**Note:** Converters work with version 3.0.13 or later of pymatgen. Earlier versions may cause errors.

---

## 1.1.8 Quantum Espresso PWscf user-tutorial

This chapter will show how to launch a single PWscf (pw.x) calculation. It is assumed that you have already performed the installation, and that you already setup a computer (with `verdi`), installed Quantum Espresso on the



cluster and in AiiDA. Although the code could be quite readable, a basic knowledge of Python and object programming is useful.

### Your classic pw.x input file

This is the input file of Quantum Espresso that we will try to execute. It consists in the total energy calculation of a 5 atom cubic cell of BaTiO<sub>3</sub>. Note also that AiiDA is a tool to use other codes: if the following input is not clear to you, please refer to the Quantum Espresso Documentation.

```
&CONTROL
  calculation = 'scf'
  outdir = './out/'
  prefix = 'aiida'
  pseudo_dir = './pseudo/'
  restart_mode = 'from_scratch'
  verbosity = 'high'
  wf_collect = .true.
/
&SYSTEM
  ecutrho = 2.4000000000d+02
  ecutwfc = 3.0000000000d+01
  ibrav = 0
  nat = 5
  ntyp = 3
/
&ELECTRONS
  conv_thr = 1.0000000000d-06
/
ATOMIC_SPECIES
Ba 137.33 Ba.pbesol-spn-rrkjus_psl.0.2.3-tot-pslib030.UPF
Ti 47.88 Ti.pbesol-spn-rrkjus_psl.0.2.3-tot-pslib030.UPF
O 15.9994 O.pbesol-n-rrkjus_psl.0.1-tested-pslib030.UPF
ATOMIC_POSITIONS angstrom
Ba 0.0000000000 0.0000000000 0.0000000000
Ti 2.0000000000 2.0000000000 2.0000000000
O 2.0000000000 2.0000000000 0.0000000000
O 2.0000000000 0.0000000000 2.0000000000
O 0.0000000000 2.0000000000 2.0000000000
K_POINTS automatic
4 4 4 0 0 0
CELL_PARAMETERS angstrom
4.0000000000 0.0000000000 0.0000000000
0.0000000000 4.0000000000 0.0000000000
0.0000000000 0.0000000000 4.0000000000
```

In the old way, not only you had to prepare ‘manually’ this file, but also prepare the scheduler submission script, send everything on the cluster, etc. We are going instead to prepare everything in a more programmatic way.

### Quantum Espresso Pw Walkthrough

We’ve got to prepare a script to submit a job to your local installation of AiiDA. This example will be a rather long script: in fact there is still nothing in your database, so that we will have to load everything, like the pseudopotential files and the structure. In a more practical situation, you might load data from the database and perform a small modification to re-use it.

Let’s say that through the `verdi` command you have already installed a cluster, say `TheHive`, and that you also compiled Quantum Espresso on the cluster, and installed the code `pw.x` with `verdi` with label `pw-5.1` for instance,

so that in the rest of this tutorial we will reference to the code as `pw-5.1@TheHive`.

Let's start writing the python script. First of all, we need to load the configuration concerning your particular installation, in particular, the details of your database installation:

```
#!/usr/bin/env python
from aiida import load_dbenv
load_dbenv()
```

## Code

Now we have to select the code. Note that in AiiDA the object 'code' in the database is meant to represent a specific executable, i.e. a given compiled version of a code. This means that if you install Quantum Espresso (QE) on two computers A and B, you will need to have two different 'codes' in the database (although the source of the code is the same, the binary file is different).

If you setup the code `pw-5.1` on machine `TheHive` correctly, then it is sufficient to write:

```
codename = 'pw-5.1@TheHive'
from aiida.orm import Code
code = Code.get_from_string(codename)
```

Where in the last line we just load the database object representing the code.

**Note:** the `.get_from_string()` method is just a helper method for user convenience, but there are some weird cases that cannot be dealt in a simple way (duplicated labels, code names that are an integer number, code names containing the '@' symbol, ...: try to not do this! This is not an error, but does not allow to use the `.get_from_string()` method to get those calculations). In this case, you can use directly the `.get()` method, for instance:

```
code = Code.get(label='pw-5.1', machinename='TheHive',
               useremail='user@domain.com')
```

or even more generally get the code from its (integer) PK:

```
code = load_node(PK)
```

---

## Structure

We now proceed in setting up the structure.

**Note:** Here we discuss only the main features of structures in AiiDA, needed to run a Quantum ESPRESSO PW calculation.

For more detailed information, give a look to the [StructureData tutorial](#).

---

There are two ways to do that in AiiDA, a first one is to use the AiiDA Structure, which we will explain in the following; the second choice is the [Atomic Simulation Environment \(ASE\)](#) which provides excellent tools to manipulate structures (the ASE Atoms object needs to be converted into an AiiDA Structure, see the note at the end of the section).

We first have to load the abstract object class that describes a structure. We do it in the following way: we load the `DataFactory`, which is a tool to load the classes by their name, and then call `StructureData` the abstract class that we loaded. (NB: it's not yet a class instance!) (If you are not familiar with the terminology of object programming, we could take [Wikipedia](#) and see their short explanation: in common speech that one refers to *a* file as a class, while *the* file is the object or the class instance. In other words, the class is our definition of the object Structure, while its instance is what will be saved as an object in the database):

```
from aiida.orm import DataFactory
StructureData = DataFactory('structure')
```

We define the cell with a 3x3 matrix (we choose the convention where each ROW represents a lattice vector), which in this case is just a cube of size 4 Angstroms:

```
alat = 4. # angstrom
cell = [[alat, 0., 0.],
        [0., alat, 0.],
        [0., 0., alat],
        ]
```

Now, we create the StructureData instance, assigning immediately the cell. Then, we append to the empty crystal cell the atoms, specifying their element name and their positions:

```
# BaTiO3 cubic structure
s = StructureData(cell=cell)
s.append_atom(position=(0., 0., 0.), symbols='Ba')
s.append_atom(position=(alat/2., alat/2., alat/2.), symbols='Ti')
s.append_atom(position=(alat/2., alat/2., 0.), symbols='O')
s.append_atom(position=(alat/2., 0., alat/2.), symbols='O')
s.append_atom(position=(0., alat/2., alat/2.), symbols='O')
```

To see more methods associated to the class StructureData, look at the [Structure](#) documentation.

**Note:** When you create a node (in this case a StructureData node) as described above, you are just creating it in the computer memory, and not in the database. This is particularly useful to run tests without filling the AiiDA database with garbage.

You will see how to store all the nodes in one shot toward the end of this tutorial; if, however, you want to directly store the structure in the database for later use, you can just call the `store()` method of the Node:

```
s.store()
```

For an extended tutorial about the creation of Structure objects, check [this tutorial](#).

**Note:** AiiDA supports also ASE structures. Once you created your structure with ASE, in an object instance called say `ase_s`, you can straightforwardly use it to create the AiiDA StructureData, as:

```
s = StructureData(ase=ase_s)
```

and then save it `s.store()`.

## Parameters

Now we need to provide also the parameters of a Quantum Espresso calculation, like the cutoff for the wavefunctions, some convergence threshold, etc... The Quantum ESPRESSO pw.x plugin requires to pass this information within a ParameterData object, that is a specific AiiDA data node that can store a dictionary (even nested) of basic data types: integers, floats, strings, lists, dates, ... We first load the class through the DataFactory, just like we did for the Structure. Then we create the instance of the object `parameter`. To represent closely the structure of the QE input file, ParameterData is a nested dictionary, at the first level the namelists (capitalized), and then the variables with their values (in lower case).

Note also that numbers and booleans are written in Python, i.e. `False` and not the Fortran string `.false.`!

```
ParameterData = DataFactory('parameter')

parameters = ParameterData(dict={
    'CONTROL': {
        'calculation': 'scf',
        'restart_mode': 'from_scratch',
        'wf_collect': True,
    },
    'SYSTEM': {
        'ecutwfc': 30.,
        'ecutrho': 240.,
    },
    'ELECTRONS': {
        'conv_thr': 1.e-6,
    })
})
```

**Note:** also in this case, we chose not to store the `parameters` node. If we wanted, we could even have done it in a single line:

```
parameters = ParameterData(dict={...}).store()
```

The experienced QE user will have noticed also that a couple of variables are missing: the prefix, the pseudo directory and the scratch directory are reserved to the plugin which will use default values, and there are specific AiiDA methods to restart from a previous calculation.

The k-points have to be saved in another kind of data, namely `KpointsData`:

```
KpointsData = DataFactory('array.kpoints')
kpoints = KpointsData()
kpoints.set_kpoints_mesh([4,4,4])
```

In this case it generates a 4\*4\*4 mesh without offset. To add an offset one can replace the last line by:

```
kpoints.set_kpoints_mesh([4,4,4], offset=(0.5,0.5,0.5))
```

**Note:** Only offsets of 0 or 0.5 are possible (this is imposed by PWscf).

You can also specify kpoints manually, by inputting a list of points in crystal coordinates (here they all have equal weights):

```
import numpy
kpoints.set_kpoints([[i,i,0] for i in numpy.linspace(0,1,10)],
    weights = [1. for i in range(10)])
```

**Note:** It is also possible to generate a gamma-only computation. To do so one has to specify additional settings, of type `ParameterData`, putting gamma-only to `True`:

```
settings = ParameterData(dict={'gamma_only':True})
```

then set the kpoints mesh to a single point (gamma):

```
kpoints.set_kpoints_mesh([1,1,1])
```

and in the end add (after `calc = code.new_calc()`, see below) a line to use these settings:

```
calc.use_settings(settings)
```

As a further comment, this is specific to the way the plugin for Quantum Espresso works. Other codes may need more than two `ParameterData`, or even none of them. And also how this parameters have to be written depends on the plugin: what is discussed here is just the format that we decided for the Quantum Espresso plugins.

## Calculation

Now we proceed to set up the calculation. Since during the setup of the code we already set the code to be a `quantumespresso.pw` code, there is a simple method to create a new calculation:

```
calc = code.new_calc()
```

We have to specify the details required by the scheduler. For example, on a SLURM or PBS scheduler, we have to specify the number of nodes (`num_machines`), possibly the number of MPI processes per node (`num_mpiprocs_per_machine`) if we want to run with a different number of MPI processes with respect to the default value configured when setting up the computer in AiiDA, the job walltime, the queue name (if desired), ...:

```
calc.set_max_wallclock_seconds(30*60) # 30 min
calc.set_resources({"num_machines": 1})
## OPTIONAL, use only if you need to explicitly specify a queue name
# calc.set_queue_name("the_queue_name")
```

(For the complete scheduler documentation, see [Supported schedulers](#))

**Note:** an alternative way of calling a method starting with the string `set_`, is to pass directly the value to the `.new_calc()` method. This is to say that the following lines:

```
calc = code.new_calc()
calc.set_max_wallclock_seconds(3600)
calc.set_resources({"num_machines": 1})
```

is equivalent to:

```
calc = code.new_calc(max_wallclock_seconds=3600,
    resources={"num_machines": 1})
```

At this point, we just created a “lone” calculation, that still does not know anything about the inputs that we created before. We need therefore to tell the calculation to use the parameters that we prepared before, by properly linking them using the `use_` methods:

```
calc.use_structure(s)
calc.use_code(code)
calc.use_parameters(parameters)
calc.use_kpoints(kpoints)
```

In practice, when you say `calc.use_structure(s)`, you are setting a link between the two nodes (`s` and `calc`), that means that `s` is the input *structure* for *calculation* `calc`. Also these links are cached and do not require to store anything in the database yet.

In the case of the gamma-only computation (see [above](#)), you also need to add:

```
calc.use_settings(settings)
```

## Pseudopotentials

There is still one missing piece of information, that is the pseudopotential files, one for each element of the structure.

In AiiDA, it is possible to specify manually which pseudopotential files to use for each atomic species. However, for any practical use, it is convenient to use the pseudopotential families. Its use is documented in *Pseudopotential families tutorial*. If you got one installed, you can simply tell the calculation to use the pseudopotential family with a given name, and AiiDA will take care of linking the proper pseudopotentials to the calculation, one for each atomic species present in the input structure. This can be done using:

```
calc.use_pseudos_from_family('my_pseudo_family')
```

## Labels and comments

Sometimes it is useful to attach some notes to the calculation, that may help you later understand why you did such a calculation, or note down what you understood out of it. Comments are a special set of properties of the calculation, in the sense that it is one of the few properties that can be changed, even after the calculation has run.

Comments come in various flavours. The most basic one is the label property, a string of max 255 characters, which is meant to be the title of the calculation. To create it, simply write:

```
calc.label = "A generic title"
```

The label can be later accessed as a class property, i.e. the command:

```
calc.label
```

will return the string you previously set (empty by default). Another important property to set is the description, which instead does not have a limitation on the maximum number of characters:

```
calc.description = "A much longer description"
```

And finally, there is the possibility to add comments to any calculation (actually, to any node). The peculiarity of comments is that they are user dependent (like the comments that you can post on facebook pages), so it is best suited to calculation exposed on a website, where you want to remember the comments of each user. To set a comment, you need first to import the django user, and then write it with a dedicated method:

```
from aiida.djsite.utils import get_automatic_user
calc.add_comment("Some comment", user=get_automatic_user())
```

The comments can be accessed with this function:

```
calc.get_comments_tuple()
```

## Execute

If we are satisfied with what you created, it is time to store everything in the database. Note that after storing it, it will not be possible to modify it (nor you should: you risk of compromising the integrity of the database)!

Unless you already stored all the inputs beforehand, you will need to store the inputs before being able to store the calculation itself. Since this is a very common operation, there is an utility method that will automatically store both all the input nodes of `calc` and then `calc` itself:

```
calc.store_all()
```

Once we store the calculation, it is useful to print its PK (principal key, that is its identifier) that is useful in the following to interact with it:

```
print "created calculation; with uuid='{ }' and PK={ }".format(calc.uuid, calc.pk)
```

**Note:** the PK will change if you give the calculation to someone else, while the UUID (the Universally Unique Identifier) is a string that is assured to be always the same also if you share your data with collaborators.

Summarizing, we created all the inputs needed by a PW calculation, that are: parameters, kpoints, pseudopotential files and the structure. We then created the calculation, where we specified that it is a PW calculation and we specified the details of the remote cluster. We set the links between the inputs and the calculation (`calc.use_***`) and finally we stored all this objects in the database (`.store_all()`).

That's all that the calculation needs. Now we just need to submit it:

```
calc.submit()
```

Everything else will be managed by AiiDA: the inputs will be checked to verify that it is consistent with a PW input. If the input is complete, the pw input file will be prepared in a folder together with all the other files required for the execution (pseudopotentials, etc.). It will be then sent on cluster, submitted, and after execution automatically retrieved and parsed.

To know how to monitor and check the state of submitted calculations, go to [Calculations](#).

To continue the tutorial with the `ph.x` phonon code of Quantum ESPRESSO, continue here: [Quantum Espresso Phonon user-tutorial](#).

## Script: source code

In this section you'll find two scripts that do what explained in the tutorial. The compact is a script with a minimal configuration required. You can copy and paste it (or download it), modify the two strings `codename` and `pseudo_family` with the correct values, and execute it with:

```
python pw_short_example.py
```

(It requires to have one family of pseudopotentials configured).

You will also find a longer version, with more exception checks, error management and user interaction. Note that the configuration of the computer resources (like number of nodes and machines) is hardware and scheduler dependent. The configuration used below should work for a pbspro or slurm cluster, asking to run on 1 node only.

## Compact script

Download: [this example script](#)

```
#!/usr/bin/env python
from aiida import load_dbenv
load_dbenv()

from aiida.orm import Code, DataFactory
StructureData = DataFactory('structure')
ParameterData = DataFactory('parameter')
KpointsData = DataFactory('array.kpoints')

#####
# Set your values here
codename = 'pw-5.1@TheHive'
pseudo_family = 'lda_pslibrary'
#####

code = Code.get_from_string(codename)
```

```
# BaTiO3 cubic structure
alat = 4. # angstrom
cell = [[alat, 0., 0.],
        [0., alat, 0.],
        [0., 0., alat,],
        ]
s = StructureData(cell=cell)
s.append_atom(position=(0.,0.,0.),symbols='Ba')
s.append_atom(position=(alat/2.,alat/2.,alat/2.),symbols='Ti')
s.append_atom(position=(alat/2.,alat/2.,0.),symbols='O')
s.append_atom(position=(alat/2.,0.,alat/2.),symbols='O')
s.append_atom(position=(0.,alat/2.,alat/2.),symbols='O')

parameters = ParameterData(dict={
    'CONTROL': {
        'calculation': 'scf',
        'restart_mode': 'from_scratch',
        'wf_collect': True,
    },
    'SYSTEM': {
        'ecutwfc': 30.,
        'ecutrho': 240.,
    },
    'ELECTRONS': {
        'conv_thr': 1.e-6,
    })
})

kpoints = KpointsData()
kpoints.set_kpoints_mesh([4,4,4])

calc = code.new_calc(max_wallclock_seconds=3600,
    resources={"num_machines": 1})
calc.label = "A generic title"
calc.description = "A much longer description"

calc.use_structure(s)
calc.use_code(code)
calc.use_parameters(parameters)
calc.use_kpoints(kpoints)
calc.use_pseudos_from_family(pseudo_family)

calc.store_all()
print "created calculation with PK={}".format(calc.pk)
calc.submit()
```

## Exception tolerant code

You can find a more sophisticated example, that checks the possible exceptions and prints nice error messages inside your AiiDA folder, under `examples/submission/test_pw.py`.

### 1.1.9 Importing previously run Quantum ESPRESSO pw.x calculations: PwImmigrant

Once you start using AiiDA to run simulations, we believe that you will find it so convenient that you will use it for all your calculations.



At the beginning, however, you may have some calculations that you already have run and are sitting in some folders, and that you want to import inside AiiDA.

This can be achieved with the `PwImmigrant` class described below.

## Quantum Espresso PWscf immigration user-tutorial

If you are a new AiiDA user, it's likely you already have a large number of calculations that you ran before installing AiiDA. This tutorial will show you how to immigrate any of these PWscf (`pw.x`) calculations into your AiiDA database. They will then exist there as if you had actually run them using AiiDA (with the exception of the times and dates the calculations were run).

It is assumed that you have already performed the installation, that you already setup a computer (with `verdi`), and that you have installed Quantum Espresso on the cluster and `pw.x` as a code in AiiDA. You should also be familiar with using AiiDA to run a PWscf calculation and the various input and output nodes of a `PwCalculation`. Please go through [Quantum Espresso PWscf user-tutorial](#) before proceeding.

---

### Example details

The rest of the tutorial will detail the steps of immigrating two example `pw.x` calculations that were run in `/scratch/`, using the code named `'pw_on_TheHive'`, on 1 node with 1 mpi process. The input/output file names of these calculations are

- `pw_job1.in/pw_job1.out`
- `pw_job2.in/pw_job2.out`

---

### Imports and database environment

As usual, we load the database environment and load the `PwimmigrantCalculation` class using the `CalculationFactory`.

```
from aiida import load_dbenv
from aiida.orm.code import Code
from aiida.orm import CalculationFactory

# Load the database environment.
load_dbenv()

# Load the PwimmigrantCalculation class.
PwimmigrantCalculation = CalculationFactory('quantumespresso.pwimmigrant')
```

### Code, computer, and resources

---

**Important:** It is up to the user to setup and link the following calculation inputs manually:

- the code
- the computer
- the resources

These input nodes should be created to be representative of those that were used for the calculation that is to be immigrated. (Eg. If the job was run using version 5.1 of Quantum-Espresso, the user should have already run `verdi code setup` to create the code's node and should load and pass this code when initializing the calculation node.)

If any of these input nodes are not representative of the actual properties the calculation was run with, there may be errors when performing a calculation restart of an immigrated calculation, for example.

Next, we load the code and computer that have already been configured to be representative of those used to perform the calculation. We also define the resources representative of those that were used to run the calculation.

```
# Load the Code node representative of the one used to perform the calculations.
code = Code.get('pw_on_TheHive')

# Get the Computer node representative of the one the calculations were run on.
computer = code.get_remote_computer()

# Define the computation resources used for the calculations.
resources = {'num_machines': 1, 'num_mpiproc_per_machine': 1}
```

### Initialization of the calculation

Now, we are ready to initialize the immigrated calculation objects from the `PwimmigrantCalculation` class. We will pass the necessary parameters as keywords during the initialization calls. Then, we link the code from above as an input node.

```
# Initialize the pw_job1 calculation node.
calc1 = PwimmigrantCalculation(computer=computer,
                               resources=resources,
                               remote_workdir='/scratch/',
                               input_file_name='pw_job1.in',
                               output_file_name='pw_job1.out')

# Initialize the pw_job2 calculation node.
calc2 = PwimmigrantCalculation(computer=computer,
                               resources=resources,
                               remote_workdir='/scratch/',
                               input_file_name='pw_job2.in',
                               output_file_name='pw_job2.out')

# Link the code that was used to run the calculations.
calc1.use_code(code)
calc2.use_code(code)
```

The user may have noticed the additional initialization keywords/parameters—`remote_workdir`, `input_file_name`, and `output_file_name`—passed here. These are necessary in order to tell AiiDA which files to use to automatically generate the calculation's input nodes in the next step.

### The immigration

Now that AiiDA knows where to look for the input files of the calculations we are immigrating, all we need to do in order to generate all the input nodes is call the `create_input_nodes` method. This method is the most helpful method of the `PwimmigrantCalculation` class. It parses the job's input file and creates and links the follow types of input nodes:

- `ParameterData` – based on the namelists and their variable-value pairs
- `KpointsData` – based on the `K_POINTS` card
- `StructureData` – based on the `ATOMIC_POSITIONS` and `CELL_PARAMETERS` cards (and the `a` or `celldm(1)` of the `&SYSTEM` namelist, if `alat` is specified through these variables)

- UpfData – one for each of the atomic species, based on the pseudopotential files specified in the ATOMIC\_SPECIES card
- settings ParameterData – if there are any fixed coordinates, or if the gamma kpoint is used

All units conversion and/or coordinate transformations are handled automatically, and the input nodes are generated in the correct units and coordinates required by AiiDA.

---

**Note:** Any existing UpfData nodes are simply linked without recreation; no duplicates are generated during this method call.

---



---

**Note:** After this method call, the calculation and the generated input nodes are still in the cached state and are not yet stored in the database. Therefore, the user may examine the input nodes that were generated (by examining the attributes of the NodeInputManager, `calc.inp`) and edit or replace any of them. The immigration can also be canceled at this point, in which case the calculation and the input nodes would not be stored in the database.

---

Finally, the last step of the immigration is to call the `prepare_for_retrieval_and_parsing` method. This method stores the calculation and its input nodes in the database, copies the original input file to the calculation's repository folder, and then tells the daemon to retrieve and parse the calculation's output files.

---

**Note:** If the daemon is not currently running, the retrieval and parsing process will not begin until it is started.

---

Because the input and pseudopotential files need to be retrieved from the computer, the computer's transport plugin needs to be open. Rather than opening and closing the transport for each calculation, we instead require the user to pass an open transport instance as a parameter to the `create_input_nodes` and `prepare_for_retrieval_and_parsing` methods. This minimizes the number of transport opening and closings, which is highly beneficial when immigrating a large number of calculations.

Calling these methods with an open transport is performed as follows:

```
# Get the computer's transport and create an instance.
Transport = computer.get_transport_class()
transport = Transport()

# Open the transport for the duration of the immigrations, so it's not
# reopened for each one. This is best performed using the transport's
# context guard through the ``with`` statement.
with transport as open_transport:

    # Parse the calculations' input files to automatically generate and link the
    # calculations' input nodes.
    calc1.create_input_nodes(open_transport)
    calc2.create_input_nodes(open_transport)

    # Store the calculations and their input nodes and tell the daemon the output
    # is ready to be retrieved and parsed.
    calc1.prepare_for_retrieval_and_parsing(open_transport)
    calc2.prepare_for_retrieval_and_parsing(open_transport)
```

The process above is easily expanded to large-scale immigrations of multiple jobs.

### Compact script

Download: [this example script](#)

```
#!/usr/bin/env python
from aiida import load_dbenv
from aiida.orm.code import Code
from aiida.orm import CalculationFactory

# Load the database environment.
load_dbenv()

# Load the PwimmigrantCalculation class.
PwimmigrantCalculation = CalculationFactory('quantumespresso.pwimmigrant')

# Load the Code node representative of the one used to perform the calculations.
code = Code.get('pw_on_TheHive')

# Get the Computer node representative of the one the calculations were run on.
computer = code.get_remote_computer()

# Define the computation resources used for the calculations.
resources = {'num_machines': 1, 'num_mpiprocs_per_machine': 1}

# Initialize the pw_job1 calculation node.
calc1 = PwimmigrantCalculation(computer=computer,
                               resources=resources,
                               remote_workdir='/scratch/',
                               input_file_name='pw_job1.in',
                               output_file_name='pw_job1.out')

# Initialize the pw_job2 calculation node.
calc2 = PwimmigrantCalculation(computer=computer,
                               resources=resources,
                               remote_workdir='/scratch/',
                               input_file_name='pw_job2.in',
                               output_file_name='pw_job2.out')

# Link the code that was used to run the calculations.
calc1.use_code(code)
calc2.use_code(code)

# Get the computer's transport and create an instance.
Transport = computer.get_transport_class()
transport = Transport()

# Open the transport for the duration of the immigrations, so it's not
# reopened for each one. This is best performed using the transport's
# context guard through the ``with`` statement.
with transport as open_transport:

    # Parse the calculations' input files to automatically generate and link the
    # calculations' input nodes.
    calc1.create_input_nodes(open_transport)
    calc2.create_input_nodes(open_transport)

    # Store the calculations and their input nodes and tell the daeomon the output
    # is ready to be retrieved and parsed.
    calc1.prepare_for_retrieval_and_parsing(open_transport)
    calc2.prepare_for_retrieval_and_parsing(open_transport)
```

### 1.1.10 Quantum Espresso PWscf immigration user-tutorial

If you are a new AiiDA user, it's likely you already have a large number of calculations that you ran before installing AiiDA. This tutorial will show you how to immigrate any of these PWscf (`pw.x`) calculations into your AiiDA database. They will then exist there as if you had actually run them using AiiDA (with the exception of the times and dates the calculations were run).

It is assumed that you have already performed the installation, that you already setup a computer (with `verdi`), and that you have installed Quantum Espresso on the cluster and `pw.x` as a code in AiiDA. You should also be familiar with using AiiDA to run a PWscf calculation and the various input and output nodes of a `PwCalculation`. Please go through [Quantum Espresso PWscf user-tutorial](#) before proceeding.

#### Example details

The rest of the tutorial will detail the steps of immigrating two example `pw.x` calculations that were run in `/scratch/`, using the code named `'pw_on_TheHive'`, on 1 node with 1 mpi process. The input/output file names of these calculations are

- `pw_job1.in/pw_job1.out`
- `pw_job2.in/pw_job2.out`

#### Imports and database environment

As usual, we load the database environment and load the `PwimmigrantCalculation` class using the `CalculationFactory`.

```
from aiida import load_dbenv
from aiida.orm.code import Code
from aiida.orm import CalculationFactory

# Load the database environment.
load_dbenv()

# Load the PwimmigrantCalculation class.
PwimmigrantCalculation = CalculationFactory('quantumespresso.pwimmigrant')
```

#### Code, computer, and resources

**Important:** It is up to the user to setup and link the following calculation inputs manually:

- the code
- the computer
- the resources

These input nodes should be created to be representative of those that were used for the calculation that is to be immigrated. (Eg. If the job was run using version 5.1 of Quantum-Espresso, the user should have already run `verdi code setup` to create the code's node and should load and pass this code when initializing the calculation node.) If any of these input nodes are not representative of the actual properties the calculation was run with, there may be errors when performing a calculation restart of an immigrated calculation, for example.

Next, we load the code and computer that have already been configured to be representative of those used to perform the calculation. We also define the resources representative of those that were used to run the calculation.

```
# Load the Code node representative of the one used to perform the calculations.
code = Code.get('pw_on_TheHive')

# Get the Computer node representative of the one the calculations were run on.
computer = code.get_remote_computer()

# Define the computation resources used for the calculations.
resources = {'num_machines': 1, 'num_mpiprocs_per_machine': 1}
```

## Initialization of the calculation

Now, we are ready to initialize the immigrated calculation objects from the `PwimmigrantCalculation` class. We will pass the necessary parameters as keywords during the initialization calls. Then, we link the code from above as an input node.

```
# Initialize the pw_job1 calculation node.
calc1 = PwimmigrantCalculation(computer=computer,
                               resources=resources,
                               remote_workdir='/scratch/',
                               input_file_name='pw_job1.in',
                               output_file_name='pw_job1.out')

# Initialize the pw_job2 calculation node.
calc2 = PwimmigrantCalculation(computer=computer,
                               resources=resources,
                               remote_workdir='/scratch/',
                               input_file_name='pw_job2.in',
                               output_file_name='pw_job2.out')

# Link the code that was used to run the calculations.
calc1.use_code(code)
calc2.use_code(code)
```

The user may have noticed the additional initialization keywords/parameters—`remote_workdir`, `input_file_name`, and `output_file_name`—passed here. These are necessary in order to tell AiiDA which files to use to automatically generate the calculation's input nodes in the next step.

## The immigration

Now that AiiDA knows where to look for the input files of the calculations we are immigrating, all we need to do in order to generate all the input nodes is call the `create_input_nodes` method. This method is the most helpful method of the `PwimmigrantCalculation` class. It parses the job's input file and creates and links the following types of input nodes:

- `ParameterData` – based on the namelists and their variable-value pairs
- `KpointsData` – based on the `K_POINTS` card
- `StructureData` – based on the `ATOMIC_POSITIONS` and `CELL_PARAMETERS` cards (and the `a` or `celldm(1)` of the `&SYSTEM` namelist, if `alat` is specified through these variables)
- `UpfData` – one for each of the atomic species, based on the pseudopotential files specified in the `ATOMIC_SPECIES` card
- settings `ParameterData` – if there are any fixed coordinates, or if the gamma kpoint is used

All units conversion and/or coordinate transformations are handled automatically, and the input nodes are generated in the correct units and coordinates required by AiiDA.

---

**Note:** Any existing UpfData nodes are simply linked without recreation; no duplicates are generated during this method call.

---



---

**Note:** After this method call, the calculation and the generated input nodes are still in the cached state and are not yet stored in the database. Therefore, the user may examine the input nodes that were generated (by examining the attributes of the `NodeInputManager`, `calc.inp`) and edit or replace any of them. The immigration can also be canceled at this point, in which case the calculation and the input nodes would not be stored in the database.

---

Finally, the last step of the immigration is to call the `prepare_for_retrieval_and_parsing` method. This method stores the calculation and its input nodes in the database, copies the original input file to the calculation's repository folder, and then tells the daemon to retrieve and parse the calculation's output files.

---

**Note:** If the daemon is not currently running, the retrieval and parsing process will not begin until it is started.

---

Because the input and pseudopotential files need to be retrieved from the computer, the computer's transport plugin needs to be open. Rather than opening and closing the transport for each calculation, we instead require the user to pass an open transport instance as a parameter to the `create_input_nodes` and `prepare_for_retrieval_and_parsing` methods. This minimizes the number of transport opening and closings, which is highly beneficial when immigrating a large number of calculations.

Calling these methods with an open transport is performed as follows:

```
# Get the computer's transport and create an instance.
Transport = computer.get_transport_class()
transport = Transport()

# Open the transport for the duration of the immigrations, so it's not
# reopened for each one. This is best performed using the transport's
# context guard through the ``with`` statement.
with transport as open_transport:

    # Parse the calculations' input files to automatically generate and link the
    # calculations' input nodes.
    calc1.create_input_nodes(open_transport)
    calc2.create_input_nodes(open_transport)

    # Store the calculations and their input nodes and tell the daemon the output
    # is ready to be retrieved and parsed.
    calc1.prepare_for_retrieval_and_parsing(open_transport)
    calc2.prepare_for_retrieval_and_parsing(open_transport)
```

The process above is easily expanded to large-scale immigrations of multiple jobs.

## Compact script

Download: [this example script](#)

```
#!/usr/bin/env python
from aiida import load_dbenv
from aiida.orm.code import Code
from aiida.orm import CalculationFactory
```

```
# Load the database environment.
load_dbenv()

# Load the PwimmigrantCalculation class.
PwimmigrantCalculation = CalculationFactory('quantumespresso.pwimmigrant')

# Load the Code node representative of the one used to perform the calculations.
code = Code.get('pw_on_TheHive')

# Get the Computer node representative of the one the calculations were run on.
computer = code.get_remote_computer()

# Define the computation resources used for the calculations.
resources = {'num_machines': 1, 'num_mpiprocs_per_machine': 1}

# Initialize the pw_job1 calculation node.
calc1 = PwimmigrantCalculation(computer=computer,
                               resources=resources,
                               remote_workdir='/scratch/',
                               input_file_name='pw_job1.in',
                               output_file_name='pw_job1.out')

# Initialize the pw_job2 calculation node.
calc2 = PwimmigrantCalculation(computer=computer,
                               resources=resources,
                               remote_workdir='/scratch/',
                               input_file_name='pw_job2.in',
                               output_file_name='pw_job2.out')

# Link the code that was used to run the calculations.
calc1.use_code(code)
calc2.use_code(code)

# Get the computer's transport and create an instance.
Transport = computer.get_transport_class()
transport = Transport()

# Open the transport for the duration of the immigrations, so it's not
# reopened for each one. This is best performed using the transport's
# context guard through the ``with`` statement.
with transport as open_transport:

    # Parse the calculations' input files to automatically generate and link the
    # calculations' input nodes.
    calc1.create_input_nodes(open_transport)
    calc2.create_input_nodes(open_transport)

    # Store the calculations and their input nodes and tell the daeomon the output
    # is ready to be retrieved and parsed.
    calc1.prepare_for_retrieval_and_parsing(open_transport)
    calc2.prepare_for_retrieval_and_parsing(open_transport)
```

### 1.1.11 Quantum Espresso Phonon user-tutorial

---

**Note:** The Phonon plugin referenced below is available in the EPFL version.

---



In this chapter will get you through the launching of a phonon calculation with Quantum Espresso, with `ph.x`, a density functional perturbation theory code. For this tutorial, it is required that you managed to launch the `pw.x` calculation, which is at the base of the phonon code; and of course it is assumed that you already know how to use the QE code.

The input of a phonon calculation can be actually simple, the only care that has to be taken, is to point to the same scratch of the previous `pw` calculation. Here we will try to compute the dynamical matrix on a mesh of points (actually consisting of a  $1 \times 1 \times 1$  mesh for brevity). The input file that we should create is more or less this one:

```
AiiDA calculation
&INPUTPH
  epsilon = .true.
  fildyn = 'DYN_MAT/dynamical-matrix-'
  iverbosity = 1
  ldisp = .true.
  nq1 = 1
  nq2 = 1
  nq3 = 1
  outdir = './out/'
  prefix = 'aiida'
  tr2_ph = 1.0000000000d-08
/
```

## Walkthrough

This input is much simpler than the previous `PWscf` work, here the only novel thing you will have to learn is how to set a parent calculation.

As before, we write a script step-by-step.

We first load a couple of useful modules that you already met in the previous tutorial, and load the database settings:

```
#!/usr/bin/env python
from aiida import load_dbenv
load_dbenv()

from aiida.orm import Code
from aiida.orm import CalculationFactory, DataFactory
```

So, you were able to launch previously a `pw.x` calculation.

## Code

Again, you need to have compiled the code on the cluster and configured a new code `ph.x` in AiiDA in the very same way you installed `pw.x` (see ....). Then we load the `Code` class-instance from the database:

```
codename = 'my-ph.x'
code = Code.get_from_string(codename)
```

## Parameter

Just like the `PWscf` calculation, here we load the class `ParameterData` and we instantiate it in parameters. Again, `ParameterData` will simply represent a nested dictionary in the database, namelists at the first level, and then variables and values. But this time of course, we need to use the variables of *PHonon*!

```
ParameterData = DataFactory('parameter')
parameters = ParameterData(dict={
    'INPUTPH': {
        'tr2_ph' : 1.0e-8,
        'epsil' : True,
        'ldisp' : True,
        'nq1' : 1,
        'nq2' : 1,
        'nq3' : 1,
    }})
```

## Calculation

Now we create the object PH-calculation. As for `pw.x`, we simply do:

```
calc = code.new_calc()
```

and we set the parameters of the scheduler (and just like the `PWscf`, this is a configuration valid for the `PBSpro` and `slurm` schedulers only, see *Supported schedulers*).

```
calc.set_max_wallclock_seconds(30*60) # 30 min
calc.set_resources({"num_machines": 1})
```

We then tell the calculation to use the code and the parameters that we prepared above:

```
calc.use_parameters(parameters)
```

## Parent calculation

The phonon calculation needs to know on which `PWscf` do the perturbation theory calculation. From the database point of view, it means that the `PHonon` calculation is always a child of a `PWscf`. In practice, this means that when you want to impose this relationship, you decided to take the input parameters of the parent `PWscf` calculation, take its charge density and use them in the phonon run. That's way we need to set the parent calculation.

You first need to remember the ID of the parent calculation that you launched before (let's say it's #6): so that you can load the class of a `QE-PWscf` calculation (with the `CalculationFactory`), and load the object that represent *the* `QE-PWscf` calculation with ID #6:

```
from aiida.orm import CalculationFactory
PwCalculation = CalculationFactory('quantumespresso.pw')
parent_id = 6
parentcalc = load_node(parent_id)
```

Now that we loaded the parent calculation, we can set the phonon calc to inherit the right information from it:

```
calc.use_parent_calculation( parentcalc )
```

Note that in our database schema relations between two calculation objects are prohibited. The link between the two is indirect and is mediated by a third Data object, which represent the scratch folder on the remote cluster. Therefore the relation between the parent `Pw` and the child `Ph` appears like: `Pw -> remotescratch -> Ph`.

## Execution

Now, everything is ready, and just like `PWscf`, you just need to store all the nodes and submit this input to AiiDA, and the calculation will launch!

```
calc.store_all()
calc.submit()
```

## Script to execute

This is the script described in the tutorial above. You can use it, just remember to customize it using the right `parent_id`, the code, and the proper scheduler info.

```
#!/usr/bin/env python
from aiida import load_dbenv
load_dbenv()

from aiida.orm import Code
from aiida.orm import CalculationFactory, DataFactory

#####
# ADAPT TO YOUR NEEDS
parent_id = 6
codename = 'my-ph.x'
#####

code = Code.get_from_string(codename)

ParameterData = DataFactory('parameter')
parameters = ParameterData(dict={
    'INPUTPH': {
        'tr2_ph' : 1.0e-8,
        'epsil' : True,
        'ldisp' : True,
        'nq1' : 1,
        'nq2' : 1,
        'nq3' : 1,
    })

QEPwCalc = CalculationFactory('quantumespresso.pw')
parentcalc = load_node(parent_id)

calc = code.new_calc()
calc.set_max_wallclock_seconds(30*60) # 30 min
calc.set_resources({"num_machines": 1})

calc.use_parameters(parameters)
calc.use_code(code)
calc.use_parent_calculation(parentcalc)

calc.store_all()
print "created calculation with PK={}".format(calc.pk)
calc.submit()
```

## Exception tolerant code

You can find a more sophisticated example, that checks the possible exceptions and prints nice error messages inside your AiiDA folder, under `examples/submission/test_ph.py`.

### 1.1.12 Quantum Espresso Car-Parrinello user-tutorial

This chapter will teach you how to set up a Car-Parrinello (CP) calculation as implemented in the Quantum Espresso distribution. Again, AiiDA is not meant to teach you how to use a Quantum-Espresso code, it is assumed that you already know CP.

It is recommended that you first learn how to launch a PWscf calculation before proceeding in this tutorial (see [Quantum Espresso PWscf user-tutorial](#)), since here we will only emphasize the differences with respect to launching a PW calculation.

We want to setup a CP run of a 5 atom cell of BaTiO<sub>3</sub>. The input file that we should create is more or less this one:

```
&CONTROL
  calculation = 'cp'
  dt = 3.0000000000d+00
  iprint = 1
  isave = 100
  max_seconds = 1500
  ndr = 50
  ndw = 50
  nstep = 10
  outdir = './out/'
  prefix = 'aiida'
  pseudo_dir = './pseudo/'
  restart_mode = 'from_scratch'
  verbosity = 'high'
  wf_collect = .false.
/
&SYSTEM
  ecutrho = 2.4000000000d+02
  ecutwfc = 3.0000000000d+01
 ibrav = 0
  nat = 5
  nrlb = 24
  nr2b = 24
  nr3b = 24
  ntyp = 3
/
&ELECTRONS
  electron_damping = 1.0000000000d-01
  electron_dynamics = 'damp'
  emass = 4.0000000000d+02
  emass_cutoff = 3.0000000000d+00
/
&IONS
  ion_dynamics = 'none'
/
ATOMIC_SPECIES
Ba 137.33 Ba.pbesol-spn-rrkjus_psl.0.2.3-tot-pslib030.UPF
Ti 47.88 Ti.pbesol-spn-rrkjus_psl.0.2.3-tot-pslib030.UPF
O 15.9994 O.pbesol-n-rrkjus_psl.0.1-tested-pslib030.UPF
ATOMIC_POSITIONS angstrom
Ba 0.0000000000 0.0000000000 0.0000000000
Ti 2.0000000000 2.0000000000 2.0000000000
O 2.0000000000 2.0000000000 0.0000000000
O 2.0000000000 0.0000000000 2.0000000000
O 0.0000000000 2.0000000000 2.0000000000
CELL_PARAMETERS angstrom
4.0000000000 0.0000000000 0.0000000000
```

0.0000000000	4.0000000000	0.0000000000
0.0000000000	0.0000000000	4.0000000000

You can immediately see that the structure of this input file closely resembles that of the PWscf: only some variables are different.

## Walkthrough

Everything works like the PW calculation: you need to get the code from the database:

```
codename = 'my_cp'
code = Code.get_from_string(codename)
```

Then create the StructureData with the structure, and a ParameterData node for the inputs. This time, of course, you have to specify the correct variables for a `cp.x` calculation:

```
StructureData = DataFactory('structure')
alat = 4. # angstrom
cell = [[alat, 0., 0.],
        [0., alat, 0.],
        [0., 0., alat],
        ]
s = StructureData(cell=cell)
s.append_atom(position=(0.,0.,0.),symbols=['Ba'])
s.append_atom(position=(alat/2.,alat/2.,alat/2.),symbols=['Ti'])
s.append_atom(position=(alat/2.,alat/2.,0.),symbols=['O'])
s.append_atom(position=(alat/2.,0.,alat/2.),symbols=['O'])
s.append_atom(position=(0.,alat/2.,alat/2.),symbols=['O'])

ParameterData = DataFactory('parameter')
parameters = ParameterData(dict={
    'CONTROL': {
        'calculation': 'cp',
        'restart_mode': 'from_scratch',
        'wf_collect': False,
        'iprint': 1,
        'isave': 100,
        'dt': 3.,
        'max_seconds': 25*60,
        'nstep': 10,
    },
    'SYSTEM': {
        'ecutwfc': 30.,
        'ecutrho': 240.,
        'nr1b': 24,
        'nr2b': 24,
        'nr3b': 24,
    },
    'ELECTRONS': {
        'electron_damping': 1.e-1,
        'electron_dynamics': 'damp',
        'emass': 400.,
        'emass_cutoff': 3.,
    },
    'IONS': {
        'ion_dynamics': 'none',
    }
}).store()
```

We then create a new calculation with the proper settings:

```
calc = code.new_calc()
calc.set_max_wallclock_seconds(30*60) # 30 min
calc.set_resources({"num_machines": 1, "num_mpiprocs_per_machine": 16})
```

And we link the input data to the calculation (and therefore set the links in the database). The main difference here is that CP does not support k-points, so you should not (and cannot) link any kpoint as input:

```
calc.use_structure(s)
calc.use_code(code)
calc.use_parameters(parameters)
```

Finally, load the proper pseudopotentials using e.g. a pseudopotential family (see *Pseudopotential families tutorial*):

```
pseudo_family = 'lda_pslib'
calc.use_pseudos_from_family(pseudo_family)
```

and store everything and submit:

```
calc.store_all()
calc.submit()
```

And now, the calculation will be executed and saved in the database automatically.

## Exception tolerant code

You can find a more sophisticated example, that checks the possible exceptions and prints nice error messages inside your AiiDA folder, under `examples/submission/test_cp.py`.

### 1.1.13 Getting parsed calculation results

In this section, we describe how to get the results of a calculation, after AiiDA parsed the output of the calculation.

When a calculation is done on the remote computer, AiiDA will retrieve the results and try to parse the results with the default parser, if one is available for the given calculation. These results are stored in new nodes, and connected as output of the calculation. Of course, it is possible for a given calculation to check output nodes and get their content. However, AiiDA provides a way to directly access the results, using the *aiida.orm.calculation.job.CalculationResultManager* class, described in the next section.

#### The CalculationResultManager

---

**Note:** In the following, we assume that `calc` is a correctly finished and parsed Quantum ESPRESSO pw.x calculation. You can load such a calculation for instance with the command:

```
calc = load_node(YOURPK)
```

either in `verdi shell`, or in a python script (see [here](#) for more information on how to use `verdi shell` or how to run python scripts for AiiDA), and where `YOURPK` is substituted by a valid calculation PK in your database.

---

Each `JobCalculation` has a `res` attribute that is a `CalculationResultManager` object and gives direct access to parsed data.

To use it, you can just use then:

```
calc.res
```

that will however just return the class. You can however convert it to a list, to get all the possible keys that were parsed. For instance, if you type:

```
print list(calc.res)
```

you will get something like this:

```
[u'rho_cutoff', u'energy', u'energy_units', ...]
```

(the list of keys has been cut for clarity: you will get many more keys).

Once you know which keys have been parsed, you can access the parsed value simply as an attribute of the `res` ResultManager. For instance, to get the final total energy, you can use:

```
print calc.res.energy
```

that will print the total energy in units of eV, as also stated in the `energy_units` key:

```
print calc.res.energy_units
```

Similarly, you can get any other parsed value, for any code that provides a parser.

**Note:** the `CalculationResultManager` is also integrated with iPython/verdi shell completion mechanism: if `calc` is a valid `JobCalculation`, you can type:

```
calc.res.
```

and then press the TAB key of the keyboard to get/complete the list of valid parsed properties for the calculation `calc`.

## 1.1.14 Pseudopotential families tutorial

### What is a pseudopotential family

As you might have seen in the previous `PWscf` tutorial, the procedure of attaching a pseudopotential file to each atomic species could be a bit tedious. In many situations, you will not produce a different pseudopotential file for every calculation you do. More likely, when you start a project you will stick to a pseudopotential file for as long as possible. Moreover, in a high-throughput calculation, you will like to do calculation over several elements keeping the same functional. That's also part of the reason why there are several projects (like [PSLibrary](#) or [GBRV](#) to name a few), that intend to develop a set of pseudopotentials that covers most of the periodic table for different functionals.

That's why we introduced the *pseudopotential families*. They are basically a set of pseudopotentials that are grouped together in a special type of AiiDA Group of nodes, with the requirement that at most one pseudopotential can be present for a given chemical element.

Of course, no requirements are enforced on the complete coverage of the periodic table (also because really complete pseudopotential sets for the whole periodic table do not exist). In other words, this means that you can create a pseudopotential family containing the pseudopotentials only for a few elements that you are interested in.

**Note:** it is your responsibility to group together pseudopotentials of the same type, or obtained using the same functionals, approximations and/or levels of theory.

## How to create a pseudopotential family

Let's say for example that we want to create a family of LDA ultrasoft pseudopotentials. As the first step, you need to get all the pseudopotential files in a single folder. For your convenience, it is useful to use a common name for your files, for example with a structure like 'Element.a-short-description.UPF'.

The utility to upload a family of pseudopotentials is accessed via `verdi`:

```
verdi data upf uploadfamily path/to/folder name_of_the_family "some description for your convenience"
```

where `path/to/folder` is the path to the folder where you collected all the UPF files that you want to add to the AiiDA database and to the family with name `name_of_the_family`, and the final parameter is a string that is set in the `description` field of the group.

---

**Note:** This command will first check the MD5 checksum of each file, and it will not create a new UPFData node if the pseudopotential is already present in the DB. In this case, it will simply add that UPFData node to the group with name `name_of_the_family`.

---

---

**Note:** if you add the optional flag `--stop-if-existing`, the code will stop (without creating any new UPFData node, nor creating a group) if at least one of the files in the folder is already found in the AiiDA DB.

---

After the upload (which may take some seconds, so please be patient) the upffamily will be ready to be used.

Note that if you pass as `name_of_the_family` a name that already exists, the pseudopotentials in the folder will be added to the existing group. The code will raise an error if you try to add two (different) pseudopotentials for the same element.

## Get the list of existing families

If you want to know what are the pseudopotential families already existing in the DB, type:

```
verdi data upf listfamilies
```

Add a `-d` (or `--with-description`) flag if you want to read also the description of the family.

You can also filter the groups to get only a list of those containing a set of given elements using the `-e` option. For instance, if you want to get only the families containing the elements Ba, Ti and O, use:

```
verdi data upf listfamilies -e Ba Ti O
```

For more help on the command line options, type:

```
verdi data upf listfamilies -h
```

### 1.1.15 Manually loading pseudopotentials

If you do not want to use pseudopotentials from a family, it is also possible to load them manually (even if this is, in general, discouraged by us).

A possible way of doing it is the following: we start by creating a list of pseudopotential filenames that we need to use:

```
raw_pseudos = [  
    "Ba.pbesol-spn-rrkjus_psl.0.2.3-tot-pslib030.UPF",  
    "Ti.pbesol-spn-rrkjus_psl.0.2.3-tot-pslib030.UPF",  
    "O.pbesol-n-rrkjus_psl.0.1-tested-pslib030.UPF"]
```



(in this simple example, we expect the pseudopotentials to be in the same folder of the script). Then, we loop over the filenames and add them to the AiiDA database. The `get_or_create` method checks if the pseudopotential is already in the database (by checking its MD5 checksum) and either stores it, or just returns the node already present in the database (the second value returned is a boolean and tells us if the pseudo was already present or not). We also store the returned nodes in a list (`pseudos_to_use`).

```
UpfData = DataFactory('upf')
pseudos_to_use = []

for filename in raw_pseudos:
    absname = os.path.abspath(filename)
    pseudo, created = UpfData.get_or_create(absname, use_first=True)
    pseudos_to_use.append(pseudo)
```

As the last step, we make a loop over the pseudopotentials, and attach its pseudopotential object to the calculation:

```
for pseudo in pseudos_to_use:
    calc.use_pseudo(pseudo, kind=pseudo.element)
```

**Note:** when the pseudopotential is created, it is parsed and the elements to which it refers is stored in the database and can be accessed using the `pseudo.element` property, as shown above.

### 1.1.16 The `verdi` commands

For some the most common operations on the AiiDA software, you can work directly on the command line using the set of `verdi` commands. You already used the `verdi install` when installing the software. There are quite some more functionalities attached to this command, here's a list:

- *calculation*: query and interact with calculations
- *code*: setup and manage codes to be used
- *comment*: manage general properties of nodes in the database
- *completioncommand*: return the bash completion function to put in `~/.bashrc`
- *computer*: setup and manage computers to be used
- *daemon*: manage the AiiDA daemon
- *data*: setup and manage data specific types
- *devel*: AiiDA commands for developers
- *export*: export nodes and group of nodes
- *group*: setup and manage groups
- *import*: export nodes and group of nodes
- *install*: install/setup aiiida for the current user/create a new profile
- *node*: manage operations on AiiDA nodes
- *profile*: list and manage AiiDA profiles
- *run*: execute an AiiDA script
- *runserver*: run the AiiDA webserver on localhost
- *shell*: run the interactive shell with the Django environment
- *user*: list and configure new AiiDA users.

- *workflow*: manage the AiiDA workflow manager

Each command above can be preceded by the `-p <profile>` or `--profile=<profile>` option, as in:

```
verdi -p <profile> calculation list
```

This allows to select a specific AiiDA profile, and therefore a specific database, on which the command is executed. Thus several databases can be handled and accessed simultaneously by AiiDA. To install a new profile, use the *install* command.

---

**Note:** This profile selection has no effect on the `verdi daemon` commands.

---

Following below, a list with the subcommands available.

### `verdi calculation`

- **kill**: stop the execution on the cluster of a calculation.
- **logshow**: shows the logs/errors produced by a calculation
- **plugins**: lists the supported calculation plugins
- **inputcat**: shows an input file of a calculation node.
- **inputls**: shows the list of the input files of a calculation node.
- **list**: list the AiiDA calculations. By default, lists only the running calculations.
- **outputcat**: shows an output file of a calculation node.
- **outputls**: shows the list of the output files of a calculation node.
- **show**: shows the database information related to the calculation: used code, all the input nodes and all the output nodes.
- **gotocomputer**: open a shell to the calc folder on the cluster
- **label**: view / set the label of a calculation
- **description**: view / set the description of a calculation

---

**Note:** When using `gotocomputer`, be careful not to change any file that AiiDA created, nor to modify the output files or resubmit the calculation, unless you **really** know what you are doing, otherwise AiiDA may get very confused!

---

### `verdi code`

- **show**: shows the information of the installed code.
- **list**: lists the installed codes
- **hide**: hide codes from *verdi code list*
- **reveal**: un-hide codes for *verdi code list*
- **setup**: setup a new code
- **rename**: change the label (name) of a code. If you like to load codes based on their labels and not on their UUID's or PK's, take care of using unique labels!
- **update**: change (some of) the installation description of the code given at the moment of the setup.

- **delete:** delete a code from the database. Only possible for disconnected codes (i.e. a code that has not been used yet)

### `verdi comment`

Manages the comments attached to a database node.

- **add:** add a new comment
- **update:** change an existing comment
- **remove:** remove a comment
- **show:** show the comments attached to a node.

### `verdi completioncommand`

Prints the string to be copied and pasted to the bashrc in order to allow for autocompletion of the verdi commands.

### `verdi computer`

- **setup:** creates a new computer object
- **configure:** set up some extra info that can be used in the connection with that computer.
- **enable:** to enable a computer. If the computer is disabled, the daemon will not try to connect to the computer, so it will not retrieve or launch calculations. Useful if a computer is under maintenance.
- **rename:** changes the name of a computer.
- **disable:** disable a computer (see enable for a larger description)
- **show:** shows the details of an installed computer
- **list:** list all installed computers
- **delete:** deletes a computer node. Works only if the computer node is a disconnected node in the database (has not been used yet)
- **test:** tests if the current user (or a given user) can connect to the computer and if basic operations perform as expected (file copy, getting the list of jobs in the scheduler queue, ...)

### `verdi daemon`

Manages the daemon, i.e. the process that runs in background and that manages submission/retrieval of calculations.

- **status:** see the status of the daemon. Typically, it will either show `Daemon not running` or you will see two processes with state `RUNNING`.
- **stop:** stops the daemon
- **configureuser:** sets the user which is running the daemon. See the installation guide for more details.
- **start:** starts the daemon.
- **logshow:** show the last lines of the daemon log (use for debugging)
- **restart:** restarts the daemon.

## `verdi data`

Manages database data objects.

- **upf**: handles the Pseudopotential Datas
  - **listfamilies**: list presently stored families of pseudopotentials
  - **uploadfamily**: install a new family (group) of pseudopotentials
  - **import**: create or return (if already present) a database node, having the contents of a supplied file
  - **exportfamily**: export a family of pseudopotential files into a folder
- **structure**: handles the StructureData
  - **list**: list currently saved nodes of StructureData kind
  - **show**: use a third-party visualizer (like vmd or xcrysden) to graphically show the StructureData
  - **export**: export the node as a string of a specified format
  - **deposit**: deposit the node to a remote database
- **parameter**: handles the ParameterData objects
  - **show**: output the content of the python dictionary in different formats.
- **cif**: handles the CifData objects
  - **list**: list currently saved nodes of CifData kind
  - **show**: use third-party visualizer (like jmol) to graphically show the CifData
  - **import**: create or return (if already present) a database node, having the contents of a supplied file
  - **export**: export the node as a string of a specified format
  - **deposit**: deposit the node to a remote database
- **trajectory**: handles the TrajectoryData objects
  - **list**: list currently saved nodes of TrajectoryData kind
  - **show**: use third-party visualizer (like jmol) to graphically show the TrajectoryData
  - **export**: export the node as a string of a specified format
  - **deposit**: deposit the node to a remote database
- **label**: view / set the label of a data
- **description**: view / set the description of a data

## `verdi devel`

Here there are some functions that are in the development stage, and that might eventually find their way outside of this placeholder. As such, they are buggy, possibly difficult to use, not necessarily documented, and they might be subject to non back-compatible changes.

- **delproperty**, **describeproperties**, **getproperty**, **listproperties**, **setproperty**: handle the properties, see [here](#) for more information.

## `verdi export`

Export data from the AiiDA database to a file. See also `verdi import` to import this data on another database.

### `verdi group`

- **list**: list all the groups in the database.
- **description**: show or change the description of a group
- **show**: show the content of a group.
- **create**: create a new empty group.
- **delete**: delete an existing group (but not the nodes belonging to it).
- **addnodes**: add nodes to a group.
- **removenodes**: remove nodes from a group.

### `verdi import`

Imports data (coming from other AiiDA databases) in the current database

### `verdi install`

Used in the installation to configure the database. If it finds an already installed database, it updates the tables migrating them to the new schema.

---

**Note:** One can also create a new profile with this command:

```
verdi -p <new_profile_name> install
```

The install procedure then works as usual, and one can select there a new database. See also the *profile* command.

---

### `verdi node`

- **repo**: Show files and their contents in the local repository
- **show**: Show basic node information (PK, UUID, class, inputs and outputs)

### `verdi profile`

- **list**: Show the list of currently available profiles, indicating which one is the default one, and showing the current one with a > symbol
- **setdefault**: Set the default profile, i.e. the one to be used when no `-p` option is specified before the `verdi` command

### `verdi run`

Run a python script for AiiDA. This is the command line equivalent of the `verdi shell`. Has also features of autogroupin: by default, every node created in one a call of `verdi run` will be grouped together.

### `verdi runserver`

Starts a lightweight Web server for development and also serves static files. Currently in ongoing development.

### `verdi shell`

Runs a Python interactive interpreter. Tries to use IPython or bpython, if one of them is available. Loads on start a good part of the AiiDA infrastructure (see [here](#) for information on how to customize it).

### `verdi user`

Manages the AiiDA users. Two valid subcommands.

- **list**: list existing users configured for your AiiDA installation.
- **configure**: configure a new AiiDA user.

### `verdi workflow`

Manages the workflow. Valid subcommands:

- **report**: display the information on how the workflow is evolving.
- **kill**: kills a workflow.
- **list**: lists the workflows present in the database. By default, shows only the running ones.

## 1.1.17 AiiDA schedulers

### Supported schedulers

The list below describes the supported *schedulers*, i.e. the batch job schedulers that manage the job queues and execution on any given computer.

#### PBSPro

The **PBSPro** scheduler is supported (and it has been tested with version 12.1).

All the main features are supported with this scheduler.

The JobResource class to be used when setting the job resources is the *NodeNumberJobResource (PBS-like)*

#### SLURM

---

**Note:** The Slurm plugin referenced below is available in the EPFL version.

---

The **SLURM** scheduler is supported (and it has been tested with version 2.5.4).

All the main features are supported with this scheduler.

The JobResource class to be used when setting the job resources is the *NodeNumberJobResource (PBS-like)*

## SGE

---

**Note:** The SGE plugin referenced below is available in the EPFL version.

---

The [SGE](#) scheduler (Sun Grid Engine, now called Oracle Grid Engine) is supported (and it has been tested with version GE 6.2u3), together with some of the main variants/forks.

All the main features are supported with this scheduler.

The JobResource class to be used when setting the job resources is the [ParEnvJobResource \(SGE-like\)](#)

## PBS/Torque & Loadleveler

PBS/Torque and Loadleveler are not fully supported yet, even if their support is one of our top priorities. For the moment, you can try the PBSPro plugin instead of PBS/Torque, that *may* also work for PBS/Torque (even if there will probably be some small issues).

## Direct execution (bypassing schedulers)

The direct scheduler, to be used mainly for debugging, is an implementation of a scheduler plugin that does not require a real scheduler installed, but instead directly executes a command, puts it in the background, and checks for its process ID (PID) to discover if the execution is completed.

**Warning:** The direct execution mode is very fragile. Currently, it spawns a separate Bash shell to execute a job and track each shell by process ID (PID). This poses following problems:

- PID numeration is reset during reboots;
- PID numeration is different from machine to machine, thus direct execution is *not* possible in multi-machine clusters, redirecting each SSH login to a different node in round-robin fashion;
- there is no real queueing, hence, all calculation started will be run in parallel.

**Warning:** Direct execution bypasses schedulers, so it should be used with care in order not to disturb the functioning of machines.

All the main features are supported with this scheduler.

The JobResource class to be used when setting the job resources is the [NodeNumberJobResource \(PBS-like\)](#)

## Job resources

When asking a scheduler to allocate some nodes/machines for a given job, we have to specify some job resources (that typically include information as, for instance, the number of required nodes or the numbers of MPI processes per node).

Unfortunately, the way of specifying this piece of information is different on different clusters. Instead of having one only abstract class, we chose to adopt different subclasses, keeping in this way the specification of the resources as similar as possible to what the user would do when writing a scheduler script. Note that only one subclass can be used, given a specific scheduler.

The base class, from which all job resource subclasses inherit, is `aiida.scheduler.datastructures.JobResource`. All classes define at least one method, `get_tot_num_mpiprocs()`, that returns the total number of MPI processes requested.

---

**Note:** to load a specific job resource subclass, you can load it manually by directly loading the correct class, e.g.:

```
from aiida.scheduler.datastructures import NodeNumberJobResource
```

However, in general, you will pass the fields to set directly to the `set_resources()` method of a `JobCalculation` object. For instance:

```
calc = JobCalculation(computer=...) # select here a given computer configured
                                     # in AiiDA

# This assumes that the computer is configured to use a scheduler with
# job resources of type NodeNumberJobResource
calc.set_resources({"num_machines": 4, "num_mpiprocs_per_machine": 16})
```

### NodeNumberJobResource (PBS-like)

This is the way of specifying the job resources in PBS and SLURM. The class is `aiida.scheduler.datastructures.NodeNumberJobResource`.

Once an instance of the class is obtained, you have the following fields that you can set:

- `res.num_machines`: specify the number of machines (also called nodes) on which the code should run
- `res.num_mpiprocs_per_machine`: number of MPI processes to use on each machine
- `res.tot_num_mpiprocs`: the total number of MPI processes that this job is requesting
- `res.num_cores_per_machine`: specify the number of cores to use on each machine
- `res.num_cores_per_mpiproc`: specify the number of cores to run each MPI process

Note that you need to specify only two among the first three fields above, for instance:

```
res = NodeNumberJobResource()
res.num_machines = 4
res.num_mpiprocs_per_machine = 16
```

asks the scheduler to allocate 4 machines, with 16 MPI processes on each machine. This will automatically ask for a total of  $4 \times 16 = 64$  total number of MPI processes.

The same can be achieved passing the fields directly to the constructor:

```
res = NodeNumberJobResource(num_machines=4, num_mpiprocs_per_machine=16)
```

or, even better, directly calling the `set_resources()` method of the `JobCalculation` class (assuming here that `calc` is your calculation object):

```
calc.set_resources({"num_machines": 4, "num_mpiprocs_per_machine": 16})
```

### Note:

If you specify `res.num_machines`, `res.num_mpiprocs_per_machine`, and `res.tot_num_mpiprocs` fields (not recommended), make sure that they satisfy:

```
res.num_machines * res.num_mpiprocs_per_machine = res.tot_num_mpiprocs
```

Moreover, if you specify `res.tot_num_mpiprocs`, make sure that this is a multiple of `res.num_machines` and/or `res.num_mpiprocs_per_machine`.



**Note:** When creating a new computer, you will be asked for a `default_mpiprocs_per_machine`. If you specify it, then you can avoid to specify `num_mpiprocs_per_machine` when creating the resources for that computer, and the default number will be used.

Of course, all the requirements between `num_machines`, `num_mpiprocs_per_machine` and `tot_num_mpiprocs` still apply.

Moreover, you can explicitly specify `num_mpiprocs_per_machine` if you want to use a value different from the default one.

The `num_cores_per_machine` and `num_cores_per_mpiproc` fields are optional. If you specify `num_mpiprocs_per_machine` and `num_cores_per_machine` fields, make sure that:

```
res.num_cores_per_mpiproc * res.num_mpiprocs_per_machine = res.num_cores_per_machine
```

If you want to specify single value in `num_mpiprocs_per_machine` and `num_cores_per_machine`, please make sure that `res.num_cores_per_machine` is multiple of `res.num_cores_per_mpiproc` and/or `res.num_mpiprocs_per_machine`.

**Note:** In PBSPro, the `num_mpiprocs_per_machine` and `num_cores_per_machine` fields are used for mpiprocs and ppn respectively.

**Note:** In Torque, the `num_mpiprocs_per_machine` field is used for ppn unless the `num_mpiprocs_per_machine` is specified.

### ParEnvJobResource (SGE-like)

In SGE and similar schedulers, one has to specify a *parallel environment* and the *total number of CPUs* requested. The class is `aiida.scheduler.datastructures.ParEnvJobResource`.

Once an instance of the class is obtained, you have the following fields that you can set:

- `res.parallel_env`: specify the parallel environment in which you want to run your job (a string)
- `res.tot_num_mpiprocs`: the total number of MPI processes that this job is requesting

Remember to always specify both fields. No checks are done on the consistency between the specified parallel environment and the total number of MPI processes requested (for instance, some parallel environments may have been configured by your cluster administrator to run on a single machine). It is your responsibility to make sure that the information is valid, otherwise the submission will fail.

Some examples:

- setting the fields one by one:

```
res = ParEnvJobResource()
res.parallel_env = 'mpi'
res.tot_num_mpiprocs = 64
```

- setting the fields directly in the class constructor:

```
res = ParEnvJobResource(parallel_env='mpi', tot_num_mpiprocs=64)
```

- even better, directly calling the `set_resources()` method of the `JobCalculation` class (assuming here that `calc` is your calculation object):

```
calc.set_resources({"parallel_env": 'mpi', "tot_num_mpiprocs": 64})
```

### 1.1.18 Calculations

AiiDA calculations can be of two kinds:

- *JobCalculation*: those who need to be run on a scheduler
- *InlineCalculation*: rapid executions that are executed by the daemon itself, on your local machine.

In the following, we will refer to the JobCalculations as a Calculation for the sake of simplicity, unless we explicitly say otherwise. In the same way, also the command `verdi calculation` refers to JobCalculation's.

### 1.1.19 Check the state of calculations

Once a calculation has been submitted to AiiDA, everything else will be managed by AiiDA: the inputs will be checked to verify that they are consistent. If the inputs are complete, the input files will be prepared, sent on cluster, and a job will be submitted. The AiiDA daemon will then monitor the scheduler, and after execution the outputs automatically retrieved and parsed.

During these phases, it is useful to be able to check and verify the state of a calculation. There are different ways to perform such an operation, described below.

#### The `verdi calculation` command

The simplest way to check the state of submitted calculations is to use the `verdi calculation list` command from the command line. To get help on its use and command line options, run it with the `-h` or `--help` option:

```
verdi calculation list --help
```

#### Possible calculation states

The calculation could be in several states. The most common you should see:

1. **NEW**: the calculation node has been created, but has not been submitted yet.
2. **WITHSCHEDULER**: the job is in some queue on the remote computer. Note that this does not mean that the job is waiting in a queue, but it may be running or finishing, but it did not finish yet. AiiDA has to wait.
3. **FINISHED**: the job on the cluster was finished, AiiDA already retrieved it and stored the results in the database. In most cases, this also means that the parser managed to parse the output file.
4. **FAILED**: something went wrong, and AiiDA rose an exception. The error could be of various nature: the inputs were not enough or were not correct, the execution on the cluster failed, or (depending on the output plugin) the code ended without completing successfully or producing a valid output file. Other possible more specific “failed” states include **SUBMISSIONFAILED**, **RETRIEVALFAILED** and **PARSINGFAILED**.
5. For very short times, when the job completes on the remote computer and AiiDA retrieves and parses it, you may happen to see a calculation in the **COMPUTED**, **RETRIEVING** and **PARSING** states.

Eventually, when the calculation has finished, you will find the computed quantities in the database, and you will be able to query the database for the results that were parsed!

#### Directly in python

If you prefer to have more flexibility or to check the state of a calculation programmatically, you can execute a script like the following, where you just need to specify the ID of the calculation you are interested in:

```

from aiida import load_dbenv
load_dbenv()

from aiida.orm import JobCalculation

## pk must be a valid integer pk
calc = load_node(pk)
## Alternatively, with the UUID (uuid must be a valid UUID string)
# calc = load_node(uuid)
print "AiiDA state:", calc.get_state()
print "Last scheduler state seen by the AiiDA daemon:", calc.get_scheduler_state()

```

Note that, as specified in the comments, you can also get a code by knowing its UUID; the advantage is that, while the numeric ID will typically change after a sync of two databases, the UUID is a unique identifier and will be preserved across different AiiDA instances.

**Note:** `calc.get_scheduler_state()` returns the state on the scheduler (queued, held, running, ...) as seen the last time that the daemon connected to the remote computer. The time at which the last check was performed is returned by the `calc.get_scheduler_lastchecktime()` method (that returns `None` if no check has been performed yet).

### The `verdi calculation gotocomputer` command

Sometimes, it may be useful to directly go to the folder on which the calculation is running, for instance to check if the output file has been created.

In this case, it is possible to run:

```
verdi calculation gotocomputer CALCULATIONPK
```

where `CALCULATIONPK` is the PK of the calculation. This will open a new connection to the computer (either simply a bash shell or a ssh connection, depending on the transport) and directly change directory to the appropriate folder where the code is running.

**Note:** Be careful not to change any file that AiiDA created, nor to modify the output files or resubmit the calculation, unless you **really** know what you are doing, otherwise AiiDA may get very confused!

## 1.1.20 Set calculation properties

There are various methods which specify the calculation properties. Here follows a brief documentation of their action.

- `c.set_max_memory_kb`: require explicitly the memory to be allocated to the scheduler job.
- `c.set_append_text`: write a set of bash commands to be executed after the call to the executable. These commands are executed only for this instance of calculations. Look also at the computer and code `append_text` to write bash commands for any job run on that computer or with that code.
- `c.set_max_wallclock_seconds`: set (as integer) the scheduler-job wall-time in seconds.
- `c.set_computer`: set the computer on which the calculation is run. Unnecessary if the calculation has been created from a code.
- `c.set_mpirun_extra_params`: set as a list of strings the parameters to be passed to the mpirun command. Example: `mpirun -np 8 extra_params[0] extra_params[1] ... exec.x` Note: the process number is set by the resources.

- `c.set_custom_scheduler_commands`: set a string (even multiline) which contains personalized job-scheduling commands. These commands are set at the beginning of the job-scheduling script, before any non-scheduler command. (`prepend_texts` instead are set after all job-scheduling commands).
- `c.set_parser_name`: set the name of the parser to be used on the output. Typically, a plugin will have already a default plugin set, use this command to change it.
- `c.set_environment_variables`: set a dictionary, whose key and values will be used to set new environment variables in the job-scheduling script before the execution of the calculation. The dictionary is translated to: `export 'keys'='values'`.
- `c.set_prepend_text`: set a string that contains bash commands, to be written in the job-scheduling script for this calculation, right before the call to the executable. (it is used for example to load modules). Note that there are also prepend text for the computer (that are used for any job-scheduling script on the given computer) and for the code (that are used for any scheduling script using the given code), the `prepend_text` here is used only for this instance of the calculation: be careful in avoiding duplication of bash commands.
- `c.set_extra`: pass a key and a value, to be stored in the `Extra` attribute table in the database.
- `c.set_extras`: like `set extra`, but you can pass a dictionary with multiple keys and values.
- `c.set_priority`: set the job-scheduler priority of the calculation (AiiDA does not have internal priorities). The function accepts a value that depends on the scheduler. plugin (but typically is an integer).
- `c.set_queue_name`: pass in a string the name of the queue to use on the job-scheduler.
- `c.set_import_sys_environment`: default=True. If True, the job-scheduling script will load the environment variables.
- `c.set_resources`: set the resources to be used by the calculation like the number of nodes, wall-time, ..., by passing a dictionary to this method. The keys of this dictionary, i.e. the resources, depend on the specific scheduler plugin that has to run them. Look at the documentation of the scheduler (type is given by: `calc.get_computer().get_scheduler_type()`).
- `c.set_withmpi`: True or False, if True (the default) it will call the executable as a parallel run.

## 1.1.21 Comments

There are various ways of attaching notes/comments to a node within AiiDA. In the first examples of scripting, you should already have noticed the possibility of storing a `label` or a `description` to any AiiDA Node. However, these properties are defined at the creation of the Node, and it is not possible to modify them after the Node has been stored.

The `Node.comment` provides a simple way to have a more dynamic management of comments, in which any user can write a comment on the Node, or modify it or delete it.

The `verdi comment` provides a set of methods that are used to manipulate the comments:

- **add**: add a new comment to a Node.
- **update**: modify a comment.
- **show**: show the existing comments attached to the Node.
- **remove**: remove a comment.

## 1.1.22 Extracting data from the Database

In this section we will overview some of the tools provided by AiiDA by means of which you can navigate through the data inside the AiiDA database.

## Finding input and output nodes

Let's start with a reference node that you loaded from the database, for example the node with PK 17:

```
n = load_node(17)
```

Now, we want to find the nodes which have a direct link to this node. There are several methods to extract this information (for developers see all the methods and their docstring: `get_outputs`, `get_outputs_dict`, `c.get_inputs` and `c.get_inputs_dict`). The most practical way to access this information, especially when working on the `verdi shell`, is by means of the `inp` and `out` methods.

The `inp` method is used to list and access the nodes with a direct link to `n` in input. The names of the input links can be printed by `list(n.inp)` or interactively by `n.inp. + TAB`. As an example, suppose that `n` has an input `KpointsData` object under the linkname `kpoints`. The command:

```
n.inp.kpoints
```

returns the `KpointsData` object.

Similar methods exist for the `out` method, which will display the names of links in output from `n` and can be used to access such output nodes. Suppose that `n` has an output `FolderData` with linkname `retrieved`, then the command:

```
n.out.retrieved
```

returns the `FolderData` object.

**Note:** At variance with input, there can be more than one output objects with the same linkname (for example: a code object can be used by several calculations always with the same linkname `code`). As such, for every output linkname, we append the string `_pk`, with the pk of the output node. There is also a linkname without pk appended, which is assigned to the oldest link. As an example, imagine that `n` is a code, which is used by calculation #18 and #19, the linknames shown by `n.out` are:

```
n.out. >>
* code
* code_18
* code_19
```

The method `n.out.code_18` and `n.out.code_19` will return two different calculation objects, and `n.out.code` will return the oldest (the reference is the creation time) between calculation 18 and 19. If one calculation (say 18) exist only in output, there is then less ambiguity, and you are sure that the output of `n.out.code` coincides with `n.out.code_18`.

### 1.1.23 Querying in AiiDA

The advantage of storing information in a database is that questions can be asked on the data, and an answer can be rapidly provided.

Here we describe different approaches to query the data in AiiDA.

**Note:** This section is still only a stub and will be significantly improved in the next versions.

#### Directly querying in Django

If you know how AiiDA stores the data internally in the database, you can directly use Django to query the database (or even use directly SQL commands, if you really feel the urge to do so). Documentation on how queries work in Django

can be found on the [official Django documentation](#). The models can be found in `aiida.djsite.db.models` and are directly accessible as `models` in the `verdi shell` or via `verdi run`.

## Using the `querytool`

We provide a Python class (`aiida.orm.querytool.QueryTool`) to perform the most common types of queries (mainly on nodes, links and their attributes) through an easy Python class interface, without the need to know anything about the SQL query language.

---

**Note:** We are working a lot on the interface for querying through the `QueryTool`, so the interface could change significantly in the future to allow for more advanced querying capabilities.

---

To use it, in your script (or within the `verdi shell`) you need first to load the `QueryTool` class:

```
from aiida.orm.querytool import QueryTool
```

Then, create an instance of this class, which will represent your query (you need to create a new instance for each different query you want to execute):

```
q = QueryTool()
```

Now, you can call a set of methods on the `q` object to decide the filters you want to apply. The first type of filter one may want to apply is on the type of nodes you want to obtain (the `QueryTool`, in the current version, always queries only nodes in the DB). You can do so passing the correct Node subclass to the `set_class()` method, for instance:

```
q.set_class(Calculation)
```

Then, if you want to query only calculations within a given group:

```
q.set_group(group_name, exclude=False)
```

where `group_name` is the name of the group you want to select. The `exclude` parameter, if `True`, negates the query (i.e., considers all objects *not* included in the give group). You can call the `set_group()` method multiple times to add more filters.

The most important query specification, though, is on the attributes of a given node.

If you want to query for attributes in the `DbAttribute` table, use the `add_attr_filter()` method:

```
q.add_attr_filter("energy", "<=", 0., relnode="res")
```

At this point, the query `q` describes a query you still have to run, which will return each calculation `calc` for which the result node `calc.res.energy` is less or equal to 0.

The `relnode` parameter allows the user to perform queries not only on the nodes you want to get out of the query (in this case, do not specify any `relnode` parameter) but also on the value of the attributes of nodes *linked* to the result nodes. For instance, specifying `"res"` as `relnode`, one gets as result of the query nodes *whose output result* has a negative energy.

Also in this case, you can add multiple filters on attributes, or you can use the same syntax also on data you stored in the `DbExtra` table using `add_extra_filter()`.

---

**Note:** We remind here that while attributes are properties that describe a node, are used internally by AiiDA and cannot be changed after the node is stored – for instance, the coordinates of atoms in a crystal structure, the input parameters for a calculation, ... – extras (stored in `DbExtra`) have the same format and are at full disposal of the user for adding metadata to each node, tagging, and later quick querying.

---

Finally, to run the query and get the results, you can use the `run_query()` method, that will return an iterator over the results of the query. For instance, if you stored A and B as extra data of a given node, you can get a list of the energy of each calculation, and the value of A and B, using the following command:

```
res = [(node.res.energy,
        node.get_extra("A"),
        node.get_extra("B"))
        for node in q.run_query()]
```

**Note:** After having run a query, if you want to run a new one, even if it is a simple modification of the current one, please discard the `q` object and create a new one with the new filters.

## The transitive closure table

Another type of query that is very common is the discovery of whether two nodes are linked through a path in the AiiDA graph database, regardless of how many nodes are in between.

This is particularly important because, for instance, you may be interested in discovering which crystal structures have, say, all phonon frequencies that are positive; but the information on the phonon frequencies is in a node that is typically not directly linked to the crystal structure (you typically have in between at least a SCF calculation, a phonon calculation on a coarse grid, and an interpolation of the phonon bands on a denser grid; moreover, each calculation may include multiple restarts).

In order to make these queries very efficient (and since we expect that typical workflows, especially in Physics and Materials Science, involve a lot of relatively small, disconnected graphs), we have implemented triggers at the database SQL level to automatically generate a *transitive closure* table, i.e., a table that for each node contains all his *parents* (at any depth level) and all the *children* (at any depth level). This means that, every time two nodes are joined by a link, this table is automatically updated to contain all the new available paths.

With the aid of such a table, discovering if two nodes are connected or not becomes a matter of a single query. This table is accessible using Django commands, and is called `DbPath`.

Transitive closure *paths* contain a parent and a child. Moreover, they also contain a *depth*, giving how many nodes have to be traversed to connect the two parent and child nodes (to make this possible, an entry in the `DbPath` table is stored for each possible path in the graph). The depth does not include the first and last node (so, a depth of zero means that two nodes are directly connected through a link).

Three further columns are stored, and they are mainly used to quickly (and recursively) discover which are the nodes that have been traversed.

### Todo

The description of the exact meaning of the three additional columns (`entry_edge_id`, `direct_edge_id`, and `exit_edge_id`, will be added soon; in the meantime, you can give a look to the implementation of the `expand()` method).

Finally, given a `DbPath` object, we provide a `expand()` method to get a list of all the nodes (in the correct order) that are traversed by the specific path. List elements are AiiDA nodes.

Here we present a simple example of how you can use the transitive closure table, imagining that you want to get the path between two nodes `n1` and `n2`. We will assume that only a single path exists between the two nodes. If no path exists, an exception will be raised in the line marked below. If more than one path exists, only the first one will be returned. The extension to manage the exception and to manage multiple paths is straightforward:

```
n1 = load_node(NODEPK1)
n2 = load_node(NODEPK2)
# In the following line, we are choosing only the first
```

```
# path returned by the query (with [0]).  
# Change here to manage zero or multiple paths!  
dbpath = models.DbPath.objects.filter(parent=n1, child=n2)[0]  
# Print all nodes in the path  
print dbpath.expand()
```

### 1.1.24 AiiDA workflows

Workflows are one of the most important components for real high-throughput calculations, allowing the user to scale well defined chains of calculations on any number of input structures, both generated or acquired from an external source.

Instead of offering a limited number of automatization schemes, crafted for some specific functions (equation of states, phonons, etc...) in AiiDA a complete workflow engine is present, where the user can script in principle any possible interaction with all the AiiDA components, from the submission engine to the materials databases connections. In AiiDA a workflow is a python script executed by a daemon, containing several user defined functions called steps. In each step all the AiiDA functions are available and calculations are launched and retrieved, as well as other sub-workflows.

In this document we'll introduce the main workflow infrastructure from the user perspective, discussing and presenting some examples that will cover all the features implemented in the code. A more detailed description of each function can be found in the developer documentation.

#### How it works

The rationale of the entire workflow infrastructure is to make efficient, reproducible and scriptable anything a user can do in the AiiDA shell. A workflow in this sense is nothing more than a list of AiiDA commands, split in different steps that depend one on each other and that are executed in a specific order. A workflow step is written with the same python language, using the same commands and libraries you use in the shell, stored in a file as a python class and managed by a daemon process.

Before starting to analyze our first workflow we should summarize very shortly the main working logic of a typical workflow execution, starting with the definition of the management daemon. The AiiDA daemon handles all the operations of a workflow, script loading, error handling and reporting, state monitoring and user interaction with the execution queue.

The daemon works essentially as an infinite loop, iterating several simple operations:

1. It checks the running step in all the active workflows, if there are new calculations attached to a step it submits them.
2. It retrieves all the finished calculations. If one step of one workflow exists where all the calculations are correctly finished it reloads the workflow and executes the next step as indicated in the script.
3. If a workflow's next step is the exit one, the workflow is terminated and the report is closed.

This simplified process is the very heart of the workflow engine, and while the process loops a user can submit a new workflow to be managed from the Verdi shell (or through a script loading the necessary Verdi environment). In the next chapter we'll initialize the daemon and analyze a simple workflow, submitting it and retrieving the results.

---

**Note:** The workflow engine of AiiDA is now fully operational but will undergo major improvements in a near future. Therefore, some of the methods or functionalities described in the following might change.

---



## The AiiDA daemon

As explained the daemon must be running to allow the execution of workflows, so the first thing needed to start it to launch the daemon. We can use the verdi script facility from your computer's shell:

```
>> verdi daemon start
```

This command will launch a background job (a daemon in fact) that will continuously check for new or running workflow to manage. Thanks to the asynchronous structure of AiiDA if the daemon gets interrupted (or the computer running the daemon restarted for example), once it will be restarted all the workflow will proceed automatically without any problem. The only thing you need to do to restart the workflow it's exactly the same command above. To stop the daemon instead we use the same command with the `stop` directive, and to have a very fast check about the execution we can use the `state` directive to obtain more information.

## A workflow demo

Now that the daemon is running we can focus on how to write our first workflow. As explained a workflow is essentially a python class, stored in a file accessible by AiiDA (in the same AiiDA path). By convention workflows are stored in `.py` files inside the `aiida/workflows` directory; in the distribution you'll find some examples (some of them analyzed here) and a user directory where user defined workflows can be stored. Since the daemon is aware only of the classes present at the time of its launch, remember to restart the daemon (`verdi daemon restart`) every time you add a new workflow to let AiiDA see it.

We can now study a very first example workflow, contained in the `wf_demo.py` file inside the distribution's `workflows` directory. Even if this is just a toy model, it helps us to introduce all the features and details on how a workflow works, helping us to understand the more sophisticated examples reported later.

```
1 import aiida.common
2 from aiida.common import aiida_logger
3 from aiida.orm.workflow import Workflow
4 from aiida.orm import Code, Computer
5
6 logger = aiida_logger.getChild('WorkflowDemo')
7
8 class WorkflowDemo(Workflow):
9
10     def __init__(self, **kwargs):
11
12         super(WorkflowDemo, self).__init__(**kwargs)
13
14     def generate_calc(self):
15
16         from aiida.orm import Code, Computer, CalculationFactory
17         from aiida.common.datastructures import calc_states
18
19         CustomCalc = CalculationFactory('simpleplugins.templatereplacer')
20
21         computer = Computer.get("localhost")
22
23         calc = CustomCalc(computer=computer, withmpi=True)
24         calc.set_resources(num_machines=1, num_mpiprocs_per_machine=1)
25         calc._set_state(calc_states.FINISHED)
26         calc.store()
27
28         return calc
29
30 @Workflow.step
```

```

31 def start(self):
32
33     from aiida.orm.node import Node
34
35     # Testing parameters
36     p = self.get_parameters()
37
38     # Testing calculations
39     self.attach_calculation(self.generate_calc())
40     self.attach_calculation(self.generate_calc())
41
42     # Testing report
43     self.append_to_report("Starting workflow with params: {0}".format(p))
44
45     # Testing attachments
46     n = Node()
47     attrs = {"a": [1,2,3], "n": n}
48     self.add_attributes(attrs)
49
50     # Test process
51     self.next(self.second_step)
52
53 @Workflow.step
54 def second_step(self):
55
56     # Test retrieval
57     calcs = self.get_step_calculations(self.start)
58     self.append_to_report("Retrieved calculation 0 (uuid): {0}".format(calcs[0].uuid))
59
60     # Testing report
61     a = self.get_attributes()
62     self.append_to_report("Execution second_step with attachments: {0}".format(a))
63
64     # Test results
65     self.add_result("scf_converged", calcs[0])
66
67     self.next(self.exit)

```

As discussed before this is native python code, meaning that a user can load any library or script accessible from their PYTHONPATH and interacting with any database or service of preference inside the workflow. We'll now go through all the details of the first workflow, line by line, discussing the most important methods and discovering along the way all the features available.

**lines 1-7** Module imports. Some are necessary for the Workflow objects but many more can be added for user defined functions and libraries.

**lines 8-12** Superclass definition, a workflow **MUST** extend the Workflow class from the `aiida.orm.workflow`. This is a fundamental requirement, since the subclassing is the way AiiDA understand if a class inside the file is an AiiDA workflow or a simple utility class. Note that for back-compatibility with python 2.7 also the explicit initialization of line 12 is necessary to make things work correctly.

**lines 14-28** Once the class is defined a user can add as many methods as he wishes, to generate calculations or to download structures or to compute new ones starting from a query in previous AiiDA calculations present in the DB. In the script above the method `generate_calc` will simply prepare a dummy calculation, setting its state to finished and returning the object after having it stored in the repository. This utility function will allow the dummy workflow run without the need of any code or machine except for localhost configured. In real cases, as we'll see, a calculation will be set up with parameters and structures defined in more sophisticated ways, but the logic underneath is identical as far as the workflow inner working is concerned.

**lines 30-51** This is the first *step*, one of the main components in the workflow logic. As you can see the `start` method is decorated as a `Workflow.step` making it a very unique kind of method, automatically stored in the database as a container of calculations and sub-workflows. Several functions are available to the user when coding a workflow step, and in this method we can see most of the basic ones:

- **line 36** `self.get_parameters()`. With this method we can retrieve the parameters passed to the workflow when it was initialized. Parameters cannot be modified during an execution, while attributes can be added and removed.
- **lines 39-40** `self.attach_calculation(JobCalculation)`. This is a key point in the workflow, and something possible only inside a step method. JobCalculations, generated in the methods or retrieved from other utility methods, are attached to the workflow's step, launched and executed completely by the daemon, without the need of user interaction. Failures, re-launching and queue management are all handled by the daemon, and thousands of calculations can be attached. The daemon will poll the servers until all the step calculations will be finished, and only after that it will pass to the next step.
- **line 43** `self.append_to_report(string)`. Once the workflow will be launched, the user interactions are limited to some events (stop, relaunch, list of the calculations) and most of the times is very useful to have custom messages during the execution. For this each workflow is equipped with a reporting facility, where the user can fill with any text and can retrieve both live and at the end of the execution.
- **lines 45-48** `self.add_attributes(dict)`. Since the workflow is instantiated every step from scratch, if a user wants to pass arguments between steps he must use the attributes facility, where a dictionary of values (accepted values are basic types and AiiDA nodes) can be saved and retrieved from other steps during future executions.
- **line 52** `self.next(Workflow.step)`. This is the final part of a step, where the user points the engine about what to do after all the calculations in the steps (on possible sub-workflows, as we'll see later) are terminated. The argument of this function has to be a `Workflow.step` decorated method of the same workflow class, or in case this is the last step to be executed you can use the common method `self.exit`, always present in each `Workflow` subclass.

---

**Note:** make sure to `store()` all input nodes for the attached calculations, as unstored nodes will be lost during the transition from one step to another.

---

**lines 53-67** When the workflow will be launched through the `start` method, the AiiDA daemon will load the workflow, execute the step, launch all the calculations and monitor their state. Once all the calculations in `start` will be finished the daemon will then load and execute the next step, in this case the one called `second_step`. In this step new features are shown:

- **line 57** `self.get_step_calculations(Workflow.step)`. Anywhere after the first step we may need to retrieve and analyze calculations executed in a previous steps. With this method we can have access to the list of calculations of a specific workflows step, passed as an argument.
- **line 61** `self.get_attributes()`. With this call we can retrieve the attributes stored in previous steps. Remember that this is the only way to pass arguments between different steps, adding them as we did in line 48.
- **line 65** `self.add_result()`. When all the calculations are done it's useful to tag some of them as results, using custom string to be later searched and retrieved. Similarly to the `get_step_calculations`, this method works on the entire workflow and not on a single step.
- **line 67** `self.next(self.exit)`. This is the final part of each workflow, setting the exit. Every workflow inheritate a fictitious step called `exit` that can be set as a next to any step. As the names suggest, this implies the workflow execution to finish correctly.

## Running a workflow

After saving the workflow inside a python file located in the `aiida/workflows` directory, we can launch the workflow simply invoking the specific workflow class and executing the `start()` method inside the Verdi shell. It's important to remember that all the AiiDA framework needs to be accessible for the workflow to be launched, and this can be achieved either with the verdi shell or by any other python environment that has previously loaded the AiiDA framework (see the developer manual for this).

To launch the verdi shell execute `verdi shell` from the command line; once inside the shell we have to import the workflow class we want to launch (this command depends on the file location and the class name we decided). In this case we expect we'll launch the `WorkflowDemo` presented before, located in the `wf_demo.py` file in the clean AiiDA distribution. In the shell we execute:

```
>> from aiida.workflows.wf_demo import WorkflowDemo
>> params = {"a": [1, 2, 3]}
>> wf = WorkflowDemo(params=params)
>> wf.start()
```

---

**Note:** If you want to write the above script in a file, remember to run it with `verdi run` and not simply with `python`, or otherwise to use the other techniques described [here](#).

---

In these four lines we loaded the class, we created some fictitious parameter and we initialized the workflow. Finally we launched it with the `start()` method, a lazy command that in the background adds the workflow to the execution queue monitored by the verdi daemon. In the background the daemon will handle all the workflow processes, stepping each method, launching and retrieving calculations and monitoring possible errors and problems.

Since the workflow is now managed by the daemon, to interact with it we need special methods. There are basically two ways to see how the workflows are running: by printing the workflow `list` or its `report`.

- **Workflow list**

From the command line we run:

```
>> verdi workflow list
```

This will list all the running workflows, showing the state of each step and each calculation (and, when present, each sub-workflow - see below). It is the fastest way to have a snapshot of what your AiiDA workflow daemon is working on. An example output right after the `WorkflowDemo` submission should be

```
+ Workflow WorkflowDemo (pk: 1) is RUNNING [0h:05m:04s]
|-* Step: start [->second_step] is RUNNING
| | Calculation (pk: 1) is FINISHED
| | Calculation (pk: 2) is FINISHED
```

For each workflow is reported the pk number, a unique id identifying that specific execution of the workflow, something necessary to retrieve it at any other time in the future (as explained in the next point).

---

**Note:** You can also print the `list` of any individual workflow from the verdi shell (here in the shell where you defined your workflow as `wf`, see above):

```
>> import aiida.orm.workflow as wfs
>> print "\n".join(wfs.get_workflow_info(wf._dbworkflowinstance))
```

- **Workflow report**

As explained, each workflow is equipped with a reporting facility the user can use to log any important intermediate information, useful to debug the state or show some details. Moreover the report is also used by AiiDA as an error reporting tool: in case of errors encountered during the execution, the AiiDA daemon will copy the

entire stack trace in the workflow report before halting it's execution. To access the report we need the specific pk of the workflow. From the command line we would run:

```
>> verdi workflow report PK_NUMBER
```

while from the verdi shell the same operation requires to use the `get_report()` method:

```
>> load_workflow(PK_NUMBER).get_report()
```

In both variants, `PK_NUMBER` is the pk number of the workflow we want the report of. The `load_workflow` function loads a `Workflow` instance from its pk number, or from its uuid (given as a string).

**Note:** It's always recommended to get the workflow instance from `load_workflow` (or from the `Workflow.get_subclass_from_pk` method) without saving this object in a variable. The information generated in the report may change and the user calling a `get_report` method of a class instantiated in the past will probably lose the most recent additions to the report.

Once launched, the workflows will be handled by the daemon until the final step or until some error occurs. In the last case, the workflow gets halted and the report can be checked to understand what happened.

#### • Killing a workflow

A user can also kill a workflow while it's running. This can be done with the following verdi command:

```
>> verdi workflow kill PK_NUMBER_1 PK_NUMBER_2 PK_NUMBER_N
```

where several pk numbers can be given. A prompt will ask for a confirmation; this can be avoided by using the `-f` option.

An alternative way to kill an individual workflow is to use the `kill` method. In the verdi shell type:

```
>> load_workflow(PK_NUMBER).kill()
```

or, equivalently:

```
>> Workflow.get_subclass_from_pk(PK_NUMBER).kill()
```

**Note:** Sometimes the `kill` operation might fail because one calculation cannot be killed (e.g. if it's running but not in the `WITHSCHEDULER`, `TOSUBMIT` or `NEW` state), or because one workflow step is in the `CREATED` state. In that case the workflow is put to the `SLEEP` state, such that no more workflow steps will be launched by the daemon. One can then simply wait until the calculation or step changes state, and try to kill it again.

## A more sophisticated workflow

In the previous chapter we've been able to see almost all the workflow features, and we're now ready to work on some more sophisticated examples, where real calculations are performed and common real-life issues are solved. As a real case example we'll compute the equation of state of a simple class of materials, `XTiO3`; the workflow will accept as an input the `X` material, it will build several structures with different crystal parameters, run and retrieve all the simulations, fit the curve and run an optimized final structure saving it as the workflow results, aside to the final optimal cell parameter value.

```
1  ## =====
2  ##      WorkflowXTiO3_EOS
3  ## =====
4
5  class WorkflowXTiO3_EOS(Workflow):
```

```

6
7     def __init__(self, **kwargs):
8
9         super(WorkflowXTiO3_EOS, self).__init__(**kwargs)
10
11     ## =====
12     ##     Object generators
13     ## =====
14
15     def get_structure(self, alat = 4, x_material = 'Ba'):
16
17         cell = [[alat, 0., 0.],
18                 [0., alat, 0.],
19                 [0., 0., alat,],
20                 ]
21
22         # BaTiO3 cubic structure
23         s = StructureData(cell=cell)
24         s.append_atom(position=(0., 0., 0.), symbols=x_material)
25         s.append_atom(position=(alat/2., alat/2., alat/2.), symbols=['Ti'])
26         s.append_atom(position=(alat/2., alat/2., 0.), symbols=['O'])
27         s.append_atom(position=(alat/2., 0., alat/2.), symbols=['O'])
28         s.append_atom(position=(0., alat/2., alat/2.), symbols=['O'])
29         s.store()
30
31         return s
32
33     def get_pw_parameters(self):
34
35         parameters = ParameterData(dict={
36             'CONTROL': {
37                 'calculation': 'scf',
38                 'restart_mode': 'from_scratch',
39                 'wf_collect': True,
40             },
41             'SYSTEM': {
42                 'ecutwfc': 30.,
43                 'ecutrho': 240.,
44             },
45             'ELECTRONS': {
46                 'conv_thr': 1.e-6,
47             })
48         parameters.store()
49
50         return parameters
51
52     def get_kpoints(self):
53
54         kpoints = KpointsData()
55         kpoints.set_kpoints_mesh([4, 4, 4])
56         kpoints.store()
57
58         return kpoints
59
60     def get_pw_calculation(self, pw_structure, pw_parameters, pw_kpoint):
61
62         params = self.get_parameters()
63
64         pw_codename = params['pw_codename']

```

```

64     num_machines          = params['num_machines']
65     num_mpiprocs_per_machine = params['num_mpiprocs_per_machine']
66     max_wallclock_seconds  = params['max_wallclock_seconds']
67     pseudo_family          = params['pseudo_family']
68
69     code = Code.get_from_string(pw_codename)
70     computer = code.get_remote_computer()
71
72     QECalc = CalculationFactory('quantumespresso.pw')
73
74     calc = QECalc(computer=computer)
75     calc.set_max_wallclock_seconds(max_wallclock_seconds)
76     calc.set_resources({"num_machines": num_machines, "num_mpiprocs_per_machine": num_mpiprocs_per_machine})
77     calc.store()
78
79     calc.use_code(code)
80
81     calc.use_structure(pw_structure)
82     calc.use_pseudos_from_family(pseudo_family)
83     calc.use_parameters(pw_parameters)
84     calc.use_kpoints(pw_kpoint)
85
86     return calc
87
88
89     ## =====
90     ##     Workflow steps
91     ## =====
92
93     @Workflow.step
94     def start(self):
95
96         params = self.get_parameters()
97         x_material = params['x_material']
98
99         self.append_to_report(x_material+"TiO3 EOS started")
100        self.next(self.eos)
101
102    @Workflow.step
103    def eos(self):
104
105        from aiida.orm import Code, Computer, CalculationFactory
106        import numpy as np
107
108        params = self.get_parameters()
109
110        x_material          = params['x_material']
111        starting_alat       = params['starting_alat']
112        alat_steps          = params['alat_steps']
113
114
115        a_sweep = np.linspace(starting_alat*0.85, starting_alat*1.15, alat_steps).tolist()
116
117        aiida.logger.info("Storing a_sweep as "+str(a_sweep))
118        self.add_attribute('a_sweep', a_sweep)
119
120        for a in a_sweep:

```

```

122         self.append_to_report("Preparing structure {0} with alat {1}".format(x_material+"TiO3",
123
124         calc = self.get_pw_calculation(self.get_structure(alat=a, x_material=x_material),
125                                     self.get_pw_parameters(),
126                                     self.get_kpoints())
127
128         self.attach_calculation(calc)
129
130
131     self.next(self.optimize)
132
133 @Workflow.step
134 def optimize(self):
135
136     from aiida.orm.data.parameter import ParameterData
137
138     x_material = self.get_parameter("x_material")
139     a_sweep = self.get_attribute("a_sweep")
140
141     aiida_logger.info("Retrieving a_sweep as {0}".format(a_sweep))
142
143     # Get calculations
144     start_calcs = self.get_step_calculations(self.eos) #.get_calculations()
145
146     # Calculate results
147     #-----
148
149     e_calcs = [c.res.energy for c in start_calcs]
150     v_calcs = [c.res.volume for c in start_calcs]
151
152     e_calcs = zip(*sorted(zip(a_sweep, e_calcs)))[1]
153     v_calcs = zip(*sorted(zip(a_sweep, v_calcs)))[1]
154
155     # Add to report
156     #-----
157     for i in range (len(a_sweep)):
158         self.append_to_report(x_material+"TiO3 simulated with a="+str(a_sweep[i])+", e="+str(e_
159
160     # Find optimal alat
161     #-----
162
163     murnpars, ier = Murnaghan_fit(e_calcs, v_calcs)
164
165     # New optimal alat
166     optimal_alat = murnpars[3]** (1 / 3.0)
167     self.add_attribute('optimal_alat', optimal_alat)
168
169     # Build last calculation
170     #-----
171
172     calc = self.get_pw_calculation(self.get_structure(alat=optimal_alat, x_material=x_material),
173                                     self.get_pw_parameters(),
174                                     self.get_kpoints())
175
176     self.attach_calculation(calc)
177
178     self.next(self.final_step)
179

```



```

180 @Workflow.step
181 def final_step(self):
182
183     from aiida.orm.data.parameter import ParameterData
184
185     x_material = self.get_parameter("x_material")
186     optimal_alat = self.get_attribute("optimal_alat")
187
188     opt_calc = self.get_step_calculations(self.optimize)[0] #.get_calculations()[0]
189     opt_e = opt_calc.get_outputs(type=ParameterData)[0].get_dict()['energy']
190
191     self.append_to_report(x_material+"TiO3 optimal with a="+str(optimal_alat)+", e="+str(opt_e))
192
193     self.add_result("scf_converged", opt_calc)
194
195     self.next(self.exit)

```

Before getting into details, you'll notice that this workflow is divided into sections by comments in the script. This is not necessary, but helps the user to differentiate the main parts of the code. In general it's useful to be able to recognize immediately which functions are steps and which are instead utility or support functions that either generate structure, modify them, add special parameters for the calculations, etc. In this case the support functions are reported first, under the Object generators part, while Workflow steps are reported later in the soundy Workflow steps section. Lets now get in deeper details for each function.

- **\_\_init\_\_** Usual initialization function, notice again the necessary super class initialization for back compatibility.
- **start** The workflow tries to get the X material from the parameters, called in this case `x_material`. If the entry is not present in the dictionary an error will be thrown and the workflow will hang, reporting the error in the report. After that a simple line in the report is added to notify the correct start and the eos step will be chained to the execution.
- **eos** This step is the heart of this workflow. At the beginning parameters needed to investigate the equation of states are retrieved. In this case we chose a very simple structure with only one interesting cell parameter, called `starting_alat`. The code will take this alat as the central point of a linear mesh going from 0.85 alat to 1.15 alat where only a total of `alat_steps` will be generated. This decision is very much problem dependent, and your workflows will certainly need more parameters or more sophisticated meshes to run a satisfactory equation of state analysis, but again this is only a tutorial and the scope is to learn the basic concepts.

After retrieving the parameters, a linear interpolation is generated between the values of interest and for each of these values a calculation is generated by the support function (see later). Each calculation is then attached to the step and finally the step chains `optimize` as the step. As told, the manager will handle all the job execution and retrieval for all the step's calculation before calling the next step, and this ensures that no optimization will be done before all the alat steps are computed with success.

- **optimize** In the first lines the step will retrieve the initial parameters, the `a_sweep` attribute computed in the previous step and all the calculations launched and successfully retrieved. Energy and volume in each calculation is retrieved thanks to the output parser functions mentioned in the other chapters, and a simple message is added to the report for each calculation.

Having the volume and the energy for each simulation we can run a Murnaghan fit to obtain the optimal cell parameter and expected energy, to do this we use a simple fitting function `Murnaghan_fit` defined at the bottom of the workflow file `wf_XTiO3.py`. The optimal alat is then saved in the attributes and a new calculation is generated for it. The calculation is attached to the step and the `final_step` is attached to the execution.

- **final\_step** In this step the main result is collected and stored. Parameters and attributes are retrieved, a new entry in the report is stored pointing to the optimal alat and to the final energy of the structure. Finally the calculation is added to the workflow results and the `exit` step is chained for execution.
- **get\_pw\_calculation (get\_kpoints, get\_pw\_parameters, get\_structure)** As you noticed to let the code clean

all the functions needed to generate AiiDA Calculation objects have been factored in the utility functions. These functions are highly specific for the task needed, and unrelated to the workflow functions. Nevertheless they're a good example of best practise on how to write clean and reusable workflows, and we'll comment the most important feature.

`get_pw_calculation` is called in the workflow's steps, and it handles the entire Calculation object creation. First it extracts the parameters from the workflow initialization necessary for the execution (the machine, the code, and the number of core, pseudos, etc..) and then it generates and stores the JobCalculation objects, returning it for later use.

`get_kpoints` generates a k-point mesh suitable for the calculation, in this case a fixed MP mesh  $4 \times 4 \times 4$ . In a real case scenario this needs much more sophisticated calculations to ensure a correct convergence, not necessary for the tutorial.

`get_pw_parameters` builds the minimum set of parameters necessary to run the Quantum Espresso simulations. In this case as well parameters are not for production.

`get_structure` generates the real atomic arrangement for the specific calculation. In this case the configuration is extremely simple, but in principle this can be substituted with an external function, implementing even very sophisticated approaches such as genetic algorithm evolution or semi-randomic modifications, or any other structure evolution function the user wants to test.

As you noticed this workflow needs several parameters to be correctly executed, something natural for real case scenarios. Nevertheless the launching procedure is identical as for the simple example before, with just a little longer dictionary of parameters:

```
>> from aiida.workflows.wf_XTiO3 import WorkflowXTiO3_EOS
>> params = {'pw_codename':'PWcode', 'num_machines':1, 'num_mpiprocs_per_machine':8, 'max_wallclock_time':10000}
>> wf = WorkflowXTiO3_EOS(params=params)
>> wf.start()
```

To run this workflow remember to update the `params` dictionary with the correct values for your AiiDA installation (namely `pw_codename` and `pseudo_family`).

## Chaining workflows

After the previous chapter we're now able to write a real case workflow that runs in a fully automatic way EOS analysis for simple structures. This covers almost all the workflow engine's features implemented in AiiDA, except for workflow chaining.

Thanks to their modular structure a user can write task-specific workflows very easily. An example is the EOS before, or an energy convergence procedure to find optimal cutoffs, or any other necessity the user can code. These self contained workflows can easily become a library of result-oriented scripts that a user would be happy to reuse in several ways. This is exactly where sub-workflows come in handy.

Workflows, in an abstract sense, are in fact calculations, that accept as input some parameters and that produce results as output. The way this calculations are handled is completely transparent for the user and the engine, and if a workflow could launch other workflows it would just be a natural extension of the step's calculation concept. This is in fact how workflow chaining has been implemented in AiiDA. Just as with calculations, in each step a workflow can attach another workflow for executions, and the AiiDA daemon will handle its execution waiting for its successful end (in case of errors in any subworkflow, such errors will be reported and the entire workflow tree will be halted, exactly as when a calculation fails).

To introduce this function we analyze our last example, where the `WorkflowXTiO3_EOS` is used as a sub workflow. The general idea of this new workflow is simple: if we're now able to compute the EOS of any XTiO3 structure we can build a workflow to loop among several X materials, obtain the relaxed structure for each material and run some more sophisticated calculation. In this case we'll compute phonon vibrational frequencies for some XTiO3 materials, namely Ba, Sr and Pb.

```

1  ## =====
2  ##      WorkflowXTiO3
3  ## =====
4
5  class WorkflowXTiO3(Workflow):
6
7      def __init__(self, **kwargs):
8
9          super(WorkflowXTiO3, self).__init__(**kwargs)
10
11      ## =====
12      ##      Calculations generators
13      ## =====
14
15      def get_ph_parameters(self):
16
17          parameters = ParameterData(dict={
18              'INPUTPH': {
19                  'tr2_ph' : 1.0e-8,
20                  'epsil' : True,
21                  'ldisp' : True,
22                  'nq1' : 1,
23                  'nq2' : 1,
24                  'nq3' : 1,
25              })
26
27          return parameters
28
29      def get_ph_calculation(self, pw_calc, ph_parameters):
30
31          params = self.get_parameters()
32
33          ph_codename          = params['ph_codename']
34          num_machines         = params['num_machines']
35          num_mpiprocs_per_machine = params['num_mpiprocs_per_machine']
36          max_wallclock_seconds = params['max_wallclock_seconds']
37
38          code = Code.get_from_string(ph_codename)
39          computer = code.get_remote_computer()
40
41          QEPhCalc = CalculationFactory('quantumespresso.ph')
42          calc = QEPhCalc(computer=computer)
43
44          calc.set_max_wallclock_seconds(max_wallclock_seconds) # 30 min
45          calc.set_resources({'num_machines': num_machines, "num_mpiprocs_per_machine": num_mpiprocs_per_machine})
46          calc.store()
47
48          calc.use_parameters(ph_parameters)
49          calc.use_code(code)
50          calc.use_parent_calculation(pw_calc)
51
52          return calc
53
54      ## =====
55      ##      Workflow steps
56      ## =====
57
58      @Workflow.step

```

```

59     def start(self):
60
61         params = self.get_parameters()
62         elements_alat = [('Ba', 4.0), ('Sr', 3.89), ('Pb', 3.9)]
63
64         for x in elements_alat:
65
66             params.update({'x_material': x[0]})
67             params.update({'starting_alat': x[1]})
68
69             aiida_logger.info("Launching workflow WorkflowXTiO3_EOS for {0} with alat {1}".format(x[0], x[1]))
70
71             w = WorkflowXTiO3_EOS(params=params)
72             w.start()
73             self.attach_workflow(w)
74
75         self.next(self.run_ph)
76
77     @Workflow.step
78     def run_ph(self):
79
80         # Get calculations
81         sub_wfs = self.get_step(self.start).get_sub_workflows()
82
83         for sub_wf in sub_wfs:
84
85             # Retrieve the pw optimized calculation
86             pw_calc = sub_wf.get_step("optimize").get_calculations()[0]
87
88             aiida_logger.info("Launching PH for PW {0}".format(pw_calc.get_job_id()))
89             ph_calc = self.get_ph_calculation(pw_calc, self.get_ph_parameters())
90             self.attach_calculation(ph_calc)
91
92         self.next(self.final_step)
93
94     @Workflow.step
95     def final_step(self):
96
97         #self.append_to_report(x_material+"TiO3 EOS started")
98         from aiida.orm.data.parameter import ParameterData
99         import aiida.tools.physics as ps
100
101         params = self.get_parameters()
102
103         # Get calculations
104         run_ph_calcs = self.get_step_calculations(self.run_ph) #.get_calculations()
105
106         for c in run_ph_calcs:
107             dm = c.get_outputs(type=ParameterData)[0].get_dict()['dynamical_matrix_1']
108             self.append_to_report("Point q: {0} Frequencies: {1}".format(dm['q_point'], dm['frequencies']))
109
110         self.next(self.exit)

```

Most of the code is now simple adaptation of previous examples, so we're going to comment only the most relevant differences where workflow chaining plays an important role.

- **start** This workflow accepts the same input as the WorkflowXTiO3\_EOS, but right at the beginning the workflow a list of X materials is defined, with their respective initial alat. This list is iterated and for each material a new

Workflow is both generated, started and attached to the step. At the end `run_ph` is chained as the following step.

- **run\_ph** Only after all the subworkflows in `start` are successfully completed this step will be executed, and it will immediately retrieve all the subworkflow, and from each of them it will get the result calculations. As you noticed the result can be stored with any user defined key, and this is necessary when someone wants to retrieve it from a completed workflow. For each result a phonon calculation is launched and then the `final_step` step is chained.

To launch this new workflow we have only to add a simple entry in the previous parameter dictionary, specifying the phonon code, as reported here:

```
>> from aiida.workflows.wf_XTiO3 import WorkflowXTiO3
>> params = {'pw_codename':'PWcode', 'ph_codename':'PHcode', 'num_machines':1, 'num_mpirprocs_per_mach':1}
>> wf = WorkflowXTiO3(params=params)
>> wf.start()
```

### 1.1.25 Import structures from external databases

We support the import of structures from external databases. The base class that defines the API for the importers can be found here: [DbImporter](#). Below, you can find a list of existing plugins that have already been implemented.

#### Available plugins

##### ICSD database importer

In this section we explain how to import CIF files from the ICSD database using the [IcsdDbImporter](#) class.

Before being able to query ICSD, provided by FIZ Karlsruhe, you should have the intranet database installed on a server ([http://www.fiz-karlsruhe.de/icsd\\_intranet.html](http://www.fiz-karlsruhe.de/icsd_intranet.html)). Follow the installation as described in the manual.

It is necessary to know the webpage of the icsd web interface and have access to the full database from the local machine.

You can either query the mysql database or the web page, the latter is restricted to a maximum of 1000 search results, which makes it unsuitable for data mining. So better set up the mysql connection.

**Setup** An instance of the [IcsdDbImporter](#) can be created as follows:

```
importer = aiida.tools.dbimporters.plugins.icsd.IcsdDbImporter(server="http://ICSDSERVER.com/", host=)
```

Here is a list of the most important input parameters with an explanation.

For both connection types (web and SQL):

- **server:** address of web interface of the icsd database; it should contain both the protocol and the domain name and end with a slash; example:

```
server = "http://ICSDSERVER.com/"
```

The following parameters are required only for the mysql query:

- **host:** database host name address.

**Tip:** If the database is not hosted on your local machine, it can be useful to create an ssh tunnel to the 3306 port of the database host:

```
ssh -L 3306:localhost:3306 username@icsddbhostname.com
```

If you get an `URLLError` with `Errno 111` (Connection refused) when you query the database, try to use instead:

```
ssh -L 3306:localhost:3306 -L 8010:localhost:80 username@icsddbhostname.com
```

The database can then be accessed using “127.0.0.1” as host:

```
host = "127.0.0.1"
```

- **user / pass\_wd / db / port:** Login username, password, name of database and port of your mysql database. If the standard installation of ICSD intranet version has been followed, the default values should work. Otherwise contact your system administrator to get the required information:

```
user = "dba", pass_wd = "sql", db = "icsd", port = 3306
```

Other settings:

- **querydb:** If True (default) the mysql database is queried, otherwise the web page is queried.

A more detailed documentation and additional settings are found under *IcsdDbImporter*.

**How to do a query** If the setup worked, you can do your first query:

```
cif_nr_list = ["50542", "617290", "35538"]
queryresults = importer.query(id= cif_nr_list)
```

All supported keywords can be obtained using:

```
importer.get_supported_keywords()
```

More information on the keywords are found under [http://www.fiz-karlsruhe.de/fileadmin/be\\_user/ICSD/PDF/sci\\_man\\_ICSD\\_v1.pdf](http://www.fiz-karlsruhe.de/fileadmin/be_user/ICSD/PDF/sci_man_ICSD_v1.pdf)

A query returns an instance of *IcsdSearchResults*

The *IcsdEntry* at position *i* can be accessed using:

```
queryresults.at(i)
```

You can also iterate through all query results:

```
for entry in query_results:
    do something
```

Instances of *IcsdEntry* have following methods:

- **get\_cif\_node():** Return an instance of *CifData*, which can be used in an AiiDA workflow.
- **get\_aiida\_structure():** Return an AiiDA structure
- **get\_ase\_structure():** Return an ASE structure

The most convenient format can be chosen for further processing.

**Full example** Here is a full example how the icsd importer can be used:

```
import aaida.tools.dbimporters.plugins.icsd

cif_nr_list = [
    "50542",
    "617290",
    "35538 ",
    "165226",
    "158366"
]

importer = aaida.tools.dbimporters.plugins.icsd.IcsdDbImporter(server="http://ICSDSERVER.com/",
    host= "127.0.0.1")
query_results = importer.query(id=cif_nr_list)
for result in query_results:
    print result.source['db_id']
    aaida_structure = result.get_aiida_structure()
    #do something with the structure
```

**Troubleshooting: Testing the mysql connection** To test your mysql connection, first make sure that you can connect to the 3306 port of the machine hosting the database. If the database is not hosted by your local machine, use the local port tunneling provided by ssh, as follows:

```
ssh -L 3306:localhost:3306 username@icsddbhostname.com
```

**Note:** If you get an `URLError` with `Errno 111 (Connection refused)` when you query the database, try to use instead:

```
ssh -L 3306:localhost:3306 -L 8010:localhost:80 username@icsddbhostname.com
```

**Note:** You need an account on the host machine.

**Note:** There are plenty of explanations online explaining how to setup an tunnel over a SSH connection using the `-L` option, just google for it in case you need more information.

Then open a new `verdi shell` and type:

```
import MySQLdb

db = MySQLdb.connect(host = "127.0.0.1", user = "dba", passwd = "sql", db = "icsd", port=3306)
```

If you do not get an error and it does not hang, you have successfully established your connection to the mysql database.

### COD database importer

COD database importer is used to import crystal structures from the [Crystallography Open Database \(COD\)](#) to AiiDA.

**Setup** An instance of `CodDbImporter` is created as follows:

```
from aaida.tools.dbimporters.plugins.cod import CodDbImporter
importer = CodDbImporter()
```

No additional parameters are required for standard queries on the main COD server.

**How to do a query** A search is initiated by supplying query statements using keyword=value syntax:

```
results = importer.query(chemical_name="caffeine")
```

List of possible keywords can be listed using:

```
importer.get_supported_keywords()
```

Values for the most of the keywords can be list. In that case the query will return entries, that match any of the values (binary *OR*) from the list. Moreover, in the case of multiple keywords, entries, that match all the conditions imposed by the keywords, will be returned (binary *AND*).

Example:

```
results = importer.query(chemical_name=["caffeine", "serotonin"],
                          year=[2000, 2001])
```

is equivalent to the following SQL statement:

```
results = SELECT * FROM data WHERE
          ( chemical_name == "caffeine" OR chemical_name == "serotonin" ) AND
          ( year = 2000 OR year = 2001 )
```

A query returns an instance of *CodSearchResults*, which can be used in a same way as a list of *CodEntry* instances:

```
print len(results)

for entry in results:
    print entry
```

**Using data from CodEntry** *CodEntry* has a few functions to access the contents of it's instances:

```
CodEntry.get_aiida_structure()
CodEntry.get_ase_structure()
CodEntry.get_cif_node()
CodEntry.get_parsed_cif()
CodEntry.get_raw_cif()
```

## 1.1.26 Export data to external databases

We support the export of data to external databases. In the most general way, the export to external databases can be viewed as a subworkflow, taking data as input and resulting in the deposition of it to external database(s). Below is a list of supported databases with deposition routines described in *comments-type* style.

### Supported databases

#### TCOD database exporter

TCOD database exporter is used to export computation results of *StructureData*, *CifData* and *TrajectoryData* (or any other data type, which can be converted to them) to the Theoretical Crystallography Open Database (TCOD).

**Setup** To be able to export data to TCOD, one has to *install dependencies for CIF manipulation* as well as *cod-tools package*, and set up an AiiDA *Code* for *cif\_cod\_deposit* script from **cod-tools**.



**How to deposit a structure** Best way to deposit data is to use the command line interface:

```
verdi DATATYPE structure deposit tcod [--type {published,prepublication,personal}]
                                     [--username USERNAME] [--password]
                                     [--user-email USER_EMAIL] [--title TITLE]
                                     [--author-name AUTHOR_NAME]
                                     [--author-email AUTHOR_EMAIL] [--url URL]
                                     [--code CODE_LABEL]
                                     [--computer COMPUTER_NAME]
                                     [--replace REPLACE] [-m MESSAGE]
                                     [--reduce-symmetry] [--no-reduce-symmetry]
                                     [--parameter-data PARAMETER_DATA]
                                     [--dump-aiida-database]
                                     [--no-dump-aiida-database]
                                     [--exclude-external-contents]
                                     [--no-exclude-external-contents] [--gzip]
                                     [--no-gzip]
                                     [--gzip-threshold GZIP_THRESHOLD]
                                     PK
```

Where:

- DATATYPE – one of AiiDA structural data types (at the moment of writing, they were *StructureData*, *CifData* and *TrajectoryData*);
- TITLE – the title of the publication, where the exported data is/will be published; in case of personal communication, the title should be chosen so as to reflect the exported dataset the best;
- CODE\_LABEL – label of AiiDA *Code*, associated with *cif\_cod\_deposit*;
- COMPUTER\_NAME – name of AiiDA *Computer*, where *cif\_cod\_deposit* script is to be launched;
- REPLACE – TCOD ID of the replaced entry in the event of redeposition;
- MESSAGE – string to describe changes for redeposited structures;
- --reduce-symmetry, --no-reduce-symmetry – turn on/off symmetry reduction of the exported structure (on by default);
- --parameter-data – specify the PK of *ParameterData* object, describing the result of the final (or single) calculation step of the workflow;
- --dump-aiida-database, --no-dump-aiida-database – turn on/off addition of relevant AiiDA database dump (on by default).

**Warning:** be aware that TCOD is an **open** database, thus **no copyright-protected data should be deposited** unless permission is given by the owner of the rights.

**Note:** data, which is deposited as pre-publication material, **will be kept private on TCOD server** and will not be disclosed to anyone without depositor's permission.

- --exclude-external-contents, --no-exclude-external-contents – exclude contents of initial input files, that contain *source* property with definitions on how to obtain the contents from external resources (on by default);
- --gzip, --no-gzip – turn on/off gzip compression for large files (off by default); --gzip-threshold sets the minimum file size to be compressed.

Other command line options correspond to the options of *cif\_cod\_deposit* of the same name. To ease the use of TCOD exporter, one can define persistent parameters in AiiDA *properties*. Corresponding command line parameters and AiiDA properties are presented in the table:

Command line parameter	AiiDA property
<code>--author-email</code>	<code>tcod.depositor_author_email</code>
<code>--author-name</code>	<code>tcod.depositor_author_name</code>
<code>--user-email</code>	<code>tcod.depositor_email</code>
<code>--username</code>	<code>tcod.depositor_password</code>
<code>--password</code>	<code>tcod.depositor_username</code>

---

**Note:** `--password` does not accept any value; instead, the option will prompt the user to enter one's password in the terminal.

---

---

**Note:** command line parameters can be used to override AiiDA properties even if properties are set.

---

**Return values** The deposition process, which is of *JobCalculation* type, returns the output of `cif_cod_deposit`, wrapped in *ParameterData*.

---

## Other guide resources

---

### 2.1 Other guide resources

#### 2.1.1 AiiDA cookbook (useful code snippets)

This cookbook is intended to be a collection of useful short scripts and code snippets that may be useful in the everyday usage of AiiDA. Please read carefully the nodes (if any) before running the scripts!

##### Deletion of nodes

At the moment, we do not support natively the deletion of nodes. This is mainly because it is very dangerous to delete data, as this is cannot be undone.

If you really feel the need to delete some code, you can use the function below.

---

**Note: WARNING!** In order to preserve the provenance, this function will delete not only the list of specified nodes, but also all the children nodes! So please be sure to double check what is going to be deleted before running this function.

---

Here is the function, pass a list of PKs as parameter to delete those nodes and all the children nodes:

```
def delete_nodes(pks_to_delete):
    """
    Delete a set of nodes.

    :note: The script will also delete
    all children calculations generated from the specified nodes.

    :param pks_to_delete: a list of the PKs of the nodes to delete
    """
    from django.db import transaction
    from django.db.models import Q
    from aiida.djsite.db import models
    from aiida.orm import load_node

    # Delete also all children of the given calculations
    # Here I get a set of all pks to actually delete, including
    # all children nodes.
    all_pks_to_delete = set(pks_to_delete)
    for pk in pks_to_delete:
        all_pks_to_delete.update(models.DbNode.objects.filter(
```

```
parents__in=pks_to_delete).values_list('pk', flat=True))

print "I am going to delete {} nodes, including ALL THE CHILDREN".format(
    len(all_pks_to_delete))
print "of the nodes you specified. Do you want to continue? [y/N]"
answer = raw_input()

if answer.strip().lower() == 'y':
    # Recover the list of folders to delete before actually deleting
    # the nodes. I will delete the folders only later, so that if
    # there is a problem during the deletion of the nodes in
    # the DB, I don't delete the folders
    folders = [load_node(pk).folder for pk in all_pks_to_delete]

    with transaction.atomic():
        # Delete all links pointing to or from a given node
        models.DbLink.objects.filter(
            Q(input__in=all_pks_to_delete) |
            Q(output__in=all_pks_to_delete)).delete()
        # now delete nodes
        models.DbNode.objects.filter(pk__in=all_pks_to_delete).delete()

    # If we are here, we managed to delete the entries from the DB.
    # I can now delete the folders
    for f in folders:
        f.erase()
```

## 2.1.2 Troubleshooting and tricks

### Some tricks

#### Using the proxy\_command option with ssh

This page explains how to use the `proxy_command` feature of `ssh`. This feature is needed when you want to connect to a computer B, but you are not allowed to connect directly to it; instead, you have to connect to computer A first, and then perform a further connection from A to B.

**Requirements** The idea is that you ask `ssh` to connect to computer B by using a proxy to create a sort of tunnel. One way to perform such an operation is to use `netcat`, a tool that simply takes the standard input and redirects it to a given TCP port.

Therefore, a requirement is to install `netcat` on computer A. You can already check if the `netcat` or `nc` command is available on your computer, since some distributions include it (if it is already installed, the output of the command:

```
which netcat
```

or:

```
which nc
```

will return the absolute path to the executable).

If this is not the case, you will need to install it on your own. Typically, it will be sufficient to look for a `netcat` distribution on the web, unzip the downloaded package, `cd` into the folder and execute something like:

```
./configure --prefix=.
make
make install
```

This usually creates a subfolder `bin`, containing the `netcat` and `nc` executables. Write down the full path to `nc` that we will need later.

**ssh/config** You can now test the proxy command with `ssh`. Edit the `~/.ssh/config` file on the computer on which you installed AiiDA (or create it if missing) and add the following lines:

```
Host FULLHOSTNAME_B
Hostname FULLHOSTNAME_B
User USER_B
ProxyCommand ssh USER_A@FULLHOSTNAME_A ABSPATH_NETCAT %h %p
```

where you have to replace:

- `FULLHOSTNAME_A` and `FULLHOSTNAME_B` with the fully-qualified hostnames of computer A and B (remembering that B is the computer you want to actually connect to, and A is the intermediate computer to which you have direct access)
- `USER_A` and `USER_B` are the usernames on the two machines (that can possibly be the same).
- `ABSPATH_PATH_NETCAT` is the absolute path to the `nc` executable that you obtained in the previous step.

Remember also to configure passwordless `ssh` connections using `ssh` keys both from your computer to A, and from A to B.

Once you add this lines and save the file, try to execute:

```
ssh FULLHOSTNAME_B
```

which should allow you to directly connect to B.

**WARNING** There are several versions of `netcat` available on the web. We found at least one case in which the executable wasn't working properly. At the end of the connection, the `netcat` executable might still be running: as a result, you may rapidly leave the cluster with hundreds of opened `ssh` connections, one for every time you connect to the cluster B. Therefore, check on both computers A and B that the number of processes `netcat` and `ssh` are disappearing if you close the connection. To check if such processes are running, you can execute:

```
ps -aux | grep <username>
```

Remember that a cluster might have more than one login node, and the `ssh` connection will randomly connect to any of them.

**AiiDA config** If the above steps work, setup and configure now the computer as explained [here](#).

If you properly set up the `~/.ssh/config` file in the previous step, AiiDA should properly parse the information in the file and provide the correct default value for the `proxy_command` during the `verdi computer configure` step.

### Some notes on the `proxy_command` option

- In the `~/.ssh/config` file, you can leave the `%h` and `%p` placeholders, that are then automatically replaced by `ssh` with the hostname and the port of the machine B when creating the proxy. However, in the AiiDA `proxy_command` option, you need to put the actual hostname and port. If you start from a properly configured

`~/.ssh/config` file, AiiDA will already replace these placeholders with the correct values. However, if you input the `proxy_command` value manually, remember to write the hostname and the port and not `%h` and `%p`.

- In the `~/.ssh/config` file, you can also insert stdout and stderr redirection, e.g. `2> /dev/null` to hide any error that may occur during the proxying/tunneling. However, you should only give AiiDA the actual command to be executed, without any redirection. Again, AiiDA will remove the redirection when it automatically reads the `~/.ssh/config` file, but be careful if entering manually the content in this field.

## Connection problems

- **When AiiDA tries to connect to the remote computer, it says** `paramiko.SSHException: Server u'FULLHOSTNAME' not found in known_hosts`

AiiDA uses the `paramiko` library to establish SSH connections. `paramiko` is able to read the remote host keys from the `~/.ssh/known_hosts` of the user under which the AiiDA daemon is running. You therefore have to make sure that the key of the remote host is stored in the file.

- As a first check, login as the user under which the AiiDA daemon is running and run a:

```
ssh FULLHOSTNAME
```

command, where `FULLHOSTNAME` is the complete host name of the remote computer configured in AiiDA. If the key of the remote host is not in the `known_hosts` file, SSH will ask confirmation and then add it to the file.

- If the above point is not sufficient, check the format of the remote host key. On some machines (we know that this issue happens at least on recent Ubuntu distributions) the default format is not RSA but ECDSA. However, `paramiko` is still not able to read keys written in this format.

To discover the format, run the following command:

```
ssh-keygen -F FULLHOSTNAME
```

that will print the remote host key. If the output contains the string `ecdsa-sha2-nistp256`, then `paramiko` will not be able to use this key (see below for a solution). If instead `ssh-rsa`, the key should be OK and `paramiko` will be able to use it.

In case your key is in `ecdsa` format, you have to first delete the key by using the command:

```
ssh-keygen -R FULLHOSTNAME
```

Then, in your `~/.ssh/config` file (create it if it does not exist) add the following lines:

```
Host *  
    HostKeyAlgorithms ssh-rsa
```

(use the same indentation, and leave an empty line before and one after). This will set the RSA algorithm as the default one for all remote hosts. In case, you can set the `HostKeyAlgorithms` attribute only to the relevant computers (use `man ssh_config` for more information).

Then, run a:

```
ssh FULLHOSTNAME
```

command. SSH will ask confirmation and then add it to the file, but this time it should use the `ssh-rsa` format (it will say so in the prompt message). You can also double-check that the host key was correctly inserted using the `ssh-keygen -F FULLHOSTNAME` command as described above. Now, the error message should not appear anymore.

## Increasing the debug level

By default, the logging level of AiiDA is minimal to avoid filling logfiles. Only warnings and errors are logged (to the `~/.aiida/daemon/log/aiida_daemon.log` file), while info and debug messages are discarded.

If you are experiencing a problem, you can change the default minimum logging level of AiiDA messages (and celery messages – celery is the library that we use to manage the daemon process) using, on the command line, the two following commands:

```
verdi devel setproperty logging.celery_loglevel DEBUG
verdi devel setproperty logging.aiida_loglevel DEBUG
```

After rebooting the daemon (`verdi daemon restart`), the number of messages logged will increase significantly and may help in understanding the source of the problem.

**Note:** In the command above, you can use a different level than `DEBUG`. The list of the levels and their order is the same of the [standard python logging module](#).

**Note:** When the problem is solved, we suggest to bring back the default logging level, using the two commands:

```
verdi devel delproperty logging.celery_loglevel
verdi devel delproperty logging.aiida_loglevel
```

to avoid to fill the logfiles.

## Tips to ease the life of the hard drive (for large databases)

Those tips are useful when your database is very large, i.e. several hundreds of thousands of nodes and workflows or more. With such large databases the hard drive may be constantly working and the computer slowed down a lot. Below are some solutions to take care of the most typical reasons.

### Repository backup

The backup of the repository takes an extensively long time if it is done through a standard rsync or backup software, since it contains as many folders as the number of nodes plus the number of workflows (and each folder can contain many files!). A solution is to use instead the incremental backup described in the [repository backup section](#).

### mlocate cron job

Under typical Linux distributions, there is a cron job (called `updatedb.mlocate`) running every day to update a database of files and folders – this is to be used by the `locate` command. This might become problematic since the repository contains many folders and will be scanned everyday. The net effect is a hard drive almost constantly working.

To avoid this issue, edit as root the file `/etc/updatedb.conf` and put in `PRUNEPATHS` the name of the repository folder.

## 2.1.3 Using AiiDA in multi-user mode

**Note:** multi-user mode is still not fully supported, and the way it works will change significantly soon. Do not use unless you know what you are doing.

---

## Todo

To be documented.

Discuss:

- Security issues
- Under which linux user (aiida) to run, and remove the pwd with `passwd -d aiida`.
- How to setup each user (`aiida@localhost` for the daemon user, correct email for the others using `verdi install --only-config`)
- How to configure a given user (verdi user configure)
- How to list users (also the `-color` option, and the meaning of colors)
- How to setup the daemon user (verdi daemon configureuser)
- How to start the daemon
- How to configure the permissions! (all AiiDA in the same group, and set the `'chmod -R g+s'` flag to all folders and subfolders of the AiiDA repository) (comment that by default now we have a flag (hardcoded to True) in `aiida.common.folders` to give write permissions to the group both to files and folders created using the `Folder` class.
- Some configuration example:

```
{u'compress': True,
 u'key_filename': u'/home/aiida/.aiida/sshkeys/KEYFILE',
 u'key_policy': u'RejectPolicy',
 u'load_system_host_keys': True,
 u'port': 22,
 u'proxy_command': u'ssh -i /home/aiida/.aiida/sshkeys/KEYFILE USERNAME@MIDDLECOMPUTER /bin/nc F
 u'timeout': 60,
 u'username': u'xxx'}
```

- Moreover, on the remote computer do:

```
ssh-keyscan FINALCOMPUTER
```

and append the output to the `known_hosts` of the aiida daemon account. Do the same also for the MIDDLE-COMPUTER if a `proxy_command` is user.

- 

---

## 2.1.4 Deploying AiiDA using Apache

---

**Note:** At this stage, this section is meant for developers only.

---

## Todo

To be documented.

Some notes:

- Configure your default site of Apache (check in `/etc/apache2/sites-enabled`, probably it is called `000-default`).



Add the full ServerName outside of any <VirtualHost> section:

```
ServerName FULLSERVERNAMEHERE
```

and inside the VirtualHost that provide access, specify the email of the server administrator (note that the email will be accessible, e.g. it is shown if a INTERNAL ERROR 500 page is shown):

```
<VirtualHost *:80>
    ServerAdmin administratoremail@xxx.xx

    # [...]

</VirtualHost>
```

- Login as the user running apache, e.g. www-data in Ubuntu; use something like:

```
sudo su www-data -s /bin/bash

and run ``verdi install`` to configure where the DB and the files stay, etc.

Be also sure to check that this apache user belongs to the group that has
read/write permissions to the AiiDA repository.
```

- If you home directory is set to /var/www, and this is published by Apache, double check that nobody can access the .aiida subfolder! By default, during verdi install AiiDA puts inside the folder a .htaccess file to disallow access, but this file is not read by some default Apache configurations.

To have Apache honor the .htaccess file, in the default Apache site (probably the same file as above) you need to set the AllowOverride all flag in the proper VirtualHost and Directory (note that there can be more than one, e.g. if you have both HTTP and HTTPS).

You should have something like:

```
<VirtualHost *:80>
    ServerAdmin xxx@xxx.xx

    DocumentRoot /var/www
    <Directory /var/www/>
        AllowOverride all
    </Directory>
</VirtualHost>
```

**Note:** Of course, you will typically have other configurations as well, the snippet above just shows where the AllowOverride all line should appear.

Double check if you cannot list/read the files (e.g. connecting to http://YOURSERVER/.aiida).

### Todo

Allow to have a trick to have only one file in .aiida, containing the url where the actual configuration stuff resides (or some other trick to physically move the configuration files out of /var/www).

- Create a /etc/apache2/sites-available/wsgi-aiida file, with content:

```
Alias /static/awi /PATH_TO_AIIDA/aiida/djsite/awi/static/awi/
Alias /favicon.ico /PATH_TO_AIIDA/aiida/djsite/awi/static/favicon.ico

WSGIScriptAlias / /PATH_TO_AIIDA/aiida/djsite/settings/wsgi.py
WSGIProcessGroup On
```

```
WSGI PythonPath /PATH_TO_AIIDA/

<Directory /PATH_TO_AIIDA/aiana/djsite/settings>
<Files wsgi.py>
Order deny,allow
Allow from all
## For Apache >= 2.4, replace the two lines above with the one below:
# Require all granted
</Files>
</Directory>
```

---

**Note:** Replace everywhere `PATH_TO_AIIDA` with the full path to the AiiDA source code. Check that the user running the Apache daemon can read/access all files in that folder and subfolders.

---

---

**Note:** in the `WSGI PythonPath` you can also add other folders that should be in the Python path (e.g. if you use other libraries that should be accessible). The different paths must be separated with `:`.

---

---

**Note:** For Apache `>= 2.4`, replace the two lines:

```
Order deny,allow
Allow from all
```

with:

```
Require all granted
```

---

**Note:** The `WSGIScriptAlias` exposes AiiDA under main address of your website (`http://SERVER/`).

If you want to serve AiiDA under a subfolder, e.g. `http://SERVER/aiana`, then change the line containing `WSGIScriptAlias` with:

```
WSGIScriptAlias /aiana /PATH_TO_AIIDA/aiana/djsite/settings/wsgi.py
```

**without any trailing slashes after `/aiana`.**

---

- Enable the given site:

```
sudo a2ensite wsgi-aiana
```

and reload the Apache configuration to load the new site:

```
sudo /etc/init.d/apache2 reload
```

- A comment on permissions (to be improved): the default Django Authorization (used e.g. in the API) does not allow a “standard” user to modify data in the DB, but only to read it, therefore if you are accessing with a user that is not a superuser, all API calls trying to modify the DB will return an HTTP UNAUTHORIZED message.

Temporarily, you can fix this by going in a `verdi shell`, loading your user with something like:

```
u = models.DbUser.objects.get(email='xxx')
```

and then upgrading the user to a superuser:

```
u.is_superuser = True
u.save()
```

### 2.1.5 AiiDA Website

To run the server:

```
verdi runserver
```

For more info:

```
verdi runserver --help
```

Anyway the options are those of Django at <https://docs.djangoproject.com/en/1.5/ref/django-admin/#runserver-port-or-address-port>



---

## Developer's guide

---

### 3.1 Developer's guide

#### 3.1.1 Developer's Guide For AiiDA

##### Python style

When writing python code, a more than reasonable guideline is given by the Google python styleguide <http://google-styleguide.googlecode.com/svn/trunk/pyguide.html>. The documentation should be written consistently in the style of sphinx.

And more generally, write verbose! Will you remember after a month why you had to write that check on that line? (Hint: no) Write comments!

##### Pylint

You can check your code style and other important code errors by using Pylint. Once installed you can run Pylint from the root source directory on the code using the command:

```
pylint aiiida
```

The most important part is the summary under the Messages table near the end.

##### Version number

The AiiDA version number is stored in both `aiida/__init__.py` and `setup.py`. Make sure to update both when changing version number.

##### Inline calculations

If an operation is extremely fast to be run, this can be done directly in Python, without being submitted to a cluster. However, this operation takes one (or more) input data nodes, and creates new data nodes, the operation itself is not recorded in the database, and provenance is lost. In order to put a Calculation object inbetween, we define the *InlineCalculation* class, that is used as the class for these calculations that are run “in-line”.

We also provide a wrapper (that also works as a decorator of a function), *make\_inline()*. This can be used to wrap suitably defined function, so that after their execution, a node representing their execution is stored in the DB, and suitable input and output nodes are also stored.

---

**Note:** See the documentation of this function for further documentation of how it should be used, and of the requirements for the wrapped function.

---

## Database schema

The Django database schema can be found in `aiida.djsite.db.models`.

If you need to change the database schema follow these steps:

1. Make all the necessary changes to `aiida.djsite.db.models`
2. Create a new migration file. From `aiida/djsite`, run:

```
python manage.py makemigrations
```

This will create the migration file in `aiida/djsite/db/migrations` whose name begins with a number followed by some description. If the description is not appropriate then change to it to something better but retain the number.

3. Open the generated file and make the following changes:

```
from aiida.djsite.db.migrations import update_schema_version
...
SCHEMA_VERSION = # choose an appropriate version number
                  # (hint: higher than the last migration!)
...
class Migration(migrations.Migration):
    ...
    operations = [
        ..
        update_schema_version(SCHEMA_VERSION)
    ]
```

5. Change the `LATEST_MIGRATION` variable in `aiida/djsite/db/migrations/__init__.py` to the name of your migration file:

```
LATEST_MIGRATION = '0003_my_db_update'
```

This let's AiiDA get the version number from your migration and check sure the database and the code are in sync.

6. Migrate your database to the new version, (again from `aiida/djsite`), run:

```
python manage.py migrate
```

## Commits and GIT usage

In order to have an efficient management of the project development, we chose to adopt the guidelines for the branching model described [here](#). In particular:

- The main branch in which one should work is called `develop`
- The `master` branch is reserved for releases: every commit there implies a new release. Therefore, one should never commit directly there (except once per every release).
- New releases should also be tagged.
- Any new modification requiring just one commit can be done in `develop`

- mid-to-long development efforts should be done in a branch, branching off from develop (e.g. a long bugfix, or a new feature)
- while working on the branch, often merge the develop branch back into it (if you also have a remote branch and there are no conflicts, that can be done with one click from the BitBucket web interface, and then you just do a local ‘git pull’)
- remember to fix generic bugs in the `develop` (or in a branch to be then merged in the develop), *not in your local branch* (except if the bug is present only in the branch); only then merge `develop` back into your branch. In particular, if it is a complex bugfix, better to have a branch because it allows to backport the fix also in old releases, if we want to support multiple versions
- only when a feature is ready, merge it back into `develop`. If it is a big change, better to instead do a *pull request* on BitBucket instead of directly merging and wait for another (or a few other) developers to accept it beforehand, to be sure it does not break anything.

For a cheatsheet of git commands, see [here](#).

**Note:** Before committing, **always** run:

```
verdi devel tests
```

to be sure that your modifications did not introduce any new bugs in existing code. Remember to do it even if you believe your modification to be small - the tests run pretty fast!

## Tests

### Running the tests

To run the tests, use the:

```
verdi devel tests
```

command. You can add a list of tests after the command to run only a selected portion of tests (e.g. while developing, if you discover that only a few tests fail). Use TAB completion to get the full list of tests. For instance, to run only the tests for transport and the generic tests on the database, run:

```
verdi devel tests aiiida.transport db.generic
```

### The test-first approach

Remember in best codes actually the tests are written even before writing the actual code, because this helps in having a clear API.

For any new feature that you add/modify, write a test for it! This is extremely important to have the project last and be as bug-proof as possible. Even more importantly, add a test that fails when you find a new bug, and then solve the bug to make the test work again, so that in the future the bug is not introduced anymore.

Remember to make unit tests as atomic as possible, and to document them so that other developers can understand why you wrote that test, in case it should fail after some modification.

### Creating a new test

There are three types of tests:

1. Tests that do not require the usage of the database (testing the creation of paths in k-space, the functionality of a transport plugin, ...)
2. Tests that require the database, but do not require submission (e.g. verifying that node attributes can be correctly queried, that the transitive closure table is correctly generated, ...)
3. Tests that require the submission of jobs

For each of the above types of tests, a different testing approach is followed (you can also see existing tests as guidelines of how tests are written):

1. Tests are written inside the package that one wants to test, creating a `test_MODULENAME.py` file. For each group of tests, create a new subclass of `unittest.TestCase`, and then create the tests as methods using the `unittests` module. Tests inside a selected number of AiiDA packages are automatically discovered when running `verdi devel tests`. To make sure that your test is discovered, verify that its parent module is listed in the `base_allowed_test_folders` property of the `Devel` class, inside `aiida.cmdline.commands.devel`.

For an example of this type of tests, see, e.g., the `aiida.common.test_utils` module.

2. In this case, we use the `testing` functionality of `Django`, adapted to run smoothly with AiiDA.

To create a new group of tests, create a new python file under `aiida.djsite.db.subtests`, and instead of inheriting each class directly from `unittest`, inherit from `aiida.djsite.db.testbase.AiidaTestCase`. In this way:

- (a) The Django testing functionality is used, and a temporary database is used
- (b) every time the class is created to run its tests, default data are added to the database, that would otherwise be empty (in particular, a computer and a user; for more details, see the code of the `AiidaTestCase.setUpClass()` method).
- (c) at the end of all tests of the class, the database is cleaned (nodes, links, ... are deleted) so that the temporary database is ready to run the tests of the following test classes.

---

**Note:** it is *extremely important* that these tests are run from the `verdi devel tests` command line interface. Not only this will ensure that a temporary database is used (via Django), but also that a temporary repository folder is used. Otherwise, you risk to corrupt your database data. (In the codes there are some checks to avoid that these classes are run without the correct environment being prepared by `verdi devel tests`.)

---

Once you create a new file in `aiida.djsite.db.subtests`, you have to add a new entry to the `db_test_list` inside `aiida.djsite.db.testbase` module in order for `verdi devel tests` to find it. In particular, the key should be the name that you want to use on the command line of `verdi devel tests` to run the test, and the value should be the full module name to load. Note that, in `verdi devel tests`, the string `db.` is prepended to the name of each test involving the database. Therefore, if you add a line:

```
db_test_list = {
    ...
    'newtests': 'aiida.djsite.db.subtests.mynewtestsmodule',
    ...
}
```

you will be able to run all all tests inside `aiida.djsite.db.subtests.mynewtestsmodule` with the command:

```
verdi devel tests db.newtests
```

---

**Note:** If in the list of parameters to `verdi devel tests` you add also a `db` parameter, then all database-



related tests will be run, i.e., all tests that start with `db.` (or, if you want, all tests in the `db_test_list` described above).

**Note:** By default, the test database is created using an in-memory SQLite database, which is much faster than creating from scratch a new test database with PostgreSQL or SQLite. However, if you want to test database-specific settings and you want to use the same type of database you are using with AiiDA, set the `tests.use_sqlite` global property to `False`:

```
verdi devel setproperty tests.use_sqlite false
```

3. These tests require an external engine to submit the calculations and then check the results at job completion. We use for this a continuous integration server, and the best approach is to write suitable workflows to run simulations and then verify the results at the end.

**Special tests** Some tests have special routines to ease and simplify the creation of new tests. One case is represented by the tests for transport. In this case, you can define tests for a specific plugin as described above (e.g., see the `aiida.transport.plugins.test_ssh` and `aiida.transport.plugins.test_local` tests). Moreover, there is a `test_all_plugins` module in the same folder. Inside this module, the discovery code is adapted so that each test method defined in that file **and decorated with** `@run_for_all_plugins` is run for *all* available plugins, to avoid to rewrite the same test code more than once and ensure that all plugins behave in the same way (e.g., to copy files, remove folders, etc.).

## Virtual environment

Sometimes it's useful to have a virtual environment that separates out the AiiDA dependencies from the rest of the system. This is especially the case when testing AiiDA against library versions that are different from those installed on the system.

First, install `virtualenv` using `pip`:

```
pip install virtualenv
```

## Basic usage

1. To create a virtual environment in folder `venv`, while in the AiiDA directory type:

```
virtualenv venv
```

This puts a copy of the Python executables and the `pip` library within the `venv` folder hierarchy.

2. Activate the environment with:

```
source venv/bin/activate
```

Your shell should now be prompt should now start with `(venv)`.

3. (optional) Install AiiDA:

```
pip install .
```

4. Deactivate the virtual environment:

```
deactivate
```

### 3.1.2 AiiDA internals

#### Node

The *Node* class is the basic class that represents all the possible objects at the AiiDA world. More precisely it is inherited by many classes including (among others) the *Calculation* class, representing computations that convert data into a different form, the *Code* class representing executables and file collections that are used by calculations and the *Data* class which represents data that can be input or output of calculations.

#### Methods & properties

In the sequel the most important methods and properties of the *Node* class will be described.

**Node subclasses organization** The *Node* class has two important variables:

- `_plugin_type_string` characterizes the class of the object.
- `_query_type_string` characterizes the class and all its subclasses (by pointing to the package or Python file that contain the class).

The convention for all the *Node* subclasses is that if a class *B* is inherited by a class *A* then there should be a package *A* under `aiida/orm` that has a file `__init__.py` and a `B.py` in that directory (or a *B* package with the corresponding `__init__.py`)

An example of this is the *ArrayData* and the *KpointsData*. *ArrayData* is placed in `aiida/orm/data/array/__init__.py` and *KpointsData* which inherits from *ArrayData* is placed in `aiida/orm/data/array/kpoints.py`

This is an implicit & quick way to check the inheritance of the *Node* subclasses.

#### General purpose methods

- `__init__()`: The initialization of the Node class can be done by not providing any attributes or by providing a *DbNode* as initialization. E.g.:

```
dbn = a_dbnode_object
n = Node(dbn=dbn.dbnode)
```

- `ctime()` and `mtime()` provide the creation and the modification time of the node.
- `_is_stored()` informs whether a node is already stored to the database.
- `query()` queries the database by filtering for the results for similar nodes (if the used object is a subclass of *Node*) or with no filtering if it is a *Node* class. Note that for this check `_plugin_type_string` should be properly set.
- `computer()` returns the computer associated to this node.
- `_validate()` does a validation check for the node. This is important for *Node* subclasses where various attributes should be checked for consistency before storing.
- `get_user()` returns the user that created the node.
- `_increment_version_number_db()`: increment the version number of the node on the DB. This happens when adding an attribute or an extra to the node. This method should not be called by the users.
- `copy()` returns a not stored copy of the node with new UUID that can be edited directly.
- `uuid()` returns the universally unique identifier (UUID) of the node.

- `pk()` returns the principal key (ID) of the node.
- `dbnode()` returns the corresponding Django object.
- `get_computer()` & `set_computer()` get and set the computer to be used & is associated to the node.

**Annotation methods** The *Node* can be annotated with labels, description and comments. The following methods can be used for the management of these properties.

*Label management:*

- `label()` returns the label of the node. The setter method can be used for the update of the label.
- `_update_db_label_field()` updates the label in the database. This is used by the setter method of the label.

*Description management:*

- `description()`: the description of the node (more detailed than the label). There is also a setter method.
- `_update_db_description_field()`: update the node description in the database.

*Comment management:*

- `add_comment()` adds a comment.
- `get_comments()` returns a sorted list of the comments.
- `_get_dbcomments()` is similar to `get_comments()`, just the sorting changes.
- `_update_comment()` updates the node comment. It can be done by `verdi comment update`.
- `_remove_comment()` removes the node comment. It can be done by `verdi comment remove`.

**Link management methods** *Node* objects and objects of its subclasses can have ancestors and descendants. These are connected with links. The following methods exist for the processing & management of these links.

- `_has_cached_links()` shows if there are cached links to other nodes.
- `_add_link_from()` adds a link to the current node from the ‘src’ node with the given label. Depending on whether the nodes are stored or node, the linked are written to the database or to the cache.
- `_add_cachelink_from()` adds a link to the cache.
- `_replace_link_from()` replaces or creates an input link.
- `_remove_link_from()` removes an input link that is stored in the database.
- `_replace_dblink_from()` is similar to `_replace_link_from()` but works directly on the database.
- `_remove_dblink_from()` is similar to `_remove_link_from()` but works directly on the database.
- `_add_dblink_from()` adds a link to the current node from the given ‘src’ node. It acts directly on the database.

*Listing links example*

Assume that the user wants to see the available links of a node in order to understand the structure of the graph and maybe traverse it. In the following example, we load a specific node and we list its input and output links. The returned dictionaries have as keys the link name and as value the linked node. Here is the code:

```
In [1]: # Let's load a node with a specific pk

In [2]: c = load_node(139168)
```

```

In [3]: c.get_inputs_dict()
Out[3]:
{'u'code': <Code: Remote code 'cp-5.1' on daint, pk: 75709, uuid: 3c9cdb7f-0cda-402e-b898-4dd0d06aa5a>,
 u'parameters': <ParameterData: uuid: 94efe64f-7f7e-46ea-922a-fe64a7fba8a5 (pk: 139166)>,
 u'parent_calc_folder': <RemoteData: uuid: becb4894-c50c-4779-b84f-713772eaceff (pk: 139118)>,
 u'pseudo_Ba': <UpfData: uuid: 5e53b22d-5757-4d50-bbe0-51f3b9ac8b7c (pk: 1905)>,
 u'pseudo_O': <UpfData: uuid: 5cccd0d9-7944-4c67-b3c7-a39a1f467906 (pk: 1658)>,
 u'pseudo_Ti': <UpfData: uuid: e5744077-8615-4927-9f97-c5f7b36ba421 (pk: 1660)>,
 u'settings': <ParameterData: uuid: a5a828b8-fdd8-4d75-b674-2e2d62792de0 (pk: 139167)>,
 u'structure': <StructureData: uuid: 3096f83c-6385-48c4-8cb2-24a427ce11b1 (pk: 139001)>}

In [4]: c.get_outputs_dict()
Out[4]:
{'u'output_parameters': <ParameterData: uuid: f7a3ca96-4594-497f-a128-9843a1f12f7f (pk: 139257)>,
 u'output_parameters_139257': <ParameterData: uuid: f7a3ca96-4594-497f-a128-9843a1f12f7f (pk: 139257)>,
 u'output_trajectory': <TrajectoryData: uuid: 7c5b65bc-22bb-4b87-ac92-e8a78cf145c3 (pk: 139256)>,
 u'output_trajectory_139256': <TrajectoryData: uuid: 7c5b65bc-22bb-4b87-ac92-e8a78cf145c3 (pk: 139256)>,
 u'remote_folder': <RemoteData: uuid: 17642a1c-8cac-4e7f-8bd0-1dcebe974aa4 (pk: 139169)>,
 u'remote_folder_139169': <RemoteData: uuid: 17642a1c-8cac-4e7f-8bd0-1dcebe974aa4 (pk: 139169)>,
 u'retrieved': <FolderData: uuid: a9037dc0-3d84-494d-9616-42b8df77083f (pk: 139255)>,
 u'retrieved_139255': <FolderData: uuid: a9037dc0-3d84-494d-9616-42b8df77083f (pk: 139255)>}

```

### Understanding link names

The nodes may have input and output links. Every input link of a node should have a unique name and this unique name is mapped to a specific node. On the other hand, given a node *c*, many output nodes may share the same output link name. To differentiate between the output nodes of *c* that have the same link name, the *pk* of the output node is added next to the link name (please see the input & output nodes in the above example).

**Input/output related methods** The input/output links of the node can be accessed by the following methods.

#### Methods to get the input data

- `get_inputs_dict()` returns a dictionary where the key is the label of the input link.
- `get_inputs()` returns the list of input nodes
- `inp()` returns a `NodeInputManager()` object that can be used to access the node's parents.
- `has_parents()` returns true or false whether the node has parents

#### Methods to get the output data

- `get_outputs_dict()` returns a dictionary where the key is the label of the output link, and the value is the input node.
- `get_outputs()` returns a list of output nodes.
- `out()` returns a `NodeOutputManager()` object that can be used to access the node's children.
- `has_children()` returns true or false whether the node has children.

### Navigating in the “node” graph

The user can easily use the `NodeInputManager()` and the `NodeOutputManager()` objects of a node (provided by the `inp()` and `out()` respectively) to traverse the node graph and access other connected nodes. `inp()` will give us access to the input nodes and `out()` to the output nodes. For example:

```

In [1]: # Let's load a node with a specific pk

In [2]: c = load_node(139168)

```

```

In [3]: c
Out[3]: <CpCalculation: uuid: 49084dcf-c708-4422-8bcf-808e4c3382c2 (pk: 139168)>

In [4]: # Let's traverse the inputs of this node.

In [5]: # By typing c.inp. we get all the input links

In [6]: c.inp.
c.inp.code                c.inp.parent_calc_folder  c.inp.pseudo_O            c.inp.settings
c.inp.parameters          c.inp.pseudo_Ba            c.inp.pseudo_Ti           c.inp.structure

In [7]: # We may follow any of these links to access other nodes. For example, let's follow the parent

In [8]: c.inp.parent_calc_folder
Out[8]: <RemoteData: uuid: becb4894-c50c-4779-b84f-713772eaceff (pk: 139118)>

In [9]: # Let's assign to r the node reached by the parent_calc_folder link

In [10]: r = c.inp.parent_calc_folder

In [11]: r.inp.__dir__()
Out[11]:
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__format__',
 '__getattr__',
 '__getattribute__',
 '__getitem__',
 '__hash__',
 '__init__',
 '__iter__',
 '__module__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 u'remote_folder']

In [12]: r.out.
r.out.parent_calc_folder      r.out.parent_calc_folder_139168

In [13]: # By following the same link from node r, you will get node c

In [14]: r.out.parent_calc_folder
Out[14]: <CpCalculation: uuid: 49084dcf-c708-4422-8bcf-808e4c3382c2 (pk: 139168)>

```

**Attributes related methods** Each `Node()` object can have attributes which are properties that characterize the node. Such properties can be the energy, the atom symbols or the lattice vectors. The following methods can be used for the management of the attributes.

- `_set_attr()` adds a new attribute to the node. The key of the attribute is the property name (e.g. `energy`, `lattice_vectors` etc) and the value of the attribute is the value of that property.
- `_del_attr()` & `_del_all_attrs()` delete a specific or all attributes.
- `get_attr()` returns a specific attribute.
- `iterattrs()` returns an iterator over the attributes. The iterators returns tuples of key/value pairs.
- `attrs()` returns the keys of the attributes.

**Extras related methods** Extras are additional information that are added to the calculations. In contrast to `files` and `attributes`, `extras` are information added by the user (user specific).

- `set_extra()` adds an extra to the database. To add a more extras at once, `set_extras()` can be used.
- `get_extra()` and `get_extras()` return a specific extra or all the available extras respectively. `extras()` returns the keys of the extras. `iterextras()` returns an iterator (returning key/value tuples) of the extras.
- `del_extra()` deletes an extra.

**Folder management** Folder objects represent directories on the disk (virtual or not) where extra information for the node are stored. These folders can be temporary or permanent.

- `folder()` returns the folder associated to the node.
- `get_folder_list()` returns the list of files that are in the `path` sub-folder of the repository folder.
- `_repository_folder()` returns the permanent repository folder.
- `_get_folder_pathsubfolder()` returns the `path` sub-folder in the repository.
- `_get_temp_folder()` returns the node folder in the temporary repository.
- `remove_path()` removes a file/directory from the repository.
- `add_path()` adds a file or directory to the repository folder.
- `get_abs_path()` returns the absolute path of the repository folder.

### Store & deletion

- `store_all()` stores all the input nodes, then it stores the current node and in the end, it stores the cached input links.
- `_store_input_nodes()` stores the input nodes.
- `_check_are_parents_stored()` checks that the parents are stored.
- `_store_cached_input_links()` stores the input links that are in memory.
- `store()` method checks that the node data is valid, then check if node's parents are stored, then moves the contents of the temporary folder to the repository folder and in the end, it stores in the database the information that are in the cache. The latter happens with a database transaction. In case this transaction fails, then the data transferred to the repository folder are moved back to the temporary folder.
- `__del__()` deletes temporary folder and it should be called when an in-memory object is deleted.

## DbNode

The *DbNode* is the Django class that corresponds to the *Node* class allowing to store and retrieve the needed information from and to the database. Other classes extending the *Node* class, like *Data*, *Calculation* and *Code* use the *DbNode* code too to interact with the database. The main methods are:

- *get\_aiida\_class()* which returns the corresponding AiiDA class instance.
- *get\_simple\_name()* which returns a string with the type of the class (by stripping the path before the class name).
- *attributes()* which returns the all the attributes of the specific node as a dictionary.
- *extras()* which returns all the extras of the specific node as a dictionary.

## Folders

AiiDA uses *Folder* and its subclasses to add an abstraction layer between the functions and methods working directly on the file-system and AiiDA. This is particularly useful when we want to easily change between different folder options (temporary, permanent etc) and storage options (plain local directories, compressed files, remote files & directories etc).

### Folder

This is the main class of the available *Folder* classes. Apart from the abstraction provided to the OS operations needed by AiiDA, one of its main features is that it can restrict all the available operations within a given folder limit. The available methods are:

- *mode\_dir()* and *mode\_file()* return the mode with which folders and files should be writable.
- *get\_subfolder()* returns the subfolder matching the given name
- *get\_content\_list()* returns the contents matching a pattern.
- *insert\_path()* adds a file/folder to a specific location and *remove\_path()* removes a file/folder
- *get\_abs\_path()* returns the absolute path of a file/folder under a given folder and *abspath()* returns the absolute path of the folder.
- *create\_symlink()* creates a symlink pointing the given location inside the folder.
- *create\_file\_from\_filelike()* creates a file from the given contents.
- *open()* opens a file in the folder.
- *folder\_limit()* returns the limit under which the creation of files/folders is restrained.
- *exists()* returns true or false depending whether a folder exists or not.
- *isfile()* and `py:meth:~aiida.common.folders.Folder.isdir` return true or false depending on the existence of the given file/folder.
- *create()* creates the folder, *erase()* deletes the folder and *replace\_with\_folder()* copies/moves a given folder.

### RepositoryFolder

Objects of this class correspond to the repository folders. The *RepositoryFolder* specific methods are:

- *\_\_init\_\_()* initializes the object with the necessary folder names and limits.

- `get_topdir()` returns the top directory.
- `section()` returns the section to which the `folder` belongs. This can be for the moment a workflow or node.
- `subfolder()` returns the subfolder within the section/uuid folder.
- `uuid()` the UUID of the corresponding node or workflow.

#### SandboxFolder

`SandboxFolder` objects correspond to temporary (“sandbox”) folders. The main methods are:

- `__init__()` creates a new temporary folder
- `__exit__()` destroys the folder on exit.

### 3.1.3 Developer calculation plugin tutorial - Integer summation

In this chapter we will give you some examples and a brief guide on how to write a plugin to support a new code. We will focus here on a very simple code (that simply adds two numbers), so that we can focus only on how AiiDA manages the calculation. At the end, you will have an overview of how a plugin is developed. You will be able then to proceed to more complex plugin guides like the guide for the Quantum Espresso plugin, or you can directly jump in and develop your own plugin!

#### Overview

Before analysing the different components of the plugin, it is important to understand which are these and their interaction.

We should keep in mind that AiiDA is a tool allowing us to perform easily calculations and to maintain data provenance. That said, it should be clear that AiiDA doesn’t perform the calculations but orchestrates the calculation procedure following the user’s directives. Therefore, AiiDA executes (external) codes and it needs to know:

- where the code is;
- how to prepare the input for the code. This is called an *input plugin* or a *Calculation* subclass;
- how to parse the output of the code. This is called an *output plugin* or a *Parser* subclass.

It is also useful, but not necessary, to have a script that prepares the calculation for AiiDA with the necessary parameters and submits it. Let’s start to see how to prepare these components.

#### Code

The code is an external program that does a useful calculation for us. For detailed information on how to setup the new codes, you can have a look at the [respective documentation page](#).

Imagine that we have the following python code that we want to install. It does the simple task of adding two numbers that are found in a JSON file, whose name is given as a command-line parameter:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import json
import sys
```



```

in_file = sys.argv[1]
out_file = sys.argv[2]

with open(in_file) as f:
    in_dict = json.load(f)

out_dict = { 'sum':in_dict['x1']+in_dict['x2'] }

with open(out_file, 'w') as f:
    json.dump(out_dict, f)

```

The result will be stored in JSON format in a file which name is also passed as parameter. The resulting file from the script will be handled by AiiDA. The code can be downloaded from [here](#). We will now proceed to prepare an AiiDA input plugin for this code.

## Input plugin

In abstract term, this plugin must contain the following two pieces of information:

- what are the input data objects of the calculation;
- how to convert the input data object in the actual input file required by the external code.

Let's have a look at the *input plugin* developed for the aforementioned summation code (a detailed description of the different sections follows):

```

# -*- coding: utf-8 -*-

from aiida.orm import JobCalculation
from aiida.orm.data.parameter import ParameterData
from aiida.common.utils import classproperty
from aiida.common.exceptions import InputValidationError
from aiida.common.exceptions import ValidationError
from aiida.common.datastructures import CalcInfo, CodeInfo
import json

class SumCalculation(JobCalculation):
    """
    A generic plugin for adding two numbers.
    """

    def __init_internal_params(self):
        super(SumCalculation, self).__init_internal_params()

        self._INPUT_FILE_NAME = 'in.json'
        self._OUTPUT_FILE_NAME = 'out.json'
        self._default_parser = 'sum'

    @classproperty
    def _use_methods(cls):
        """
        Additional use_* methods for the namelists class.
        """
        retdict = JobCalculation._use_methods
        retdict.update({
            "parameters": {
                'valid_types': ParameterData,
                'additional_parameter': None,
            }
        })

```

```

        'linkname': 'parameters',
        'docstring': ("Use a node that specifies the input parameters "
                      "for the namelists"),
    },
    })
    return retdict

def _prepare_for_submission(self, tempfolder, inputdict):
    """
    This is the routine to be called when you want to create
    the input files and related stuff with a plugin.

    :param tempfolder: a aiida.common.folders.Folder subclass where
                       the plugin should put all its files.
    :param inputdict: a dictionary with the input nodes, as they would
                     be returned by get_inputs_dict (with the Code!)
    """
    try:
        parameters = inputdict.pop(self.get_linkname('parameters'))
    except KeyError:
        raise InputValidationError("No parameters specified for this "
                                   "calculation")
    if not isinstance(parameters, ParameterData):
        raise InputValidationError("parameters is not of type "
                                   "ParameterData")
    try:
        code = inputdict.pop(self.get_linkname('code'))
    except KeyError:
        raise InputValidationError("No code specified for this "
                                   "calculation")
    if inputdict:
        raise ValidationError("Cannot add other nodes beside parameters")

    #####
    # END OF INITIAL INPUT CHECK #
    #####

    input_json = parameters.get_dict()

    # write all the input to a file
    input_filename = tempfolder.get_abs_path(self._INPUT_FILE_NAME)
    with open(input_filename, 'w') as infile:
        json.dump(input_json, infile)

    # ===== calcinfo =====

    calcinfo = CalcInfo()
    calcinfo.uuid = self.uuid
    calcinfo.local_copy_list = []
    calcinfo.remote_copy_list = []
    calcinfo.retrieve_list = [self._OUTPUT_FILE_NAME]

    codeinfo = CodeInfo()
    codeinfo.cmdline_params = [self._INPUT_FILE_NAME, self._OUTPUT_FILE_NAME]
    codeinfo.code_uuid = code.uuid
    calcinfo.codes_info = [codeinfo]

    return calcinfo

```

The above input plugin can be downloaded from [\(here\)](#) and should be placed at `aiida/orm/calculation/job/sum.py`.

In order the plugin to be automatically discoverable by AiiDA, it is important to:

- give the right name to the file. This should be the name of your input plugin (all lowercase);
- place the plugin under `aiida/orm/calculation/job`;
- name the class inside the plugin as `PluginnameCalculation`. For example, the class name of the summation input plugin is, as you see above, `SumCalculation`. The first letter must be capitalized, the other letters must be lowercase;
- inherit the class from `JobCalculation`.

By doing the above, your plugin will be discoverable and loadable using `CalculationFactory`.

**Note:** The base `Calculation` class should only be used as the abstract base class. Any calculation that needs to run on a remote scheduler must inherit from `JobCalculation`, that contains all the methods to run on a remote scheduler, get the calculation state, copy files remotely and retrieve them, ...

### Defining the accepted input Data nodes

The input data nodes that the input plugin expects are those returned by the `_use_methods` class property. It is important to always extend the dictionary returned by the parent class, starting this method with:

```
retdict = JobCalculation._use_methods
```

(or the correct parent class, instead of `JobCalculation`, if you are inheriting from a subclass).

The specific parameters needed by the plugin are defined by the following code snippet:

```
retdict.update({
    "parameters": {
        'valid_types': ParameterData,
        'additional_parameter': None,
        'linkname': 'parameters',
        'docstring': ("Use a node that specifies the input parameters "
                     "for the namelists"),
    },
})
```

This means that this specific summation plugin expects only one input data node, which is of the type `ParameterData` and with link name `parameters`.

### The main plugin logic

The main logic of the plugin (called by AiiDA just before submission, in order to read the AiiDA input data nodes and create the actual input files for the external code) must be defined inside a method `_prepare_for_submission`, that will receive (beside `self`) two parameters, a temporary folder `tempfolder` in which content can be written, and a dictionary containing all the input nodes that AiiDA will retrieve from the database (in this way, the plugin does not need to browse the database).

The input data node with the parameter is retrieved using its link name `parameters` specified above:

```
parameters = inputdict.pop(self.get_linkname('parameters'))
```

A few additional checks are performed to retrieve also the input code (the AiiDA node representing the code executable, that we are going to setup in the next section) and verify that there are no unexpected additional input nodes.

The following lines do the actual job, and prepare the input file for the external code, creating a suitable JSON file:

```
input_json = parameters.get_dict()

# write all the input to a file
input_filename = tempfolder.get_abs_path(self._INPUT_FILE_NAME)
with open(input_filename, 'w') as infile:
    json.dump(input_json, infile)
```

### The last step: the calcinfo

We can now create the calculation info: an object containing some additional information that AiiDA needs (beside the files you generated in the folder) in order to submit the calculation. In the calcinfo object, you need to store the calculation UUID:

```
calcinfo.uuid = self.uuid
```

You should also define a list of output files that will be retrieved automatically after the code execution, and that will be stored permanently into the AiiDA database:

```
calcinfo.retrieve_list = [self._OUTPUT_FILE_NAME]
```

For the time being, just define also the following variables as empty lists (we will describe them in the next sections):

```
calcinfo.local_copy_list = []
calcinfo.remote_copy_list = []
```

Finally, you need to specify which code executable(s) need to be called link the code to the `codeinfo` object. For each code, you need to create a `CodeInfo` object, specify the code UUID, and define the command line parameters that should be passed to the code as a list of strings (only parameters after the executable name must be specified. Moreover, AiiDA takes care of escaping spaces and other symbols). In our case, our code requires the name of the input file, followed by the name of the output file, so we write:

```
codeinfo.cmdline_params = [self._INPUT_FILE_NAME, self._OUTPUT_FILE_NAME]
```

Finally, we link the just created `codeinfo` to the `calcinfo`, and return it:

```
calcinfo.codes_info = [codeinfo]

return calcinfo
```

---

**Note:** `calcinfo.codes_info` is a list of `CodeInfo` objects. This allows to support the execution of more than one code, and will be described later.

---

**Note:** All content stored in the `tempfolder` will be then stored into the AiiDA database, potentially *forever*. Therefore, before generating huge files, you should carefully think at how to design your plugin interface. In particular, give a look to the `local_copy_list` and `remote_copy_list` attributes of `calcinfo`, described in more detail in the [Quantum ESPRESSO developer plugin tutorial](#).

---

By doing all the above, we have clarified what parameters should be passed to which code, we have prepared the input file that the code will access and we let also AiiDA know the name of the output file: our first input plugin is ready!

**Note:** A few class internal parameters can (or should) be defined inside the `_init_internal_params` method:

```
def _init_internal_params(self):
    super(SumCalculation, self)._init_internal_params()

    self._INPUT_FILE_NAME = 'in.json'
    self._OUTPUT_FILE_NAME = 'out.json'
    self._default_parser = 'sum'
```

In particular, it is good practice to define a `_INPUT_FILE_NAME` and `_OUTPUT_FILE_NAME` attributes (pointing to the default input and output file name – these variables are then used by some `verdi` commands, such as `verdi calculation outputcat`). Also, you need to define the name of the default parser that will be invoked when the calculation completes in `_default_parser`. In the example above, we choose the ‘sum’ plugin (that we are going to define later on). If you don’t want to call any parser, set this variable to `None`.

As a final step, after copying the file in the location specified above, we can check if AiiDA recognised the plugin, by running the command `verdi calculation plugins` and verifying that our new `sum` plugin is now listed.

### Setup of the code

Now that we know the executable that we want to run, and we have setup the input plugin, we can proceed to configure AiiDA by setting up a new code to execute:

```
verdi code setup
```

During the setup phase, you can either configure a *remote* code (meaning that you are going to place the python executable in the right folder of the remote computer, and then just instruct AiiDA on the location), or as a *local* folder, meaning that you are going to store (during the setup phase) the python executable into the AiiDA DB, and AiiDA will copy it to the remote computer when needed. In this second case, put the `sum_executable.py` in an empty folder and pass this folder in the setup phase.

**Note:** In both cases, remember to set the executable flag to the code by running `chmod +x sum_executable.py`.

After defining the code, we should be able to see it in the list of our installed codes by typing:

```
verdi code list
```

A typical output of the above command is:

```
$ verdi code list
# List of configured codes:
# (use 'verdi code show CODEID' to see the details)
* Id 73: sum
```

Where we can see the already installed summation code. We can further see the specific parameters that we gave when we set-up the code by typing:

```
verdi code show 73
```

Which will give us an output similar to the following:

```
$ verdi code show 73
* PK: 73
* UUID: 34b44d33-86c1-478b-88ff-baadfb6f30bf
* Label: sum
* Description: A simple sum executable
* Default plugin: sum
* Used by: 0 calculations
```

```
* Type:                local
* Exec name:            ./sum_executable.py
* List of files/folders:
  * [file] sum_executable.py
* prepend text:
  # No prepend text.
* append text:
  # No append text.
```

What is important to keep from the above is that we have informed AiiDA for the existence of a code that resides at a specific location and we have also specified the *default (input) plugin* that will be used.

### Output plugin: the parser

In general, it is useful to parse files generated by the code to import relevant data into the database. This has two advantages:

- we can store information in specific data classes to facilitate their use (e.g. crystal structures, parameters, ...)
- we can then make use of efficient database queries if, e.g., output quantities are stored as integers or floats rather than as strings in a long text file.

The following is a sample output plugin for the summation code, described in detail later:

```
# -*- coding: utf-8 -*-

from aiida.orm.calculation.job.sum import SumCalculation
from aiida.parsers.parser import Parser
from aiida.parsers.exceptions import OutputParsingError
from aiida.orm.data.parameter import ParameterData

import json

class SumParser(Parser):
    """
    This class is the implementation of the Parser class for Sum.
    """
    def parse_with_retrieved(self, retrieved):
        """
        Parses the datafolder, stores results.
        This parser for this simple code does simply store in the DB a node
        representing the file of forces in real space
        """

        successful = True
        # select the folder object
        # Check that the retrieved folder is there
        try:
            out_folder = retrieved[self._calc._get_linkname_retrieved()]
        except KeyError:
            self.logger.error("No retrieved folder found")
            return False, ()

        # check what is inside the folder
        list_of_files = out_folder.get_folder_list()
        # at least the stdout should exist
        if self._calc._OUTPUT_FILE_NAME not in list_of_files:
            successful = False
```

```

self.logger.error("Output json not found")
return successful, ()

try:
    with open( out_folder.get_abs_path(self._calc._OUTPUT_FILE_NAME) ) as f:
        out_dict = json.load(f)
except ValueError:
    successful=False
    self.logger.error("Error parsing the output json")
    return successful, ()

# save the arrays
output_data = ParameterData(dict=out_dict)
link_name = self.get_linkname_outparams()
new_nodes_list = [(link_name, output_data)]

return successful, new_nodes_list

```

As mentioned above the *output plugin* will *parse* the output of the executed code at the remote computer and it will store the results to the AiiDA database.

All the parsing code is enclosed in a single method `parse_with_retrieved`, that will receive as a single parameter `retrieved`, a dictionary of retrieved nodes. The default behavior is to create a single `FolderData` node, that can be retrieved using:

```
out_folder = retrieved[self._calc.get_linkname_retrieved()]
```

We then read and parse the output file that will contain the result:

```

with open( out_folder.get_abs_path(self._calc._OUTPUT_FILE_NAME) ) as f:
    out_dict = json.load(f)

```

**Note:** all parsers have a `self._calc` attribute that points to the calculation being parsed. This is automatically set in the parent `Parser` class.

After loading the code result data to the dictionary `out_dict`, we construct a `ParameterData` object (`ParameterData(dict=out_dict)`) that will be linked to the calculation in the AiiDA graph to be later in the database:

```

output_data = ParameterData(dict=out_dict)
link_name = self.get_linkname_outparams()
new_nodes_list = [(link_name, output_data)]

return successful, new_nodes_list

```

**Note:** Parsers should not store nodes manually. Instead, they should return a list of output unstored nodes (together with a link name string, as shown above). AiiDA will then take care of storing the node, and creating the appropriate links in the DB.

**Note:** the `self.get_linkname_outparams()` is a string automatically defined in all `Parser` classes and subclasses. In general, you can have multiple output nodes with any name, but it is good practice so have also one of the output nodes with link name `self.get_linkname_outparams()` and of type `ParameterData`. The reason is that this node is the one exposed with the `calc.res` interface (for instance, later we will be able to get the results using `print calc.res.sum`).

The above *output plugin* can be downloaded from [here](#) and should be placed at `aiida/parsers/plugins/sum.py`.

**Note:** Before continuing, it is important to restart the daemon, so that it can recognize the new files added into the aiida code and use the new plugins. To do so, run now:

```
verdi daemon restart
```

---

## Submission script

It's time to calculate how much  $2+3$  is! We need to submit a new calculation. To this aim, we don't necessarily need a submission script, but it definitely facilitates the calculation submission. A very minimal sample script follows (other examples can be found in the `aiida/examples/submission` folder):

```
#!/usr/bin/env runaiida
# -*- coding: utf-8 -*-
import sys
import os

from aiida.common.exceptions import NotExistent
ParameterData = DataFactory('parameter')

# The name of the code setup in AiiDA
codename = 'sum'
computer_name = 'localhost'

#####
try:
    dontsend = sys.argv[1]
    if dontsend == "--dont-send":
        submit_test = True
    elif dontsend == "--send":
        submit_test = False
    else:
        raise IndexError
except IndexError:
    print >> sys.stderr, ("The first parameter can only be either "
                          "--send or --dont-send")
    sys.exit(1)

code = Code.get_from_string(codename)
# The following line is only needed for local codes, otherwise the
# computer is automatically set from the code
computer = Computer.get(computer_name)

# These are the two numbers to sum
parameters = ParameterData(dict={'x1':2, 'x2':3})

calc = code.new_calc()
calc.label = "Test sum"
calc.description = "Test calculation with the sum code"
calc.set_max_wallclock_seconds(30*60) # 30 min
calc.set_computer(computer)
calc.set_withmpi(False)
calc.set_resources({"num_machines": 1})

calc.use_parameters(parameters)
```



```

if submit_test:
    subfolder, script_filename = calc.submit_test()
    print "Test submit file in {}".format(os.path.join(
        os.path.relpath(subfolder.abspath),
        script_filename
    ))
else:
    calc.store_all()
    calc.submit()
    print "submitted calculation; calc=Calculation(uuid='{}') # ID={}".format(
        calc.uuid, calc.dbnode.pk)

```

What is important to note in the script above is the definition of the code to be used:

```

codename = 'sum'
code = Code.get_from_string(codename)

```

and the definition of the parameters:

```

parameters = ParameterData(dict={'x1':2, 'x2':3})
calc.use_parameters(parameters)

```

If everything is done correctly, by running the script a new calculation will be generated and submitted to AiiDA (to run the script, remember to change its permissions with `chmod +x filename` first, and then run it with `./scriptname.py`). When the code finishes its execution, AiiDA will retrieve the results, parse and store them back to the AiiDA database using the output plugin. You can download the submission script from [here](#).

## Conclusion

We have just managed to write our first AiiDA plugin! What is important to remember is that:

- AiiDA doesn't know how to execute your code. Therefore, you have to setup your code (with `verdi code setup`) and let AiiDA know how to prepare the data that will be given to the code (*input plugin* or *calculation*) and how to handle the result of the code (*output plugin* or *parser*).
- you need to do pass the actual data for the calculation you want to submit, either in the interactive shell, or via a submission script.

As usual, we can see the executed calculations by doing a `verdi calculation list`. To see the calculations of the last day:

```

$ verdi calculation list -a -p1
# Last daemon state_updater check: 0h:00m:06s ago (at 20:10:31 on 2015-10-20)
# Pk|State      |Creation|Sched. state|Computer  |Type
327 |FINISHED     |4h ago  |DONE        |localhost |sum

```

and we can see the result of the sum by running in the verdi shell the following commands (change 327 with the correct calculation PK):

```

>>> calc = load_node(327)
>>> print calc.res.sum
<<< 5

```

So we verified that, indeed,  $2+3=5$ .

### 3.1.4 Developer data plugin tutorial - Float summation

Now that you have written [your first AiiDA plugin](#), we can try to extend it to see how we can introduce different type of parameters and how the plugins have to be modified to encompass these changes.

#### Introducing a new data type

We will start by describing what is a data plugin, and by creating a new one.

A data plugin is a subclass of `Data`. What you have to do is just to define a subclass with a suitable name inside the `aiida/orm/data` folder (with the same name convention of Calculation plugins: the class should be called `NameData` (with `Name` being a name of your choice) and put in a `aiida/orm/data/name.py` file. In the class, you should provide methods that the end user should use to store high-level objects (for instance, for a crystal structure, there can be a method for setting the unit cell, one for adding an atom in a given position, ...). Internally, you should choose where to store the content. There are two options:

- **In the AiiDA database.** This is useful for small amounts of data, that you plan to query. In this case, use `self._set_attr(attr_name, attr_value)` to store the required value.
- **In the AiiDA file repository (as a file on the disk).** This is suitable for big files and quantities that you do not want to query. In this case, access the folder using `self.folder` and use the methods of `self.folder` to create files, subfolders, ...

Of course, it is also good practice to provide “getter” methods to retrieve the data in the database and return it back to the user. The idea is that the user can operate directly only with the methods you provide, and should not need to know how you decided to store the data inside the AiiDA database.

As a simple example that we will use for the exercise below, imagine that we want to introduce a new type of data node that simply stores a float number. We will call it `FloatData`, and the class implementation can look like this:

```
from aiida.orm.data import Data

class FloatData(Data):

    @property
    def value(self):
        """
        The value of the Float
        """
        return self.get_attr('number')

    @value.setter
    def value(self, value):
        """
        Set the value of the Float

        :raise ValueError:
        """
        self._set_attr('number', float(value))
```

This file should be placed under `aiida/orm/data/float.py` and it should extend the class `Data`.

#### Exercise: Modifying the calculation plugin

Your exercise consists in creating a new code plugin (let's call it for instance `floatsum`) that will also perform the sum, but accept as input two `FloatData` node and return also a `FloatData` node containing the sum.

Below, you will find some hints on the parts you need to modify with respect to the [previous tutorial](#) using instead `ParameterData` both as inputs and outputs.

**Note:** remember to create copies of your files with a new name `floatsum.py` instead of `sum.py`, and to change the class name accordingly.

### Changes to the parser

The plugin should now return a `FloatData` instead of a `ParameterData`, therefore the parser code should contain something like the following:

```
output_data = FloatData()
output_data.value = out_dict["sum"]
linkname = 'output_data'
```

### Changes to the input plugin

To be able to run your new `FloatsumParser`, you will need the corresponding input plugin (`FloatsumCalculation`). The first modification is then to link to the correct parser class:

```
self._default_parser = 'floatsum'
```

For consistency, we also want that the input plugin accepts two `FloatData` instead of a single `ParameterData`. Therefore, you have to update the `retdict` object accordingly:

```
retdict.update({
    "float_data_1": {
        'valid_types': FloatData,
        'additional_parameter': None,
        'linkname': 'float_data_1',
        'docstring': ("The first addend"),
    },
    "float_data_2": {
        'valid_types': FloatData,
        'additional_parameter': None,
        'linkname': 'float_data_2',
        'docstring': ("The second addend"),
    },
})
```

You need then to change the main code to use the values obtained from the two nodes, rather than from a single node as before. This should be easy, so we leave this task to you. Note that we plan to use the same python code to actually perform the sum, so the JSON file to be generated should have the same format.

We also suggest that you add utility methods (to the benefit of the end user) to provide the addends to the code, something like:

```
def set_addendl(self, value):
    fl = FloatData()
    fl.value = value
    self.use_float_data_1(fl)
```

and similarly for the second addend.

The final input plugin should be placed at `aiida/orm/calculation/job/floatsum.py`.

## Code

The python code that actually performs the calculation does not need to be modified. We can reuse the same file, but we suggest to setup a new code in AiiDA, with a different name, using as default plugin the `floatsum` plugin.

## Submission script

Finally, adapt your submission script to create the correct input nodes, and try to perform a sum of two numbers to verify that you did all correctly!

---

**Note:** After placing your files, do not forget to restart the daemon so that it will recognize the files! The same should be done if you do any change to the plugin, otherwise the daemon may have cached the old file and will keep using it.

---

## 3.1.5 Developer code plugin tutorial - Quantum Espresso

In this section we will focus on AiiDA's Quantum Espresso plugin that we are going to analyse and show how a physics oriented plugin is developed. It will be assumed that you have already tried to run an example of Quantum Espresso, and you know more or less how the AiiDA interface works. We hope that in the end you will be able to replicate the task for other codes.

In fact, when writing your own plugin, keep in mind that you need to satisfy multiple users, and the interface needs to be simple (not the code below). But always try to follow the Zen of Python:

Simple is better than complex.

Complex is better than complicated.

Readability counts.

As demonstrated in previous sections, there will be two kinds of plugins: the input and the output. The former has the purpose to convert python object in text inputs that can be executed by external software. The latter will convert the text output of these software back into python dictionaries/objects that can be put back in the database.

## InputPlugin

Create a new file, which has the same name as the class you are creating (in this way, it will be possible to load it with `CalculationFactory`). Save it in a subfolder at the path `aiida/orm/calculation/job`.

### Step 1: inheritance

First define the class:

```
class SubclassCalculation(JobCalculation):
```

(Substitute `Subclass` with the name of your plugin). Take care of inheriting the `JobCalculation` class, or the plugin will not work.

Now, you will likely need to define some variables that belong to `SubclassCalculation`. In order to be sure that you don't lose any variables belonging to the inherited class, every subclass of calculation needs to have a method which is called `_init_internal_params()`. An example of it would look like:

```
def __init_internal_params(self):
    super(SubclassCalculation, self).__init_internal_params()

    self.A_NEW_VARIABLE = 'nabucco'
```

This function will be called by the `__init__` method and will initialize the variable `A_NEW_VARIABLE` at the moment of the instancing. The second line will call the `__init_internal_params()` of the parent class and load other variables eventually defined there. Now you are able to access the variable `A_NEW_VARIABLE` also in the rest of the class by calling `self.A_NEW_VARIABLE`.

---

**Note:** Even if you don't need to define new variables, it is safer to define the method with the call to `super()`.

---

**Note:** It is not recommended to rewrite an `__init__` by yourself: this method is inherited from the classes `Node` and `Calculation`, and you shouldn't alter it unless you really know the code down to the lowest-level.

---

**Note:** The following is a list of relevant parameters you may want to (re)define in `__init_internal_params`:

- `self.__default_parser`: set to the string of the default parser to be used, in the form accepted by the plugin loader (e.g., for the Quantum ESPRESSO plugin for phonons, this would be “quantumespresso.ph”, loaded from the `aiida.parsers.plugins` module).
  - `self.__DEFAULT_INPUT_FILE`: specify here the relative path to the filename of the default file that should be shown by `verdi calculation outputcat --default`. If not specified, the default value is `None` and `verdi calculation outputcat` will not accept the `--default` option, but it will instead always ask for a specific path name.
  - `self.__DEFAULT_OUTPUT_FILE`: same of `__DEFAULT_INPUT_FILE`, but for the default output file.
- 

## Step 2: define input nodes

First, you need to specify what are the objects that are going to be accepted as input to the calculation class. This is done by the class property `_use_methods`. An example is as follows:

```
from aiida.common.utils import classproperty

class SubclassCalculation(JobCalculation):

    def __init_internal_params(self):
        super(SubclassCalculation, self).__init_internal_params()

    @classproperty
    def _use_methods(cls):
        retdict = JobCalculation._use_methods
        retdict.update({
            "settings": {
                'valid_types': ParameterData,
                'additional_parameter': None,
                'linkname': 'settings',
                'docstring': "Use an additional node for special settings",
            },
            "pseudo": {
                'valid_types': UpfData,
                'additional_parameter': 'kind',
                'linkname': cls._get_pseudo_linkname,
                'docstring': ("Use a remote folder as parent folder (for "
```

```
        "restarts and similar"),
    },
    })
    return retdict

@classmethod
def _get_pseudo_linkname(cls, kind):
    """
    Return the linkname for a pseudopotential associated to a given
    structure kind.
    """
    return "pseudo_{}".format(kind)
```

After this piece of code is written, we now have defined two methods of the calculation that specify what DB object could be set as input (and draw the graph in the DB). Specifically, here we will find the two methods:

```
calculation.use_settings(an_object)
calculation.use_pseudo(another_object, 'object_kind')
```

What did we do?

1. We added implicitly the two new `use_settings` and `use_pseudo` methods (because the dictionary returned by `_use_methods` now contains a `settings` and a `pseudo` key)
2. We did not lose the `use_code` call defined in the `Calculation` base class, because we are extending `Calculation._use_methods`. Therefore: don't specify a code as input in the plugin!
3. `use_settings` will accept only one parameter, the node specifying the settings, since the `additional_parameter` value is `None`.
4. `use_pseudo` will require two parameters instead, since `additional_parameter` value is *not* `None`. If the second parameter is passed via kwargs, its name must be 'kind' (the value of `additional_parameters`). That is, you can call `use_pseudo` in one of the two following ways:

```
use_pseudo(pseudo_node, 'He')
use_pseudo(pseudo_node, kind='He')
```

to associate the pseudopotential node `pseudo_node` (that you must have loaded before) to helium (He) atoms.

5. The type of the node that you pass as first parameter will be checked against the type (or the tuple of types) specified with `valid_types` (the check is internally done using the `isinstance` python call).
6. The name of the link is taken from the `linkname` value. Note that if `additional_parameter` is `None`, this is simply a string; otherwise, it must be a callable that accepts one single parameter (the further parameter passed to the `use_XXX` function) and returns a string with the proper name. This functionality is provided to have a single `use_XXX` method to define more than one input node, as it is the case for pseudopotentials, where one input pseudopotential node must be specified for each atomic species or kind.
7. Finally, `docstring` will contain the documentation of the function, that the user can obtain by printing e.g. `use_pseudo.__doc__`.

---

**Note:** The actual implementation of the `use_pseudo` method in the Quantum ESPRESSO tutorial is slightly different, as it allows the user to specify a list of kinds that are associated with the same pseudopotential file (while in the example above only one kind string can be passed).

---

### Step 3: prepare a text input

How are the input nodes used internally? Every plugin class is required to have the following method:

```
def _prepare_for_submission(self, tempfolder, inputdict):
```

This function is called by the daemon when it is trying to create a new calculation.

There are two arguments:

1. `tempfolder`: is an object of kind `SandboxFolder`, which behaves exactly as a folder. In this placeholder, you are going to write the input files. This tempfolder is gonna be copied to the remote cluster.
2. `inputdict`: contains all the input data nodes as a dictionary, in the same format that is returned by the `get_inputs_dict()` method, i.e. a linkname as key, and the object as value.

Changed in version 0.5: `inputdict` should contain all input `Data` nodes, *and* the code. (this is what the `get_inputs_dict()` method returns, by the way). In older versions, the code is not present.

In general, you simply want to do:

```
inputdict = self.get_inputs_dict()
```

right before calling `_prepare_for_submission`. The reason for having this explicitly passed is that the plugin does not have to perform explicit database queries, and moreover this is useful to test for submission without the need to store all nodes on the DB.

For the sake of clarity, it's probably going to be easier looking at an implemented example. Take a look at the `NamelistCalculation` located in `aiida.orm.calculation.job.quantumespresso.namelist`.

How does the method `_prepare_for_submission` work in practice?

1. You should start by checking if the input nodes passed in `inputdict` are logically sufficient to run an actual calculation. Remember to raise an exception (for example `InputValidationError`) if something is missing or if something unexpected is found. Ideally, it is better to discover now if something is missing, rather than waiting the queue on the cluster and see that your job has crashed. Also, if there are some nodes left unused, you are gonna leave a DB more complicated than what has really been, and therefore is better to stop the calculation now.
2. create an input file (or more if needed). In the `Namelist` plugin is done like:

```
input_filename = tempfolder.get_abs_path(self.INPUT_FILE_NAME)
with open(input_filename, 'w') as infile:
    # Here write the information of a ParameterData inside this
    # file
```

Note that here it all depends on how you decided the `ParameterData` to be written. In the `namelist` plugin we decided the convention that a `ParameterData` of the format:

```
ParameterData(dict={"INPUT": {'smearing': 2,
                               'cutoff': 30}})
```

is written in the input file as:

```
&INPUT
  smearing = 2,
  cutoff=30,
/
```

Of course, it's up to you to decide a convention which defines how to convert the dictionary to the input file. You can also impose some default values for simplicity. For example, the location of the scratch directory, if needed, should be imposed by the plugin and not by the user, and similarly you can/should decide the naming of output files.

**Note:** it is convenient to avoid hard coding of all the variables that your code has. The convention stated above is sufficient for all inputs structured as fortran cards, without the need of knowing which variables are accepted. Hard coding variable names implies that every time the external software is updated, you need to modify the plugin: in practice the plugin will easily become obsolete if poor maintained. Easyness of maintainance here win over user comfort!

---

3. copy inside this folder some auxiliary files that resides on your local machine, like for example pseudopotentials.
4. return a CalcInfo object.

This object contains some accessory information. Here's a template of what it may look like:

```
calcinfo = CalcInfo()

calcinfo.uuid = self.uuid
calcinfo.local_copy_list = local_copy_list
calcinfo.remote_copy_list = remote_copy_list

calcinfo.retrieve_list = []
### Modify here !
calcinfo.retrieve_list.append('Every file/folder you want to store back locally')
### Modify here!
calcinfo.retrieve_singlefile_list = []

### Modify here and put a name for standard input/output files
codeinfo = CodeInfo()
codeinfo.cmdline_params = settings_dict.pop('CMDLINE', [])
codeinfo.stdin_name = self.INPUT_FILE_NAME
codeinfo.stdout_name = self.OUTPUT_FILE_NAME
codeinfo.withmpi = self.get_withmpi()
codeinfo.code_pk = code.pk

calcinfo.codes_info = [codeinfo]

return calcinfo
```

There are a couple of things to be set on calcinfo.

- (a) `retrieve_list`: a list of relative file pathnames, that will be copied from the cluster to the aiida server, after the calculation has run on cluster. Note that all the file names you need to modify are not absolute path names (you don't know the name of the folder where it will be created) but rather the path relative to the scratch folder.
- (b) `local_copy_list`: a list of length-two-tuples: (localabspath, relativedestpath). Files to be copied from the aiida server to the cluster.
- (c) `remote_copy_list`: a list of tuples: (remotemachinename, remoteabspath, relativedestpath). Files/folders to be copied from a remote source to a remote destination, sitting both on the same machine.
- (d) `retrieve_singlefile_list`: a list of triplets, in the form ["linkname\_from calc to singlefile", "subclass of singlefile", "filename"]. If this is specified, at the end of the calculation it will be created a SinglefileData-like object in the Database, children of the calculation, if of course the file is found on the cluster.
- (e) `codes_info`: a list of informations that needs to be passed on the command line to the code, passed in the form of a list of CalcInfo objects (see later). Every element in this list corresponds to a call to a code that will be executed in the *same* scheduling job. This can be useful if a code needs to execute a short preprocessing. For long preprocessings, consider to develop a separate plugin.
- (f) `codes_run_mode`: a string, only necessary if you want to run more than one code in the same



scheduling job. Determines the order in which the multiple codes are run (i.e. sequentially or all at the same time. It assumes one of the values of `aiida.common.datastructures.code_run_modes`, like `code_run_modes.PARALLEL` or `code_run_modes.SERIAL`

A `CodeInfo` object, as said before, describes how a code has to be executed. The list of `CodeInfo` objects passed to `calcinfo` will determined the ordered execution of one (or more) calls to executables. The attributes that can be set to `CodeInfo` are:

- (a) `stdin_name`: the name of the standard input.
- (b) `stdout_name`: the name of the standard output.
- (c) `cmdline_params`: like parallelization flags, that will be used when running the code.
- (d) `stderr_name`: the name of the error output.
- (e) `withmpi`: whether the code has to be called with mpi or not.
- (f) `code_pk`: the pk of the code associated to the `CodeInfo` instance.

If you need to change other settings to make the plugin work, you likely need to add more information to the `calcinfo` than what we showed here. For the full definition of `CalcInfo()` and `CodeInfo()`, refer to the source `aiida.common.datastructures`.

That's what is needed to write an input plugin. To test that everything is done properly, remember to use the `calculation.submit_test()` method, which creates locally the folder to be sent on cluster, without submitting the calculation on the cluster.

## OutputPlugin

Well done! You were able to have a successful input plugin. Now we are going to see what you need to do for an output plugin. First of all let's create a new folder: `$path_to_aiida/aiida/parsers/plugins/the_name_of_new_code`, and put there an empty `__init__.py` file. Here you will write in a new python file the output parser class. It is actually a rather simple class, performing only a few (but tedious) tasks.

After the calculation has been computed and retrieved from the cluster, that is, at the moment when the parser is going to be called, the calculation has two children: a `RemoteData` and a `FolderData`. The `RemoteData` is an object which represents the scratch folder on the cluster: you don't need it for the parsing phase. The `FolderData` is the folder in the AiiDA server which contains the files that have been retrieved from the cluster. Moreover, if you specified a `retrieve_singlefile_list`, at this stage there is also going to be some children of `SinglefileData` kind.

Let's say that you copied the standard output in the `FolderData`. The parser then has just a couple of tasks:

1. open the files in the `FolderData`
2. read them
3. convert the information into objects that can be saved in the Database
4. return the objects and the linkname.

---

**Note:** The parser should not save any object in the DB, that is a task of the daemon: never use a `.store()` method!

---

Basically, you just need to specify an `__init__()` method, and a function `parse_with_retrieved(calc, retrieved)___`, which does the actual work.

The difficult and long part is the point 3, which is the actual parsing stage, which convert text into python objects. Here, you should try to parse as much as you can from the output files. The more you will write, the better it will be.

---

**Note:** You should not only parse physical values, a very important thing that could be used by workflows are

exceptions or others errors occurring in the calculation. You could save them in a dedicated key of the dictionary (say 'warnings'), later a workflow can easily read the exceptions from the results and perform a dedicated correction!

---

In principle, you can save the information in an arbitrary number of objects. The most useful classes to store the information back into the DB are:

1. **ParameterData**: This is the DB representation of a python dictionary. If you put everything in a single **ParameterData**, then this could be easily accessed from the calculation with the `.res` method. If you have to store arrays / large lists or matrices, consider using **ArrayData** instead.
2. **ArrayData**: If you need to store large arrays of values, for example, a list of points or a molecular dynamic trajectory, we strongly encourage you to use this class. At variance with **ParameterData**, the values are not stored in the DB, but are written to a file (mapped back in the DB). If instead you store large arrays of numbers in the DB with **ParameterData**, you might soon realize that: a) the DB grows large really rapidly; b) the time it takes to save an object in the DB gets very large.
3. **StructureData**: If your code relaxes an input structure, you can end up with an output structure.

Of course, you can create new classes to be stored in the DB, and use them at your own advantage.

A kind of template for writing such parser for the calculation class **NewCalculation** is as follows:

```
class NewParser(Parser):
    """
    A doc string
    """

    def __init__(self, calc):
        """
        Initialize the instance of NewParser
        """
        # check for valid input
        if not isinstance(calc, NewCalculation):
            raise ParsingError("Input must calc must be a NewCalculation")

        super(NewParser, self).__init__(calc)

    def parse_with_retrieved(self, retrieved):
        """
        Parses the calculation-output datafolder, and stores
        results.

        :param retrieved: a dictionary of retrieved nodes, where the keys
            are the link names of retrieved nodes, and the values are the
            nodes.
        """
        # check the calc status, not to overwrite anything
        state = calc.get_state()
        if state != calc_states.PARSING:
            raise InvalidOperation("Calculation not in {} state"
                                   .format(calc_states.PARSING) )

        # retrieve the whole list of input links
        calc_input_parameterdata = self._calc.get_inputs(type=ParameterData,
                                                         also_labels=True)

        # then look for parameterdata only
        input_param_name = self._calc.get_linkname('parameters')
        params = [i[1] for i in calc_input_parameterdata if i[0]==input_param_name]
```

```

if len(params) != 1:
    # Use self.logger to log errors, warnings, ...
    # This will also add an entry to the DbLog table associated
    # to the calculation that we are trying to parse, that can
    # be then seen using 'verdi calculation logshow'
    self.logger.error("Found {} input_params instead of one"
                      .format(params))

    successful = False
    calc_input = params[0]

    # Check that the retrieved folder is there
    try:
        out_folder = retrieved[self._calc._get_linkname_retrieved()]
    except KeyError:
        self.logger.error("No retrieved folder found")
        return False, ()

    # check what is inside the folder
    list_of_files = out_folder.get_folder_list()
    # at least the stdout should exist
    if not calc.OUTPUT_FILE_NAME in list_of_files:
        raise QEOutputParsingError("Standard output not found")
    # get the path to the standard output
    out_file = os.path.join( out_folder.get_abs_path('.'),
                             calc.OUTPUT_FILE_NAME )

    # read the file
    with open(out_file) as f:
        out_file_lines = f.readlines()

    # call the raw parsing function. Here it was thought to return a
    # dictionary with all keys and values parsed from the out_file (i.e. enery, forces, etc...)
    # and a boolean indicating whether the calculation is successfull or not
    # In practice, this is the function deciding the final status of the calculation
    out_dict,successful = parse_raw_output(out_file_lines)

    # convert the dictionary into an AiiDA object, here a
    # ParameterData for instance
    output_params = ParameterData(dict=out_dict)

    # prepare the list of output nodes to be returned
    # this must be a list of tuples having 2 elements each: the name of the
    # linkname in the database (the one below, self.get_linkname_outparams(),
    # is defined in the Parser class), and the object to be saved
    new_nodes_list = [ (self.get_linkname_outparams(),output_params) ]

    # The calculation state will be set to failed if successful=False,
    # to finished otherwise
    return successful, new_nodes_list

```

### 3.1.6 Parser warnings policy

As a rule of thumb, always include two keys in the output parameters of a calculation, warnings and parser\_warnings. These two keys contain a list of messages (strings) that are useful for debugging problems in the execution of calculations. Below are the guidelines for the usage of the keys warnings and

`parser_warnings` in the output parameters of a calculation.

## Warnings

These should be devoted to warnings or error messages relative to the **execution of the code**. As a (non-exhaustive) list of examples, for Quantum-ESPRESSO, run-time messages such as

- Maximum CPU time exceeded.
- `c_bands: 2 eigenvalues not converged`
- Not enough space allocated for radial FFT
- The scf cycle did not reach convergence.
- The FFT is incommensurate: some symmetries may be lost.
- Error in routine [...]

should be put in the warnings. In the above cases the warning messages are directly copied from the output of the code, but a warning can also be elaborated by the parser when it finds out that something strange went on during the execution of the code. For QE an example is `QE pw run did not reach the end of the execution`.

Among the code-based warnings, some can be identified as “critical”, meaning that when present the calculation should be set in `FAILED` state. There should be an internal list in the parser, e.g. `critical_messages`, defining such critical warnings. Other non-critical warnings instead might be used to signal the presence of some possible source of troubles, but that nevertheless did not prevent the calculation to be considered `FINISHED`.

## Parser\_warnings

These should be reserved to warnings occurring **during parsing**, i.e. when the parser does not find an information it was looking for in the output files. For Quantum-ESPRESSO (PW), examples are

- Skipping the parsing of the xml file.
- Error while parsing for energy terms.
- etc.

Therefore, these warnings should be placed just to notify that the output was not found in the way the developer had expected, and they signal the necessity of improving the parser code.

### 3.1.7 Automated parser tests

AiiDA testing facility can check for the proper functionality of parsers automatically. To facilitate the creation of new tests, we provide a simple tool to create a new parser test from a calculation that you already run in your AiiDA database, described below.

## Test folders

Each folder inside the path `aiida/djsite/db/subtests/parser_tests` constitutes a single test. The naming convention for folders is the following:

- it should contain only digits, letters and underscores, otherwise the folder will be ignored when running `verdi devel tests db.parsers;`
- the folder name should start with `test_;`

- the name should be followed by the parser plugin name, as returned by `calculation.get_parser_name()`, and with dots replaced with underscores;
- it should be followed by an underscore;
- finally it should be followed by a string that explains what is tested.

For instance, a valid name is `test_quantumespresso_pw_vanderwaals`. Note that the naming scheme is only a convention, and that the parser to use for the test is selected automatically.

### Creation of a test from an existing calculation

In order to create the folder, you can open `verdi shell` while being in the folder `aiida/djsite/db/subtests/parser_tests`, import the following function:

```
from aiida.djsite.db.subtests.parsers import output_test
```

and then run it with the correct parameters. The documentation of the function can be found [here](#).

An example call could be:

```
output_test (
    pk=21,
    testname='vanderwaals',
    skip_uuids_from_inputs=[
        'f579974c-6a9e-4eb4-9b41-e72486f86ac5',
        'ee0df234-955e-4f99-9808-17e168e6a769']
)
```

where:

- 21 is the PK of the calculation that you want to export
- `vanderwaals` is the name of the test: if for instance the node with `pk=21` is a Quantum ESPRESSO pw.x calculation, the script will create a folder named `test_quantumespresso_pw_vanderwaals`
- the (optional) `skip_uuids_from_inputs` is a list of UUIDs of input nodes that will *not* be exported.

The script will create a new folder, containing the exported content of the calculation, its direct inputs (except those listed in the `skip_uuids_from_inputs` list), and the output retrieved node. The format of the exported data is the same of the export files of AiiDA, but the folder is not zipped.

**Note:** The `skip_uuids_from_inputs` parameter is typically useful for input nodes containing large files that are not needed for parsing and would just create a large test; a typical example is given by pseudopotential input nodes for Quantum ESPRESSO.

After having run the command, the existence of the folder will only test that the parser is able to parse the calculation without errors. Typically, however, you will also want to check some parsed values.

In this case, you need to modify the `_aiida_checks.json` JSON file inside the folder. The syntax is the following:

- each key represents an output node that should be generated by the parser;
- each value is a dictionary with multiple keys (an empty dictionary will just check for the existence of the output node);
- each key of the subdictionary is an attribute to check for. The value is a list of dictionaries, one for each test to perform on the given value; multiple tests are therefore possible. The dictionary should have at least have one key: “comparison”, a string to specifies the type of comparison. The other keys depend on the type of comparison, and typically there is at least a “value” key, the value to compare with. An example:

```
{
  "output_parameters": {
    "energy": [
      {
        "comparison": "AlmostEqual",
        "value": -3699.26590536037
      }
    ],
    "energy_units": [
      {
        "comparison": "Equal",
        "value": "eV"
      }
    ]
  },
  "output_array": {
  }
}
```

The list of valid comparisons is hardcoded inside the `aiida.djsite.db.subtests.parsers` module; if you need new comparison types, add them directly to the module.

## Running tests

Finally, in order to run all tests contained in the folder `aiida/djsite/db/subtests/parser_tests` one can use the following `verdi` command:

```
verdi devel tests db.parsers
```

If no fail message appears it means that the test was successful.

## 3.1.8 Workflow's Guide For AiiDA

### Creating new workflows

New user specific workflows should be put in `aiida/workflows/user`. If the workflow is general enough to be of interest for the community, the best is to create a git repository (e.g. on [Bitbucket](#)) and clone the content of the repository in a subfolder of `aiida/workflows/user`, e.g. in `aiida/workflows/user/epfl_theos` for workflows from the group THEOS at EPFL.

Put `__init__.py` files in all subdirectories of `aiida/workflows/user` to be able to import any workflows. Also, it may be a good idea to create a specific workflow factory to load easily workflows of the subdirectory. To do so place in your `__init__.py` file in the main workflow directory (e.g. in `aiida/workflows/user/epfl_theos/__init__.py` in the example above):

```
from aiida.orm.workflow import Workflow

def TheosWorkflowFactory(module):
    """
    Return a suitable Workflow subclass for the workflows defined here.
    """
    from aiida.common.pluginloader import BaseFactory

    return BaseFactory(module, Workflow, "aiida.workflows.user.epfl_theos")
```

In this example, a workflow located in e.g. `aiida/workflows/user/epfl_theos/quantumespresso/pw.py` can be loaded simply by typing:

```
TheosWorkflowFactory('quantumespresso.pw')
```

**Note:** The class name of the workflow should be compliant with the `BaseFactory` syntax. In the above example, it should be called `PwWorkflow` otherwise the workflow factory won't work.

You can also customize your verdi shell by adding this function to the modules to be loaded automatically – see [here](#) for more information.

### 3.1.9 Developer Workflow tutorial

#### Creating new workflows

In this section we are going to write a very simple AiiDA workflow. Before starting this tutorial, we assume that you have successfully completed the [Developer calculation plugin tutorial](#) and have your input and output plugins ready to use with this tutorial.

This tutorial creates a workflow for the addition of three numbers. Number could be an integer or a float value. All three numbers will be passed as parameters to the workflow in dictionary format (e.g. `{"a": 1, "b": 2.2, "c": 3}`).

To demonstrate how a workflow works, we will perform the sum of three numbers in two steps:

1. Step 1: `temp_value = a + b`
2. Step 2: `sum = temp_value + c`

A workflow in AiiDA is a python script with several user defined functions called `steps`. All AiiDA functions are available inside “steps” and calculations or sub-workflows can be launched and retrieved. The AiiDA daemon executes a workflow and handles all the operations starting from script loading, error handling and reporting, state monitoring and user interaction with the execution queue. The daemon works essentially as an infinite loop, iterating several simple operations:

1. It checks the running step in all the active workflows, if there are new calculations attached to a step it submits them.
2. It retrieves all the finished calculations. If one step of one workflow exists where all the calculations are correctly finished it reloads the workflow and executes the next step as indicated in the script.
3. If a workflow's next step is the exit one, the workflow is terminated and the report is closed.

**Note:** Since the daemon is aware only of the classes present at the time of its launch, make sure you restart the daemon every time you add a new workflow, or modify an existing one. To restart a daemon, use following command:

```
verdi daemon restart
```

Let's start to write a workflow step by step. First we have to import some packages:

```
from aiida.common import aiida_logger
from aiida.orm.workflow import Workflow
from aiida.orm import Code, Computer
from aiida.orm.data.parameter import ParameterData
from aiida.common.exceptions import InputValidationError
```

In order to write a workflow, we must create a class by extending the `Workflow` class from `aiida.orm.workflow`. This is a fundamental requirement, since the subclassing is the way AiiDA understand if

a class inside the file is an AiiDA workflow or a simple utility class. In the class, you need to re-define an `__init__` method as shown below (in the current code version, this is a requirement). Create a new file, which has the same name as the class you are creating (in this way, it will be possible to load it with `WorkflowFactory`), in this case `addnumbers.py`, with the following content:

```
class AddnumbersWorkflow(Workflow):
    """
    This workflow takes 3 numbers as an input and gives
    its addition as an output.
    Workflow steps:
    passed parameters: a,b,c
    1st step: a + b = step1_result
    2nd step: step1_result + c = final_result
    """

    def __init__(self, **kwargs):
        super(AddnumbersWorkflow, self).__init__(**kwargs)
```

Once the class is defined a user can add methods to generate calculations, download structures or compute new structures starting from a query in previous AiiDA calculations present in the DB. Here we will add simple helper function to validate the input parameters which will be the dictionary with keys `a`, `b` and `c`. All dictionary values should be of type integer or float.

```
def validate_input(self):
    """
    Check if the passed parameters are of type int or float
    else raise exception
    """
    # get parameters passed to workflow when it was
    # initialised. These parameters can not be modified
    # during an execution
    params = self.get_parameters()

    for k in ['a', 'b', 'c']:
        try:
            # check if value is int or float
            if not (isinstance(params[k], int) or isinstance(params[k], float)):
                raise InputValidationError("Value of {} is not of type int or float".format(k))
        except KeyError:
            raise InputValidationError("Missing input key {}".format(k))

    # add in report
    self.append_to_report("Starting workflow with params: {}".format(params))
```

In the above method we have used `append_to_report` workflow method. Once the workflow is launched, the user interactions are limited to some events (stop, relaunch, list of the calculations). So most of the times it is very useful to have custom messages during the execution. Hence, workflow is equipped with a reporting facility `self.append_to_report(string)`, where the user can fill with any text and can retrieve both live and at the end of the execution.

Now we will add the method to launch the actual calculations. We have already done this as part of plugin exercise and hence we do not discuss it in detail here.

```
def get_calculation_sum(self, a, b):
    """
    launch new calculation
    :param a: number
    :param b: number
    :return: calculation object, already stored
```



```

"""
# get code/executable file
codename = 'sum'
code = Code.get_from_string(codename)

computer_name = 'localhost'
computer = Computer.get(computer_name)

# create new calculation
calc = code.new_calc()
calc.set_computer(computer)
calc.label = "Add two numbers"
calc.description = "Calculation step in a workflow to add more than two numbers"
calc.set_max_wallclock_seconds(30*60) # 30 min
calc.set_withmpi(False)
calc.set_resources({"num_machines": 1})

# pass input to the calculation
parameters = ParameterData(dict={'x1': a, 'x2': b,})
calc.use_parameters(parameters)

# store calculation in database
calc.store_all()
return calc

```

Now we will write the first step which is one of the main components in the workflow. In the example below, the start method is decorated with `Workflow.step` making it a very unique kind of method, automatically stored in the database as a container of calculations and sub-workflows.

```

@Workflow.step
def start(self):
    """
    Addition for first two parameters passed to workflow
    when it was initialised
    """

    try:
        self.validate_input()
    except InputValidationError:
        self.next(self.exit)
        return

    # get first parameter passed to workflow when it was initialised.
    a = self.get_parameter("a")
    # get second parameter passed to workflow when it was initialised.
    b = self.get_parameter("b")

    # start first calculation
    calc = self.get_calculation_sum(a, b)

    # add in report
    self.append_to_report("First step calculation is running...")

    # attach calculation in workflow to access in next steps
    self.attach_calculation(calc)

    # go to next step
    self.next(self.stage2)

```

Several functions are available to the user when coding a workflow step, and in the above method we have used basic ones discussed below:

- `self.get_parameters()`: with this method we can retrieve the parameters passed to the workflow when it was initialized. Parameters cannot be modified during an execution, while attributes can be added and removed.
- `self.attach_calculation(calc)`: this is a key point in the workflow, and something possible only inside a step method. Every `JobCalculation`, generated in the method itself or retrieved from other utility methods, is attached to the workflow's step. They are then launched and executed completely by the daemon, without the need of user interaction. Any number of calculations can be attached. The daemon will poll the servers until all the step calculations will be finished, and only after that it will call the next step.
- `self.next(Workflow.step)`: this is the final part of a step, where the user points the engine about what to do after all the calculations in the steps (on possible sub-workflows, as we will see later) are terminated. The argument of this function has to be a `Workflow.step` decorated method of the same workflow class, or in case this is the last step to be executed, you can use the common method `self.exit` which is always present in each `Workflow` subclass. Note that while this call typically occurs at the end of the function, this is not required and you can call the `next()` method as soon as you can decide which method should follow the current one. As it can be seen above, we can use some python logic (`if, ...`) to decide what the next method is going to be (above, we directly point to `self.exit` if the input is invalid).

---

**Note:** remember to call `self.next(self.stage2)` and NOT `self.next(self.stage2())`!! In the first case, we are correctly passing the *method* `stage2` to `next`. In the second case we are instead immediately running the `stage2` method, something we do not want to do (we need to wait for the current step to finish), and passing its *return value* to `self.next` (which is wrong).

---

The above start step calls method `validate_input()` to validate the input parameters. When the workflow will be launched through the `start` method, the AiiDA daemon will load the workflow, execute the step, launch all the calculations and monitor their state.

Now we will create a second step to retrieve the addition of first two numbers from the first step and then we will add the third input number. Once all the calculations in the start step will be finished, the daemon will load and execute the next step i.e. `stage2`, shown below:

```
@Workflow.step
def stage2(self):
    """
    Get result from first calculation and add third value passed
    to workflow when it was initialised
    """
    # get third parameter passed to workflow when it was initialised.
    c = self.get_parameter("c")
    # get result from first calculation
    start_calc = self.get_step_calculations(self.start)[0]

    # add in report
    self.append_to_report("Result of first step calculation is {}".format(
        start_calc.res.sum))

    # start second calculation
    result_calc = self.get_calculation_sum(start_calc.res.sum, c)

    # add in report
    self.append_to_report("Second step calculation is done..")

    # attach calculation in workflow to access in next steps
    self.attach_calculation(result_calc)
```

```
# go to next step
self.next(self.stage3)
```

The new feature used in the above step is:

- `self.get_step_calculations(Workflow.step)`: anywhere after the first step we may need to retrieve and analyze calculations executed in a previous steps. With this method we can have access to the list of calculations of a specific workflows step, passed as an argument.

Now in the last step of the workflow we will retrieve the results from `stage2` and exit the workflow by calling `self.next(self.exit)` method:

```
@Workflow.step
def stage3(self):
    """
    Get the result from second calculation and add it as final
    result of this workflow
    """
    # get result from second calculation
    second_calc = self.get_step_calculations(self.stage2)[0]

    # add in report
    self.append_to_report("Result of second step calculation is {}".format(
        second_calc.res.sum))

    # add workflow result
    self.add_result('value', second_calc.res.sum)

    # add in report
    self.append_to_report("Added value to workflow results")

    # Exit workflow
    self.next(self.exit)
```

The new features used in the above step are:

- `self.add_result()`: When all calculations are done it is useful to tag some of them as results, using custom string to be later searched and retrieved. Similarly to the `get_step_calculations`, this method works on the entire workflow and not on a single step.
- `self.next(self.exit)`: This is the final part of each workflow. Every workflow inherits a fictitious step called `exit` that can be set as a next to any step. As the names suggest, this implies the workflow execution finished correctly.

## Running a workflow

After saving the workflow inside a python file (i.e. `addnumbers.py`) located in the `aiida/workflows` directory, we can launch the workflow simply invoking the specific workflow class and executing the `start()` method inside the `verdi shell` or in a python script (with the AiiDA framework loaded).

**Note:** Don't forget to (re)start your daemon at this point!

In this case, let's use the `verdi shell`. In the shell we execute:

```
AddnumbersWorkflow = WorkflowFactory("addnumbers")
params = {"a":2, "b": 1.4, "c": 1}
```

```
wobject = AddnumbersWorkflow(params=params)
wobject.store()
wobject.start()
```

In the above example we initialized the workflow with input parameters as a dictionary. The `WorkflowFactory` will work only if you gave the correct name both the python file and to the class. Otherwise, you can just substitute that line with a suitable import like:

```
from aiida.orm.workflows.addnumbers import AddnumbersWorkflow
```

We launched the workflow using `start()` method after storing it. Since `start` is a decorated workflow step, the workflow is added to the workflow to the execution queue monitored by the AiiDA daemon.

We now need to know what is going on. There are basically two main ways to see the workflows that are running: by printing the workflow `list` or a single workflow `report`.

- **Workflow list**

From the command line we run:

```
>> verdi workflow list
```

This will list all the running workflows, showing the state of each step and each calculation (and, when present, each sub-workflow). It is the fastest way to have a snapshot of what your AiiDA workflow daemon is working on. An example output right after the `AddnumbersWorkflow` submission should be:

```
+ Workflow AddnumbersWorkflow (pk: 76) is RUNNING [0h:00m:14s ago]
|-* Step: start [->stage2] is RUNNING
| | Calculation ('Number sum', pk: 739) is TOSUBMIT
|
```

The `pk` number of each workflow is reported, a unique ID identifying that specific execution of the workflow, something necessary to retrieve it at any other time in the future (as explained in the next point).

- **Workflow report**

As explained, each workflow is equipped with a reporting facility the user can use to log any intermediate information, useful to debug the state or show some details. Moreover the report is also used by AiiDA as an error reporting tool: in case of errors encountered during the execution, the AiiDA daemon will copy the entire stack trace in the workflow report before halting its execution. To access the report we need the specific `pk` of the workflow. From the command line you would run:

```
verdi workflow report PK_NUMBER
```

while from the verdi shell the same operation requires to use the `get_report()` method:

```
>> load_workflow(PK_NUMBER).get_report()
```

In both variants, `PK_NUMBER` is the `pk` number of the workflow we want the report of. The `load_workflow` function loads a `Workflow` instance from its `pk` number, or from its `uuid` (given as a string).

Once launched, the workflows will be handled by the daemon until the final step or until some error occurs. In the last case, the workflow gets halted and the report can be checked to understand what happened.

- **Workflow result**

As explained, when all the calculations are done it is useful to tag some nodes or quantities as results, using a custom string to be later searched and retrieved. This method works on the entire workflow and not on a single step.

To access the results we need the specific pk of the workflow. From the verdi shell, you can use the `get_report()` method:

```
>> load_workflow(PK_NUMBER).get_results()
```

In both variants, PK\_NUMBER is the pk number of the workflow we want the report of.

- **Killing a workflow**

A user can also kill a workflow while it is running. This can be done with the following verdi command:

```
>> verdi workflow kill PK_NUMBER_1 PK_NUMBER_2 PK_NUMBER_N
```

where several pk numbers can be given. A prompt will ask for a confirmation; this can be avoided by using the `-f` option.

An alternative way to kill an individual workflow is to use the `kill` method. In the verdi shell type:

```
>> load_workflow(PK_NUMBER).kill()
```

## Exercise

In the exercise you have to write a workflow for the addition of six numbers, using the workflow we just wrote as subworkflows.

For this workflow use:

- **Input parameters:** `params = {"w1": {"a": 2, "b": 2.1, "c": 1}, "w2": {"a": 2, "b": 2.1, "c": 4}}`
- **start step:** Use two sub workflows (the ones developed above) for the addition of three numbers:
  - Sub workflow with input w1 and calculate its sum (`temp_result1`)
  - Sub workflow with input w2 and calculate its sum (`temp_result2`)
- **stage2 step:** `final_result = temp_result1 + temp_result2` Add `final_result` to the workflow results and exit the workflow.

Some notes and tips:

- You can attach a subworkflow similarly to how you attach a calculation: in the step, create the new sub-workflow, set its parameters using `set_parameters`, store it, call the `start()` method, and then call `self.attach_workflow(wobject)` to attach it to the current step.
- If you want to pass intermediate data from one step to another, you can set the data as a workflow attribute: in a step, call `self.set_attribute(attr_name, attr_value)`, and retrieve it in another step using `attr_value = self.get_attribute(attr_name)`. Values can be any JSON-serializable value, or an AiiDA node.

### 3.1.10 Verdi command line plugins

AiiDA can be extended by adding custom means of use to interact with it via the command line, by extending the ‘verdi’ commands.

We will describe in particular how to extend `verdi data` by adding a new subcommand.

## Framework for `verdi data`

The code for each of the `verdi data <datatype> <action> [--format <plugin>]` commands is placed in `__<Datatype>` class inside `aiida.cmdline.commands.data.py`. Standard actions, such as

- `list`
- `show`
- `import`
- `export`

are implemented in corresponding classes:

- `Listable`
- `Visualizable`
- `Importable`
- `Exportable`,

which are inherited by `__<Datatype>` classes (multiple inheritance is possible). Actions `show`, `import` and `export` can be extended with new format plugins simply by adding additional methods in `__<Datatype>` (these are automatically detected). Action `list` can be extended by overriding default methods of the `Listable`.

## Adding plugins for `show`, `import`, `export` and like

A plugin to `show`, `import` or `export` the data node can be added by inserting a method to `__<Datatype>` class. Each new method is automatically detected, provided it starts with `__<action>__` (that means `__show__` for `show`, `__import__` for `import` and `__export__` for `export`). Node for each of such method is passed using a parameter.

---

**Note:** plugins for `show` are passed a list of nodes, while plugins for `import` and `export` are passed a single node.

---

As the `--format` option is optional, the default plugin can be specified by setting the value for `__default__<action>__plugin` in the inheriting class, for example:

```
class __Parameter(VerdiCommandWithSubcommands, Visualizable):
    """
    View and manipulate Parameter data classes.
    """

    def __init__(self):
        """
        A dictionary with valid commands and functions to be called.
        """
        from aiida.orm.data.parameter import ParameterData
        self.dataclass = ParameterData
        self.__default_show_format = 'json_date'
        self.valid_subcommands = {
            'show': (self.show, self.complete_visualizers),
        }

    def __show_json_date(self, exec_name, node_list):
        """
        Show contents of ParameterData nodes.
        """
```

If the default plugin is not defined and there are more than one plugin, an exception will be raised upon issuing `verdi data <datatype> <action>` to be caught and explained for the user.

**Plugin-specific command line options** Plugin-specific command line options can be appended in plugin-specific methods `_<action>\_<plugin>\_parameters\(self, parser\)`. All these methods are called before parsing command line arguments, and are passed an `argparse.ArgumentParser` instance, to which command line argument descriptions can be appended using `parser.add_argument()`. For example:

```
def _show_jmol_parameters(self, parser):
    """
    Describe command line parameters.
    """
    parser.add_argument('--step',
                        help="ID of the trajectory step. If none is "
                             "supplied, all steps are exported.",
                        type=int, action='store')
```

**Note:** as all `_<action>\_<plugin>\_parameters\(self, parser\)` methods are called, it requires some attention in order not to make conflicting command line argument names!

**Note:** it's a good practice to set `default=None` for all command line arguments, since `None`-valued arguments are excluded before passing the parsed argument dictionary to a desired plugin.

## Implementing `list`

As listing of data nodes can be extended with filters, controllable using command line parameters, the code of `Listable` is split into a few separate methods, that can be individually overridden:

- **`list`:** the main method, parsing the command line arguments and printing the data node information to the standard output;
- **`query`:** takes the parsed command line arguments and performs a query on the database, returns table of unformatted strings, representing the hits;
- **`append_list_cmdline_arguments`:** informs the command line argument parser about additional, user-defined parameters, used to control the `query` function;
- **`get_column_names`:** returns the names of columns to be printed by `list` method.

## Adding a `verdi` command

Here we will add a new `verdi` command for the `FloatData` datatype we created and used in [Developer code plugin tutorial](#) exercise.

The new command will be:

```
>> verdi data float show <pk>
```

To create the above `verdi` command, we will write a `_Float` class inheriting from both `VerdiCommandWithSubcommands` and `Visualizable` classes; this class will be added inside `aiida.cmdline.commands.data.py` file. By inheriting from `Visualizable`, our class will have a `show()` method, that we can use as the default action for `verdi data float show`:

```

class _Float(VerdiCommandWithSubcommands, Visualizable):
    """
    View and manipulate Float data classes.
    """

    def __init__(self):
        """
        A dictionary with valid commands and functions to be called.
        """
        from aiida.orm.data.float import FloatData

        self.dataclass = FloatData
        self.valid_subcommands = {
            'show': (self.show, self.complete_none),
        }
        self._default_show_format = 'simple'

```

The features used in init method are:

- `self.dataclass`: It is the data type for which the command is written. In this example it is `FloatData`.
- `self.valid_subcommands`: It is the dictionary of valid subcommands and the two functions to be called when the given command is called, or when bash completion is needed. Each key will be the command for the defined data type. For `FloatData` we are therefore adding a `show` command, that will call `self.show()` as method from base class to be called on. We pass `self.complete_none` as completion function to disable further bash completion after the command (this method is defined in the `VerdiCommandWithSubcommands` base class). The `self.show()` method creates a list of all methods of the current class with prefix `_show_` in their name, and provides them as possible formats.
- `self._default_show_format`: It is the default format to be displayed for the `show` command when no specific format is passed as an argument. For `FloatData`, we will show data in a simple format by default. To display node in simple format, we will simply add a method called `_show_simple()` in the `_Float` class. Please note that the method name should follow the convention `_show_ + format_name`.

The `_show_simple()` method will be:

```

def _show_simple(self, exec_name, node_list):
    """
    Show contents of FloatData nodes.
    """
    from aiida.cmdline import print_dictionary

    for node in node_list:
        print node.value

```

In this method we have passed the executable name and the list of nodes. To print `FloatData` in simple format we are just printing the corresponding value on screen.

Once the `_Float` class is added, make sure to add entry in `self.routed_subcommands` dictionary in the `__init__` method of the `Data` class in `aiida.cmdline.commands.data.py` file as shown below.

```

class Data(VerdiCommandRouter):
    """
    Setup and manage data specific types

    There is a list of subcommands for managing specific types of data.
    For instance, 'data upf' manages pseudopotentials in the UPF format.
    """

    def __init__(self):

```



```

"""
A dictionary with valid commands and functions to be called.
"""
## Add here the classes to be supported.
self.routed_subcommands = {
    .
    .
    # other entries
    'float': _Float,
}

```

The new verdi command `float`, is now ready!

Try experimenting by adding other formats for `show` command or by adding other commands like `list`, `import` and `export` for `FloatData` data type.

### 3.1.11 Exporting structures to TCOD

Export of *StructureData* and *CifData* (or any other data type, which can be converted to them) to the **Theoretical Crystallography Open Database** (TCOD) can be divided into following workflow steps:

No.	Description	Input	Output	Type	Imple- mented?
0	Conversion of the StructureData to CifData	<i>StructureData</i>	<i>CifData</i>	In- line	+
1	Detection of the symmetry and reduction to the unit cell	<i>CifData</i>	<i>CifData</i>	In- line	+
2	Niggli reduction of the unit cell	<i>CifData</i>	<i>CifData</i>	In- line	—
3	Addition of structure properties (total energy, residual forces)	<i>CifData</i> , <i>ParameterData</i>	<i>CifData</i>	In- line	PW and CP
4	Addition of the metadata for reproduction of the results	<i>CifData</i>	<i>CifData</i>	In- line	~
5	Deposition to the TCOD	<i>CifData</i>	<i>ParameterData</i>	<i>Job</i>	+

Type of each step's calculation (*InlineCalculation* or *JobCalculation*) defined in column *Type*. Each step is described in more detail below:

- **Conversion of the StructureData to CifData** Conversion between the *StructureData* and *CifData* is done via ASE atoms object.
- **Detection of the symmetry and reduction to the unit cell** Detection of the symmetry and reduction to the unit cell is performed using `pyspglib.spplib.refine_cell()` function.
- **Niggli reduction of the unit cell** Reduction of the unit cell to Niggli cell is a *nice to have* feature, as it would allow to represent structure as an unambiguously selected unit cell.
- **Addition of structure properties (energy, remaining forces)** The structure properties from the calculations, such as total energy and residual forces can be extracted from *ParameterData* nodes and put into related **TCOD CIF dictionaries** tags using calculation-specific parameter translator, derived from *BaseTcodtranslator*.
- **Addition of the metadata for reproduction of the results** Current metadata, added for reproducibility, includes scripts for re-running of calculations, outputs from the calculations and exported subset of AiiDA database. It's not quite clear what/how to record the metadata for calculations of type *InlineCalculation*.

- **Deposition to the TCOD** Deposition of the final *CifData* to the TCOD is performed using *cif\_cod\_deposit* script from *cod-tools* package.

### 3.1.12 GIT cheatsheet

Excellent and thorough documentation on how to use GIT can be found online on the official GIT documentation or by searching on Google. We summarize here only a set of commands that may be useful.

#### Interesting online resources

- [Atlassian forking-workflow guide](#)
- [Gitflow model](#)

#### Set the push default behavior to push only the current branch

The default push behavior may not be what you expect: if a branch you are not working on changes, you may not be able to push your own branch, because git tries to check them all. To avoid this, use:

```
git config push.default upstream
```

to set the default push.default behaviour to push the current branch to its upstream branch. Note the actual string to set depends on the version of git; newer versions allow to use:

```
git config push.default simple
```

which is better; see also discussion on [this stackoverflow page](#).

#### View commits that would be pushed

If you want to see which commits would be sent to the remote repository upon a `git push` command, you can use (e.g. if you want to compare with the `origin/develop` remote branch):

```
git log origin/develop..HEAD
```

to see the logs of the commits, or:

```
git diff origin/develop..HEAD
```

to see also the differences among the current HEAD and the version on `origin/develop`.

#### Switch to another branch

You can switch to another branch with:

```
git checkout newbranchname
```

and you can see the list of checked-out branches, and the one you are in, with:

```
git branch
```

(or `git branch -a` to see also the list of remote branches).

## Associate a local and remote branch

To tell GIT to always push a local branch (checked-out) to a remote branch called `remotebranchname`, check out the correct local branch and then do:

```
git push --set-upstream origin remotebranchname
```

From now on, you will just need to run `git push`. This will create a new entry in `.git/config` similar to:

```
[branch "localbranchname"]
  remote = origin
  merge = refs/heads/remotebranchname
```

## Branch renaming

To rename a branch *locally*, from `oldname` to `newname`:

```
git checkout oldname
git branch -m oldname newname
```

If you want also to rename it remotely, you have to create a new branch and then delete the old one. One way to do it, is first editing `~/.git/config` so that the branch points to the new remote name, changing `refs/heads/oldname` to `refs/heads/newname` in the correct section:

```
[branch "newname"]
  remote = origin
  merge = refs/heads/newname
```

Then, do a:

```
git push origin newname
```

to create the new branch, and finally delete the old one with:

```
git push origin :oldname
```

(notice the `:` symbol). Note that if you are working e.g. on BitBucket, there may be a filter to disallow the deletion of branches (check in the repository settings, and then under “Branch management”). Moreover, the “Main branch” (set in the repository settings, under “Repository details”) cannot be deleted.

## Create a new (lightweight) tag

If you want to create a new tag, e.g. for a new version, and you have checked out the commit that you want to tag, simply run:

```
git tag TAGNAME
```

(e.g., `git tag v0.2.0`). Afterwards, remember to push the tag to the remote repository (otherwise it will remain only local):

```
git push --tags
```

## Create a new branch from a given tag

This will create a new `newbranchname` branch starting from tag `v0.2.0`:

```
git checkout -b newbranchname v0.2.0
```

Then, if you want to push the branch remotely and have git remember the association:

```
git push --set-upstream origin remotebranchname
```

(for the meaning of `--set-upstream` see the section *Associate a local and remote branch* above).

## Disallow a branch deletion, or committing to a branch, on BitBucket

You can find these settings in the repository settings of the web interface, and then under “Branch management”.

**Note:** if you commit to a branch (locally) and then discover that you cannot push (e.g. you mistakenly committed to the master branch), you can remove your last commit using:

```
git reset --hard HEAD~1
```

(this removes one commit only, and you should have no local modifications; if you do it, be sure to avoid losing your modifications!)

## Merge from a different repository

It is possible to do a pull request of a forked repository from the BitBucket web interface. However, if one just wants to keep in sync, e.g., the main AiiDA repository with a fork you are working into without creating a pull request (e.g., for daily merge of your fork’s develop into the main repo’s develop), you can:

- commit and pull all your changes in your fork
- from the BitBucket web interface, sync your fork with the main repository, if needed
- go in a local cloned version of the main repository
- [only the first time] add a remote pointing to the new repository, with the name you prefer (here: `myfork`):

```
git remote add myfork git@bitbucket.org:BUTBUCKETUSER/FORKEDREPO.git
```

- checkout to the correct branch you want to merge into (`git checkout develop`)
- do a `git pull` (just in case)
- Fetch the correct branch of the other repository (e.g., the develop branch):

```
git fetch myfork develop
```

(this will fetch that branch into a temporary location called `FETCH_HEAD`).

- Merge the modifications:

```
git merge FETCH_HEAD
```

- Fix any merge conflicts (if any) and commit.
- Finally, push the merged result into the main repository:

```
git push
```

(or, if you did not use the default remote with `--set-upstream`, specify the correct remote branch, e.g. `git push origin develop`).

---

**Note:** If you want to fetch and transfer also tags, use instead:

```
git fetch -t myfork develop
git merge FETCH_HEAD
git push --tags
```

to get the tags from myfork and then push them in the current repository.

---

### 3.1.13 Sphinx cheatsheet

A brief overview of some of the main functions of Sphinx as used in the aiiida documentation. View [This Page](#) to see how this page was formatted. This is only a brief outline for more please see [the Sphinx documentation](#)

#### Main Titles and Subtitles

This is an example of a main title.

#### subtitles are made like this

This is an example of a subtitle.

#### Formatting

##### Basic Paragraph Formatting

Words can be written in *italics* or in **bold**. Text describing a specific `computer_thing` can be formatted as well.

##### Paragraph and Indentation

Much like in regular python, the indentation plays a strong role in the formatting.

For example all of this sentence will appear on the same line.

**While this sentence will appear** differently because there is an indent.

##### Terminal and Code Formatting

Something to be run in command line can be formatted like this:

```
>> Some command
```

As can be seen above, while snippets of python on code can be done like this:

```
import module
print('hello world')
```

## Notes

---

**Note:** Notes can be added like this.

---

## Bullet Points and Lists

- Bullet points can be added
  - Just like this \* With sub-bullets like this
1. While numerical bullets
  2. Can be added
  3. Like this

## Links, Code Display, Cross References

### External Links

Can be done like here for AiiDA

### Code Download

Code can be downloaded like this.

Download: `this example script`

### Code Display

Can be done like this. This entire document can be seen unformatted below using this method.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import json
import sys

in_file = sys.argv[1]
out_file = sys.argv[2]

print "Some output from the code"

with open(in_file) as f:
    in_dict = json.load(f)

out_dict = { 'sum':in_dict['x1']+in_dict['x2'] }

with open(out_file,'w') as f:
    json.dump(out_dict,f)
```

## Cross Reference Docs

Here is an example of a reference to the *StructureData tutorial* which is on *another page*

Here is an example of a reference to something on the same page, *Cross Reference Docs*

---

**Note:** References within the same document need a reference label, see `.. _self-reference:` used in this section for an example. *Hidden in formatted page, can only be seen in the input text.*

---

## Cross Reference Classes and Methods

Any class can be referenced for example *StructureData* references the StructureData data class.

Similarly any method can be referenced for example *append\_atom()* shows the StructureData class' append atom method.

## Table of Contents Docs and Code

### Table of Contents for Docs

An example of the table of contents syntax for the *GIT cheatsheet* can be seen here note that these are especially important in the global structure of the document, as found in index.rst files.

**GIT cheatsheet** Excellent and thorough documentation on how to use GIT can be found online on the official GIT documentation or by searching on Google. We summarize here only a set of commands that may be useful.

### Interesting online resources

- [Atlassian forking-workflow guide](#)
- [Gitflow model](#)

**Set the push default behavior to push only the current branch** The default push behavior may not be what you expect: if a branch you are not working on changes, you may not be able to push your own branch, because git tries to check them all. To avoid this, use:

```
git config push.default upstream
```

to set the default push.default behaviour to push the current branch to its upstream branch. Note the actual string to set depends on the version of git; newer versions allow to use:

```
git config push.default simple
```

which is better; see also discussion on [this stackoverflow page](#).

**View commits that would be pushed** If you want to see which commits would be sent to the remote repository upon a `git push` command, you can use (e.g. if you want to compare with the `origin/develop` remote branch):

```
git log origin/develop..HEAD
```

to see the logs of the commits, or:

```
git diff origin/develop..HEAD
```

to see also the differences among the current HEAD and the version on origin/develop.

**Switch to another branch** You can switch to another branch with:

```
git checkout newbranchname
```

and you can see the list of checked-out branches, and the one you are in, with:

```
git branch
```

(or `git branch -a` to see also the list of remote branches).

**Associate a local and remote branch** To tell GIT to always push a local branch (checked-out) to a remote branch called `remotebranchname`, check out the correct local branch and then do:

```
git push --set-upstream origin remotebranchname
```

From now on, you will just need to run `git push`. This will create a new entry in `.git/config` similar to:

```
[branch "localbranchname"]
  remote = origin
  merge = refs/heads/remotebranchname
```

**Branch renaming** To rename a branch *locally*, from `oldname` to `newname`:

```
git checkout oldname
git branch -m oldname newname
```

If you want also to rename it remotely, you have to create a new branch and then delete the old one. One way to do it, is first editing `~/.git/config` so that the branch points to the new remote name, changing `refs/heads/oldname` to `refs/heads/newname` in the correct section:

```
[branch "newname"]
  remote = origin
  merge = refs/heads/newname
```

Then, do a:

```
git push origin newname
```

to create the new branch, and finally delete the old one with:

```
git push origin :oldname
```

(notice the `:` symbol). Note that if you are working e.g. on BitBucket, there may be a filter to disallow the deletion of branches (check in the repository settings, and then under “Branch management”). Moreover, the “Main branch” (set in the repository settings, under “Repository details”) cannot be deleted.

**Create a new (lightweight) tag** If you want to create a new tag, e.g. for a new version, and you have checked out the commit that you want to tag, simply run:

```
git tag TAGNAME
```



(e.g., `git tag v0.2.0`). Afterwards, remember to push the tag to the remote repository (otherwise it will remain only local):

```
git push --tags
```

**Create a new branch from a given tag** This will create a new `newbranchname` branch starting from tag `v0.2.0`:

```
git checkout -b newbranchname v0.2.0
```

Then, if you want to push the branch remotely and have git remember the association:

```
git push --set-upstream origin remotebranchname
```

(for the meaning of `--set-upstream` see the section [Associate a local and remote branch](#) above).

**Disallow a branch deletion, or committing to a branch, on BitBucket** You can find these settings in the repository settings of the web interface, and then under “Branch management”.

**Note:** if you commit to a branch (locally) and then discover that you cannot push (e.g. you mistakenly committed to the master branch), you can remove your last commit using:

```
git reset --hard HEAD~1
```

(this removes one commit only, and you should have no local modifications; if you do it, be sure to avoid losing your modifications!)

**Merge from a different repository** It is possible to do a pull request of a forked repository from the BitBucket web interface. However, if one just wants to keep in sync, e.g., the main AiiDA repository with a fork you are working into without creating a pull request (e.g., for daily merge of your fork’s develop into the main repo’s develop), you can:

- commit and pull all your changes in your fork
- from the BitBucket web interface, sync your fork with the main repository, if needed
- go in a local cloned version of the main repository
- [*only the first time*] add a remote pointing to the new repository, with the name you prefer (here: `myfork`):

```
git remote add myfork git@bitbucket.org:BUTBUCKETUSER/FORKEDREPO.git
```

- checkout to the correct branch you want to merge into (`git checkout develop`)
- do a `git pull` (just in case)
- Fetch the correct branch of the other repository (e.g., the develop branch):

```
git fetch myfork develop
```

(this will fetch that branch into a temporary location called `FETCH_HEAD`).

- Merge the modifications:

```
git merge FETCH_HEAD
```

- Fix any merge conflicts (if any) and commit.
- Finally, push the merged result into the main repository:

```
git push
```

(or, if you did not use the default remote with `--set-upstream`, specify the correct remote branch, e.g. `git push origin develop`).

**Note:** If you want to fetch and transfer also tags, use instead:

```
git fetch -t myfork develop
git merge FETCH_HEAD
git push --tags
```

to get the tags from myfork and then push them in the current repository.

**Note:** The *maxdepth* parameter can be used to change how deep the title indexing goes. See [This Page](#).

---

## Table of Contents for Code

Table of contents, that cross reference code, can be done very similarly to how it is done for documents. For example the parser docs can be indexed like this

**ORM documentation: generic aiida.orm** This section describes the aiida/django object-relational mapping.

Some generic methods of the module aiida.orm

`aiida.orm.CalculationFactory` (*module*, *from\_abstract=False*)

Return a suitable JobCalculation subclass.

### Parameters

- **module** – a valid string recognized as a Calculation plugin
- **from\_abstract** – A boolean. If False (default), actually look only to subclasses to JobCalculation, not to the base Calculation class. If True, check for valid strings for plugins of the Calculation base class.

`aiida.orm.DataFactory` (*module*)

Return a suitable Data subclass.

`aiida.orm.WorkflowFactory` (*module*)

Return a suitable Workflow subclass.

`aiida.orm.load_node` (*node\_id=None*, *pk=None*, *uuid=None*, *parent\_class=None*)

Return an AiiDA node given PK or UUID.

### Parameters

- **node\_id** – PK (integer) or UUID (string) or a node
- **pk** – PK of a node
- **uuid** – UUID of a node
- **parent\_class** – if specified, checks whether the node loaded is a subclass of `parent_class`

**Returns** an AiiDA node

**Raises**

- **ValueError** – if none or more than one of parameters is supplied or type of `node_id` is neither string nor integer.
- **NotExistent** – if the `parent_class` is specified and no matching Node is found.

`aiida.orm.load_workflow(wf_id=None, pk=None, uuid=None)`

Return an AiiDA workflow given PK or UUID.

#### Parameters

- **wf\_id** – PK (integer) or UUID (string) or a workflow
- **pk** – PK of a workflow
- **uuid** – UUID of a workflow

**Returns** an AiiDA workflow

**Raises** ValueError if none or more than one of parameters is supplied or type of `wf_id` is neither string nor integer

### Computer

**class** `aiida.orm.computer.Computer(**kwargs)`

Base class to map a node in the DB + its permanent repository counterpart.

Stores attributes starting with an underscore.

Caches files and attributes before the first save, and saves everything only on `store()`. After the call to `store()`, in general attributes cannot be changed, except for those listed in the `self._updatable_attributes` tuple (empty for this class, can be extended in a subclass).

Only after storing (or upon loading from `uuid`) metadata can be modified and in this case they are directly set on the db.

In the plugin, also set the `_plugin_type_string`, to be set in the DB in the ‘type’ field.

**copy()**

Return a copy of the current object to work with, not stored yet.

**full\_text\_info**

Return a (multiline) string with a human-readable detailed information on this computer.

**classmethod get (computer)**

Return a computer from its name (or from another Computer or DbComputer instance)

**get\_dbauthinfo (user)**

Return the `aiida.djsite.db.models.DbAuthInfo` instance for the given user on this computer, if the computer is not configured for the given user.

**Parameters** **user** – a `DbUser` instance.

**Returns** a `aiida.djsite.db.models.DbAuthInfo` instance

**Raises** **NotExistent** if the computer is not configured for the given user.

**get\_default\_mpiprocs\_per\_machine()**

Return the default number of CPUs per machine (node) for this computer, or None if it was not set.

**get\_mpirun\_command()**

Return the mpirun command. Must be a list of strings, that will be then joined with spaces when submitting.

I also provide a sensible default that may be ok in many cases.

**is\_user\_configured** (*user*)

Return True if the computer is configured for the given user, False otherwise.

**Parameters** *user* – a DbUser instance.

**Returns** a boolean.

**is\_user\_enabled** (*user*)

Return True if the computer is enabled for the given user (looking only at the per-user setting: the computer could still be globally disabled).

**Note** Return False also if the user is not configured for the computer.

**Parameters** *user* – a DbUser instance.

**Returns** a boolean.

**classmethod list\_names** ()

Return a list with all the names of the computers in the DB.

**logging** = <module 'logging' from '/usr/lib/python2.7/logging/\_\_init\_\_.pyc'>

**pk**

Return the principal key in the DB.

**set\_default\_mpiprocs\_per\_machine** (*def\_cpus\_per\_machine*)

Set the default number of CPUs per machine (node) for this computer. Accepts None if you do not want to set this value.

**set\_mpirun\_command** (*val*)

Set the mpirun command. It must be a list of strings (you can use string.split() if you have a single, space-separated string).

**store** ()

Store the computer in the DB.

Differently from Nodes, a computer can be re-stored if its properties are to be changed (e.g. a new mpirun command, etc.)

**uuid**

Return the UUID in the DB.

**validate** ()

Check if the attributes and files retrieved from the DB are valid. Raise a ValidationError if something is wrong.

Must be able to work even before storing: therefore, use the get\_attr and similar methods that automatically read either from the DB or from the internal attribute cache.

For the base class, this is always valid. Subclasses will reimplement this. In the subclass, always call the super().validate() method first!

**aiida.orm.computer.delete\_computer** (*computer*)

Delete a computer from the DB. It assumes that the DB backend does the proper checks and avoids to delete computers that have nodes attached to them.

Implemented as a function on purpose, otherwise complicated logic would be needed to set the internal state of the object after calling computer.delete().

## Node

**class** **aiida.orm.node.AttributeManager** (*node*)

An object used internally to return the attributes as a dictionary.

**Note** Important! It cannot be used to change variables, just to read them. To change values (of unstored nodes), use the proper Node methods.

**\_\_init\_\_** (*node*)

**Parameters** *node* – the node object.

**class** `aiida.orm.node.Node` (*\*\*kwargs*)

Base class to map a node in the DB + its permanent repository counterpart.

Stores attributes starting with an underscore.

Caches files and attributes before the first save, and saves everything only on `store()`. After the call to `store()`, in general attributes cannot be changed, except for those listed in the `self._updatable_attributes` tuple (empty for this class, can be extended in a subclass).

Only after storing (or upon loading from uuid) extras can be modified and in this case they are directly set on the db.

In the plugin, also set the `_plugin_type_string`, to be set in the DB in the ‘type’ field.

**\_\_init\_\_** (*\*\*kwargs*)

Initialize the object Node.

**Parameters** *uuid* (*optional*) – if present, the Node with given uuid is loaded from the database. (It is not possible to assign a uuid to a new Node.)

**add\_comment** (*content*, *user=None*)

Add a new comment.

**Parameters** *content* – string with comment

**add\_path** (*src\_abs*, *dst\_path*)

Copy a file or folder from a local file inside the repository directory. If there is a subpath, folders will be created.

Copy to a cache directory if the entry has not been saved yet.

**Parameters**

- **src\_abs** (*str*) – the absolute path of the file to copy.
- **dst\_filename** (*str*) – the (relative) path on which to copy.

**Todo** in the future, add an `add_attachment()` that has the same meaning of a extras file. Decide also how to store. If in two separate subfolders, remember to reset the limit.

**attrs** ()

Returns the keys of the attributes.

**Returns** a list of strings

**copy** ()

Return a copy of the current object to work with, not stored yet.

This is a completely new entry in the DB, with its own UUID. Works both on stored instances and with not-stored ones.

Copies files and attributes, but not the extras. Does not store the Node to allow modification of attributes.

**Returns** an object copy

**ctime**

Return the creation time of the node.

**dbnode**

**Returns** the corresponding Django DbNode object.

**del\_extra** (*key*)

Delete a extra, acting directly on the DB! The action is immediately performed on the DB. Since extras can be added only after storing the node, this function is meaningful to be called only after the .store() method.

**Parameters** **key** (*str*) – key name

**Raise** AttributeError: if key starts with underscore

**Raise** ModificationNotAllowed: if the node is not stored yet

**description**

Get the description of the node.

**Returns** a string

**extras** ()

Get the keys of the extras.

**Returns** a list of strings

**folder**

Get the folder associated with the node, whether it is in the temporary or the permanent repository.

**Returns** the RepositoryFolder object.

**get\_abs\_path** (*path=None, section=None*)

Get the absolute path to the folder associated with the Node in the AiiDA repository.

**Parameters**

- **path** (*str*) – the name of the subfolder inside the section. If None returns the abspath of the folder. Default = None.
- **section** – the name of the subfolder ('path' by default).

**Returns** a string with the absolute path

For the moment works only for one kind of files, 'path' (internal files)

**get\_attr** (*key, \*args*)

Get the attribute.

**Parameters**

- **key** – name of the attribute
- **value** (*optional*) – if no attribute key is found, returns value

**Returns** attribute value

**Raises**

- **IndexError** – If no attribute is found and there is no default
- **ValueError** – If more than two arguments are passed to get\_attr

**get\_attrs** ()

Return a dictionary with all attributes of this node.

**get\_comments** (*pk=None*)

Return a sorted list of comment values, one for each comment associated to the node.

**Parameters** **pk** – integer or list of integers. If it is specified, returns the comment values with desired pks. (pk refers to DbComment.pk)

**Returns** the list of comments, sorted by pk; each element of the list is a dictionary, containing (pk, email, ctime, mtime, content)

**get\_computer** ()

Get the computer associated to the node.

**Returns** the Computer object or None.

**get\_extra** (key, \*args)

Get the value of a extras, reading directly from the DB! Since extras can be added only after storing the node, this function is meaningful to be called only after the .store() method.

**Parameters**

- **key** (str) – key name
- **value** (optional) – if no attribute key is found, returns value

**Returns** the key value

**Raises ValueError** If more than two arguments are passed to get\_extra

**get\_extras** ()

Get the value of extras.

**Returns** the dictionary of extras ({} if no extras)

**get\_folder\_list** (subfolder='.')

Get the the list of files/directory in the repository of the object.

**Parameters** **subfolder** (str,optional) – get the list of a subfolder

**Returns** a list of strings.

**get\_inputs** (type=None, also\_labels=False, only\_in\_db=False)

Return a list of nodes that enter (directly) in this node

**Parameters**

- **type** – If specified, should be a class, and it filters only elements of that specific type (or a subclass of 'type')
- **also\_labels** – If False (default) only return a list of input nodes. If True, return a list of tuples, where each tuple has the following format: ('label', Node), with 'label' the link label, and Node a Node instance or subclass
- **only\_in\_db** – Return only the inputs that are in the database, ignoring those that are in the local cache. Otherwise, return all links.

**get\_inputs\_dict** (only\_in\_db=False)

Return a dictionary where the key is the label of the input link, and the value is the input node.

**Returns** a dictionary {label:object}

**get\_outputs** (type=None, also\_labels=False)

Return a list of nodes that exit (directly) from this node

**Parameters**

- **type** – if specified, should be a class, and it filters only elements of that specific type (or a subclass of 'type')
- **also\_labels** – if False (default) only return a list of input nodes. If True, return a list of tuples, where each tuple has the following format: ('label', Node), with 'label' the link label, and Node a Node instance or subclass

**get\_outputs\_dict ()**

Return a dictionary where the key is the label of the output link, and the value is the input node. As some Nodes (Datas in particular) can have more than one output with the same label, all keys have the name of the link with appended the pk of the node in output. The key without pk appended corresponds to the oldest node.

**Returns** a dictionary {linkname:object}

**classmethod get\_subclass\_from\_pk (pk)**

Get a node object from the pk, with the proper subclass of Node. (integer primary key used in this database), but loading the proper subclass where appropriate.

**Parameters** **pk** – a string with the pk of the object to be loaded.

**Returns** the object of the proper subclass.

**Raise** `NotExist`: if there is no entry of the desired object kind with the given pk.

**classmethod get\_subclass\_from\_uuid (uuid)**

Get a node object from the uuid, with the proper subclass of Node. (if `Node(uuid=...)` is called, only the Node class is loaded).

**Parameters** **uuid** – a string with the uuid of the object to be loaded.

**Returns** the object of the proper subclass.

**Raise** `NotExist`: if there is no entry of the desired object kind with the given uuid.

**get\_user ()**

Get the user.

**Returns** a Django `DbUser` model object

**has\_children**

Property to understand if children are attached to the node :return: a boolean

**has\_parents**

Property to understand if parents are attached to the node :return: a boolean

**inp**

Traverse the graph of the database. Returns a databaseobject, linked to the current node, by means of the linkname. Example: `B = A.inp.parameters`: returns the object (B), with link from B to A, with linkname parameters `C = A.inp`: returns an `InputManager`, an object that is meant to be accessed as the previous example

**iterattrs (also\_updatable=True)**

Iterator over the attributes, returning tuples (key, value)

**Todo** optimize! At the moment, the call is very slow because it is also calling `attr.getvalue()` for each attribute, that has to perform complicated queries to rebuild the object.

**Parameters** **also\_updatable** (*bool*) – if False, does not iterate over attributes that are updatable

**iterextras ()**

Iterator over the extras, returning tuples (key, value)

**Todo** verify that I am not creating a list internally

**label**

Get the label of the node.

**Returns** a string.



**logger**

Get the logger of the Node object.

**Returns** Logger object

**mtime**

Return the modification time of the node.

**out**

Traverse the graph of the database. Returns a databaseobject, linked to the current node, by means of the linkname. Example: `B = A.out.results`: Returns the object B, with link from A to B, with linkname parameters

**pk**

**Returns** the principal key (the ID) as an integer, or None if the node was not stored yet

**classmethod query** (*\*args*, *\*\*kwargs*)

Map to the aiiDAobjects manager of the DbNode, that returns Node objects (or their subclasses) instead of DbNode entities.

# TODO: VERY IMPORTANT: the recognition of a subclass from the type # does not work if the modules defining the subclasses are not # put in subfolders. # In the future, fix it either to make a cache and to store the # full dependency tree, or save also the path.

**remove\_path** (*path*)

Remove a file or directory from the repository directory. Can be called only before storing.

**Parameters** **path** (*str*) – relative path to file/directory.

**set** (*\*\*kwargs*)

For each `k=v` pair passed as kwargs, call the corresponding `set_k(v)` method (e.g., calling `self.set(property=5, mass=2)` will call `self.set_property(5)` and `self.set_mass(2)`). Useful especially in the `__init__`.

**Note** it uses the `_set_incompatibilities` list of the class to check that we are not setting methods that cannot be set at the same time. `_set_incompatibilities` must be a list of tuples, and each tuple specifies the elements that cannot be set at the same time. For instance, if `_set_incompatibilities = [('property', 'mass')]`, then the call `self.set(property=5, mass=2)` will raise a `ValueError`. If a tuple has more than two values, it raises `ValueError` if *all* keys are provided at the same time, but it does not give any error if at least one of the keys is not present.

**Note** If one element of `_set_incompatibilities` is a tuple with only one element, this element will not be settable using this function (and in particular,

**Raises ValueError** if the corresponding `set_k` method does not exist in self, or if the methods cannot be set at the same time.

**set\_computer** (*computer*)

Set the computer to be used by the node.

Note that the computer makes sense only for some nodes: Calculation, RemoteData, ...

**Parameters** **computer** – the computer object

**set\_extra** (*key*, *value*, *exclusive=False*)

Immediately sets an extra of a calculation, in the DB! No `.store()` to be called. Can be used *only* after saving.

**Parameters**

- **key** (*string*) – key name

- **value** – key value
- **exclusive** – (default=False). If exclusive is True, it raises a UniquenessError if an Extra with the same name already exists in the DB (useful e.g. to “lock” a node and avoid to run multiple times the same computation on it).

**Raises UniquenessError** if extra already exists and exclusive is True.

**set\_extras** (*the\_dict*)

Immediately sets several extras of a calculation, in the DB! No .store() to be called. Can be used *only* after saving.

**Parameters the\_dict** – a dictionary of key:value to be set as extras

**store** (*with\_transaction=True*)

Store a new node in the DB, also saving its repository directory and attributes.

Can be called only once. Afterwards, attributes cannot be changed anymore! Instead, extras can be changed only AFTER calling this store() function.

**Note** After successful storage, those links that are in the cache, and for which also the parent node is already stored, will be automatically stored. The others will remain unstored.

**Parameters with\_transaction** – if False, no transaction is used. This is meant to be used ONLY if the outer calling function has already a transaction open!

**store\_all** (*with\_transaction=True*)

Store the node, together with all input links, if cached, and also the linked nodes, if they were not stored yet.

**Parameters with\_transaction** – if False, no transaction is used. This is meant to be used ONLY if the outer calling function has already a transaction open!

**uuid**

**Returns** a string with the uuid

**class** `aiida.orm.node.NodeInputManager` (*node*)

To document

`__init__` (*node*)

**Parameters node** – the node object.

**class** `aiida.orm.node.NodeOutputManager` (*node*)

To document

`__init__` (*node*)

**Parameters node** – the node object.

`aiida.orm.node.from_type_to_pluginclassname` (*typestr*)

Return the string to pass to the load\_plugin function, starting from the ‘type’ field of a Node.

## Workflow

**class** `aiida.orm.workflow.Workflow` (*\*\*kwargs*)

Base class to represent a workflow. This is the superclass of any workflow implementations, and provides all the methods necessary to interact with the database.

The typical use case are workflow stored in the aiida.workflow packages, that are initiated either by the user in the shell or by some scripts, and that are monitored by the aiida daemon.

Workflow can have steps, and each step must contain some calculations to be executed. At the end of the step’s calculations the workflow is reloaded in memory and the next methods is called.

**add\_attribute** (*\_name*, *\_value*)

Add one attributes to the Workflow. If another attribute is present with the same name it will be overwritten.  
:param name: a string with the attribute name to store :param value: a storable object to store

**add\_attributes** (*\_params*)

Add a set of attributes to the Workflow. If another attribute is present with the same name it will be overwritten.  
:param name: a string with the attribute name to store :param value: a storable object to store

**add\_path** (*src\_abs*, *dst\_path*)

Copy a file or folder from a local file inside the repository directory. If there is a subpath, folders will be created.

Copy to a cache directory if the entry has not been saved yet. *src\_abs*: the absolute path of the file to copy.  
*dst\_filename*: the (relative) path on which to copy.

**add\_result** (*\_name*, *\_value*)

Add one result to the Workflow. If another result is present with the same name it will be overwritten.  
:param name: a string with the result name to store :param value: a storable object to store

**add\_results** (*\_params*)

Add a set of results to the Workflow. If another result is present with the same name it will be overwritten.  
:param name: a string with the result name to store :param value: a storable object to store

**append\_to\_report** (*text*)

Adds text to the Workflow report.

**Note** Once, in case the workflow is a subworkflow of any other Workflow this method calls the parent `append_to_report` method; now instead this is not the case anymore

**attach\_calculation** (*calc*)

Adds a calculation to the caller step in the database. This is a lazy call, no calculations will be launched until the `next` method gets called. For a step to be completed all the calculations linked have to be in RETRIEVED state, after which the next method gets called from the workflow manager. :param calc: a JobCalculation object :raise: AiidaException: in case the input is not of JobCalculation type

**attach\_workflow** (*sub\_wf*)

Adds a workflow to the caller step in the database. This is a lazy call, no workflow will be started until the `next` method gets called. For a step to be completed all the workflows linked have to be in FINISHED state, after which the next method gets called from the workflow manager. :param next\_method: a Workflow object

**clear\_report** ()

Wipe the Workflow report. In case the workflow is a subworkflow of any other Workflow this method calls the parent `clear_report` method.

**current\_folder**

Get the current repository folder, whether the temporary or the permanent.

**Returns** the RepositoryFolder object.

**dbworkflowinstance**

Get the DbWorkflow object stored in the super class.

**Returns** DbWorkflow object from the database

**description**

Get the description of the workflow.

**Returns** a string

**exit()**

This is the method to call in `next` to finish the Workflow. When `exit` is the next method, and no errors are found, the Workflow is set to `FINISHED` and removed from the execution manager duties.

**get\_abs\_path** (*path*, *section=None*)

TODO: For the moment works only for one kind of files, 'path' (internal files)

**get\_all\_calcs** (*calc\_class=<class 'aiida.orm.calculation.job.JobCalculation'>*, *calc\_state=None*, *depth=16*)

Get all calculations connected with this workflow and all its subworkflows up to a given depth. The list of calculations can be restricted to a given calculation type and state :param *calc\_class*: the calculation class to which the calculations should belong (default: `JobCalculation`)

**Parameters**

- **calc\_state** – a specific state to filter the calculations to retrieve
- **depth** – the maximum depth level the recursion on sub-workflows will try to reach (0 means we stay at the step level and don't go into sub-workflows, 1 means we go down to one step level of the sub-workflows, etc.)

**Returns** a list of `JobCalculation` objects

**get\_attribute** (*\_name*)

Get one Workflow attribute :param *name*: a string with the attribute name to retrieve :return: a dictionary of storable objects

**get\_attributes** ()

Get the Workflow attributes :return: a dictionary of storable objects

**get\_folder\_list** (*subfolder='.'*)

Get the the list of files/directory in the repository of the object.

**Parameters** **subfolder** (*str, optional*) – get the list of a subfolder

**Returns** a list of strings.

**get\_parameter** (*\_name*)

Get one Workflow parameter :param *name*: a string with the parameters name to retrieve :return: a dictionary of storable objects

**get\_parameters** ()

Get the Workflow parameters :return: a dictionary of storable objects

**get\_report** ()

Return the Workflow report.

**Note** once, in case the workflow is a subworkflow of any other Workflow this method calls the parent `get_report` method. This is not the case anymore.

**Returns** a list of strings

**get\_result** (*\_name*)

Get one Workflow result :param *name*: a string with the result name to retrieve :return: a dictionary of storable objects

**get\_results** ()

Get the Workflow results :return: a dictionary of storable objects

**get\_state** ()

Get the Workflow's state :return: a state from `wf_states` in `aiida.common.datastructures`

**get\_step** (*step\_method*)

Retrieves by name a step from the Workflow. :param *step\_method*: a string with the name of the step to

retrieve or a method :raise: `ObjectDoesNotExist`: if there is no step with the specific name. :return: a `DbWorkflowStep` object.

**get\_step\_calculations** (*step\_method*, *calc\_state=None*)

Retrieves all the calculations connected to a specific step in the database. If the step is not existent it returns `None`, useful for simpler grammatic in the workflow definition. :param *next\_method*: a Workflow step (decorated) method :param *calc\_state*: a specific state to filter the calculations to retrieve :return: a list of `JobCalculations` objects

**get\_step\_workflows** (*step\_method*)

Retrieves all the workflows connected to a specific step in the database. If the step is not existent it returns `None`, useful for simpler grammatic in the workflow definition. :param *next\_method*: a Workflow step (decorated) method

**get\_steps** (*state=None*)

Retrieves all the steps from a specific workflow Workflow with the possibility to limit the list to a specific step's state. :param *state*: a state from `wf_states` in `aiida.common.datastructures` :return: a list of `DbWorkflowStep` objects.

**classmethod get\_subclass\_from\_dbnode** (*wf\_db*)

Loads the workflow object and reoads the python script in memory with the `importlib` library, the main class is searched and then loaded. :param *wf\_db*: a specific `DbWorkflowNode` object representing the Workflow :return: a Workflow subclass from the specific source code

**classmethod get\_subclass\_from\_pk** (*pk*)

Calls the `get_subclass_from_dbnode` selecting the `DbWorkflowNode` from the input *pk*. :param *pk*: a primary key index for the `DbWorkflowNode` :return: a Workflow subclass from the specific source code

**classmethod get\_subclass\_from\_uuid** (*uuid*)

Calls the `get_subclass_from_dbnode` selecting the `DbWorkflowNode` from the input *uuid*. :param *uuid*: a uuid for the `DbWorkflowNode` :return: a Workflow subclass from the specific source code

**get\_temp\_folder** ()

Get the folder of the Node in the temporary repository.

**Returns** a `SandboxFolder` object mapping the node in the repository.

**has\_failed** ()

Returns True is the Workflow's state is `ERROR`

**has\_finished\_ok** ()

Returns True is the Workflow's state is `FINISHED`

**has\_step** (*step\_method*)

Return if the Workflow has a step with a specific name. :param *step\_method*: a string with the name of the step to retrieve or a method

**info** ()

Returns an array with all the informations about the modules, file, class to locate the workflow source code

**is\_new** ()

Returns True is the Workflow's state is `CREATED`

**is\_running** ()

Returns True is the Workflow's state is `RUNNING`

**is\_subworkflow** ()

Return True is this is a subworkflow (i.e., if it has a parent), False otherwise.

**kill** (*verbose=False*)

Stop the Workflow execution and change its state to `FINISHED`.

This method calls the `kill` method for each Calculation and each subworkflow linked to each RUNNING step.

**Parameters** `verbose` – True to print the pk of each subworkflow killed

**Raises** `InvalidOperation` if some calculations cannot be killed (the workflow will be also put to SLEEP so that it can be killed later on)

**kill\_step\_calculations** (*step*)

Calls the `kill` method for each Calculation linked to the step method passed as argument. :param step: a Workflow step (decorated) method

**label**

Get the label of the workflow.

**Returns** a string

**logger**

Get the logger of the Workflow object, so that it also logs to the DB.

**Returns** LoggerAdapter object, that works like a logger, but also has the ‘extra’ embedded

**next** (*next\_method*)

Adds the a new step to be called after the completion of the caller method’s calculations and subworkflows.

This method must be called inside a Workflow step, otherwise an error is thrown. The code finds the caller method and stores in the database the input `next_method` as the next method to be called. At this point no execution in made, only configuration updates in the database.

If during the execution of the caller method the user launched calculations or subworkflows, this method will add them to the database, making them available to the workflow manager to be launched. In fact all the calculation and subworkflow submissions are lazy method, really executed by this call.

**Parameters** `next_method` – a Workflow step method to execute after the caller method

**Raise** `AiidaException`: in case the caller method cannot be found or validated

**Returns** the wrapped methods, decorated with the correct step name

**pk**

Returns the DbWorkflow pk

**classmethod query** (*\*args, \*\*kwargs*)

Map to the aiidaobjects manager of the DbWorkflow, that returns Workflow objects instead of DbWorkflow entities.

**remove\_path** (*path*)

Remove a file or directory from the repository directory.

Can be called only before storing.

**repo\_folder**

Get the permanent repository folder. Use preferentially the `current_folder` method.

**Returns** the permanent RepositoryFolder object

**set\_params** (*params, force=False*)

Adds parameters to the Workflow that are both stored and used every time the workflow engine re-initialize the specific workflow to launch the new methods.

**set\_state** (*state*)

Set the Workflow’s state :param name: a state from `wf_states` in `aiida.common.datastructures`

**sleep** ()

Changes the workflow state to SLEEP, only possible to call from a Workflow step decorated method.

**classmethod `step`** (*fun*)

This method is used as a decorator for workflow steps, and handles the method's execution, the state updates and the eventual errors.

The decorator generates a wrapper around the input function to execute, adding with the correct step name and a utility variable to make it distinguishable from non-step methods.

When a step is launched, the wrapper tries to run the function in case of error the state of the workflow is moved to ERROR and the traceback is stored in the report. In general the input method is a step obtained from the Workflow object, and the decorator simply handles a controlled execution of the step allowing the code not to break in case of error in the step's source code.

The wrapper also tests not to run two times the same step, unless a Workflow is in ERROR state, in this case all the calculations and subworkflows of the step are killed and a new execution is allowed.

**Parameters** *fun* – a methods to wrap, making it a Workflow step

**Raise** `AiidaException`: in case the workflow state doesn't allow the execution

**Returns** the wrapped methods,

**`store()`**

Stores the DbWorkflow object data in the database

**`uuid`**

Returns the DbWorkflow uuid

**exception** `aiida.orm.workflow.WorkflowKillError` (*\*args, \*\*kwargs*)

An exception raised when a workflow failed to be killed. The `error_message_list` attribute contains the error messages from all the subworkflows.

**exception** `aiida.orm.workflow.WorkflowUnkillable`

Raised when a workflow cannot be killed because it is in the FINISHED or ERROR state.

`aiida.orm.workflow.get_workflow_info` (*w, tab\_size=2, short=False, pre\_string='', depth=16*)

Return a string with all the information regarding the given workflow and all its calculations and subworkflows. This is a recursive function (to print all subworkflows info as well).

**Parameters**

- **w** – a DbWorkflow instance
- **tab\_size** – number of spaces to use for the indentation
- **short** – if True, provide a shorter output (only total number of calculations, rather than the state of each calculation)
- **pre\_string** – string appended at the beginning of each line
- **depth** – the maximum depth level the recursion on sub-workflows will try to reach (0 means we stay at the step level and don't go into sub-workflows, 1 means we go down to one step level of the sub-workflows, etc.)

**Return lines** list of lines to be outputted

`aiida.orm.workflow.kill_all()`

Kills all the workflows not in FINISHED state running the `kill_from_uuid` method in a loop.

**Parameters** **uuid** – the UUID of the workflow to kill

`aiida.orm.workflow.kill_from_pk` (*pk, verbose=False*)

Kills a workflow without loading the class, useful when there was a problem and the workflow definition module was changed/deleted (and the workflow cannot be reloaded).

**Parameters**

- **pk** – the principal key (id) of the workflow to kill
- **verbose** – True to print the pk of each subworkflow killed

`aiida.orm.workflow.kill_from_uuid(uuid)`

Kills a workflow without loading the class, useful when there was a problem and the workflow definition module was changed/deleted (and the workflow cannot be reloaded).

**Parameters** `uuid` – the UUID of the workflow to kill

## Code

**class** `aiida.orm.code.Code(**kwargs)`

A code entity. It can either be ‘local’, or ‘remote’.

- **Local code:** it is a collection of files/dirs (added using the `add_path()` method), where one file is flagged as executable (using the `set_local_executable()` method).
- **Remote code:** it is a pair (`remotecomputer`, `remotepath_of_executable`) set using the `set_remote_computer_exec()` method.

For both codes, one can set some code to be executed right before or right after the execution of the code, using the `set_preexec_code()` and `set_postexec_code()` methods (e.g., the `set_preexec_code()` can be used to load specific modules required for the code to be run).

**can\_run\_on** (*computer*)

Return True if this code can run on the given computer, False otherwise.

Local codes can run on any machine; remote codes can run only on the machine on which they reside.

TODO: add filters to mask the remote machines on which a local code can run.

**full\_text\_info**

Return a (multiline) string with a human-readable detailed information on this computer.

**classmethod** `get(label, computername=None, useremail=None)`

Get a code from its label.

### Parameters

- **label** – the code label
- **computername** – filter only codes on computers with this name
- **useremail** – filter only codes belonging to a user with this email

### Raises

- **NotExistent** – if no matches are found
- **MultipleObjectsError** – if multiple matches are found. In this case you may want to pass the additional parameters to filter the codes, or relabel the codes.

**get\_append\_text** ()

Return the `postexec_code`, or an empty string if no post-exec code was defined.

**get\_execname** ()

Return the executable string to be put in the script. For local codes, it is `./LOCAL_EXECUTABLE_NAME`. For remote codes, it is the absolute path to the executable.

**classmethod** `get_from_string(code_string)`

Get a Computer object with given identifier string, that can either be the numeric ID (pk), or the label (if unique); the label can either be simply the label, or in the format `label@machinename`. See the note below for details on the string detection algorithm.



**Note:** If a string that can be converted to an integer is given, the numeric ID is verified first (therefore, is a code A with a label equal to the ID of another code B is present, code A cannot be referenced by label). Similarly, the (leftmost) '@' symbol is always used to split code and computername. Therefore do not use '@' in the code name if you want to use this function ('@' in the computer name are instead valid).

**Parameters** `code_string` – the code string identifying the code to load

**Raises**

- **NotExistent** – if no code identified by the given string is found
- **MultipleObjectsError** – if the string cannot identify uniquely a code

**get\_input\_plugin\_name** ()

Return the name of the default input plugin (or None if no input plugin was set).

**get\_prepend\_text** ()

Return the code that will be put in the scheduler script before the execution, or an empty string if no pre-exec code was defined.

**is\_local** ()

Return True if the code is 'local', False if it is 'remote' (see also documentation of the `set_local` and `set_remote` functions).

**classmethod list\_for\_plugin** (*plugin, labels=True*)

Return a list of valid code strings for a given plugin.

**Parameters**

- **plugin** – The string of the plugin.
- **labels** – if True, return a list of code names, otherwise return the code PKs (integers).

**Returns** a list of string, with the code names if labels is True, otherwise a list of integers with the code PKs.

**new\_calc** (\*args, \*\*kwargs)

Create and return a new Calculation object (unstored) with the correct plugin subclass, as obtained by the `self.get_input_plugin_name()` method.

Parameters are passed to the calculation `__init__` method.

**Note** it also directly creates the link to this code (that will of course be cached, since the new node is not stored yet).

**Raises**

- **MissingPluginError** – if the specified plugin does not exist.
- **ValueError** – if no plugin was specified.

**set\_append\_text** (*code*)

Pass a string of code that will be put in the scheduler script after the execution of the code.

**set\_files** (*files*)

Given a list of filenames (or a single filename string), add it to the path (all at level zero, i.e. without folders). Therefore, be careful for files with the same name!

**Todo** decide whether to check if the Code must be a local executable to be able to call this function.

**set\_input\_plugin\_name** (*input\_plugin*)

Set the name of the default input plugin, to be used for the automatic generation of a new calculation.

**set\_local\_executable** (*exec\_name*)

Set the filename of the local executable. Implicitly set the code as local.

**set\_prepend\_text** (*code*)

Pass a string of code that will be put in the scheduler script before the execution of the code.

**set\_remote\_computer\_exec** (*remote\_computer\_exec*)

Set the code as remote, and pass the computer on which it resides and the absolute path on that computer.

**Args:**

**remote\_computer\_exec:** a tuple (**computer**, **remote\_exec\_path**), where **computer** is a `aiida.orm.Computer` or an `aiida.djsite.db.models.DbComputer` object, and **remote\_exec\_path** is the absolute path of the main executable on remote computer.

`aiida.orm.code.delete_code` (*code*)

Delete a code from the DB. Check before that there are no output nodes.

NOTE! Not thread safe... Do not use with many users accessing the DB at the same time.

Implemented as a function on purpose, otherwise complicated logic would be needed to set the internal state of the object after calling `computer.delete()`.

### ORM documentation: Data

**class** `aiida.orm.data.Data` (*\*\*kwargs*)

This class is base class for all data objects.

**convert** (*object\_format=None, \*args*)

Convert the AiiDA StructureData into another python object

**Parameters** **object\_format** – Specify the output format

**export** (*fname, fileformat=None*)

Save a Data object to a file.

**Parameters**

- **fname** – string with file name. Can be an absolute or relative path.
- **fileformat** – kind of format to use for the export. If not present, it will try to use the extension of the file name.

**importfile** (*fname, fileformat=None*)

Populate a Data object from a file.

**Parameters**

- **fname** – string with file name. Can be an absolute or relative path.
- **fileformat** – kind of format to use for the export. If not present, it will try to use the extension of the file name.

**importstring** (*inputstring, fileformat, \*\*kwargs*)

Converts a Data object to other text format.

**Parameters** **fileformat** – a string (the extension) to describe the file format.

**Returns** a string with the structure description.

**set\_source** (*source*)

Sets the dictionary describing the source of Data object.

**source**

Gets the dictionary describing the source of Data object. Possible fields:

- **db\_name**: name of the source database.
- **db\_uri**: URI of the source database.
- **uri**: URI of the object's source. Should be a permanent link.
- **id**: object's source identifier in the source database.
- **version**: version of the object's source.
- **extras**: a dictionary with other fields for source description.
- **source\_md5**: MD5 checksum of object's source.
- **description**: **human-readable free form description of the** object's source.
- **license**: a string with a type of license.

---

**Note:** some limitations for setting the data source exist, see `_validate()`.

---

**Returns** dictionary describing the source of Data object.

**Structure** This module defines the classes for structures and all related functions to operate on them.

**class** `aiida.orm.data.structure.Kind(**kwargs)`

This class contains the information about the species (kinds) of the system.

It can be a single atom, or an alloy, or even contain vacancies.

**\_\_init\_\_** (*\*\*kwargs*)

Create a site. One can either pass:

**Parameters**

- **raw** – the raw python dictionary that will be converted to a Kind object.
- **ase** – an ase Atom object
- **kind** – a Kind object (to get a copy)

Or alternatively the following parameters:

**Parameters**

- **symbols** – a single string for the symbol of this site, or a list of symbol strings
- **(optional) (mass)** – the weights for each atomic species of this site. If only a single symbol is provided, then this value is optional and the weight is set to 1.
- **(optional)** – the mass for this site in atomic mass units. If not provided, the mass is set by the `self.reset_mass()` function.
- **name** – a string that uniquely identifies the kind, and that is used to identify the sites.

**compare\_with** (*other\_kind*)

Compare with another Kind object to check if they are different.

---

**Note:** This does NOT check the 'type' attribute. Instead, it compares (with reasonable thresholds, where applicable): the mass, and the list of symbols and of weights. Moreover, it compares the `_internal_tag`, if defined (at the moment, defined automatically only when importing the Kind from ASE, if the atom has a non-zero tag). Note that the `_internal_tag` is only used while the class is loaded, but is not persisted on the database.

---

**Returns** A tuple with two elements. The first one is True if the two sites are ‘equivalent’ (same mass, symbols and weights), False otherwise. The second element of the tuple is a string, which is either None (if the first element was True), or contains a ‘human-readable’ description of the first difference encountered between the two sites.

**get\_raw()**

Return the raw version of the site, mapped to a suitable dictionary. This is the format that is actually used to store each kind of the structure in the DB.

**Returns** a python dictionary with the kind.

**get\_symbols\_string()**

Return a string that tries to match as good as possible the symbols of this kind. If there is only one symbol (no alloy) with 100% occupancy, just returns the symbol name. Otherwise, groups the full string in curly brackets, and try to write also the composition (with 2 precision only).

---

**Note:** If there is a vacancy (sum of weights<1), we indicate it with the X symbol followed by 1-sum(weights) (still with 2 digits precision, so it can be 0.00)

---



---

**Note:** Note the difference with respect to the symbols and the symbol properties!

---

**has\_vacancies()**

Returns True if the sum of the weights is less than one. It uses the internal variable `_sum_threshold` as a threshold.

**Returns** a boolean

**is\_alloy()**

To understand if kind is an alloy.

**Returns** True if the kind has more than one element (i.e., `len(self.symbols) != 1`), False otherwise.

**mass**

The mass of this species kind.

**Returns** a float

**name**

Return the name of this kind. The name of a kind is used to identify the species of a site.

**Returns** a string

**reset\_mass()**

Reset the mass to the automatic calculated value.

The mass can be set manually; by default, if not provided, it is the mass of the constituent atoms, weighted with their weight (after the weight has been normalized to one to take correctly into account vacancies).

This function uses the internal `_symbols` and `_weights` values and thus assumes that the values are validated.

It sets the mass to None if the sum of weights is zero.

**set\_automatic\_kind\_name(tag=None)**

Set the type to a string obtained with the symbols appended one after the other, without spaces, in alphabetical order; if the site has a vacancy, a X is appended at the end too.

**set\_symbols\_and\_weights(symbols, weights)**

Set the chemical symbols and the weights for the site.

---

---

**Note:** Note that the kind name remains unchanged.

---

**symbol**

If the kind has only one symbol, return it; otherwise, raise a `ValueError`.

**symbols**

List of symbols for this site. If the site is a single atom, pass a list of one element only, or simply the string for that atom. For alloys, a list of elements.

---

**Note:** Note that if you change the list of symbols, the kind name remains unchanged.

---

**weights**

Weights for this species kind. Refer also to `:func:validate_symbols_tuple` for the validation rules on the weights.

**class** `aiida.orm.data.structure.Site(**kwargs)`

This class contains the information about a given site of the system.

It can be a single atom, or an alloy, or even contain vacancies.

`__init__` (*\*\*kwargs*)

Create a site.

**Parameters**

- **kind\_name** – a string that identifies the kind (species) of this site. This has to be found in the list of kinds of the `StructureData` object. Validation will be done at the `StructureData` level.
- **position** – the absolute position (three floats) in angstrom

`get_ase` (*kinds*)

Return a `ase.Atom` object for this site.

**Parameters** **kinds** – the list of kinds from the `StructureData` object.

---

**Note:** If any site is an alloy or has vacancies, a `ValueError` is raised (from the `site.get_ase()` routine).

---

`get_raw` ()

Return the raw version of the site, mapped to a suitable dictionary. This is the format that is actually used to store each site of the structure in the DB.

**Returns** a python dictionary with the site.

**kind\_name**

Return the kind name of this site (a string).

The type of a site is used to decide whether two sites are identical (same mass, symbols, weights, ...) or not.

**position**

Return the position of this site in absolute coordinates, in angstrom.

**class** `aiida.orm.data.structure.StructureData(**kwargs)`

This class contains the information about a given structure, i.e. a collection of sites together with a cell, the boundary conditions (whether they are periodic or not) and other related useful information.

`append_atom` (*\*\*kwargs*)

Append an atom to the Structure, taking care of creating the corresponding kind.

**Parameters**

- **ase** – the ase Atom object from which we want to create a new atom (if present, this must be the only parameter)
- **position** – the position of the atom (three numbers in angstrom)
- **symbols, weights, name** (..) – any further parameter is passed to the constructor of the Kind object. For the ‘name’ parameter, see the note below.

---

**Note:** Note on the ‘name’ parameter (that is, the name of the kind):

- if specified, no checks are done on existing species. Simply, a new kind with that name is created. If there is a name clash, a check is done: if the kinds are identical, no error is issued; otherwise, an error is issued because you are trying to store two different kinds with the same name.
- if not specified, the name is automatically generated. Before adding the kind, a check is done. If other species with the same properties already exist, no new kinds are created, but the site is added to the existing (identical) kind. (Actually, the first kind that is encountered). Otherwise, the name is made unique first, by adding to the string containing the list of chemical symbols a number starting from 1, until an unique name is found

---

**Note:** checks of equality of species are done using the `compare_with()` method.

---

**append\_kind** (*kind*)

Append a kind to the *StructureData*. It makes a copy of the kind.

**Parameters** **kind** – the site to append, must be a Kind object.

**append\_site** (*site*)

Append a site to the *StructureData*. It makes a copy of the site.

**Parameters** **site** – the site to append. It must be a Site object.

**cell**

Returns the cell shape.

**Returns** a 3x3 list of lists.

**cell\_angles**

Get the angles between the cell lattice vectors in degrees.

**cell\_lengths**

Get the lengths of cell lattice vectors in angstroms.

**clear\_kinds** ()

Removes all kinds for the StructureData object.

---

**Note:** Also clear all sites!

---

**clear\_sites** ()

Removes all sites for the StructureData object.

**get\_ase** ()

Get the ASE object. Requires to be able to import ase.

**Returns** an ASE object corresponding to this *StructureData* object.

---

**Note:** If any site is an alloy or has vacancies, a ValueError is raised (from the site.get\_ase() routine).

---

**get\_cell\_volume** ()

Returns the cell volume in Angstrom^3.

**Returns** a float.

**get\_formula** (*mode='hill', separator=''*)

Return a string with the chemical formula.

#### Parameters

- **mode** – a string to specify how to generate the formula, can assume one of the following values:
  - ‘hill’ (default): count the number of atoms of each species, then use Hill notation, i.e. alphabetical order with C and H first if one or several C atom(s) is (are) present, e.g. [‘C’, ‘H’, ‘H’, ‘H’, ‘O’, ‘C’, ‘H’, ‘H’, ‘H’] will return ‘C2H6O’ [‘S’, ‘O’, ‘O’, ‘H’, ‘O’, ‘H’, ‘O’] will return ‘H2O4S’ From E. A. Hill, J. Am. Chem. Soc., 22 (8), pp 478–494 (1900)
  - ‘hill\_compact’: same as hill but the number of atoms for each species is divided by the greatest common divisor of all of them, e.g. [‘C’, ‘H’, ‘H’, ‘H’, ‘O’, ‘C’, ‘H’, ‘H’, ‘H’, ‘O’, ‘O’, ‘O’] will return ‘CH3O2’
  - ‘reduce’: group repeated symbols e.g. [‘Ba’, ‘Ti’, ‘O’, ‘O’, ‘O’, ‘Ba’, ‘Ti’, ‘O’, ‘O’, ‘O’, ‘Ba’, ‘Ti’, ‘Ti’, ‘O’, ‘O’, ‘O’] will return ‘BaTiO3BaTiO3BaTiO3’
  - ‘group’: will try to group as much as possible parts of the formula e.g. [‘Ba’, ‘Ti’, ‘O’, ‘O’, ‘O’, ‘Ba’, ‘Ti’, ‘O’, ‘O’, ‘O’, ‘Ba’, ‘Ti’, ‘Ti’, ‘O’, ‘O’, ‘O’] will return ‘(BaTiO3)2BaTi2O3’
  - ‘count’: same as hill (i.e. one just counts the number of atoms of each species) without the re-ordering (take the order of the atomic sites), e.g. [‘Ba’, ‘Ti’, ‘O’, ‘O’, ‘O’, ‘Ba’, ‘Ti’, ‘O’, ‘O’, ‘O’, ‘O’] will return ‘Ba2Ti2O6’
  - ‘count\_compact’: same as count but the number of atoms for each species is divided by the greatest common divisor of all of them, e.g. [‘Ba’, ‘Ti’, ‘O’, ‘O’, ‘O’, ‘Ba’, ‘Ti’, ‘O’, ‘O’, ‘O’, ‘O’] will return ‘BaTiO3’
- **separator** – a string used to concatenate symbols. Default empty.

**Returns** a string with the formula

---

**Note:** in modes reduce, group, count and count\_compact, the initial order in which the atoms were appended by the user is used to group and/or order the symbols in the formula

---

**get\_kind** (*kind\_name*)

Return the kind object associated with the given kind name.

**Parameters** *kind\_name* – String, the name of the kind you want to get

**Returns** The Kind object associated with the given *kind\_name*, if a Kind with the given name is present in the structure.

**Raise** ValueError if the *kind\_name* is not present.

**get\_kind\_names** ()

Return a list of kind names (in the same order of the `self.kinds` property, but return the names rather than Kind objects)

---

**Note:** This is NOT necessarily a list of chemical symbols! Use `get_symbols_set` for chemical symbols

---

**Returns** a list of strings.

**get\_pymatgen()**

Get pymatgen object. Returns Structure for structures with periodic boundary conditions (in three dimensions) and Molecule otherwise.

---

**Note:** Requires the pymatgen module (version  $\geq 3.0.13$ , usage of earlier versions may cause errors).

---

**get\_pymatgen\_molecule()**

Get the pymatgen Molecule object.

---

**Note:** Requires the pymatgen module (version  $\geq 3.0.13$ , usage of earlier versions may cause errors).

---

**Returns** a pymatgen Molecule object corresponding to this *StructureData* object.

**get\_pymatgen\_structure()**

Get the pymatgen Structure object.

---

**Note:** Requires the pymatgen module (version  $\geq 3.0.13$ , usage of earlier versions may cause errors).

---

**Returns** a pymatgen Structure object corresponding to this *StructureData* object.

**Raises ValueError** if periodic boundary conditions do not hold in at least one dimension of real space.

**get\_site\_kindnames()**

Return a list with length equal to the number of sites of this structure, where each element of the list is the kind name of the corresponding site.

---

**Note:** This is NOT necessarily a list of chemical symbols! Use `[self.get_kind(s.kind_name).get_symbols_string() for s in self.sites]` for chemical symbols

---

**Returns** a list of strings

**get\_symbols\_set()**

Return a set containing the names of all elements involved in this structure (i.e., for it joins the list of symbols for each kind *k* in the structure).

**Returns** a set of strings of element names.

**has\_vacancies()**

To understand if there are vacancies in the structure.

**Returns** a boolean, True if at least one kind has a vacancy

**is\_alloy()**

To understand if there are alloys in the structure.

**Returns** a boolean, True if at least one kind is an alloy

**kinds**

Returns a list of kinds.

**pbc**

Get the periodic boundary conditions.



**Returns** a tuple of three booleans, each one tells if there are periodic boundary conditions for the i-th real-space direction (i=1,2,3)

**reset\_cell** (*new\_cell*)

Reset the cell of a structure not yet stored to a new value.

**Parameters** *new\_cell* – list specifying the cell vectors

**Raises** `ModificationNotAllowed`: if object is already stored

**reset\_sites\_positions** (*new\_positions*, *conserve\_particle=True*)

Replace all the Site positions attached to the Structure

**Parameters**

- **new\_positions** – list of (3D) positions for every sites.
- **conserve\_particle** – if True, allows the possibility of removing a site. currently not implemented.

**Raises**

- **ModificationNotAllowed** – if object is stored already
- **ValueError** – if positions are invalid

---

**Note:** it is assumed that the order of the *new\_positions* is given in the same order of the one it's substituting, i.e. the kind of the site will not be checked.

---

**set\_ase** (*aseatoms*)

Load the structure from a ASE object

**set\_pymatgen** (*obj*, *\*\*kwargs*)

Load the structure from a pymatgen object.

---

**Note:** Requires the pymatgen module (version  $\geq 3.0.13$ , usage of earlier versions may cause errors).

---

**set\_pymatgen\_molecule** (*mol*, *margin=5*)

Load the structure from a pymatgen Molecule object.

**Parameters** *margin* – the margin to be added in all directions of the bounding box of the molecule.

---

**Note:** Requires the pymatgen module (version  $\geq 3.0.13$ , usage of earlier versions may cause errors).

---

**set\_pymatgen\_structure** (*struct*)

Load the structure from a pymatgen Structure object.

---

**Note:** periodic boundary conditions are set to True in all three directions.

---



---

**Note:** Requires the pymatgen module (version  $\geq 3.0.13$ , usage of earlier versions may cause errors).

---

**sites**

Returns a list of sites.

`aiida.orm.data.structure.ase_refine_cell` (*aseatoms*, *\*\*kwargs*)

Detect the symmetry of the structure, remove symmetric atoms and refine unit cell.

**Parameters**

- **aseatoms** – an ase.atoms.Atoms instance
- **symprec** – symmetry precision, used by pyspglib

**Return newase** refined cell with reduced set of atoms

**Return symmetry** a dictionary describing the symmetry space group

`aiida.orm.data.structure.calc_cell_volume(cell)`

Calculates the volume of a cell given the three lattice vectors.

It is calculated as  $\text{cell}[0] \cdot (\text{cell}[1] \times \text{cell}[2])$ , where  $\cdot$  represents a dot product and  $\times$  a cross product.

**Parameters** **cell** – the cell vectors; there must be a 3x3 list of lists of floats, no other checks are done.

**Returns** the cell volume.

`aiida.orm.data.structure.get_formula(symbol_list, mode='hill', separator='')`

Return a string with the chemical formula.

**Parameters**

- **symbol\_list** – a list of symbols, e.g. `['H', 'H', 'O']`
- **mode** – a string to specify how to generate the formula, can assume one of the following values:
  - ‘hill’ (default): count the number of atoms of each species, then use Hill notation, i.e. alphabetical order with C and H first if one or several C atom(s) is (are) present, e.g. `['C', 'H', 'H', 'H', 'O', 'C', 'H', 'H', 'H']` will return `'C2H6O'` `['S', 'O', 'O', 'H', 'O', 'H', 'O']` will return `'H2O4S'` From E. A. Hill, J. Am. Chem. Soc., 22 (8), pp 478–494 (1900)
  - ‘hill\_compact’: same as hill but the number of atoms for each species is divided by the greatest common divisor of all of them, e.g. `['C', 'H', 'H', 'H', 'O', 'C', 'H', 'H', 'H', 'O', 'O', 'O']` will return `'CH3O2'`
  - ‘reduce’: group repeated symbols e.g. `['Ba', 'Ti', 'O', 'O', 'O', 'Ba', 'Ti', 'O', 'O', 'O', 'Ba', 'Ti', 'Ti', 'O', 'O', 'O']` will return `'BaTiO3BaTiO3BaTi2O3'`
  - ‘group’: will try to group as much as possible parts of the formula e.g. `['Ba', 'Ti', 'O', 'O', 'O', 'Ba', 'Ti', 'O', 'O', 'O', 'Ba', 'Ti', 'Ti', 'O', 'O', 'O']` will return `'(BaTiO3)2BaTi2O3'`
  - ‘count’: same as hill (i.e. one just counts the number of atoms of each species) without the re-ordering (take the order of the atomic sites), e.g. `['Ba', 'Ti', 'O', 'O', 'O', 'Ba', 'Ti', 'O', 'O', 'O']` will return `'Ba2Ti2O6'`
  - ‘count\_compact’: same as count but the number of atoms for each species is divided by the greatest common divisor of all of them, e.g. `['Ba', 'Ti', 'O', 'O', 'O', 'Ba', 'Ti', 'O', 'O', 'O']` will return `'BaTiO3'`
- **separator** – a string used to concatenate symbols. Default empty.

**Returns** a string with the formula

---

**Note:** in modes reduce, group, count and count\_compact, the initial order in which the atoms were appended by the user is used to group and/or order the symbols in the formula

---

`aiida.orm.data.structure.get_formula_from_symbol_list(_list, separator='')`

Return a string with the formula obtained from the list of symbols. Examples: \*  
`[[1, 'Ba'], [1, 'Ti'], [3, 'O']]` will return `'BaTiO3'` \* `[[2, [ [1, 'Ba'], [1, 'Ti'] ] ]]` will return `'(BaTi)2'`

#### Parameters

- **\_list** – a list of symbols and multiplicities as obtained from the function `group_symbols`
- **separator** – a string used to concatenate symbols. Default empty.

**Returns** a string

`aiida.orm.data.structure.get_formula_group(symbol_list, separator='')`

Return a string with the chemical formula from a list of chemical symbols. The formula is written in a compact way, i.e. trying to group as much as possible parts of the formula.

---

**Note:** it works for instance very well if structure was obtained from an ASE supercell.

---

Example of result: `['Ba', 'Ti', 'O', 'O', 'O', 'Ba', 'Ti', 'O', 'O', 'O', 'Ba', 'Ti', 'Ti', 'O', 'O', 'O']` will return `'(BaTiO3)2BaTi2O3'`.

#### Parameters

- **symbol\_list** – list of symbols (e.g. `['Ba','Ti','O','O','O']`)
- **separator** – a string used to concatenate symbols. Default empty.

**Returns** a string with the chemical formula for the given structure.

`aiida.orm.data.structure.get_pymatgen_version()`

**Returns** string with pymatgen version, None if can not import.

`aiida.orm.data.structure.get_symbols_string(symbols, weights)`

Return a string that tries to match as good as possible the symbols and weights. If there is only one symbol (no alloy) with 100% occupancy, just returns the symbol name. Otherwise, groups the full string in curly brackets, and try to write also the composition (with 2 precision only). If (sum of weights<1), we indicate it with the X symbol followed by 1-sum(weights) (still with 2 digits precision, so it can be 0.00)

#### Parameters

- **symbols** – the symbols as obtained from `<kind>._symbols`
- **weights** – the weights as obtained from `<kind>._weights`

---

**Note:** Note the difference with respect to the symbols and the symbol properties!

---

`aiida.orm.data.structure.get_valid_pbc(inputpbc)`

Return a list of three booleans for the periodic boundary conditions, in a valid format from a generic input.

**Raises ValueError** if the format is not valid.

`aiida.orm.data.structure.group_symbols(_list)`

Group a list of symbols to a list containing the number of consecutive identical symbols, and the symbol itself.

Examples:

- `['Ba', 'Ti', 'O', 'O', 'O', 'Ba']` will return `[[1, 'Ba'], [1, 'Ti'], [3, 'O'], [1, 'Ba']]`
- `[[ [1, 'Ba'], [1, 'Ti'] ], [ [1, 'Ba'], [1, 'Ti'] ] ]` will return `[[2, [ [1, 'Ba'], [1, 'Ti'] ] ]]`

**Parameters** **\_list** – a list of elements representing a chemical formula

**Returns** a list of length-2 lists of the form [ multiplicity , element ]

`aiida.orm.data.structure.has_ase()`

**Returns** True if the ase module can be imported, False otherwise.

`aiida.orm.data.structure.has_pymatgen()`

**Returns** True if the pymatgen module can be imported, False otherwise.

`aiida.orm.data.structure.has_pyspglib()`

**Returns** True if the pyspglib module can be imported, False otherwise.

`aiida.orm.data.structure.has_vacancies(weights)`

**Returns** True if the sum of the weights is less than one. It uses the internal variable `_sum_threshold` as a threshold. :param weights: the weights :return: a boolean

`aiida.orm.data.structure.is_ase_atoms(ase_atoms)`

Check if the ase\_atoms parameter is actually a ase.Atoms object.

**Parameters** `ase_atoms` – an object, expected to be an ase.Atoms.

**Returns** a boolean.

Requires the ability to import ase, by doing ‘import ase’.

`aiida.orm.data.structure.is_valid_symbol(symbol)`

Validates the chemical symbol name.

**Returns** True if the symbol is a valid chemical symbol (with correct capitalization), False otherwise.

Recognized symbols are for elements from hydrogen (Z=1) to lawrencium (Z=103).

`aiida.orm.data.structure.symop_fract_from_ortho(cell)`

Creates a matrix for conversion from fractional to orthogonal coordinates.

Taken from [svn://www.crystallography.net/cod-tools/trunk/lib/perl5/Fractional.pm](http://svn://www.crystallography.net/cod-tools/trunk/lib/perl5/Fractional.pm), revision 850.

**Parameters** `cell` – array of cell parameters (three lengths and three angles)

`aiida.orm.data.structure.symop_ortho_from_fract(cell)`

Creates a matrix for conversion from orthogonal to fractional coordinates.

Taken from [svn://www.crystallography.net/cod-tools/trunk/lib/perl5/Fractional.pm](http://svn://www.crystallography.net/cod-tools/trunk/lib/perl5/Fractional.pm), revision 850.

**Parameters** `cell` – array of cell parameters (three lengths and three angles)

`aiida.orm.data.structure.validate_symbols_tuple(symbols_tuple)`

Used to validate whether the chemical species are valid.

**Parameters** `symbols_tuple` – a tuple (or list) with the chemical symbols name.

**Raises** ValueError if any symbol in the tuple is not a valid chemical symbols (with correct capitalization).

Refer also to the documentation of :func:is\_valid\_symbol

`aiida.orm.data.structure.validate_weights_tuple(weights_tuple, threshold)`

Validates the weight of the atomic kinds.

**Raise** ValueError if the weights\_tuple is not valid.

**Parameters**

- **weights\_tuple** – the tuple to validate. It must be a a tuple of floats (as created by :func:create\_weights\_tuple).

- **threshold** – a float number used as a threshold to check that the sum of the weights is  $\leq 1$ .

If the sum is less than one, it means that there are vacancies. Each element of the list must be  $\geq 0$ , and the sum must be  $\leq 1$ .

## Folder

**class** `aiida.orm.data.folder.FolderData(**kwargs)`

Stores a folder with subfolders and files.

No special attributes are set.

**get\_file\_content** (*path*)

Return the content of a path stored inside the folder as a string.

**Raises** `NotExistent` if the path does not exist.

**replace\_with\_folder** (*folder, overwrite=True*)

Replace the data with another folder, always copying and not moving the original files.

**Args:** *folder*: the folder to copy from *overwrite*: if to overwrite the current content or not

**Singlefile** Implement subclass for a single file in the permanent repository files = [one\_single\_file] jsons = { }

methods: \* `get_content` \* `get_path` \* `get_aiidaurl` (?) \* `get_md5` \* ...

To discuss: do we also need a simple directory class for full directories in the perm repo?

**class** `aiida.orm.data.singlefile.SinglefileData(**kwargs)`

Pass as input a file parameter with the (absolute) path of a file on the hard drive. It will get copied inside the node.

Internally must have a single file, and stores as internal attribute the filename in the 'filename' attribute.

**add\_path** (*src\_abs, dst\_filename=None*)

Add a single file

**del\_file** (*filename*)

Remove a file from SingleFileData :param filename: name of the file stored in the DB

**filename**

Returns the name of the file stored

**get\_file\_abs\_path** ()

Return the absolute path to the file in the repository

**set\_file** (*filename*)

Add a file to the singlefiledata :param filename: absolute path to the file

**Upf** This module manages the UPF pseudopotentials in the local repository.

**class** `aiida.orm.data.upf.UpfData(**kwargs)`

Function not yet documented.

**classmethod** `from_md5` (*md5*)

Return a list of all UPF pseudopotentials that match a given MD5 hash.

Note that the hash has to be stored in a `_md5` attribute, otherwise the pseudo will not be found.

**classmethod** `get_or_create` (*filename, use\_first=False, store\_upf=True*)

Pass the same parameter of the init; if a file with the same md5 is found, that UpfData is returned.

### Parameters

- **filename** – an absolute filename on disk
- **use\_first** – if False (default), raise an exception if more than one potential is found. If it is True, instead, use the first available pseudopotential.
- **store\_upf** (*bool*) – If false, the UpfData objects are not stored in the database. default=True.

**Return (upf, created)** where upf is the UpfData object, and create is either True if the object was created, or False if the object was retrieved from the DB.

**get\_upf\_family\_names** ()

Get the list of all upf family names to which the pseudo belongs

**classmethod get\_upf\_group** (*group\_name*)

Return the UpfFamily group with the given name.

**classmethod get\_upf\_groups** (*filter\_elements=None, user=None*)

Return all names of groups of type UpfFamily, possibly with some filters.

### Parameters

- **filter\_elements** – A string or a list of strings. If present, returns only the groups that contains one Upf for every element present in the list. Default=None, meaning that all families are returned.
- **user** – if None (default), return the groups for all users. If defined, it should be either a DbUser instance, or a string for the username (that is, the user email).

**set\_file** (*filename*)

I pre-parse the file to store the attributes.

**store** (*\*args, \*\*kwargs*)

Store the node, reparsing the file so that the md5 and the element are correctly reset.

`aiida.orm.data.upf.get_pseudos_from_structure` (*structure, family\_name*)

Given a family name (a UpfFamily group in the DB) and a AiiDA structure, return a dictionary associating each kind name with its UpfData object.

### Raises

- **MultipleObjectsError** – if more than one UPF for the same element is found in the group.
- **NotExistent** – if no UPF for an element in the group is found in the group.

`aiida.orm.data.upf.parse_upf` (*fname, check\_filename=True*)

Try to get relevant information from the UPF. For the moment, only the element name. Note that even UPF v.2 cannot be parsed with the XML minidom! (e.g. due to the & characters in the human-readable section).

If check\_filename is True, raise a ParsingError exception if the filename does not start with the element name.

`aiida.orm.data.upf.upload_upf_family` (*folder, group\_name, group\_description, stop\_if\_existing=True*)

Upload a set of UPF files in a given group.

### Parameters

- **folder** – a path containing all UPF files to be added. Only files ending in .UPF (case-insensitive) are considered.
- **group\_name** – the name of the group to create. If it exists and is non-empty, a UniquenessError is raised.

- **group\_description** – a string to be set as the group description. Overwrites previous descriptions, if the group was existing.
- **stop\_if\_existing** – if True, check for the md5 of the files and, if the file already exists in the DB, raises a MultipleObjectsError. If False, simply adds the existing UPFData node to the group.

**Cif**

**class** `aiida.orm.data.cif.CifData` (*\*\*kwargs*)  
 Wrapper for Crystallographic Interchange File (CIF)

---

**Note:** the file (physical) is held as the authoritative source of information, so all conversions are done through the physical file: when setting `ase` or `values`, a physical CIF file is generated first, the values are updated from the physical CIF file.

---

**ase**  
 ASE object, representing the CIF.

---

**Note:** requires ASE module.

---

**classmethod** `from_md5` (*md5*)  
 Return a list of all CIF files that match a given MD5 hash.

---

**Note:** the hash has to be stored in a `_md5` attribute, otherwise the CIF file will not be found.

---

**generate\_md5** ()  
 Generate MD5 hash of the file's contents on-the-fly.

---

**get\_ase** (*\*\*kwargs*)  
 Returns ASE object, representing the CIF. This function differs from the property `ase` by the possibility to pass the keyworded arguments (*kwargs*) to `ase.io.cif.read_cif()`.

---

**Note:** requires ASE module.

---

**get\_formulae** (*mode='sum'*)  
 Get the formula.

---

**classmethod** `get_or_create` (*filename, use\_first=False, store\_cif=True*)  
 Pass the same parameter of the `init`; if a file with the same md5 is found, that `CifData` is returned.

**Parameters**

- **filename** – an absolute filename on disk
- **use\_first** – if False (default), raise an exception if more than one CIF file is found. If it is True, instead, use the first available CIF file.
- **store\_cif** (*bool*) – If false, the `CifData` objects are not stored in the database. default=True.

**Return** (**cif, created**) where `cif` is the `CifData` object, and `created` is either True if the object was created, or False if the object was retrieved from the DB.

**get\_spacegroup\_numbers** ()  
 Get the spacegroup international number.

**has\_attached\_hydrogens()**

Check if there are hydrogens without coordinates, specified as attached to the atoms of the structure.  
:return: True if there are attached hydrogens, False otherwise.

**has\_partial\_occupancies()**

Check if there are float values in the atom occupancies. :return: True if there are partial occupancies, False otherwise.

**set\_file(filename)**

Set the file. If the source is set and the MD5 checksum of new file is different from the source, the source has to be deleted.

**store(\*args, \*\*kwargs)**

Store the node.

**values**

PyCifRW structure, representing the CIF datablocks.

---

**Note:** requires PyCifRW module.

---

`aiida.orm.data.cif.cif_from_ase(ase, full_occupancies=False, add_fake_biso=False)`

Construct a CIF datablock from the ASE structure. The code is taken from [https://wiki.fysik.dtu.dk/ase/epydoc/ase.io.cif-pysrc.html#write\\_cif](https://wiki.fysik.dtu.dk/ase/epydoc/ase.io.cif-pysrc.html#write_cif), as the original ASE code contains a bug in printing the Hermann-Mauguin symmetry space group symbol.

**Parameters** *ase* – ASE “images”

**Returns** array of CIF datablocks

`aiida.orm.data.cif.has_pycifrw()`

**Returns** True if the PyCifRW module can be imported, False otherwise.

`aiida.orm.data.cif.parse_formula(formula)`

Parses the Hill formulae, written with spaces for separators.

`aiida.orm.data.cif.pycifrw_from_cif(datablocks, loops={}, names=None)`

Constructs PyCifRW’s CifFile from an array of CIF datablocks.

**Parameters**

- **datablocks** – an array of CIF datablocks
- **loops** – optional list of lists of CIF tag loops.
- **names** – optional list of datablock names

**Returns** CifFile

`aiida.orm.data.cif.symop_string_from_symop_matrix_tr(matrix, tr=[0, 0, 0], eps=0)`

Construct a CIF representation of symmetry operator plus translation. See International Tables for Crystallography Vol. A. (2002) for definition.

**Parameters**

- **matrix** – 3x3 matrix, representing the symmetry operator
- **tr** – translation vector of length 3 (default [0, 0, 0])
- **eps** – epsilon parameter for fuzzy comparison  $x == 0$

**Returns** CIF representation of symmetry operator



**Parameter**

**class** `aiida.orm.data.parameter.ParameterData` (*\*\*kwargs*)

Pass as input in the init a dictionary, and it will get stored as internal attributes.

Usual rules for attribute names apply (in particular, keys cannot start with an underscore). If this is the case, a `ValueError` will be raised.

You can then change/delete/add more attributes before storing with the usual methods of `aiida.orm.Node`

**dict**

To be used to get direct access to the underlying dictionary with the syntax `node.dict.key` or `node.dict['key']`.

**Returns** an instance of the `AttributeResultManager`.

**get\_dict** ()

Return a dict with the parameters

**keys** ()

Iterator of valid keys stored in the `ParameterData` object

**set\_dict** (dict)

Replace the current dictionary with another one.

**Parameters** **dict** – The dictionary to set.

**update\_dict** (dict)

Update the current dictionary with the keys provided in the dictionary.

**Parameters** **dict** – a dictionary with the keys to substitute. It works like `dict.update()`, adding new keys and overwriting existing keys.

**Remote**

**class** `aiida.orm.data.remote.RemoteData` (*\*\*kwargs*)

Store a link to a file or folder on a remote machine.

Remember to pass a computer!

**add\_path** (src\_abs, dst\_filename=None)

Disable adding files or directories to a `RemoteData`

**is\_empty** ()

Check if remote folder is empty

**ArrayData**

**class** `aiida.orm.data.array.ArrayData` (*\*args, \*\*kwargs*)

Store a set of arrays on disk (rather than on the database) in an efficient way using `numpy.save()` (therefore, this class requires `numpy` to be installed).

Each array is stored within the Node folder as a different `.npy` file.

**Note** Before storing, no caching is done: if you perform a `get_array()` call, the array will be re-read from disk. If instead the `ArrayData` node has already been stored, the array is cached in memory after the first read, and the cached array is used thereafter. If too much RAM memory is used, you can clear the cache with the `clear_internal_cache()` method.

**arraynames** ()

Return a list of all arrays stored in the node, listing the files (and not relying on the properties).

**clear\_internal\_cache()**

Clear the internal memory cache where the arrays are stored after being read from disk (used in order to reduce at minimum the readings from disk). This function is useful if you want to keep the node in memory, but you do not want to waste memory to cache the arrays in RAM.

**delete\_array(name)**

Delete an array from the node. Can only be called before storing.

**Parameters** **name** – The name of the array to delete from the node.

**get\_array(name)**

Return an array stored in the node

**Parameters** **name** – The name of the array to return.

**get\_shape(name)**

Return the shape of an array (read from the value cached in the properties for efficiency reasons).

**Parameters** **name** – The name of the array.

**iterarrays()**

Iterator that returns tuples (name, array) for each array stored in the node.

**set\_array(name, array)**

Store a new numpy array inside the node. Possibly overwrite the array if it already existed.

Internally, it stores a name.npy file in numpy format.

**Parameters**

- **name** – The name of the array.
- **array** – The numpy array to store.

**ArrayData subclasses** The following are Data classes inheriting from ArrayData.

**KpointsData** This module defines the classes related to band structures or dispersions in a Brillouin zone, and how to operate on them.

**class aiida.orm.data.array.kpoints.KpointsData(\*args, \*\*kwargs)**

Class to handle array of kpoints in the Brillouin zone. Provide methods to generate either user-defined k-points or path of k-points along symmetry lines. Internally, all k-points are defined in terms of crystal (fractional) coordinates. Cell and lattice vector coordinates are in Angstroms, reciprocal lattice vectors in  $\text{\AA}^{-1}$ . :note: The methods setting and using the Bravais lattice info assume the PRIMITIVE unit cell is provided in input to the set\_cell or set\_cell\_from\_structure methods.

**cell**

The crystal unit cell. Rows are the crystal vectors in Angstroms. :return: a 3x3 numpy.array

**get\_kpoints(also\_weights=False, cartesian=False)**

Return the list of kpoints

**Parameters**

- **also\_weights** – if True, returns also the list of weights. Default = False
- **cartesian** – if True, returns points in cartesian coordinates, otherwise, returns in crystal coordinates. Default = False.

**get\_kpoints\_mesh(print\_list=False)**

Get the mesh of kpoints.

**Parameters** **print\_list** – default=False. If True, prints the mesh of kpoints as a list

**Raises `AttributeError`** if no mesh has been set

**Return `mesh, offset`** (if `print_list=False`) a list of 3 integers and a list of three floats  $0 < x < 1$ , representing the mesh and the offset of kpoints

**Return `kpoints`** (if `print_list = True`) an explicit list of kpoints coordinates, similar to what returned by `get_kpoints()`

#### **labels**

Labels associated with the list of kpoints. List of tuples with kpoint index and kpoint name: `[(0,'G'),(13,'M'),...]`

#### **pbcs**

The periodic boundary conditions along the vectors `a1,a2,a3`.

**Returns** a tuple of three booleans, each one tells if there are periodic boundary conditions for the *i*-th real-space direction (*i*=1,2,3)

#### **set\_cell** (*cell*, *pbcs=None*)

Set a cell to be used for symmetry analysis. To set a cell from an AiiDA structure, use “`set_cell_from_structure`”.

##### **Parameters**

- **cell** – 3x3 matrix of cell vectors. Orientation: each row represent a lattice vector. Units are Angstroms.
- **pbcs** – list of 3 booleans, True if in the *n*th crystal direction the structure is periodic. Default = [True,True,True]

#### **set\_cell\_from\_structure** (*structuredata*)

Set a cell to be used for symmetry analysis from an AiiDA structure. Inherits both the cell and the pbcs. To set manually a cell, use “`set_cell`”

**Parameters** **structuredata** – an instance of `StructureData`

#### **set\_kpoints** (*kpoints*, *cartesian=False*, *labels=None*, *weights=None*, *fill\_values=0*)

Set the list of kpoints. If a mesh has already been stored, raise a `ModificationNotAllowed`

##### **Parameters**

- **kpoints** – a list of kpoints, each kpoint being a list of one, two or three coordinates, depending on `self.pbcs`: if structure is 1D (only one True in `self.pbcs`) one allows singletons or scalars for each k-point, if it's 2D it can be a length-2 list, and in all cases it can be a length-3 list. Examples:
  - `[[0.,0.,0.],[0.1,0.1,0.1],...]` for 1D, 2D or 3D
  - `[[0.,0.],[0.1,0.1],...]` for 1D or 2D
  - `[[0.],[0.1],...]` for 1D
  - `[0., 0.1, ...]` for 1D (list of scalars)

For 0D (all pbcs are False), the list can be any of the above or empty - then only Gamma point is set. The value of *k* for the non-periodic dimension(s) is set by `fill_values`

- **cartesian** – if True, the coordinates given in input are treated as in cartesian units. If False, the coordinates are crystal, i.e. in units of `b1,b2,b3`. Default = False
- **labels** – optional, the list of labels to be set for some of the kpoints. See labels for more info
- **weights** – optional, a list of floats with the weight associated to the kpoint list

- **fill\_values** – scalar to be set to all non-periodic dimensions (indicated by False in self.pbc), or list of values for each of the non-periodic dimensions.

**set\_kpoints\_mesh** (*mesh*, *offset*=[0.0, 0.0, 0.0])

Set KpointsData to represent a uniformly spaced mesh of kpoints in the Brillouin zone. This excludes the possibility of set/get kpoints

#### Parameters

- **mesh** – a list of three integers, representing the size of the kpoint mesh along b1,b2,b3.
- **offset** (*optional*) – a list of three floats between 0 and 1. [0.,0.,0.] is Gamma centered mesh [0.5,0.5,0.5] is half shifted [1.,1.,1.] by periodicity should be equivalent to [0.,0.,0.] Default = [0.,0.,0.].

### TrajectoryData

**class** aiida.orm.data.array.trajectory.TrajectoryData (\*args, \*\*kwargs)

Stores a trajectory (a sequence of crystal structures with timestamps, and possibly with velocities).

**get\_cells** ()

Return the array of cells, if it has already been set.

**Raises KeyError** if the trajectory has not been set yet.

**get\_positions** ()

Return the array of positions, if it has already been set.

**Raises KeyError** if the trajectory has not been set yet.

**get\_step\_data** (*index*)

Return a tuple with all information concerning the step with given index (0 is the first step, 1 the second step and so on). If you know only the step value, use the `get_step_index()` method to get the corresponding index.

If no velocities were specified, None is returned as the last element.

**Returns** A tuple in the format (step, time, cell, symbols, positions, velocities), where step is an integer, time is a float, cell is a  $3 \times 3$  matrix, symbols is an array of length n, positions is a  $n \times 3$  array, and velocities is either None or a  $n \times 3$  array

**Parameters index** – The index of the step that you want to retrieve, from 0 to self.numsteps - 1.

#### Raises

- **IndexError** – if you require an index beyond the limits.
- **KeyError** – if you did not store the trajectory yet.

**get\_step\_index** (*step*)

Given a value for the step (i.e., a value among those of the steps array), return the array index of that step, that can be used in other methods such as `get_step_data()` or `step_to_structure()`.

---

**Note:** Note that this function returns the first index found (i.e. if multiple steps are present with the same value, only the index of the first one is returned).

---

**Raises ValueError** if no step with the given value is found.

**get\_steps** ()

Return the array of steps, if it has already been set.

**Raises `KeyError`** if the trajectory has not been set yet.

**`get_symbols()`**

Return the array of symbols, if it has already been set.

**Raises `KeyError`** if the trajectory has not been set yet.

**`get_times()`**

Return the array of times (in ps), if it has already been set.

**Raises `KeyError`** if the trajectory has not been set yet.

**`get_velocities()`**

Return the array of velocities, if it has already been set.

---

**Note:** This function (differently from all other `get_*` functions, will not raise an exception if the velocities are not set, but rather return `None` (both if no trajectory was not set yet, and if it the trajectory was set but no velocities were specified).

---

**`numsites`**

Return the number of stored sites, or zero if nothing has been stored yet.

**`numsteps`**

Return the number of stored steps, or zero if nothing has been stored yet.

**`set_structurelist(structurelist)`**

Create trajectory from the list of `aiida.orm.data.structure.StructureData` instances.

**Parameters `structurelist`** – a list of `aiida.orm.data.structure.StructureData` instances.

**Raises `ValueError`** if symbol lists of supplied structures are different

**`set_trajectory(steps, cells, symbols, positions, times=None, velocities=None)`**

Store the whole trajectory, after checking that types and dimensions are correct. Velocities are optional, if they are not passed, nothing is stored.

**Parameters**

- **`steps`** – integer array with dimension  $s$ , where  $s$  is the number of steps. Typically represents an internal counter within the code. For instance, if you want to store a trajectory with one step every 10, starting from step 65, the array will be `[65, 75, 85, ...]`. No checks are done on duplicate elements or on the ordering, but anyway this array should be sorted in ascending order, without duplicate elements. If your code does not provide an internal counter, just provide for instance `arange(s)`.
- **`cells`** – float array with dimension  $s \times 3 \times 3$ , where  $s$  is the length of the `steps` array. Units are angstrom. In particular, `cells[i, j, k]` is the  $k$ -th component of the  $j$ -th cell vector at the time step with index  $i$  (identified by step number `step[i]` and with timestamp `times[i]`).
- **`symbols`** – string array with dimension  $n$ , where  $n$  is the number of atoms (i.e., sites) in the structure. The same array is used for each step. Normally, the string should be a valid chemical symbol, but actually any unique string works and can be used as the name of the atomic kind (see also the `step_to_structure()` method).
- **`positions`** – float array with dimension  $s \times n \times 3$ , where  $s$  is the length of the `steps` array and  $n$  is the length of the `symbols` array. Units are angstrom. In particular, `positions[i, j, k]` is the  $k$ -th component of the  $j$ -th atom (or site) in the structure at the time step with index  $i$  (identified by step number `step[i]` and with timestamp `times[i]`).

- **times** – if specified, float array with dimension  $s$ , where  $s$  is the length of the `steps` array. Contains the timestamp of each step in picoseconds (ps).
- **velocities** – if specified, must be a float array with the same dimensions of the `positions` array. The array contains the velocities in the atoms.

---

**Todo**

Choose suitable units for velocities

---

**step\_to\_structure** (*index*, *custom\_kinds=None*)

Return an AiiDA `aiida.orm.data.structure.StructureData` node (not stored yet!) with the coordinates of the given step, identified by its index. If you know only the step value, use the `get_step_index()` method to get the corresponding index.

---

**Note:** The periodic boundary conditions are always set to True.

---

**Parameters**

- **index** – The index of the step that you want to retrieve, from 0 to `self.numsteps-1`.
- **custom\_kinds** – (Optional) If passed must be a list of `aiida.orm.data.structure.Kind` objects. There must be one kind object for each different string in the `symbols` array, with `kind.name` set to this string. If this parameter is omitted, the automatic kind generation of AiiDA `aiida.orm.data.structure.StructureData` nodes is used, meaning that the strings in the `symbols` array must be valid chemical symbols.

**ORM documentation: Calculations**

**class** `aiida.orm.calculation.Calculation` (\*\*kwargs)

This class provides the definition of an “abstract” AiiDA calculation. A calculation in this sense is any computation that converts data into data.

You will typically use one of its subclasses, often a `JobCalculation` for calculations run via a scheduler.

**get\_code** ()

Return the code for this calculation, or None if the code was not set.

**get\_linkname** (*link*, \*args, \*\*kwargs)

Return the linkname used for a given input link

Pass as parameter “NAME” if you would call the `use_NAME` method. If the `use_NAME` method requires a further parameter, pass that parameter as the second parameter.

**logger**

Get the logger of the Calculation object, so that it also logs to the DB.

**Returns** `LoggerAdapter` object, that works like a logger, but also has the ‘extra’ embedded

**class** `aiida.orm.calculation.inline.InlineCalculation` (\*\*kwargs)

Subclass used for calculations that are automatically generated using the `make_inline` wrapper/decorator.

This is used to automatically create a calculation node for a simple calculation

**get\_function\_name** ()

Get the function name.

**Returns** a string

`aiida.orm.calculation.inline.make_inline(func)`

This `make_inline` wrapper/decorator takes a function with specific requirements, runs it and stores the result as an `InlineCalculation` node. It will also store all other nodes, including any possibly unstored input node! The return value of the wrapped calculation will also be slightly changed, see below.

The wrapper:

- checks that the function name ends with the string `'_inline'`
- checks that each input parameter is a valid `Data` node (can be stored or unstored)
- runs the actual function
- gets the result values
- checks that the result value is a dictionary, where the key are all strings and the values are all **unstored** data nodes
- creates an `InlineCalculation` node, links all the kwargs as inputs and the returned nodes as outputs, using the keys as link labels
- stores all the nodes (including, possibly, unstored input nodes given as kwargs)
- returns a length-two tuple, where the first element is the `InlineCalculation` node, and the second is the dictionary returned by the wrapped function

To use this function, you can use it as a decorator of a wrapped function:

```
@make_inline
def copy_inline(source):
    return {copy: source.copy() }
```

In this way, every time you call `copy_inline`, the wrapped version is actually called, and the return value will be a tuple with the `InlineCalculation` instance, and the returned dictionary. For instance, if `s` is a valid `Data` node, with the following lines:

```
c, s_copy_dict = copy_inline(source=s)
s_copy = s_copy_dict['copy']
```

`c` will contain the new `InlineCalculation` instance, `s_copy` the (stored) copy of `s` (with the side effect that, if `s` was not stored, after the function call it will be automatically stored).

**Note** If you use a wrapper, make sure to write explicitly in the docstrings that the function is going to store the nodes.

The second possibility, if you want that by default the function does not store anything, but can be wrapped when it is necessary, is the following. You simply define the function you want to wrap (`copy_inline` in the example above) without decorator:

```
def copy_inline(source):
    return {copy: source.copy() }
```

This is a normal function, so to call it you will normally do:

```
s_copy_dict = copy_inline(s)
```

while if you want to wrap it, so that an `InlineCalculation` is created, and everything is stored, you will run:

```
c, s_copy_dict = make_inline(f)(s=s)
```

Note that, with the wrapper, all the parameters to `f()` have to be passed as keyworded arguments. Moreover, the return value is different, i.e. `(c, s_copy_dict)` instead of simply `s_copy_dict`.

---

**Note:** EXTREMELY IMPORTANT! The wrapped function **MUST** have the following requirements in order to be reproducible. These requirements cannot be enforced, but must be followed when writing the wrapped function.

- The function **MUST NOT USE** information that is not passed in the kwargs. In particular, it cannot read files from the hard-drive (that will not be present in another user's computer), it cannot connect to external databases and retrieve the current entries in that database (that could change over time), etc.
- The only exception to the above rule is the access to the AiiDA database for the *parents* of the input nodes. That is, you can take the input nodes passed as kwargs, and use also the data given in their inputs, the inputs of their inputs, ... but you **CANNOT** use any output of any of the above-mentioned nodes (that could change over time).
- The function **MUST NOT** have side effects (creating files on the disk, adding entries to an external database, ...).

---

**Note:** The function will also store:

- the source of the function in an attribute "source\_code", and the first line at which the function appears (attribute "first\_line\_source\_code"), as returned by `inspect.getsourcelines`;
- the full source file in "source\_file", if it is possible to retrieve it (this will be set to `None` otherwise, e.g. if the function was defined in the interactive shell).

For this reason, try to keep, if possible, all the code to be run within the same file, so that it is possible to keep the provenance of the functions that were run (if you instead call a function in a different file, you will never know in the future what that function did). If you call external modules and you matter about provenance, it would be good to also return in a suitable dictionary the version of these modules (e.g., after importing a module `XXX`, you can check if the module defines a variable `XXX.__version__` or `XXX.VERSION` or something similar, and store it in an output node).

---

**Todo** For the time being, I am storing the function source code and the full source code file in the attributes of the calculation. To be moved to an input Code node!

**Note** All nodes will be stored, including unstored input nodes!!

**Parameters** **kwargs** – all kwargs are passed to the wrapped function

**Returns** a length-two tuple, where the first element is the `InlineCalculation` node, and the second is the dictionary returned by the wrapped function. All nodes are stored.

**Raises**

- **TypeError** – if the return value is not a dictionary, the keys are not strings, or the values are not data nodes. Raise also if the input values are not data nodes.
- **ModificationNotAllowed** – if the returned Data nodes are already stored.
- **Exception** – All other exceptions from the wrapped function are not caught.

`aiida.orm.calculation.inline.optional_inline(func)`

`optional_inline` wrapper/decorator takes a function, which can be called either as wrapped in `InlineCalculation` or a simple function, depending on 'store' keyworded argument (`True` stands for `InlineCalculation`, `False` for simple function). The wrapped function has to adhere to the requirements by `make_inline` wrapper/decorator.

Usage example:



```
@optional_inline
def copy_inline(source=None):
    return {'copy': source.copy() }
```

Function `copy_inline` will be wrapped in `InlineCalculation` when invoked in following way:

```
copy_inline(source=node, store=True)
```

while it will be called as a simple function when invoked:

```
copy_inline(source=node)
```

In any way the `copy_inline` will return the same results.

**class** `aiida.orm.calculation.job.CalculationResultManager` (*calc*)

An object used internally to interface the calculation object with the Parser and consequentially with the ParameterData object result. It shouldn't be used explicitly by a user.

`__init__` (*calc*)

**Parameters** *calc* – the calculation object.

**class** `aiida.orm.calculation.job.JobCalculation` (\*\*kwargs)

This class provides the definition of an AiiDA calculation that is run remotely on a job scheduler.

`get_append_text` ()

Get the calculation-specific append text, which is going to be appended in the scheduler-job script, just after the code execution.

`get_custom_scheduler_commands` ()

Return a (possibly multiline) string with the commands that the user wants to manually set for the scheduler. See also the documentation of the corresponding `set_` method.

**Returns** the custom scheduler command, or an empty string if no custom command was defined.

`get_environment_variables` ()

Return a dictionary of the environment variables that are set for this calculation.

Return an empty dictionary if no special environment variables have to be set for this calculation.

`get_import_sys_environment` ()

To check if it's loading the system environment on the submission script.

**Returns** a boolean. If True the system environment will be load.

`get_job_id` ()

Get the scheduler job id of the calculation.

**Returns** a string

`get_max_memory_kb` ()

Get the memory (in KiloBytes) requested to the scheduler.

**Returns** an integer

`get_max_wallclock_seconds` ()

Get the max wallclock time in seconds requested to the scheduler.

**Returns** an integer

`get_mpirun_extra_params` ()

Return a list of strings, that are the extra params to pass to the mpirun (or equivalent) command after the one provided in `computer.mpirun_command`. Example: `mpirun -np 8 extra_params[0] extra_params[1] ... exec.x`

Return an empty list if no parameters have been defined.

**get\_parser\_name()**

Return a string locating the module that contains the output parser of this calculation, that will be searched in the 'aiida/parsers/plugins' directory. None if no parser is needed/set.

**Returns** a string.

**get\_parserclass()**

Return the output parser object for this calculation, or None if no parser is set.

**Returns** a Parser class.

**Raise** MissingPluginError from ParserFactory no plugin is found.

**get\_prepend\_text()**

Get the calculation-specific prepend text, which is going to be prepended in the scheduler-job script, just before the code execution.

**get\_priority()**

Get the priority, if set, of the job on the cluster.

**Returns** a string or None

**get\_queue\_name()**

Get the name of the queue on cluster.

**Returns** a string or None.

**get\_resources(full=False)**

Returns the dictionary of the job resources set.

**Parameters** **full** – if True, also add the default values, e.g.  
default\_mpiproc\_per\_machine

**Returns** a dictionary

**get\_retrieved\_node()**

Return the retrieved data folder, if present.

**Returns** the retrieved data folder object, or None if no such output node is found.

**Raises** MultipleObjectsError if more than one output node is found.

**get\_scheduler\_error()**

Return the output of the scheduler error (a string) if the calculation has finished, and output node is present, and the output of the scheduler was retrieved.

Return None otherwise.

**get\_scheduler\_output()**

Return the output of the scheduler output (a string) if the calculation has finished, and output node is present, and the output of the scheduler was retrieved.

Return None otherwise.

**get\_scheduler\_state()**

Return the status of the calculation according to the cluster scheduler.

**Returns** a string.

**get\_state(from\_attribute=False)**

Get the state of the calculation.

---

**Note:** the 'most recent' state is obtained using the logic in the `aiida.common.datastructures.sort_states` function.

---

**Todo**

Understand if the state returned when no state entry is found in the DB is the best choice.

---

**Parameters** `from_attribute` – if set to True, read it from the attributes (the attribute is also set with `set_state`, unless the state is set to IMPORTED; in this way we can also see the state before storing).

**Returns** a string. If `from_attribute` is True and no attribute is found, return None. If `from_attribute` is False and no entry is found in the DB, also return None.

**get\_withmpi ()**

Get whether the job is set with mpi execution.

**Returns** a boolean. Default=True.

**has\_failed ()**

Get whether the calculation is in a failed status, i.e. SUBMISSIONFAILED, RETRIEVALFAILED, PARSINGFAILED or FAILED.

**Returns** a boolean

**has\_finished\_ok ()**

Get whether the calculation is in the FINISHED status.

**Returns** a boolean

**kill ()**

Kill a calculation on the cluster.

Can only be called if the calculation is in status WITHSCHEDULER.

The command tries to run the kill command as provided by the scheduler, and raises an exception if something goes wrong. No changes of calculation status are done (they will be done later by the calculation manager).

**res**

To be used to get direct access to the parsed parameters.

**Returns** an instance of the CalculationResultManager.

**Note** a practical example on how it is meant to be used: let's say that there is a key 'energy' in the dictionary of the parsed results which contains a list of floats. The command `calc.res.energy` will return such a list.

**set\_append\_text (val)**

Set the calculation-specific append text, which is going to be appended in the scheduler-job script, just after the code execution.

**Parameters** `val` – a (possibly multiline) string

**set\_custom\_scheduler\_commands (val)**

Set a (possibly multiline) string with the commands that the user wants to manually set for the scheduler.

The difference of this method with respect to the `set_prepend_text` is the position in the scheduler submission file where such text is inserted: with this method, the string is inserted before any non-scheduler command.

**set\_environment\_variables (env\_vars\_dict)**

Set a dictionary of custom environment variables for this calculation.

Both keys and values must be strings.

In the remote-computer submission script, it's going to export variables as `export 'keys'='values'`

**set\_import\_sys\_environment** (*val*)

If set to true, the submission script will load the system environment variables.

**Parameters** **val** (*bool*) – load the environment if True

**set\_max\_memory\_kb** (*val*)

Set the maximum memory (in KiloBytes) to be asked to the scheduler.

**Parameters** **val** – an integer. Default=None

**set\_max\_wallclock\_seconds** (*val*)

Set the wallclock in seconds asked to the scheduler.

**Parameters** **val** – An integer. Default=None

**set\_mpirun\_extra\_params** (*extra\_params*)

Set the extra params to pass to the mpirun (or equivalent) command after the one provided in `computer.mpirun_command`. Example: `mpirun -np 8 extra_params[0] extra_params[1] ... exec.x`

**Parameters** **extra\_params** – must be a list of strings, one for each extra parameter

**set\_parser\_name** (*parser*)

Set a string for the output parser Can be None if no output plugin is available or needed.

**Parameters** **parser** – a string identifying the module of the parser. Such module must be located within the folder 'aiida/parsers/plugins'

**set\_prepend\_text** (*val*)

Set the calculation-specific prepend text, which is going to be prepended in the scheduler-job script, just before the code execution.

See also `set_custom_scheduler_commands`

**Parameters** **val** – a (possibly multiline) string

**set\_priority** (*val*)

Set the priority of the job to be queued.

**Parameters** **val** – the values of priority as accepted by the cluster scheduler.

**set\_queue\_name** (*val*)

Set the name of the queue on the remote computer.

**Parameters** **val** (*str*) – the queue name

**set\_resources** (*resources\_dict*)

Set the dictionary of resources to be used by the scheduler plugin, like the number of nodes, cpus, ... This dictionary is scheduler-plugin dependent. Look at the documentation of the scheduler. (scheduler type can be found with `calc.get_computer().get_scheduler_type()`)

**set\_withmpi** (*val*)

Set the calculation to use mpi.

**Parameters** **val** – A boolean. Default=True

**store** (*\*args, \*\*kwargs*)

Override the `store()` method to store also the calculation in the NEW state as soon as this is stored for the first time.

**submit ()**

Puts the calculation in the TOSUBMIT status.

Actual submission is performed by the daemon.

**submit\_test** (*folder=None, subfolder\_name=None*)

Test submission, creating the files in a local folder.

**Note** this submit\_test function does not require any node (neither the calculation nor the input links) to be stored yet.

#### Parameters

- **folder** – A Folder object, within which each calculation files are created; if not passed, a subfolder ‘submit\_test’ of the current folder is used.
- **subfolder\_name** – the name of the subfolder to use for this calculation (within Folder). If not passed, a unique string starting with the date and time in the format `yymmdd-HHMMSS-` is used.

**Quantum Espresso - PW** Plugin to create a Quantum Espresso pw.x file.

**class** `aiida.orm.calculation.job.quantum.espresso.pw.PwCalculation` (*\*\*kwargs*)

Main DFT code (PWscf, pw.x) of the Quantum ESPRESSO distribution. For more information, refer to <http://www.quantum-espresso.org/>

**Quantum Espresso - PW immigrant** Plugin to immigrate a Quantum Espresso pw.x job that was not run using AiiDa.

**class** `aiida.orm.calculation.job.quantum.espresso.pwimmigrant.PwimmigrantCalculation` (*\*\*kwargs*)

Create a PwCalculation object that can be used to import old jobs.

This is a subclass of `aiida.orm.calculation.quantum.espresso.PwCalculation` with slight modifications to some of the class variables and additional methods that

1. parse the job’s input file to create the calculation’s input nodes that would exist if the calculation were submitted using AiiDa,
2. bypass the functions of the daemon, and prepare the node’s attributes such that all the processes (copying of the files to the repository, results parsing, ect.) can be performed

---

**Note:** The keyword arguments of PwCalculation are also available.

---

#### Parameters

- **remote\_workdir** (*str*) – Absolute path to the directory where the job was run. The transport of the computer you link ask input to the calculation is the transport that will be used to retrieve the calculation’s files. Therefore, `remote_workdir` should be the absolute path to the job’s directory on that computer.
- **input\_file\_name** (*str*) – The file name of the job’s input file.
- **output\_file\_name** (*str*) – The file name of the job’s output file (i.e. the file containing the stdout of QE).

**create\_input\_nodes** (*open\_transport, input\_file\_name=None, output\_file\_name=None, remote\_workdir=None*)

Create calculation input nodes based on the job’s files.

**Parameters** `open_transport` (`aiida.transport.plugins.local.LocalTransport` | `aiida.transport.plugins.ssh.SshTransport`) – An open instance of the transport class of the calculation’s computer. See the tutorial for more information.

This method parses the files in the job’s remote working directory to create the input nodes that would exist if the calculation were submitted using AiiDa. These nodes are

- a ‘parameters’ `ParameterData` node, based on the namelists and their variable-value pairs;
- a ‘kpoints’ `KpointsData` node, based on the `K_POINTS` card;
- a ‘structure’ `StructureData` node, based on the `ATOMIC_POSITIONS` and `CELL_PARAMETERS` cards;
- one ‘pseudo\_X’ `UpfData` node for the pseudopotential used for the atomic species with name X, as specified in the `ATOMIC_SPECIES` card;
- a ‘settings’ `ParameterData` node, if there are any fixed coordinates, or if the gamma kpoint is used;

and can be retrieved as a dictionary using the `get_inputs_dict()` method. *These input links are cached-links; nothing is stored by this method (including the calculation node itself).*

---

**Note:** QE stores the calculation’s pseudopotential files in the `<outdir>/<prefix>.save/` subfolder of the job’s working directory, where `outdir` and `prefix` are QE *CONTROL* variables (see [pw input file description](#)). This method uses these files to either get—if the a node already exists for the pseudo—or create a `UpfData` node for each pseudopotential.

---

### Keyword arguments

---

**Note:** These keyword arguments can also be set when instantiating the class or using the `set_` methods (e.g. `set_remote_workdir`). Offering to set them here simply offers the user an additional place to set their values. *Only the values that have not yet been set need to be specified.*

---

### Parameters

- `input_file_name` (*str*) – The file name of the job’s input file.
- `output_file_name` (*str*) – The file name of the job’s output file (i.e. the file containing the stdout of QE).
- `remote_workdir` (*str*) – Absolute path to the directory where the job was run. The transport of the computer you link ask input to the calculation is the transport that will be used to retrieve the calculation’s files. Therefore, `remote_workdir` should be the absolute path to the job’s directory on that computer.

### Raises

- `aiida.common.exceptions.InputValidationError` – if `open_transport` is a different type of transport than the computer’s.
- `aiida.common.exceptions.InvalidOperation` – if `open_transport` is not open.
- `aiida.common.exceptions.InputValidationError` – if `remote_workdir`, `input_file_name`, and/or `output_file_name` are not set prior to or during the call of this method.
- `aiida.common.exceptions.FeatureNotAvailable` – if the input file uses anything other than `ibrav = 0`, which is not currently implimented in aiiida.

- `aiida.common.exceptions.ParsingError` – if there are issues parsing the input file.
- `IOError` – if there are issues reading the input file.

#### **prepare\_for\_retrieval\_and\_parsing** (*open\_transport*)

Tell the daemon that the calculation is computed and ready to be parsed.

**Parameters** `open_transport` (*aiida.transport.plugins.local.LocalTransport* | *aiida.transport.plugins.ssh.SshTransport*) – An open instance of the transport class of the calculation’s computer. See the tutorial for more information.

The next time the daemon updates the status of calculations, it will see this job is in the ‘COMPUTED’ state and will retrieve its output files and parse the results.

If the daemon is not currently running, nothing will happen until it is started again.

This method also stores the calculation and all input nodes. It also copies the original input file to the calculation’s repository folder.

#### **Raises**

- `aiida.common.exceptions.InputValidationError` – if `open_transport` is a different type of transport than the computer’s.
- `aiida.common.exceptions.InvalidOperation` – if `open_transport` is not open.

#### **set\_input\_file\_name** (*input\_file\_name*)

Set the file name of the job’s input file (e.g. ‘pw.in’).

**Parameters** `input_file_name` (*str*) – The file name of the job’s input file.

#### **set\_output\_file\_name** (*output\_file\_name*)

Set the file name of the job’s output file (e.g. ‘pw.out’).

**Parameters** `output_file_name` (*str*) – The file name of file containing the job’s stdout.

#### **set\_output\_subfolder** (*output\_subfolder*)

Manually set the job’s `outdir` variable (e.g. ‘./out/’).

---

**Note:** The `outdir` variable is normally set automatically by

1. looking for the `outdir` CONTROL namelist variable
2. looking for the `$ESPRESSO_TMPDIR` environment variable on the calculation’s computer (using the transport)
3. using the QE default, the calculation’s `remote_workdir`

but this method is made available to the user, in the event that they wish to set it manually.

---

**Parameters** `output_subfolder` (*str*) – The job’s `outdir` variable.

#### **set\_prefix** (*prefix*)

Manually set the job’s `prefix` variable (e.g. ‘pwscf’).

---

**Note:** The `prefix` variable is normally set automatically by

1. looking for the `prefix` CONTROL namelist variable
2. using the QE default, ‘pwscf’

but this method is made available to the user, in the event that they wish to set it manually.

---

**Parameters** `prefix` (*str*) – The job’s prefix variable.

**set\_remote\_workdir** (*remote\_workdir*)

Set the job’s remote working directory.

**Parameters** `remote_workdir` (*str*) – Absolute path of the job’s remote working directory.

**TemplateReplacer** This is a simple plugin that takes two node inputs, both of type `ParameterData`, with the following labels: `template` and `parameters`. You can also add other `SinglefileData` nodes as input, that will be copied according to what is written in ‘`template`’ (see below).

- `parameters`: a set of parameters that will be used for substitution.
- `template`: can contain the following parameters:
  - `input_file_template`: a string with substitutions to be managed with the `format()` function of python, i.e. if you want to substitute a variable called ‘`varname`’, you write `{varname}` in the text. See <http://www.python.org/dev/peps/pep-3101/> for more details. The replaced file will be the input file.
  - `input_file_name`: a string with the file name for the input. If it is not provided, no file will be created.
  - `output_file_name`: a string with the file name for the output. If it is not provided, no redirection will be done and the output will go in the scheduler output file.
  - `cmdline_params`: a list of strings, to be passed as command line parameters. Each one is substituted with the same rule of `input_file_template`. Optional
  - `input_through_stdin`: if `True`, the input file name is passed via `stdin`. Default is `False` if missing.
  - **files\_to\_copy**: if defined, a list of tuple pairs, with format (`‘link_name’`, `‘dest_rel_path’`); for each tuple, an input link and with file type ‘`Singlefile`’, and the content is copied to a remote file named ‘`dest_rel_path`’ Errors are raised in the input links are non-existent, or of the wrong type, or if there are unused input files.

TODO: probably use Python’s Template strings instead?? TODO: catch exceptions

**class** `aiida.orm.calculation.job.simpleplugins.template_replacer.TemplateReplacerCalculation` (\*\*)  
 Simple stub of a plugin that can be used to replace some text in a given template. Can be used for many different codes, or as a starting point to develop a new plugin.

**Calculation parsers** This section describes the different parsers classes for calculations.

### Quantum ESPRESSO parsers

`aiida.parsers.plugins.quantum.espresso.convert_qe2aiida_structure` (*output\_dict*,  
*in-*  
*put\_structure=None*)  
 Receives the dictionary cell parsed from quantum espresso Convert it into an AiiDA structure object

### Basic Raw Cp Parser

`aiida.parsers.plugins.quantum.espresso.basic_raw_parser_cp.parse_cp_text_output` (*data*,  
*xml\_data*)  
 data must be a list of strings, one for each lines, as returned by `readlines()`. On output, a dictionary with parsed values



`aiida.parsers.plugins.quantumespresso.basic_raw_parser_cp.parse_cp_traj_stanzas` (*num\_elements*,  
*split\_lines*,  
*prepend\_name*,  
*rescale=1.0*)

*num\_elements*: Number of lines (with three elements) between lines with two only elements (containing step number and time in ps). *num\_elements* is 3 for cell, and the number of atoms for coordinates and positions.

*splitlines*: a list of lines of the file, already split in pieces using `string.split`

*prepend\_name*: a string to be prepended to the name of keys returned in the return dictionary.

*rescale*: the values in each stanza are multiplied by this factor, for units conversion

`aiida.parsers.plugins.quantumespresso.basic_raw_parser_cp.parse_cp_xml_counter_output` (*data*)  
 Parse xml file `print_counter.xml` data must be a single string, as returned by `file.read()` (notice the difference with `parse_text_output`!) On output, a dictionary with parsed values.

`aiida.parsers.plugins.quantumespresso.basic_raw_parser_cp.parse_cp_xml_output` (*data*)  
 Parse xml data data must be a single string, as returned by `file.read()` (notice the difference with `parse_text_output`!) On output, a dictionary with parsed values. Democratically, we have decided to use picoseconds as units of time, eV for energies, Angstrom for lengths.

**Basic Raw Pw Parser** A collection of function that are used to parse the output of Quantum Espresso PW. The function that needs to be called from outside is `parse_raw_output()`. The functions mostly work without aiiDA specific functionalities. The parsing will try to convert whatever it can in some dictionary, which by operative decision doesn't have much structure encoded, [the values are simple ]

`aiida.parsers.plugins.quantumespresso.basic_raw_parser_pw.cell_volume` (*a1*,  
*a2*,  
*a3*)

returns the volume of the primitive cell:  $a_1(a_2 \times a_3)$

`aiida.parsers.plugins.quantumespresso.basic_raw_parser_pw.convert_list_to_matrix` (*in\_matrix*,  
*n\_rows*,  
*n\_columns*)

converts a list into a list of lists (a matrix like) with *n\_rows* and *n\_columns*

`aiida.parsers.plugins.quantumespresso.basic_raw_parser_pw.convert_qe_time_to_sec` (*timestr*)  
 Given the walltime string of Quantum Espresso, converts it in a number of seconds (float).

`aiida.parsers.plugins.quantumespresso.basic_raw_parser_pw.parse_QE_errors` (*lines*,  
*count*,  
*warnings*)

Parse QE errors messages (those appearing between some lines with ' %%%%%%%%%%')

#### Parameters

- **lines** – list of strings, the output text file as read by `readlines()` or as obtained by `data.split('n')` when data is the text file read by `read()`
- **count** – the line at which we identified some ' %%%%%%%%%%'
- **warnings** – the warnings already parsed in the file

**Return messages** a list of QE error messages

`aiida.parsers.plugins.quantumespresso.basic_raw_parser_pw.parse_pw_text_output` (*data*,  
*xml\_data=None*,  
*structure\_data=None*,  
*input\_dict=None*)

Parses the text output of QE-PWscf.

#### Parameters

- **data** – a string, the file as read by `read()`
- **xml\_data** – the dictionary with the keys read from xml.
- **structure\_data** – dictionary, coming from the xml, with info on the structure

**Return parsed\_data** dictionary with key values, referring to quantities at the last scf step.

**Return trajectory\_data** key,values referring to intermediate scf steps, as in the case of vc-relax.  
Empty dictionary if no value is present.

**Return critical\_messages** a list with critical messages. If any is found in `parsed_data['warnings']`,  
the calculation is FAILED!

`aiida.parsers.plugins.quantumespresso.basic_raw_parser_pw.parse_pw_xml_output` (*data*,  
*dir\_with\_bands=None*)

Parse the xml data of QE v5.0.x Input data must be a single string, as returned by `file.read()` Returns a dictionary  
with parsed values

`aiida.parsers.plugins.quantumespresso.basic_raw_parser_pw.parse_raw_output` (*out\_file*,  
*input\_dict*,  
*parser\_opts=None*,  
*xml\_file=None*,  
*dir\_with\_bands=None*)

Parses the output of a calculation Receives in input the paths to the output file and the xml file.

#### Parameters

- **out\_file** – path to pw std output
- **input\_dict** – not used
- **parser\_opts** – not used
- **dir\_with\_bands** – path to directory with all k-points (Kxxxxx) folders
- **xml\_file** – path to QE data-file.xml

**Returns out\_dict** a dictionary with parsed data

**Return successful** a boolean that is False in case of failed calculations

#### Raises

- **QEOutputParsingError** – for errors in the parsing,
- **AssertionError** – if two keys in the parsed dicts are found to be qual

3 different keys to check in output: `parser_warnings`, `xml_warnings` and `warnings`. On an upper level, these  
flags MUST be checked. The first two are expected to be empty unless QE failures or unfinished jobs.

**Basic Pw Parser**

**class** `aiida.parsers.plugins.quantumespresso.basicpw.BasicpwParser` (*calc*)

This class is the implementation of the Parser class for PWscf.

**get\_linkname\_out\_kpoints** ()

Returns the name of the link to the output\_kpoints Node exists if cell has changed and no bands are stored.

**get\_linkname\_outarray** ()

Returns the name of the link to the output\_array Node may exist in case of calculation='scf'

**get\_linkname\_outstructure** ()

Returns the name of the link to the output\_structure Node exists if positions or cell changed.

**get\_linkname\_outtrajectory** ()

Returns the name of the link to the output\_trajectory. Node exists in case of calculation='md', 'vc-md', 'relax', 'vc-relax'

**get\_parser\_settings\_key** ()

Return the name of the key to be used in the calculation settings, that contains the dictionary with the parser\_options

**parse\_with\_retrieved** (*retrieved*)

Receives in input a dictionary of retrieved nodes. Does all the logic here.

**Constants** Physical or mathematical constants. Since every code has its own conversion units, this module defines what QE understands as for an eV or other quantities. Whenever possible, we try to use the constants defined in `:py:mod:aiida.common.constants:`, but if some constants are slightly different among different codes (e.g., different standard definition), we define the constants in this file.

**Cp Parser**

**class** `aiida.parsers.plugins.quantumespresso.cp.CpParser` (*calc*)

This class is the implementation of the Parser class for Cp.

**get\_linkname\_trajectory** ()

Returns the name of the link to the output\_structure (None if not present)

**parse\_with\_retrieved** (*retrieved*)

Receives in input a dictionary of retrieved nodes. Does all the logic here.

**Automodules Example**

This module defines the main data structures used by the Calculation.

**class** `aiida.common.datastructures.CalcInfo` (*init={}*)

This object will store the data returned by the calculation plugin and to be passed to the ExecManager

**class** `aiida.common.datastructures.CodeInfo` (*init={}*)

This attribute-dictionary contains the information needed to execute a code. Possible attributes are:

- `cmdline_params`: a list of strings, containing parameters to be written on the command line right after the call to the code, as for example:

```
code.x cmdline_params[0] cmdline_params[1] ... < stdin > stdout
```

- `stdin_name`: (optional) the name of the standard input file. Note, it is only possible to use the stdin with the syntax:

```
code.x < stdin_name
```

If no `stdin_name` is specified, the string “< stdin\_name” will not be passed to the code. Note: it is not possible to substitute/remove the ‘<’ if `stdin_name` is specified; if that is needed, avoid `stdin_name` and use instead the `cmdline_params` to specify a suitable syntax.

- `stdout_name`: (optional) the name of the standard output file. Note, it is only possible to pass output to `stdout_name` with the syntax:

```
code.x ... > stdout_name
```

If no `stdout_name` is specified, the string “> stdout\_name” will not be passed to the code. Note: it is not possible to substitute/remove the ‘>’ if `stdout_name` is specified; if that is needed, avoid `stdout_name` and use instead the `cmdline_params` to specify a suitable syntax.

- `stderr_name`: (optional) a string, the name of the error file of the code.
- `join_files`: (optional) if `True`, redirects the error to the output file. If `join_files=True`, the code will be called as:

```
code.x ... > stdout_name 2>&1
```

otherwise, if `join_files=False` and `stderr` is passed:

```
code.x ... > stdout_name 2> stderr_name
```

- `withmpi`: if `True`, executes the code with `mpirun` (or another MPI installed on the remote computer)
- `code_uuid`: the `uuid` of the code associated to the `CodeInfo`

`aiida.common.datastructures.sort_states` (*list\_states*)

Given a list of state names, return a sorted list of states (the first is the most recent) sorted according to their logical appearance in the DB (i.e., `NEW` before of `SUBMITTING` before of `FINISHED`).

---

**Note:** The order of the internal variable `_sorted_datastates` is used.

---

**Parameters** `list_states` – a list (or tuple) of state strings.

**Returns** a sorted list of the given data states.

**Raises** `ValueError` if any of the given states is not a valid state.

---

**Note:** A `:noindex:` directive was added to avoid duplicate object description for this example. Do not put the keyword in a real documentation.

---

## How To Format Docstrings

Much of the work will be done automatically by Sphinx, just format the docstrings with the same syntax used here, a few extra examples of use would include:

```
:param parameters: some notes on input parameters

:return returned: some note on what is returned

:raise Errors: Notes on warnings raised
```

## Changing The Docs

If you are creating a new .rst file, make sure to add it in the relevant index.rst tree. This can be done by:

- Modifying relevant doc strings or .rst files (be sure to modify them in the /doc/source/ folder and not /doc/build)
- Making sure that all relevant .rst files are added to the relevant index.rst file
- Running *make html* in /aiida/docs/ folder
- Be sure to check for any warnings and correct if possible

## This Page

```
Sphinx cheatsheet
+++++

A brief overview of some of the main functions of Sphinx
as used in the aiida documentation. View :ref:`this-page` to see
how this page was formatted. This is only a brief outline for more
please see `the Sphinx documentation <http://sphinx-doc.org/contents.html>`_

Main Titles and Subtitles
-----

This is an example of a main title.

subtitles are made like this
=====

This is an example of a subtitle.

Formatting
-----

Basic Paragraph Formatting
=====

Words can be written in italics or in bold. Text describing a specific
``computer_thing`` can be formatted as well.

Paragraph and Indentation
=====

Much like in regular python, the indentation plays a strong role in the formatting.

For example all of this sentence will
appear on the same line.

While this sentence will appear
differently because there is an indent.

Terminal and Code Formatting
=====

Something to be run in command line can be formatted like this::
```

```
>> Some command
```

As can be seen above, while snippets of python on code can be done like this::

```
import module
print('hello world')
```

Notes

```
=====
.. note:: Notes can be added like this.
```

Bullet Points and Lists

```
=====

* Bullet points can be added
* Just like this
  * With sub-bullets like this

#. While numerical bullets
#. Can be added
#. Like this
```

Links, Code Display, Cross References

```
-----
```

External Links

```
=====
Can be done like here for `AiiDA <www.aiida.net/>`_
```

Code Download

```
=====

Code can be downloaded like this.

Download: :download:`this example script <devel_tutorial/sum_executable.py>`
```

Code Display

```
=====

Can be done like this. This entire document can be seen unformatted below using this method.

.. literalinclude:: devel_tutorial/sum_executable.py

.. _self-reference:
```

Cross Reference Docs

```
=====

Here is an example of a reference to the :ref:`structure_tutorial` which is on *another page*

Here is an example of a reference to something on the same page, :ref:`self-reference`

.. note:: References within the same document need a reference label, see `.. _self-reference:`
         used in this section for an example. *Hidden in formatted page, can only be seen in the
         input text.*
```

Cross Reference Classes and Methods

```
=====
```

Any class can be referenced for example `:py:class:`~aiida.orm.data.structure.StructureData`` reference the `StructureData` data class.

Similarly any method can be referenced for example `:py:meth:`~aiida.orm.data.structure.StructureData.append_atom`` shows the `StructureData` class' `append_atom` method.

#### Table of Contents Docs and Code

##### Table of Contents for Docs

An example of the table of contents syntax for the `:ref:`git-cheatsheet`` can be seen here note that these are especially important in the global structure of the document, as found in `index.rst` files.

```
.. toctree::
    :maxdepth: 2

    git_cheatsheet
```

.. note:: The ``maxdepth`` parameter can be used to change how deep the title indexing goes. See `:ref:``

##### Table of Contents for Code

Table of contents, that cross reference code, can be done very similarly to how it is done for documents. For example the parser docs can be indexed like this

```
.. toctree::
    :maxdepth: 1

    aiida.orm <../orm/dev>
    ../parsers/dev
```

##### Automodules Example

```
.. toctree::
    :maxdepth: 2

.. automodule:: aiida.common.datastructures
    :members:
    :noindex:

.. note:: A `noindex:` directive was added to avoid duplicate object
    description for this example. Do not put the keyword in a real
    documentation.
```

#### How To Format Docstrings

Much of the work will be done automatically by Sphinx, just format the docstrings with the same syntax a few extra examples of use would include::

```
:param parameters: some notes on input parameters
```

```

: return returned: some note on what is returned

: raise Errors: Notes on warnings raised

Changing The Docs
-----

If you are creating a new .rst file, make sure to add it in
the relevant index.rst tree. This can be done by:

* Modifying relevant doc strings or .rst files (be sure to modify them in the /doc/source/ folder and
* Making sure that all relevant .rst files are added
  to the relevant index.rst file
* Running `make html` in /aiida/docs/ folder
* Be sure to check for any warnings and correct if possible

.. _this-page:

This Page
=====

.. literalinclude:: sphinx_cheatsheet.rst

```

### 3.1.14 Properties

Properties are configuration options that are stored in the `config.json` file (within the `.aiida` directory). They can be accessed and modified thanks to `verdi devel` commands:

- **delproperty**: delete a given property.
- **describeproperty**: describe the content of a given property.
- **getproperty**: get the value of a given property.
- **listproperties**: list all user defined properties. With `-a` option, list all of them including those still at the default values.
- **setproperty**: set a given property (usage: `verdi devel setproperty PROPERTYNAME PROPERTYVALUE`).

For instance, modules to be loaded automatically in the `verdi shell` can be added by putting their paths (separated by colons `:`) in the property `verdishell.modules`, e.g. by typing something like:

```
verdi devel setproperty verdishell.modules aiida.common.exceptions.NotFound:aiida.orm.autogroup.Autogroup
```

More information can be found in the source code: see `setup.py`.



---

## Modules provided with aiida

---

### 4.1 Modules

#### 4.1.1 aiida.common documentation

##### Calculation datastructures

This module defines the main data structures used by the Calculation.

**class** `aiida.common.datastructures.CalcInfo` (*init={}*)

This object will store the data returned by the calculation plugin and to be passed to the ExecManager

**class** `aiida.common.datastructures.CodeInfo` (*init={}*)

This attribute-dictionary contains the information needed to execute a code. Possible attributes are:

- `cmdline_params`: a list of strings, containing parameters to be written on the command line right after the call to the code, as for example:

```
code.x cmdline_params[0] cmdline_params[1] ... < stdin > stdout
```

- `stdin_name`: (optional) the name of the standard input file. Note, it is only possible to use the stdin with the syntax:

```
code.x < stdin_name
```

If no `stdin_name` is specified, the string “< stdin\_name” will not be passed to the code. Note: it is not possible to substitute/remove the ‘<’ if `stdin_name` is specified; if that is needed, avoid `stdin_name` and use instead the `cmdline_params` to specify a suitable syntax.

- `stdout_name`: (optional) the name of the standard output file. Note, it is only possible to pass output to `stdout_name` with the syntax:

```
code.x ... > stdout_name
```

If no `stdout_name` is specified, the string “> stdout\_name” will not be passed to the code. Note: it is not possible to substitute/remove the ‘>’ if `stdout_name` is specified; if that is needed, avoid `stdout_name` and use instead the `cmdline_params` to specify a suitable syntax.

- `stderr_name`: (optional) a string, the name of the error file of the code.
- `join_files`: (optional) if True, redirects the error to the output file. If `join_files=True`, the code will be called as:

```
code.x ... > stdout_name 2>&1
```

otherwise, if `join_files=False` and `stderr` is passed:

```
code.x ... > stdout_name 2> stderr_name
```

- `withmpi`: if `True`, executes the code with `mpirun` (or another MPI installed on the remote computer)
- `code_uuid`: the uuid of the code associated to the `CodeInfo`

`aiida.common.datastructures.sort_states(list_states)`

Given a list of state names, return a sorted list of states (the first is the most recent) sorted according to their logical appearance in the DB (i.e., NEW before of SUBMITTING before of FINISHED).

---

**Note:** The order of the internal variable `_sorted_datastates` is used.

---

**Parameters** `list_states` – a list (or tuple) of state strings.

**Returns** a sorted list of the given data states.

**Raises** `ValueError` if any of the given states is not a valid state.

## Exceptions

**exception** `aiida.common.exceptions.AiidaException`

Base class for all AiiDA exceptions.

Each module will have its own subclass, inherited from this (e.g. `ExecManagerException`, `TransportException`, ...)

**exception** `aiida.common.exceptions.AuthenticationError`

Raised when a user tries to access a resource for which it is not authenticated, e.g. an `aiidauser` tries to access a computer for which there is no entry in the `AuthInfo` table.

**exception** `aiida.common.exceptions.ConfigurationError`

Error raised when there is a configuration error in AiiDA.

**exception** `aiida.common.exceptions.ContentNotExistent`

Raised when trying to access an attribute, a key or a file in the result nodes that is not present

**exception** `aiida.common.exceptions.DbContentError`

Raised when the content of the DB is not valid. This should never happen if the user does not play directly with the DB.

**exception** `aiida.common.exceptions.FailedError`

Raised when accessing a calculation that is in the FAILED status

**exception** `aiida.common.exceptions.FeatureDisabled`

Raised when a feature is requested, but the user has chosen to disable it (e.g., for submissions on disabled computers).

**exception** `aiida.common.exceptions.FeatureNotAvailable`

Raised when a feature is requested from a plugin, that is not available.

**exception** `aiida.common.exceptions.InputValidationError`

The input data for a calculation did not validate (e.g., missing required input data, wrong data, ...)

**exception** `aiida.common.exceptions.InternalError`

Error raised when there is an internal error of AiiDA.

**exception** `aiida.common.exceptions.InvalidOperation`

The allowed operation is not valid (e.g., when trying to add a non-internal attribute before saving the entry), or deleting an entry that is protected (e.g., because it is referenced by foreign keys)

**exception** `aiida.common.exceptions.LicensingException`

Raised when requirements for data licensing are not met.

**exception** `aiida.common.exceptions.LockPresent`

Raised when a lock is requested, but cannot be acquired.

**exception** `aiida.common.exceptions.MissingPluginError`

Raised when the user tries to use a plugin that is not available or does not exist.

**exception** `aiida.common.exceptions.ModificationNotAllowed`

Raised when the user tries to modify a field, object, property, ... that should not be modified.

**exception** `aiida.common.exceptions.MultipleObjectsError`

Raised when more than one entity is found in the DB, but only one was expected.

**exception** `aiida.common.exceptions.NotExistent`

Raised when the required entity does not exist.

**exception** `aiida.common.exceptions.ParsingError`

Generic error raised when there is a parsing error

**exception** `aiida.common.exceptions.PluginInternalError`

Error raised when there is an internal error which is due to a plugin and not to the AiiDA infrastructure.

**exception** `aiida.common.exceptions.ProfileConfigurationError`

Configuration error raised when a wrong/inexistent profile is requested.

**exception** `aiida.common.exceptions.RemoteOperationError`

Raised when an error in a remote operation occurs, as in a failed kill() of a scheduler job.

**exception** `aiida.common.exceptions.UniquenessError`

Raised when the user tries to violate a uniqueness constraint (on the DB, for instance).

**exception** `aiida.common.exceptions.ValidationError`

Error raised when there is an error during the validation phase of a property.

**exception** `aiida.common.exceptions.WorkflowInputValidationError`

The input data for a workflow did not validate (e.g., missing required input data, wrong data, ...)

**Extended dictionaries****class** `aiida.common.extendeddicts.AttributeDict (init={})`

This class internally stores values in a dictionary, but exposes the keys also as attributes, i.e. asking for `attrdict.key` will return the value of `attrdict['key']` and so on.

Raises an `AttributeError` if the key does not exist, when called as an attribute, while the usual `KeyError` if the key does not exist and the dictionary syntax is used.

**copy ()**

Shallow copy.

**class** `aiida.common.extendeddicts.DefaultFieldsAttributeDict (init={})`

A dictionary with access to the keys as attributes, and with an internal value storing the 'default' keys to be distinguished from extra fields.

Extra methods `defaultkeys()` and `extrakeys()` divide the set returned by `keys()` in default keys (i.e. those defined at definition time) and other keys. There is also a method `get_default_fields()` to return the internal list.

Moreover, for undefined default keys, it returns None instead of raising a KeyError/AttributeError exception.

Remember to define the `_default_fields` in a subclass! E.g.:

```
class TestExample(DefaultFieldsAttributeDict):
    _default_fields = ('a', 'b', 'c')
```

When the `validate()` method is called, it calls in turn all `validate_KEY` methods, where KEY is one of the default keys. If the method is not present, the field is considered to be always valid. Each `validate_KEY` method should accept a single argument 'value' that will contain the value to be checked.

It raises a `ValidationError` if any of the `validate_KEY` function raises an exception, otherwise it simply returns. NOTE: the `validate_` functions are called also for unset fields, so if the field can be empty on validation, you have to start your validation function with something similar to:

```
if value is None:
    return
```

---

### Todo

Decide behavior if I set to None a field. Current behavior, if `a` is an instance and 'def\_field' one of the default fields, that is undefined, we get:

- `a.get('def_field'): None`
- `a.get('def_field', 'whatever'): 'whatever'`
- Note that `a.defaultkeys()` does NOT contain 'def\_field'

if we do `a.def_field = None`, then the behavior becomes

- `a.get('def_field'): None`
- `a.get('def_field', 'whatever'): None`
- Note that `a.defaultkeys()` DOES contain 'def\_field'

See if we want that setting a default field to None means deleting it.

---

### **defaultkeys()**

Return the default keys defined in the instance.

### **extrakeys()**

Return the extra keys defined in the instance.

### **classmethod get\_default\_fields()**

Return the list of default fields, either defined in the instance or not.

### **validate()**

Validate the keys, if any `validate_*` method is available.

**class** `aiida.common.extendeddicts.FixedFieldsAttributeDict` (`init={}`)

A dictionary with access to the keys as attributes, and with filtering of valid attributes. This is only the base class, without valid attributes; use a derived class to do the actual work. E.g.:

```
class TestExample(FixedFieldsAttributeDict):
    _valid_fields = ('a', 'b', 'c')
```

### **classmethod get\_valid\_fields()**

Return the list of valid fields.

## Folders

**class** `aiida.common.folders.Folder` (*abspath*, *folder\_limit=None*)

A class to manage generic folders, avoiding to get out of specific given folder borders.

---

### Todo

fix this, `os.path.commonprefix` of `/a/b/c` and `/a/b2/c` will give `a/b`, check if this is wanted or if we want to put trailing slashes. (or if we want to use `os.path.relpath` and check for a string starting with `os.pardir`?)

---

### Todo

rethink whether the `folder_limit` option is still useful. If not, remove it altogether (it was a nice feature, but unfortunately all the calls to `os.path.abspath` or `normpath` are quite slow).

---

### **abspath**

The absolute path of the folder.

### **create()**

Creates the folder, if it does not exist on the disk yet.

It will also create top directories, if absent.

It is always safe to call it, it will do nothing if the folder already exists.

### **create\_file\_from\_filelike** (*src\_filelike*, *dest\_name*)

Create a file from a file-like object.

**Note** if the current file position in `src_filelike` is not 0, only the contents from the current file position to the end of the file will be copied in the new file.

#### Parameters

- **src\_filelike** – the file-like object (e.g., if you have a string called `s`, you can pass `StringIO.StringIO(s)`)
- **dest\_name** – the destination filename will have this file name.

### **create\_symlink** (*src*, *name*)

Create a symlink inside the folder to the location 'src'.

#### Parameters

- **src** – the location to which the symlink must point. Can be either a relative or an absolute path. Should, however, be relative to work properly also when the repository is moved!
- **name** – the filename of the symlink to be created.

### **erase** (*create\_empty\_folder=False*)

Erases the folder. Should be called only in very specific cases, in general folder should not be erased!

Doesn't complain if the folder does not exist.

**Parameters** **create\_empty\_folder** – if True, after erasing, creates an empty dir.

### **exists()**

Return True if the folder exists, False otherwise.

### **folder\_limit**

The folder limit that cannot be crossed when creating files and folders.

### **get\_abs\_path** (*relpath*, *check\_existence=False*)

Return an absolute path for a file or folder in this folder.

The advantage of using this method is that it checks that filename is a valid filename within this folder, and not something e.g. containing slashes.

#### Parameters

- **filename** – The file or directory.
- **check\_existence** – if False, just return the file path. Otherwise, also check if the file or directory actually exists. Raise OSError if it does not.

**get\_content\_list** (*pattern='\*', only\_paths=True*)

Return a list of files (and subfolders) in the folder, matching a given pattern.

Example: If you want to exclude files starting with a dot, you can call this method with `pattern='[!.]*'`

#### Parameters

- **pattern** – a pattern for the file/folder names, using Unix filename pattern matching (see Python standard module `fnmatch`). By default, pattern is `'*'`, matching all files and folders.
- **only\_paths** – if False (default), return pairs (name, is\_file). if True, return only a flat list.

**Returns** a list of tuples of two elements, the first is the file name and the second is True if the element is a file, False if it is a directory.

**get\_subfolder** (*subfolder, create=False, reset\_limit=False*)

Return a Folder object pointing to a subfolder.

#### Parameters

- **subfolder** – a string with the relative path of the subfolder, relative to the absolute path of this object. Note that this may also contain `'..'` parts, as far as this does not go beyond the `folder_limit`.
- **create** – if True, the new subfolder is created, if it does not exist.
- **reset\_limit** – when doing `b = a.get_subfolder('xxx', reset_limit=False)`, the limit of b will be the same limit of a. if True, the limit will be set to the boundaries of folder b.

**Returns** a Folder object pointing to the subfolder.

**insert\_path** (*src, dest\_name=None, overwrite=True*)

Copy a file to the folder.

#### Parameters

- **src** – the source filename to copy
- **dest\_name** – if None, the same basename of src is used. Otherwise, the destination filename will have this file name.
- **overwrite** – if False, raises an error on existing destination; otherwise, delete it first.

**isdir** (*relpath*)

Return True if 'relpath' exists inside the folder and is a directory, False otherwise.

**isfile** (*relpath*)

Return True if 'relpath' exists inside the folder and is a file, False otherwise.

**mode\_dir**

Return the mode with which the folders should be created

**mode\_file**

Return the mode with which the files should be created

**open** (*name*, *mode*='r')

Open a file in the current folder and return the corresponding file object.

**remove\_path** (*filename*)

Remove a file or folder from the folder.

**Parameters** **filename** – the relative path name to remove

**replace\_with\_folder** (*srcdir*, *move*=False, *overwrite*=False)

This routine copies or moves the source folder 'srcdir' to the local folder pointed by this Folder object.

**Parameters**

- **srcdir** – the source folder on the disk; this must be a string with an absolute path
- **move** – if True, the srcdir is moved to the repository. Otherwise, it is only copied.
- **overwrite** – if True, the folder will be erased first. if False, a IOError is raised if the folder already exists. Whatever the value of this flag, parent directories will be created, if needed.

**Raises** OSError or IOError: in case of problems accessing or writing the files.

**Raises** ValueError: if the section is not recognized.

**class** `aiida.common.folders.RepositoryFolder` (*section*, *uuid*, *subfolder*='.')

A class to manage the local AiiDA repository folders.

**get\_topdir** ()

Returns the top directory, i.e., the section/uuid folder object.

**section**

The section to which this folder belongs.

**subfolder**

The subfolder within the section/uuid folder.

**uuid**

The uuid to which this folder belongs.

**class** `aiida.common.folders.SandboxFolder`

A class to manage the creation and management of a sandbox folder.

Note: this class must be used within a context manager, i.e.:

**with** `SandboxFolder` **as** `f`: `## do something with f`

In this way, the sandbox folder is removed from disk (if it wasn't removed already) when exiting the 'with' block.

**Todo**

Implement check of whether the folder has been removed.

## Plugin loaders

`aiida.common.pluginloader.BaseFactory` (*module*, *base\_class*, *base\_modname*, *suffix*=None)

Return a given subclass of Calculation, loading the correct plugin.

**Example** If `module='quantumespresso.pw'`, `base_class=JobCalculation`, `base_modname = 'aiida.orm.calculation.job'`, and `suffix='Calculation'`, the code will first look for a `pw` subclass of `JobCalculation` inside the `quantumespresso` module. Lacking such a class, it will try to look for a `PwCalculation` inside the `quantumespresso.pw` module. In the latter case, the plugin class must have a specific name and be located in a specific file: if for instance `plugin_name == 'ssh'` and `base_class.__name__ == 'Transport'`, then there must be a class named `'SshTransport'` which is a subclass of `base_class` in a file `'ssh.py'` in the `plugins_module` folder. To create the class name to look for, the code will attach the string passed in the `base_modname` (after the last dot) and the `suffix` parameter, if passed, with the proper CamelCase capitalization. If `suffix` is not passed, the default suffix that is used is the `base_class` class name.

#### Parameters

- **module** – a string with the module of the plugin to load, e.g. `'quantumespresso.pw'`.
- **base\_class** – a base class from which the returned class should inherit. e.g.: `JobCalculation`
- **base\_modname** – a basic module name, under which the module should be found. E.g., `'aiida.orm.calculation.job'`.
- **suffix** – If specified, the suffix that the class name will have. By default, use the name of the `base_class`.

```
aiida.common.pluginloader.existing_plugins(base_class, plugins_module_name,
                                           max_depth=5, suffix=None)
```

Return a list of strings of valid plugins.

#### Parameters

- **base\_class** – Identify all subclasses of the `base_class`
- **plugins\_module\_name** – a string with the full module name separated with dots that points to the folder with plugins. It must be importable by python.
- **max\_depth** – Maximum depth (of nested modules) to be used when looking for plugins
- **suffix** – The suffix that is appended to the basename when looking for the (sub)class name. If not provided (or `None`), use the base class name.

**Returns** a list of valid strings that can be used using a `Factory` or with `load_plugin`.

```
aiida.common.pluginloader.get_class_typestring(type_string)
```

Given the type string, return three strings: the first one is one of the first-level classes that the `Node` can be: “node”, “calculation”, “code”, “data”. The second string is the one that can be passed to the `DataFactory` or `CalculationFactory` (or an empty string for nodes and codes); the third one is the name of the python class that would be loaded.

```
aiida.common.pluginloader.load_plugin(base_class, plugins_module, plugin_type)
```

Load a specific plugin for the given base class.

This is general and works for any plugin used in AiiDA.

**NOTE:** actually, now `plugins_module` and `plugin_type` are joined with a dot, and the plugin is retrieved splitting using the last dot of the resulting string.

**TODO:** understand if it is probably better to join the two parameters above to a single one.

#### Args:

**base\_class** the abstract base class of the plugin.

**plugins\_module** a string with the full module name separated with dots that points to the folder with plugins. It must be importable by python.



**plugin\_type** the name of the plugin.

**Return:** the class of the required plugin.

**Raise:** MissingPluginError if the plugin cannot be loaded

**Example:**

```
plugin_class = load_plugin( aiida.transport.Transport, 'aiida.transport.plugins', 'ssh.SshTransport' )
```

and `plugin_class` will be the class `'aiida.transport.plugins.ssh.SshTransport'`

## Utilities

**class** `aiida.common.utils.classproperty` (*getter*)

A class that, when used as a decorator, works as if the two decorators `@property` and `@classmethod` were applied together (i.e., the object works as a property, both for the Class and for any of its instance; and is called with the class `cls` rather than with the instance as its first argument).

`aiida.common.utils.conv_to_fortran` (*val*)

**Parameters** **val** – the value to be read and converted to a Fortran-friendly string.

`aiida.common.utils.create_display_name` (*field*)

Given a string, creates the suitable “default” display name: replace underscores with spaces, and capitalize each word.

**Returns** the converted string

`aiida.common.utils.escape_for_bash` (*str\_to\_escape*)

This function takes any string and escapes it in a way that bash will interpret it as a single string.

Explanation:

At the end, in the return statement, the string is put within single quotes. Therefore, the only thing that I have to escape in bash is the single quote character. To do this, I substitute every single quote ‘ with ‘`''''`’ which means:

First single quote: exit from the enclosing single quotes

Second, third and fourth character: ‘`'''`’ is a single quote character, escaped by double quotes

Last single quote: reopen the single quote to continue the string

Finally, note that for python I have to enclose the string ‘`'''`’ within triple quotes to make it work, getting finally: the complicated string found below.

`aiida.common.utils.export_shard_uuid` (*uuid*)

Sharding of the UUID for the import/export

`aiida.common.utils.get_class_string` (*obj*)

Return the string identifying the class of the object (module + object name, joined by dots).

It works both for classes and for class instances.

`aiida.common.utils.get_extremas_from_positions` (*positions*)

returns the minimum and maximum value for each dimension in the positions given

`aiida.common.utils.get_new_uuid` ()

Return a new UUID (typically to be used for new nodes). It uses the UUID version specified in `aiida.djsite.settings.settings_profile.AIIDANODES_UUID_VERSION`

`aiida.common.utils.get_object_from_string` (*string*)

Given a string identifying an object (as returned by the `get_class_string` method) load and return the actual object.

`aiida.common.utils.get_repository_folder(subfolder=None)`

Return the top folder of the local repository.

`aiida.common.utils.get_suggestion(provided_string, allowed_strings)`

Given a string and a list of allowed\_strings, it returns a string to print on screen, with sensible text depending on whether no suggestion is found, or one or more than one suggestions are found.

**Args:** provided\_string: the string to compare allowed\_strings: a list of valid strings

**Returns:** A string to print on output, to suggest to the user a possible valid value.

`aiida.common.utils.get_unique_filename(filename, list_of_filenames)`

Return a unique filename that can be added to the list\_of\_filenames.

If filename is not in list\_of\_filenames, it simply returns the filename string itself. Otherwise, it appends a integer number to the filename (before the extension) until it finds a unique filename.

#### Parameters

- **filename** – the filename to add
- **list\_of\_filenames** – the list of filenames to which filename should be added, without name duplicates

**Returns** Either filename or its modification, with a number appended between the name and the extension.

`aiida.common.utils.grouper(n, iterable)`

Given an iterable, returns an iterable that returns tuples of groups of elements from iterable of length n, except the last one that has the required length to exhaust iterable (i.e., there is no filling applied).

#### Parameters

- **n** – length of each tuple (except the last one, that will have length <= n)
- **iterable** – the iterable to divide in groups

`aiida.common.utils.gunzip_string(string)`

Gunzip string contents.

**Parameters** **string** – a gzipped string

**Returns** a string

`aiida.common.utils.gzip_string(string)`

Gzip string contents.

**Parameters** **string** – a string

**Returns** a gzipped string

`aiida.common.utils.md5_file(filename, block_size_factor=128)`

Open a file and return its md5sum (hexdigested).

#### Parameters

- **filename** – the filename of the file for which we want the md5sum
- **block\_size\_factor** – the file is read at chunks of size `block_size_factor * md5.block_size`, where `md5.block_size` is the block\_size used internally by the hashlib module.

**Returns** a string with the hexdigest md5.

**Raises** No checks are done on the file, so if it doesn't exist it may raise IOError.

`aiida.common.utils.shal_file(filename, block_size_factor=128)`

Open a file and return its shalsum (hexdigested).

#### Parameters

- **filename** – the filename of the file for which we want the shalsum
- **block\_size\_factor** – the file is read at chunks of size `block_size_factor * shal.block_size`, where `shal.block_size` is the `block_size` used internally by the `hashlib` module.

**Returns** a string with the hexdigest shal.

**Raises** No checks are done on the file, so if it doesn't exists it may raise `IOError`.

`aiida.common.utils.str_timedelta(dt, max_num_fields=3, short=False, negative_to_zero=False)`

Given a `dt` in seconds, return it in a HH:MM:SS format.

#### Parameters

- **dt** – a `TimeDelta` object
- **max\_num\_fields** – maximum number of non-zero fields to show (for instance if the number of days is non-zero, shows only days, hours and minutes, but not seconds)
- **short** – if `False`, print always `max_num_fields` fields, even if they are zero. If `True`, do not print the first fields, if they are zero.
- **negative\_to\_zero** – if `True`, set `dt = 0` if `dt < 0`.

`aiida.common.utils.validate_list_of_string_tuples(val, tuple_length)`

Check that:

1. `val` is a list or tuple
2. each element of the list:
  1. is a list or tuple
  2. is of length equal to the parameter `tuple_length`
  3. each of the two elements is a string

Return if valid, raise `ValidationError` if invalid

`aiida.common.utils.xyz_parser_iterator(string)`

Yields a tuple (*natoms*, *comment*, *atomiter*) for each frame in a XYZ file where *atomiter* is an iterator yielding a nested tuple (*symbol*, (*x*, *y*, *z*)) for each entry.

**Parameters** **string** – a string containing XYZ-structured text

## 4.1.2 aiida.transport documentation

This chapter describes the generic implementation of a transport plugin. The currently implemented are the local and the ssh plugin. The local plugin makes use only of some standard python modules like `os` and `shutil`. The ssh plugin is a wrapper to the library `paramiko`, that you installed with AiiDA.

A generic set of tests is contained in `plugin_test.py`, while plugin-specific tests are written separately.

## Generic transport class

**class** `aiida.transport.__init__.FileAttribute` (*init={}*)

A class, resembling a dictionary, to describe the attributes of a file, that is returned by `get_attribute()`. Possible keys: `st_size`, `st_uid`, `st_gid`, `st_mode`, `st_atime`, `st_mtime`

**class** `aiida.transport.__init__.Transport` (*\*args, \*\*kwargs*)

Abstract class for a generic transport (ssh, local, ...) Contains the set of minimal methods

**`__enter__`** ()

For transports that require opening a connection, opens all required channels (used in 'with' statements)

**`__exit__`** (*type, value, traceback*)

Closes connections, if needed (used in 'with' statements).

**`chdir`** (*path*)

Change directory to 'path'

**Parameters** **`path`** (*str*) – path to change working directory into.

**Raises** `IOError`, if the requested path does not exist

**Return type** `string`

**`chmod`** (*path, mode*)

Change permissions of a path.

**Parameters**

- **`path`** (*str*) – path to file
- **`mode`** (*int*) – new permissions

**`chown`** (*path, uid, gid*)

Change the owner (uid) and group (gid) of a file. As with python's `os.chown` function, you must pass both arguments, so if you only want to change one, use `stat` first to retrieve the current owner and group.

**Parameters**

- **`path`** (*str*) – path to the file to change the owner and group of
- **`uid`** (*int*) – new owner's uid
- **`gid`** (*int*) – new group id

**`close`** ()

Closes the local transport channel

**`copy`** (*remotesource, remotedestination, \*args, \*\*kwargs*)

Copy a file or a directory from remote source to remote destination (On the same remote machine)

**Parameters**

- **`remotesource`** (*str*) – path of the remote source directory / file
- **`remotedestination`** (*str*) – path of the remote destination directory / file

**Raises** `IOError`, if one of src or dst does not exist

**`copy_from_remote_to_remote`** (*transportdestination, remotesource, remotedestination, \*\*kwargs*)

Copy files or folders from a remote computer to another remote computer.

**Parameters**

- **`transportdestination`** – transport to be used for the destination computer

- **remotesource** (*str*) – path to the remote source directory / file
- **remotedestination** (*str*) – path to the remote destination directory / file
- **kwargs** – keyword parameters passed to the call to `transportdestination.put`, except for ‘dereference’ that is passed to `self.get`

---

**Note:** the keyword ‘dereference’ SHOULD be set to `False` for the final put (onto the destination), while it can be set to the value given in `kwargs` for the get from the source. In that way, a symbolic link would never be followed in the final copy to the remote destination. That way we could avoid getting unknown (potentially malicious) files into the destination computer. HOWEVER, since `dereference=False` is currently NOT supported by all plugins, we still force it to `True` for the final put.

---



---

**Note:** the supported keys in `kwargs` are `callback`, `dereference`, `overwrite` and `ignore_nonexisting`.

---

**copyfile** (*remotesource*, *remotedestination*, \*args, \*\*kwargs)

Copy a file from remote source to remote destination (On the same remote machine)

**Parameters**

- **remotesource** (*str*) – path of the remote source directory / file
- **remotedestination** (*str*) – path of the remote destination directory / file

**Raises IOError** if one of `src` or `dst` does not exist

**copytree** (*remotesource*, *remotedestination*, \*args, \*\*kwargs)

Copy a folder from remote source to remote destination (On the same remote machine)

**Parameters**

- **remotesource** (*str*) – path of the remote source directory / file
- **remotedestination** (*str*) – path of the remote destination directory / file

**Raises IOError** if one of `src` or `dst` does not exist

**exec\_command\_wait** (*command*, \*\*kwargs)

Execute the command on the shell, waits for it to finish, and return the `retcode`, the `stdout` and the `stderr`.

Enforce the execution to be run from the `pwd` (as given by `self.getcwd`), if this is not `None`.

**Parameters** **command** (*str*) – execute the command given as a string

**Returns** a list: the `retcode` (int), `stdout` (str) and `stderr` (str).

**get** (*remotepath*, *localpath*, \*args, \*\*kwargs)

Retrieve a file or folder from remote source to local destination `dst` must be an absolute path (`src` not necessarily)

**Parameters**

- **remotepath** – (str) `remote_folder_path`
- **localpath** – (str) `local_folder_path`

**get\_attribute** (*path*)

Return an object `FixedFieldsAttributeDict` for file in a given path, as defined in `aiida.common.extendeddicts` Each attribute object consists in a dictionary with the following keys:

- `st_size`: size of files, in bytes
- `st_uid`: user id of owner
- `st_gid`: group id of owner

- **st\_mode**: protection bits
- **st\_atime**: time of most recent access
- **st\_mtime**: time of most recent modification

**Parameters** **path** (*str*) – path to file

**Returns** object FixedFieldsAttributeDict

**get\_mode** (*path*)

Return the portion of the file's mode that can be set by chmod().

**Parameters** **path** (*str*) – path to file

**Returns** the portion of the file's mode that can be set by chmod()

**classmethod** **get\_short\_doc** ()

Return the first non-empty line of the class docstring, if available

**classmethod** **get\_valid\_auth\_params** ()

Return the internal list of valid auth\_params

**classmethod** **get\_valid\_transports** ()

**Returns** a list of existing plugin names

**getcwd** ()

Get working directory

**Returns** a string identifying the current working directory

**getfile** (*remotepath*, *localpath*, *\*args*, *\*\*kwargs*)

Retrieve a file from remote source to local destination dst must be an absolute path (src not necessarily)

**Parameters**

- **remotepath** (*str*) – remote\_folder\_path
- **localpath** (*str*) – local\_folder\_path

**gettree** (*remotepath*, *localpath*, *\*args*, *\*\*kwargs*)

Retrieve a folder recursively from remote source to local destination dst must be an absolute path (src not necessarily)

**Parameters**

- **remotepath** (*str*) – remote\_folder\_path
- **localpath** (*str*) – local\_folder\_path

**glob** (*pathname*)

Return a list of paths matching a pathname pattern.

The pattern may contain simple shell-style wildcards a la fnmatch.

**gotocomputer\_command** (*remotedir*)

Return a string to be run using os.system in order to connect via the transport to the remote directory.

Expected behaviors:

- A new bash session is opened
- A reasonable error message is produced if the folder does not exist

**Parameters** **remotedir** (*str*) – the full path of the remote directory

**iglob** (*pathname*)

Return an iterator which yields the paths matching a pathname pattern.

The pattern may contain simple shell-style wildcards a la fnmatch.

**isdir** (*path*)

True if path is an existing directory.

**Parameters** **path** (*str*) – path to directory

**Returns** boolean

**isfile** (*path*)

Return True if path is an existing file.

**Parameters** **path** (*str*) – path to file

**Returns** boolean

**listdir** (*path='.', pattern=None*)

Return a list of the names of the entries in the given path. The list is in arbitrary order. It does not include the special entries '.' and '..' even if they are present in the directory.

**Parameters**

- **path** (*str*) – path to list (default to '.')
- **pattern** (*str*) – if used, listdir returns a list of files matching filters in Unix style. Unix only.

**Returns** a list of strings

**logger**

Return the internal logger. If you have set extra parameters using `_set_logger_extra()`, a suitable Logger-Adapter instance is created, bringing with itself also the extras.

**makedirs** (*path, ignore\_existing=False*)

Super-mkdir; create a leaf directory and all intermediate ones. Works like mkdir, except that any intermediate path segment (not just the rightmost) will be created if it does not exist.

**Parameters**

- **path** (*str*) – directory to create
- **ignore\_existing** (*bool*) – if set to true, it doesn't give any error if the leaf directory does already exist

**Raises** OSError, if directory at path already exists

**mkdir** (*path, ignore\_existing=False*)

Create a folder (directory) named path.

**Parameters**

- **path** (*str*) – name of the folder to create
- **ignore\_existing** (*bool*) – if True, does not give any error if the directory already exists

**Raises** OSError, if directory at path already exists

**normalize** (*path='.'*)

Return the normalized path (on the server) of a given path. This can be used to quickly resolve symbolic links or determine what the server is considering to be the "current folder".

**Parameters** **path** (*str*) – path to be normalized

**Raises `IOError`** if the path can't be resolved on the server

**`open()`**

Opens a local transport channel

**`path_exists(path)`**

Returns True if path exists, False otherwise.

**`put(localpath, remotepath, *args, **kwargs)`**

Put a file or a directory from local src to remote dst. src must be an absolute path (dst not necessarily)  
Redirects to `putfile` and `puttree`.

**Parameters**

- **`localpath`** (*str*) – absolute path to local source
- **`remotepath`** (*str*) – path to remote destination

**`putfile(localpath, remotepath, *args, **kwargs)`**

Put a file from local src to remote dst. src must be an absolute path (dst not necessarily))

**Parameters**

- **`localpath`** (*str*) – absolute path to local file
- **`remotepath`** (*str*) – path to remote file

**`puttree(localpath, remotepath, *args, **kwargs)`**

Put a folder recursively from local src to remote dst. src must be an absolute path (dst not necessarily))

**Parameters**

- **`localpath`** (*str*) – absolute path to local folder
- **`remotepath`** (*str*) – path to remote folder

**`remove(path)`**

Remove the file at the given path. This only works on files; for removing folders (directories), use `rmdir`.

**Parameters** **`path`** (*str*) – path to file to remove

**Raises `IOError`** if the path is a directory

**`rename(oldpath, newpath)`**

Rename a file or folder from oldpath to newpath.

**Parameters**

- **`oldpath`** (*str*) – existing name of the file or folder
- **`newpath`** (*str*) – new name for the file or folder

**Raises**

- **`IOError`** – if oldpath/newpath is not found
- **`ValueError`** – if oldpath/newpath is not a valid string

**`rmdir(path)`**

Remove the folder named path. This works only for empty folders. For recursive remove, use `rmtree`.

**Parameters** **`path`** (*str*) – absolute path to the folder to remove

**`rmtree(path)`**

Remove recursively the content at path

**Parameters** **`path`** (*str*) – absolute path to remove



**symlink** (*remotesource*, *remotedestination*)

Create a symbolic link between the remote source and the remote destination.

**Parameters**

- **remotesource** – remote source
- **remotedestination** – remote destination

**whoami** ()

Get the remote username

**Returns** list of username (str), retval (int), stderr (str)

`aiida.transport.__init__.TransportFactory` (*module*)

Used to return a suitable Transport subclass.

**Parameters** **module** (*str*) – name of the module containing the Transport subclass

**Returns** the transport subclass located in module ‘module’

**exception** `aiida.transport.__init__.TransportInternalError`

Raised if there is a transport error that is raised to an internal error (e.g. a transport method called without opening the channel first).

`aiida.transport.__init__.copy_from_remote_to_remote` (*transportsource*, *transportdestination*, *remotesource*, *remotedestination*, *\*\*kwargs*)

Copy files or folders from a remote computer to another remote computer.

**Parameters**

- **transportsource** – transport to be used for the source computer
- **transportdestination** – transport to be used for the destination computer
- **remotesource** (*str*) – path to the remote source directory / file
- **remotedestination** (*str*) – path to the remote destination directory / file
- **kwargs** – keyword parameters passed to the final put, except for ‘dereference’ that is passed to the initial get

---

**Note:** it uses the method `transportsource.copy_from_remote_to_remote`

---

## Developing a plugin

The transport class is actually almost never used in first person by the user. It is mostly utilized by the ExecutionManager, that use the transport plugin to connect to the remote computer to manage the calculation. The ExecutionManager has to be able to use always the same function, or the same interface, regardless of which kind of connection is actually really using.

The generic transport class contains a set of minimal methods that an implementation must support, in order to be fully compatible with the other plugins. If not, a `NotImplementedError` will be raised, interrupting the managing of the calculation or whatever is using the transport plugin.

Since it is important that all plugins have the same interface, or the same response behavior, a set of generic tests has been written (alongside with set of tests that are implementation specific). After **every** modification, or when implementing a new plugin, it is crucial to run the tests and verify that everything is passed. The modification of tests possibly means breaking back-compatibility and/or modifications to every piece of code using a transport plugin.

If an unexpected behavior is observed during the usage, the way of fixing it is:

1. Write a new test that shows the problem (one test for one problem when possible)
2. Fix the bug
3. Verify that the test is passed correctly

The importance of point 1) is often neglected, but unittesting is a useful tool that helps you avoiding the repetition of errors. Despite the appearance, it's a time-saver! Not only, the tests help you seeing how the plugin is used.

As for the general functioning of the plugin, the `__init__` method is used only to initialize the class instance, without actually opening the transport channel. The connection must be opened only by the `__enter__` method, (and closed by `__exit__`. The `__enter__` method let you use the transport class using the `with` statement (see [Python docs](#)), in a way similar to the following:

```
t = TransportPlugin()
with open(t):
    t.do_something_remotely
```

To ensure this, for example, the local plugin uses a hidden boolean variable `_is_open` that is set when the `__enter__` and `__exit__` methods are called. The Ssh logic is instead given by the property `sftp`.

The other functions that require some care are the copying functions, called using the following terminology:

1. `put`: from local source to remote destination
2. `get`: from remote source to local destination
3. `copy`: copying files from remote source to remote destination

Note that these functions must copy files or folders regardless, internally, they will fallback to functions like `putfile` or `puttree`.

The last function requiring care is `exec_command_wait`, which is an analogue to the [subprocess](#) Python module. The function gives the freedom to execute a string as a remote command, thus it could produce nasty effects if not written with care. Be sure to escape any string for bash!

Currently, the implemented plugins are the Local and the Ssh transports. The Local one is simply a wrapper to some standard Python modules, like `shutil` or `os`, those functions are simply interfaced in a different way with AiiDA. The Ssh instead is an interface to the [Paramiko](#) library.

Below, you can find a template to fill for a new transport plugin, with a minimal docstring that also work for the sphinx documentation.

```
class NewTransport(aiida.transport.Transport):

    def __init__(self, machine, **kwargs):
        """
        Initialize the Transport class.

        :param machine: the machine to connect to
        """

    def __enter__(self):
        """
        Open the connection
        """

    def __exit__(self, type, value, traceback):
        """
        Close the connection
        """
```

```

def chdir(self, path):
    """
    Change directory to 'path'

    :param str path: path to change working directory into.
    :raises: IOError, if the requested path does not exist
    :rtype: string
    """

def chmod(self, path, mode):
    """
    Change permissions of a path.

    :param str path: path to file
    :param int mode: new permissions
    """

def copy(self, remotesource, remotedestination, *args, **kwargs):
    """
    Copy a file or a directory from remote source to remote destination
    (On the same remote machine)

    :param str remotesource: path of the remote source directory / file
    :param str remotedestination: path of the remote destination directory / file

    :raises: IOError, if source or destination does not exist
    """
    raise NotImplementedError

def copyfile(self, remotesource, remotedestination, *args, **kwargs):
    """
    Copy a file from remote source to remote destination
    (On the same remote machine)

    :param str remotesource: path of the remote source directory / file
    :param str remotedestination: path of the remote destination directory / file

    :raises IOError: if one of src or dst does not exist
    """

def copytree(self, remotesource, remotedestination, *args, **kwargs):
    """
    Copy a folder from remote source to remote destination
    (On the same remote machine)

    :param str remotesource: path of the remote source directory / file
    :param str remotedestination: path of the remote destination directory / file

    :raise IOError: if one of src or dst does not exist
    """

def exec_command_wait(self, command, **kwargs):
    """
    Execute the command on the shell, waits for it to finish,
    and return the retcode, the stdout and the stderr.

    Enforce the execution to be run from the pwd (as given by
    self.getcwd), if this is not None.

```

```
:param str command: execute the command given as a string
:return: a tuple: the retcode (int), stdout (str) and stderr (str).
"""

def get_attribute(self, path):
    """
    Return an object FixedFieldsAttributeDict for file in a given path,
    as defined in aiida.common.extendeddicts
    Each attribute object consists in a dictionary with the following keys:

    * st_size: size of files, in bytes

    * st_uid: user id of owner

    * st_gid: group id of owner

    * st_mode: protection bits

    * st_atime: time of most recent access

    * st_mtime: time of most recent modification

    :param str path: path to file
    :return: object FixedFieldsAttributeDict
    """

def getcwd(self):
    """
    Get working directory

    :return: a string identifying the current working directory
    """

def get(self, remotepath, localpath, *args, **kwargs):
    """
    Retrieve a file or folder from remote source to local destination
    dst must be an absolute path (src not necessarily)

    :param remotepath: (str) remote_folder_path
    :param localpath: (str) local_folder_path
    """

def getfile(self, remotepath, localpath, *args, **kwargs):
    """
    Retrieve a file from remote source to local destination
    dst must be an absolute path (src not necessarily)

    :param str remotepath: remote_folder_path
    :param str localpath: local_folder_path
    """

def gettree(self, remotepath, localpath, *args, **kwargs):
    """
    Retrieve a folder recursively from remote source to local destination
    dst must be an absolute path (src not necessarily)

    :param str remotepath: remote_folder_path
    :param str localpath: local_folder_path
```

```

"""

def gotocomputer_command(self, remotedir):
    """
    Return a string to be run using os.system in order to connect
    via the transport to the remote directory.

    Expected behaviors:

    * A new bash session is opened

    * A reasonable error message is produced if the folder does not exist

    :param str remotedir: the full path of the remote directory
    """

def isdir(self, path):
    """
    True if path is an existing directory.

    :param str path: path to directory
    :return: boolean
    """

def isfile(self, path):
    """
    Return True if path is an existing file.

    :param str path: path to file
    :return: boolean
    """

def listdir(self, path='.', pattern=None):
    """
    Return a list of the names of the entries in the given path.
    The list is in arbitrary order. It does not include the special
    entries '.' and '..' even if they are present in the directory.

    :param str path: path to list (default to '.')
    :param str pattern: if used, listdir returns a list of files matching
                        filters in Unix style. Unix only.
    :return: a list of strings
    """

def makedirs(self, path, ignore_existing=False):
    """
    Super-mkdir; create a leaf directory and all intermediate ones.
    Works like mkdir, except that any intermediate path segment (not
    just the rightmost) will be created if it does not exist.

    :param str path: directory to create
    :param bool ignore_existing: if set to true, it doesn't give any error
                                if the leaf directory does already exist

    :raises: OSError, if directory at path already exists
    """

def mkdir(self, path, ignore_existing=False):

```

```
"""
Create a folder (directory) named path.

:param str path: name of the folder to create
:param bool ignore_existing: if True, does not give any error if the
                           directory already exists

:raises: OSError, if directory at path already exists
"""

def normalize(self, path='.'):
    """
    Return the normalized path (on the server) of a given path.
    This can be used to quickly resolve symbolic links or determine
    what the server is considering to be the "current folder".

    :param str path: path to be normalized

    :raise IOError: if the path can't be resolved on the server
    """

def put(self, localpath, remotepath, *args, **kwargs):
    """
    Put a file or a directory from local src to remote dst.
    src must be an absolute path (dst not necessarily)
    Redirects to putfile and puttree.

    :param str localpath: path to remote destination
    :param str remotepath: absolute path to local source
    """

def putfile(self, localpath, remotepath, *args, **kwargs):
    """
    Put a file from local src to remote dst.
    src must be an absolute path (dst not necessarily))

    :param str localpath: path to remote file
    :param str remotepath: absolute path to local file
    """

def puttree(self, localpath, remotepath, *args, **kwargs):
    """
    Put a folder recursively from local src to remote dst.
    src must be an absolute path (dst not necessarily))

    :param str localpath: path to remote folder
    :param str remotepath: absolute path to local folder
    """

def rename(src, dst):
    """
    Rename a file or folder from src to dst.

    :param str oldpath: existing name of the file or folder
    :param str newpath: new name for the file or folder

    :raises IOError: if src/dst is not found
    :raises ValueError: if src/dst is not a valid string
    """
```

```

"""

def remove(self, path):
    """
    Remove the file at the given path. This only works on files;
    for removing folders (directories), use rmdir.

    :param str path: path to file to remove

    :raise IOError: if the path is a directory
    """

def rmdir(self, path):
    """
    Remove the folder named path.
    This works only for empty folders. For recursive remove, use rmtree.

    :param str path: absolute path to the folder to remove
    """
    raise NotImplementedError

def rmtree(self, path):
    """
    Remove recursively the content at path

    :param str path: absolute path to remove
    """

```

### 4.1.3 aiiDA.scheduler documentation

We report here the generic AiiDA scheduler implementation.

#### Generic scheduler class

**class** `aiida.scheduler.__init__.Scheduler`

Base class for all schedulers.

**classmethod** `create_job_resource` (*\*\*kwargs*)

Create a suitable job resource from the kwargs specified

**getJobs** (*jobs=None, user=None, as\_dict=False*)

Get the list of jobs and return it.

Typically, this function does not need to be modified by the plugins.

#### Parameters

- **jobs** (*list*) – a list of jobs to check; only these are checked
- **user** (*str*) – a string with a user: only jobs of this user are checked
- **as\_dict** (*list*) – if False (default), a list of JobInfo objects is returned. If True, a dictionary is returned, having as key the job\_id and as value the JobInfo object.

Note: typically, only either jobs or user can be specified. See also comments in `_get_joblist_command`.

**get\_detailed\_jobinfo** (*jobid*)

Return a string with the output of the `detailed_jobinfo` command.

At the moment, the output text is just retrieved and stored for logging purposes, but no parsing is performed.

**classmethod** `get_short_doc()`

Return the first non-empty line of the class docstring, if available

**get\_submit\_script** (*job\_tmpl*)

Return the submit script as a string. :parameter job\_tmpl: a aiida.scheduler.datastructures.JobTemplate object.

The plugin returns something like

```
#!/bin/bash <- this shebang line could be configurable in the future scheduler_dependent stuff to choose
numnodes, numcores, walltime, ... prepend_computer [also from calcinfo, joined with the following?]
prepend_code [from calcinfo] output of _get_script_main_content postpend_code postpend_computer
```

**kill** (*jobid*)

Kill a remote job, and try to parse the output message of the scheduler to check if the scheduler accepted the command.

..note:: On some schedulers, even if the command is accepted, it may take some seconds for the job to actually disappear from the queue.

**Parameters** *jobid* (*str*) – the job id to be killed

**Returns** True if everything seems ok, False otherwise.

**logger**

Return the internal logger.

**set\_transport** (*transport*)

Set the transport to be used to query the machine or to submit scripts. This class assumes that the transport is open and active.

**submit\_from\_script** (*working\_directory*, *submit\_script*)

Goes in the working directory and submits the submit\_script.

Return a string with the JobID in a valid format to be used for querying.

Typically, this function does not need to be modified by the plugins.

**transport**

Return the transport set for this scheduler.

`aiida.scheduler.__init__.SchedulerFactory` (*module*)

Used to load a suitable Scheduler subclass.

**Parameters** *module* (*str*) – a string with the module name

**Returns** the scheduler subclass contained in module ‘module’

## Scheduler datastructures

This module defines the main data structures used by the Scheduler.

In particular, there is the definition of possible job states (`job_states`), the data structure to be filled for job submission (`JobTemplate`), and the data structure that is returned when querying for jobs in the scheduler (`JobInfo`).

**class** `aiida.scheduler.datastructures.JobInfo` (*init={}*)

Contains properties for a job in the queue. Most of the fields are taken from DRMAA v.2.



Note that default fields may be undefined. This is an expected behavior and the application must cope with this case. An example for instance is the `exit_status` for jobs that have not finished yet; or features not supported by the given scheduler.

Fields:

- `job_id`: the job ID on the scheduler
- `title`: the job title, as known by the scheduler
- `exit_status`: the exit status of the job as reported by the operating system on the execution host
- `terminating_signal`: the UNIX signal that was responsible for the end of the job.
- `annotation`: human-readable description of the reason for the job being in the current state or substate.
- `job_state`: the job state (one of those defined in `aiida.scheduler.datastructures.job_states`)
- `job_substate`: a string with the implementation-specific sub-state
- `allocated_machines`: a list of machines used for the current job. This is a list of *MachineInfo* objects.
- `job_owner`: the job owner as reported by the scheduler
- `num_mpi_procs`: the *total* number of requested MPI procs
- `num_cpus`: the *total* number of requested CPUs (cores) [may be undefined]
- `num_machines`: the number of machines (i.e., nodes), required by the job. If `allocated_machines` is not `None`, this number must be equal to `len(allocated_machines)`. Otherwise, for schedulers not supporting the retrieval of the full list of allocated machines, this attribute can be used to know at least the number of machines.
- `queue_name`: The name of the queue in which the job is queued or running.
- `wallclock_time_seconds`: the accumulated wallclock time, in seconds
- `requested_wallclock_time_seconds`: the requested wallclock time, in seconds
- `cpu_time`: the accumulated cpu time, in seconds
- `submission_time`: the absolute time at which the job was submitted, of type `datetime.datetime`
- `dispatch_time`: the absolute time at which the job first entered the ‘started’ state, of type `datetime.datetime`
- `finish_time`: the absolute time at which the job first entered the ‘finished’ state, of type `datetime.datetime`

**class** `aiida.scheduler.datastructures.JobResource` (*init={}*)

A class to store the job resources. It must be inherited and redefined by the specific plugin, that should contain a `_job_resource_class` attribute pointing to the correct `JobResource` subclass.

It should at least define the `get_tot_num_mpi_procs()` method, plus an `__init__` to accept its set of variables.

Typical attributes are:

- `num_machines`
- `num_mpi_procs_per_machine`

or (e.g. for SGE)

- `tot_num_mpi_procs`
- `parallel_env`

The `__init__` should take care of checking the values. The init should raise only `ValueError` or `TypeError` on invalid parameters.

**classmethod `accepts_default_mpiproc_per_machine()`**

Return True if this `JobResource` accepts a 'default\_mpiproc\_per\_machine' key, False otherwise.

Should be implemented in each subclass.

**method `get_tot_num_mpiproc()`**

Return the total number of cpus of this job resource.

**classmethod `get_valid_keys()`**

Return a list of valid keys to be passed to the `__init__`

**class `aiida.scheduler.datastructures.JobTemplate` (`init={}`)**

A template for submitting jobs. This contains all required information to create the job header.

**The required fields are: `working_directory`, `job_name`, `num_machines`, `num_mpiproc_per_machine`, `argv`.**

Fields:

- `submit_as_hold`: if set, the job will be in a 'hold' status right after the submission
- `rerunnable`: if the job is rerunnable (boolean)
- `job_environment`: a dictionary with environment variables to set before the execution of the code.
- `working_directory`: the working directory for this job. During submission, the transport will first do a 'chdir' to this directory, and then possibly set a scheduler parameter, if this is supported by the scheduler.
- `email`: an email address for sending emails on job events.
- `email_on_started`: if True, ask the scheduler to send an email when the job starts.
- `email_on_terminated`: if True, ask the scheduler to send an email when the job ends. This should also send emails on job failure, when possible.
- `job_name`: the name of this job. The actual name of the job can be different from the one specified here, e.g. if there are unsupported characters, or the name is too long.
- `sched_output_path`: a (relative) file name for the stdout of this job
- `sched_error_path`: a (relative) file name for the stderr of this job
- `sched_join_files`: if True, write both stdout and stderr on the same file (the one specified for stdout)
- `queue_name`: the name of the scheduler queue (sometimes also called partition), on which the job will be submitted.
- `job_resource`: a suitable `JobResource` subclass with information on how many nodes and cpus it should use. It must be an instance of the `aiida.scheduler.Scheduler._job_resource_class` class. Use the `Scheduler.create_job_resource` method to create it.
- `num_machines`: how many machines (or nodes) should be used
- `num_mpiproc_per_machine`: how many MPI procs should be used on each machine (or node).
- `priority`: a priority for this job. Should be in the format accepted by the specific scheduler.
- `max_memory_kb`: The maximum amount of memory the job is allowed to allocate ON EACH NODE, in kilobytes
- `max_wallclock_seconds`: The maximum wall clock time that all processes of a job are allowed to exist, in seconds

- custom\_scheduler\_commands**: a string that will be inserted right after the last scheduler command, and before any other non-scheduler command; useful if some specific flag needs to be added and is not supported by the plugin
- prepend\_text**: a (possibly multi-line) string to be inserted in the scheduler script before the main execution line
- append\_text**: a (possibly multi-line) string to be inserted in the scheduler script after the main execution line
- import\_sys\_environment**: import the system environment variables
- codes\_info**: a list of `aiida.common.datastructures.CalcInfo` objects. Each contains the information necessary to run a single code. At the moment, it can contain:
  - cmdline\_parameters**: a list of strings with the command line arguments of the program to run. This is the main program to be executed. NOTE: The first one is the executable name. For MPI runs, this will probably be “`mpirun`” or a similar program; this has to be chosen at a upper level.
  - stdin\_name**: the (relative) file name to be used as stdin for the program specified with `argv`.
  - stdout\_name**: the (relative) file name to be used as stdout for the program specified with `argv`.
  - stderr\_name**: the (relative) file name to be used as stderr for the program specified with `argv`.
  - join\_files**: if `True`, stderr is redirected on the same file specified for stdout.
- codes\_run\_mode**: sets the `run_mode` with which the (multiple) codes have to be executed. For example, parallel execution:

```
mpirun -np 8 a.x &
mpirun -np 8 b.x &
wait
```

The serial execution would be without the `&`'s. Values are given by `aiida.common.datastructures.code_run_modes`.

**class** `aiida.scheduler.datastructures.MachineInfo` (*init={}*)

Similarly to what is defined in the DRMAA v.2 as `SlotInfo`; this identifies each machine (also called ‘node’ on some schedulers) on which a job is running, and how many CPUs are being used. (Some of them could be undefined)

- name**: name of the machine
- num\_cpus**: number of cores used by the job on this machine
- num\_mpiprocs**: number of MPI processes used by the job on this machine

**class** `aiida.scheduler.datastructures.NodeNumberJobResource` (*\*\*kwargs*)

An implementation of `JobResource` for schedulers that support the specification of a number of nodes and a number of cpus per node

**classmethod** `accepts_default_mpiprocs_per_machine()`

Return `True` if this `JobResource` accepts a ‘`default_mpiprocs_per_machine`’ key, `False` otherwise.

**get\_tot\_num\_mpiprocs()**

Return the total number of cpus of this job resource.

**classmethod** `get_valid_keys()`

Return a list of valid keys to be passed to the `__init__`

**class** `aiida.scheduler.datastructures.ParEnvJobResource` (*\*\*kwargs*)

An implementation of `JobResource` for schedulers that support the specification of a parallel environment (a string) + the total number of nodes

```
classmethod accepts_default_mpiprocs_per_machine ()
    Return True if this JobResource accepts a 'default_mpiprocs_per_machine' key, False otherwise.

get_tot_num_mpiprocs ()
    Return the total number of cpus of this job resource.
```

## 4.1.4 aiida.cmdline documentation

### Baseclass

**class** aiida.cmdline.baseclass.**VerdiCommand**

This command has no documentation yet.

**complete** (*subargs\_idx*, *subargs*)  
 Method called when the user asks for the bash completion. Print a list of valid keywords. Returning without printing will use standard bash completion.

#### Parameters

- **subargs\_idx** – the index of the subargs where the TAB key was pressed (0 is the first element of subargs)
- **subargs** – a list of subarguments to this command

**classmethod** **get\_command\_name** ()  
 Return the name of the verdi command associated to this class. By default, the lower-case version of the class name.

**get\_full\_command\_name** (*with\_exec\_name=True*)  
 Return the current command name. Also tries to get the subcommand name.

**Parameters with\_exec\_name** – if True, return the full string, including the executable name ('verdi'). If False, omit it.

**run** (*\*args*)  
 Method executed when the command is called from the command line.

**class** aiida.cmdline.baseclass.**VerdiCommandWithSubcommands**

Used for commands with subcommands. Just define, in the `__init__`, the `self.valid_subcommands` dictionary, in the format:

```
self.valid_subcommands = {
    'uploadfamily': (self.uploadfamily, self.complete_auto),
    'listfamilies': (self.listfamilies, self.complete_none),
}
```

where the key is the subcommand name to give on the command line, and the value is a tuple of length 2, the first is the function to call on execution, the second is the function to call on complete.

This class already defined the `complete_auto` and `complete_none` commands, that respectively call the default bash completion for filenames/folders, or do not give any completion suggestion. Other functions can of course be defined.

---

### Todo

Improve the docstrings for commands with subcommands.

---

**get\_full\_command\_name** (*\*args*, *\*\*kwargs*)  
 Return the current command name. Also tries to get the subcommand name.

Also tries to see if the caller function was one specific submethod.

**Parameters with\_exec\_name** – if True, return the full string, including the executable name ('verdi'). If False, omit it.

## Verdi lib

Command line commands for the main executable 'verdi' of aiida

If you want to define a new command line parameter, just define a new class inheriting from `VerdiCommand`, and define a `run(self,*args)` method accepting a variable-length number of parameters `args` (the command-line parameters), which will be invoked when this executable is called as `verdi NAME`

Don't forget to add the docstring to the class: the first line will be the short description, the following ones the long description.

**class** `aiida.cmdline.verdilib.Completion`

Manage bash completion

Return a list of available commands, separated by spaces. Calls the correct function of the command if the TAB has been pressed after the first command.

Returning without printing will use the default bash completion.

**class** `aiida.cmdline.verdilib.CompletionCommand`

Return the bash completion function to put in ~/.bashrc

This command prints on screen the function to be inserted in your .bashrc command. You can copy and paste the output, or simply add `eval "verdi completioncommand"` to your .bashrc, *AFTER* having added the aiida/bin directory to the path.

**run** (*\*args*)

I put the documentation here, and I don't print it, so we don't clutter too much the .bashrc.

- “\${THE\_WORDS[@]}" (with the @) puts each element as a different parameter; note that the variable expansion etc. is performed
- I add a 'x' at the end and then remove it; in this way, \$( ) will not remove trailing spaces
- If the completion command did not print anything, we use the default bash completion for filenames
- If instead the code prints something empty, thanks to the workaround above \$OUTPUT is not empty, so we do go the the 'else' case and then, no substitution is suggested.

**class** `aiida.cmdline.verdilib.Help`

Describe a specific command

Pass a further argument to get a description of a given command.

**class** `aiida.cmdline.verdilib.Install`

Install/setup aiida for the current user

This command creates the ~/.aiida folder in the home directory of the user, interactively asks for the database settings and the repository location, does a setup of the daemon and runs a migrate command to create/setup the database.

**complete** (*subargs\_idx*, *subargs*)

No completion after 'verdi install'.

**class** `aiida.cmdline.verdilib.ListParams`

List available commands

List available commands and their short description. For the long description, use the 'help' command.

**exception** `aiida.cmdline.verdilib.ProfileParsingException(*args, **kwargs)`

Exception raised when parsing the profile command line option, if only -p is provided, and no profile is specified

**class** `aiida.cmdline.verdilib.Run`

Execute an AiiDA script

**class** `aiida.cmdline.verdilib.Runserver`

Run the AiiDA webserver on localhost

This command runs the webserver on the default port. Further command line options are passed to the Django manage runserver command

**class** `aiida.cmdline.verdilib.Shell`

Run the interactive shell with the AiiDA environment loaded.

This command opens an ipython shell with the AiiDA environment loaded.

`aiida.cmdline.verdilib.exec_from_cmdline(argv)`

The main function to be called. Pass as parameter the sys.argv.

`aiida.cmdline.verdilib.get_command_suggestion(command)`

A function that prints on stderr a list of similar commands

`aiida.cmdline.verdilib.get_listparams()`

Return a string with the list of parameters, to be printed

The advantage of this function is that the calling routine can choose to print it on stdout or stderr, depending on the needs.

`aiida.cmdline.verdilib.parse_profile(argv, merge_equal=False)`

Parse the argv to see if a profile has been specified, return it with the command position shift (index where the commands start)

**Parameters** `merge_equal` – if True, merge things like ('verdi', '-profile', '=', 'x', 'y') to ('verdi', '-profile=x', 'y') but then return the correct index for the original array.

**Raises** `ProfileParsingException` if there is only 'verdi' specified, or if only 'verdi -p' (in these cases, one has respectively exception.minus\_p\_provided equal to False or True)

`aiida.cmdline.verdilib.update_environment(*args, **kws)`

Used as a context manager, changes sys.argv with the new\_argv argument, and restores it upon exit.

## Daemon

**class** `aiida.cmdline.commands.daemon.Daemon`

Manage the AiiDA daemon

This command allows to interact with the AiiDA daemon. Valid subcommands are:

- start: start the daemon
- stop: restart the daemon
- restart: restart the aiida daemon, waiting for it to cleanly exit before restarting it.
- status: inquire the status of the Daemon.
- logshow: show the log in a continuous fashion, similar to the 'tail -f' command. Press CTRL+C to exit.

`__init__()`

A dictionary with valid commands and functions to be called: start, stop, status and restart.

`configure_user(*args)`

Configure the user that can run the daemon.

**daemon\_logshow** (\*args)

Show the log of the daemon, press CTRL+C to quit.

**daemon\_restart** (\*args)

Restart the daemon. Before restarting, wait for the daemon to really shut down.

**daemon\_start** (\*args)

Start the daemon

**daemon\_status** (\*args)

Print the status of the daemon

**daemon\_stop** (\*args, \*\*kwargs)

Stop the daemon.

**Parameters** **wait\_for\_death** – If True, also verifies that the process was already killed. It attempts at most `max_retries` times, with `sleep_between_retries` seconds between one attempt and the following one (both variables are for the time being hardcoded in the function).

**Returns** None if `wait_for_death` is False. True/False if the process was actually dead or after all the retries it was still alive.

**get\_daemon\_pid** ()

Return the daemon pid, as read from the `supervisord.pid` file. Return None if no pid is found (or the pid is not valid).

**kill\_daemon** ()

This is the actual call that kills the daemon.

There are some print statements inside, but no `sys.exit`, so it is safe to be called from other parts of the code.

`aiida.cmdline.commands.daemon.is_daemon_user` ()

Return True if the user is the current daemon user, False otherwise.

## Data

**class** `aiida.cmdline.commands.data.Data`

Setup and manage data specific types

There is a list of subcommands for managing specific types of data. For instance, ‘data upf’ manages pseudopotentials in the UPF format.

**\_\_init\_\_** ()

A dictionary with valid commands and functions to be called.

**class** `aiida.cmdline.commands.data.Depositable`

Provides shell completion for depositable data nodes.

---

**Note:** classes, inheriting `Depositable`, MUST NOT contain attributes, starting with `_deposit_`, which are not plugins for depositing.

---

**deposit** (\*args)

Deposit the data node to a given database.

**Parameters** **args** – a namespace with parsed command line parameters.

**get\_deposit\_plugins** ()

Get the list of all implemented deposition methods for data class.

**class** `aiida.cmdline.commands.data.Exportable`

Provides shell completion for exportable data nodes.

---

**Note:** classes, inheriting `Exportable`, MUST NOT contain attributes, starting with `_export_`, which are not plugins for exporting.

---

**export** (\*args)

Export the data node to a given format.

**get\_export\_plugins** ()

Get the list of all implemented exporters for data class.

**class** `aiida.cmdline.commands.data.Importable`

Provides shell completion for importable data nodes.

---

**Note:** classes, inheriting `Importable`, MUST NOT contain attributes, starting with `_import_`, which are not plugins for importing.

---

**get\_import\_plugins** ()

Get the list of all implemented importers for data class.

**class** `aiida.cmdline.commands.data.Listable`

Provides shell completion for listable data nodes.

---

**Note:** classes, inheriting `Listable`, MUST define value for property `dataclass` (preferably in `__init__`), which has to point to correct `*Data` class.

---

**append\_list\_cmdline\_arguments** (parser)

Append additional command line parameters, that are later parsed and used in the query construction.

**Parameters** `parser` – instance of `argparse.ArgumentParser`

**get\_column\_names** ()

Return the list with column names.

---

**Note:** neither the number nor correspondence of column names and actual columns in the output from the `query` are checked.

---

**list** (\*args)

List all instances of given data class.

**Parameters** `args` – a list of command line arguments.

**query** (args)

Perform the query and return information for the list.

**Parameters** `args` – a namespace with parsed command line parameters.

**Returns** table (list of lists) with information, describing nodes. Each row describes a single hit.

**query\_group** (q\_object, args)

Subselect to filter data nodes by their group.

**Parameters**

- `q_object` – a query object
- `args` – a namespace with parsed command line parameters.



**query\_past\_days** (*q\_object, args*)  
 Subselect to filter data nodes by their age.

**Parameters**

- **q\_object** – a query object
- **args** – a namespace with parsed command line parameters.

**class** `aiida.cmdline.commands.data.Visualizable`  
 Provides shell completion for visualizable data nodes.

---

**Note:** classes, inheriting `Visualizable`, MUST NOT contain attributes, starting with `_show_`, which are not plugins for visualization.

---

In order to specify a default visualization format, one has to override `_default_show_format` property (preferably in `__init__`), setting it to the name of default visualization tool.

**get\_show\_plugins** ()  
 Get the list of all implemented plugins for visualizing the structure.

**show** (\*args)  
 Show the data node with a visualization program.

## 4.1.5 aiida.execmanager documentation

### Execution Manager

This file contains the main routines to submit, check and retrieve calculation results. These are general and contain only the main logic; where appropriate, the routines make reference to the suitable plugins for all plugin-specific operations.

`aiida.execmanager.get_authinfo` (*computer, aiidauser*)  
`aiida.execmanager.retrieve_computed_for_authinfo` (*authinfo*)  
`aiida.execmanager.retrieve_jobs` ()  
`aiida.execmanager.submit_calc` (*calc, authinfo, transport=None*)  
 Submit a calculation

**Note** if no transport is passed, a new transport is opened and then closed within this function. If you want to use an already opened transport, pass it as further parameter. In this case, the transport has to be already open, and must coincide with the transport of the the computer defined by the `authinfo`.

**Parameters**

- **calc** – the calculation to submit (an instance of the `aiida.orm.JobCalculation` class)
- **authinfo** – the `authinfo` for this calculation.
- **transport** – if passed, must be an already opened transport. No checks are done on the consistency of the given transport with the transport of the computer defined in the `authinfo`.

`aiida.execmanager.submit_jobs` ()  
 Submit all jobs in the TOSUBMIT state.  
`aiida.execmanager.submit_jobs_with_authinfo` (*authinfo*)  
 Submit jobs in TOSUBMIT status belonging to user and machine as defined in the 'dbauthinfo' table.

`aiida.execmanager.update_jobs()`

calls an update for each set of pairs (machine, aiidauser)

`aiida.execmanager.update_running_calcs_status(authinfo)`

Update the states of calculations in WITHSCHEDULER status belonging to user and machine as defined in the 'dbauthinfo' table.

## 4.1.6 aiida.djsite documentation

### Database schema

**class** `aiida.djsite.db.models.DbAttribute(*args, **kwargs)`

This table stores attributes that uniquely define the content of the node. Therefore, their modification corrupts the data.

**class** `aiida.djsite.db.models.DbAttributeBaseClass(*args, **kwargs)`

Abstract base class for tables storing element-attribute-value data. Element is the dbnode; attribute is the key name. Value is the specific value to store.

This table had different SQL columns to store different types of data, and a datatype field to know the actual datatype.

Moreover, this class unpacks dictionaries and lists when possible, so that it is possible to query inside recursive lists and dicts.

**classmethod** `del_value_for_node(dbnode, key)`

Delete an attribute from the database for the given dbnode.

**Note** no exception is raised if no attribute with the given key is found in the DB.

#### Parameters

- **dbnode** – the dbnode for which you want to delete the key.
- **key** – the key to delete.

**classmethod** `get_all_values_for_node(dbnode)`

Return a dictionary with all attributes for the given dbnode.

**Returns** a dictionary where each key is a level-0 attribute stored in the Db table, correctly converted to the right type.

**classmethod** `get_value_for_node(dbnode, key)`

Get an attribute from the database for the given dbnode.

**Returns** the value stored in the Db table, correctly converted to the right type.

**Raises AttributeError** if no key is found for the given dbnode

**classmethod** `has_key(dbnode, key)`

Return True if the given dbnode has an attribute with the given key, False otherwise.

**classmethod** `list_all_node_elements(dbnode)`

Return a django queryset with the attributes of the given node, only at deepness level zero (i.e., keys not containing the separator).

**classmethod** `set_value_for_node(dbnode, key, value, with_transaction=True, stop_if_existing=False)`

This is the raw-level method that accesses the DB. No checks are done to prevent the user from (re)setting a valid key. To be used only internally.

**Todo** there may be some error on concurrent write; not checked in this unlucky case!

### Parameters

- **dbnode** – the dbnode for which the attribute should be stored; in an integer is passed, this is used as the PK of the dbnode, without any further check (for speed reasons)
- **key** – the key of the attribute to store; must be a level-zero attribute (i.e., no separators in the key)
- **value** – the value of the attribute to store
- **with\_transaction** – if True (default), do this within a transaction, so that nothing gets stored if a subitem cannot be created. Otherwise, if this parameter is False, no transaction management is performed.
- **stop\_if\_existing** – if True, it will stop with an UniquenessError exception if the key already exists for the given node. Otherwise, it will first delete the old value, if existent. The use with True is useful if you want to use a given attribute as a “locking” value, e.g. to avoid to perform an action twice on the same node. Note that, if you are using transactions, you may get the error only when the transaction is committed.

**Raises ValueError** if the key contains the separator symbol used internally to unpack dictionaries and lists (defined in `cls._sep`).

**class** `aiida.djsite.db.models.DbAuthInfo` (*\*args, \*\*kwargs*)

Table that pairs aiida users and computers, with all required authentication information.

**get\_transport** ()

Given a computer and an aiida user (as entries of the DB) return a configured transport to connect to the computer.

**class** `aiida.djsite.db.models.DbCalcState` (*\*args, \*\*kwargs*)

Store the state of calculations.

The advantage of a table (with uniqueness constraints) is that this disallows entering twice in the same state (e.g., retrieving twice).

**class** `aiida.djsite.db.models.DbComment` (*id, uuid, dbnode\_id, ctime, mtime, user\_id, content*)

**class** `aiida.djsite.db.models.DbComputer` (*\*args, \*\*kwargs*)

Table of computers or clusters.

Attributes: \* **name**: A name to be used to refer to this computer. Must be unique. \* **hostname**: Fully-qualified hostname of the host \* **transport\_type**: a string with a valid transport type

Note: other things that may be set in the metadata:

- **mpirun command**
- **num cores per node**
- **max num cores**
- **workdir**: Full path of the aiida folder on the host. It can contain the string {username} that will be substituted by the username of the user on that machine. The actual workdir is then obtained as `workdir.format(username=THE_ACTUAL_USERNAME)` Example: `workdir = "/scratch/{username}/aiida/"`
- **allocate full node** = True or False
- ... (further limits per user etc.)

**classmethod** **get\_dbcomputer** (*computer*)

Return a DbComputer from its name (or from another Computer or DbComputer instance)

**class** `aiida.djsite.db.models.DbExtra (*args, **kwargs)`

This table stores extra data, still in the key-value format, that the user can attach to a node. Therefore, their modification simply changes the user-defined data, but does not corrupt the node (it will still be loadable without errors). Could be useful to add “duplicate” information for easier querying, or for tagging nodes.

**class** `aiida.djsite.db.models.DbGroup (*args, **kwargs)`

A group of nodes.

Any group of nodes can be created, but some groups may have specific meaning if they satisfy specific rules (for instance, groups of UpdData objects are pseudopotential families - if no two pseudos are included for the same atomic element).

**class** `aiida.djsite.db.models.DbLink (*args, **kwargs)`

Direct connection between two dbnodes. The label is identifying the link type.

**class** `aiida.djsite.db.models.DbLock (key, creation, timeout, owner)`

**class** `aiida.djsite.db.models.DbLog (id, time, loggename, levelname, objname, objpk, message, metadata)`

**classmethod** `add_from_logrecord (record)`

Add a new entry from a LogRecord (from the standard python logging facility). No exceptions are managed here.

**class** `aiida.djsite.db.models.DbMultipleValueAttributeBaseClass (*args, **kwargs)`

Abstract base class for tables storing attribute + value data, of different data types (without any association to a Node).

**classmethod** `create_value (key, value, subspecifier_value=None, other_attribs={})`

Create a new list of attributes, without storing them, associated with the current key/value pair (and to the given subspecifier, e.g. the DbNode for DbAttributes and DbExtras).

**Note** No hits are done on the DB, in particular no check is done on the existence of the given nodes.

#### Parameters

- **key** – a string with the key to create (can contain the separator `cls._sep` if this is a sub-attribute: indeed, this function calls itself recursively)
- **value** – the value to store (a basic data type or a list or a dict)
- **subspecifier\_value** – must be None if this class has no subspecifier set (e.g., the DbSetting class). Must be the value of the subspecifier (e.g., the dbnode) for classes that define it (e.g. DbAttribute and DbExtra)
- **other\_attribs** – a dictionary of other parameters, to store only on the level-zero attribute (e.g. for description in DbSetting).

**Returns** always a list of class instances; it is the user responsibility to store such entries (typically with a Django `bulk_create()` call).

**classmethod** `del_value (key, only_children=False, subspecifier_value=None)`

Delete a value associated with the given key (if existing).

**Note** No exceptions are raised if no entry is found.

#### Parameters

- **key** – the key to delete. Can contain the separator `cls._sep` if you want to delete a subkey.
- **only\_children** – if True, delete only children and not the entry itself.

- **subspecifier\_value** – must be None if this class has no subspecifier set (e.g., the DbSetting class). Must be the value of the subspecifier (e.g., the dbnode) for classes that define it (e.g. DbAttribute and DbExtra)

**classmethod** `get_query_dict (value)`

Return a dictionary that can be used in a django filter to query for a specific value. This takes care of checking the type of the input parameter ‘value’ and to convert it to the right query.

**Parameters** **value** – The value that should be queried. Note: can only be base datatype, not a list or dict. For those, query directly for one of the sub-elements.

**Todo** see if we want to give the possibility to query for the existence of a (possibly empty) dictionary or list, or for their length.

**Note** this will of course not find a data if this was stored in the DB as a serialized JSON.

**Returns** a dictionary to be used in the django .filter() method. For instance, if ‘value’ is a string, it will return the dictionary {'datatype': 'txt', 'tval': value}.

**Raise** ValueError if value is not of a base datatype (string, integer, float, bool, None, or date)

**getvalue ()**

This can be called on a given row and will get the corresponding value, casting it correctly.

**long\_field\_length ()**

Return the length of “long” fields. This is used, for instance, for the ‘key’ field of attributes. This returns 1024 typically, but it returns 255 if the backend is mysql.

**Note** Call this function only AFTER having called load\_dbenv!

**classmethod** `set_value (key, value, with_transaction=True, subspecifier_value=None, other_attribs={}, stop_if_existing=False)`

Set a new value in the DB, possibly associated to the given subspecifier.

**Note** This method also stored directly in the DB.

**Parameters**

- **key** – a string with the key to create (must be a level-0 attribute, that is it cannot contain the separator cls.\_sep).
- **value** – the value to store (a basic data type or a list or a dict)
- **subspecifier\_value** – must be None if this class has no subspecifier set (e.g., the DbSetting class). Must be the value of the subspecifier (e.g., the dbnode) for classes that define it (e.g. DbAttribute and DbExtra)
- **with\_transaction** – True if you want this function to be managed with transactions. Set to False if you already have a manual management of transactions in the block where you are calling this function (useful for speed improvements to avoid recursive transactions)
- **other\_attribs** – a dictionary of other parameters, to store only on the level-zero attribute (e.g. for description in DbSetting).
- **stop\_if\_existing** – if True, it will stop with an UniquenessError exception if the new entry would violate an uniqueness constraint in the DB (same key, or same key+node, depending on the specific subclass). Otherwise, it will first delete the old value, if existent. The use with True is useful if you want to use a given attribute as a “locking” value, e.g. to avoid to perform an action twice on the same node. Note that, if you are using transactions, you may get the error only when the transaction is committed.

**subspecifier\_pk**

Return the subspecifier PK in the database (or None, if no subspecifier should be used)

**subspecifiers\_dict**

Return a dict to narrow down the query to only those matching also the subspecifier.

**classmethod validate\_key** (*key*)

Validate the key string to check if it is valid (e.g., if it does not contain the separator symbol.).

**Returns** None if the key is valid

**Raises ValidationError** if the key is not valid

**class** `aiida.djsite.db.models.DbNode` (*\*args, \*\*kwargs*)

Generic node: data or calculation or code.

Nodes can be linked (DbLink table) Naming convention for Node relationships: A → C → B.

- A is 'input' of C.
- C is 'output' of A.
- A is 'parent' of B,C
- C,B are 'children' of A.

**Note** parents and children are stored in the DbPath table, the transitive closure table, automatically updated via DB triggers whenever a link is added to or removed from the DbLink table.

Internal attributes, that define the node itself, are stored in the DbAttribute table; further user-defined attributes, called 'extra', are stored in the DbExtra table (same schema and methods of the DbAttribute table, but the code does not rely on the content of the table, therefore the user can use it at his will to tag or annotate nodes.

**Note** Attributes in the DbAttribute table have to be thought as belonging to the DbNode, (this is the reason for which there is no 'user' field in the DbAttribute field). Moreover, Attributes define uniquely the Node so should be immutable (except for the few ones defined in the `_updatable_attributes` attribute of the `Node()` class, that are updatable: these are Attributes that are set by AiiDA, so the user should not modify them, but can be changed (e.g., the `append_text` of a code, that can be redefined if the code has to be recompiled).

**attributes**

Return all attributes of the given node as a single dictionary.

**extras**

Return all extras of the given node as a single dictionary.

**get\_aiida\_class** ()

Return the corresponding aiida instance of class `aiida.orm.Node` or a appropriate subclass.

**get\_simple\_name** (*invalid\_result=None*)

Return a string with the last part of the type name.

If the type is empty, use 'Node'. If the type is invalid, return the content of the input variable `invalid_result`.

**Parameters** `invalid_result` – The value to be returned if the node type is not recognized.

**class** `aiida.djsite.db.models.DbPath` (*\*args, \*\*kwargs*)

Transitive closure table for all dbnode paths.

**expand** ()

Method to expand a DbPath (recursive function), i.e., to get a list of all dbnodes that are traversed in the given path.

**Returns** list of DbNode objects representing the expanded DbPath

```

class aiida.djsite.db.models.DbSetting(*args, **kwargs)
    This will store generic settings that should be database-wide.

class aiida.djsite.db.models.DbUser(*args, **kwargs)
    This class replaces the default User class of Django

class aiida.djsite.db.models.DbWorkflow(id, uuid, ctime, mtime, user_id, label, description,
                                         nodeversion, lastsyncedversion, state, report, module,
                                         module_class, script_path, script_md5)

    get_aiida_class()
        Return the corresponding aiida instance of class aiida.workflow

    is_subworkflow()
        Return True if this is a subworkflow, False if it is a root workflow, launched by the user.

class aiida.djsite.db.models.DbWorkflowData(id, parent_id, name, time, data_type, value_type,
                                             json_value, aiida_obj_id)

class aiida.djsite.db.models.DbWorkflowStep(id, parent_id, name, user_id, time, nextcall,
                                             state)

aiida.djsite.db.models.deserialize_attributes(data, sep, original_class=None, original_pk=None)
    Deserialize the attributes from the format internally stored in the DB to the actual format (dictionaries, lists,
    integers, ...

```

#### Parameters

- **data** – must be a dictionary of dictionaries. In the top-level dictionary, the key must be the key of the attribute. The value must be a dictionary with the following keys: datatype, tval, fval, ival, bval, dval. Other keys are ignored. NOTE that a type check is not performed! tval is expected to be a string, dval a date, etc.
- **sep** – a string, the separator between subfields (to separate the name of a dictionary from the keys it contains, for instance)
- **original\_class** – if these elements come from a specific subclass of DbMultipleValueAttributeBaseClass, pass here the class (note: the class, not the instance!). This is used only in case the wrong number of elements is found in the raw data, to print a more meaningful message (if the class has a dbnode associated to it)
- **original\_pk** – if the elements come from a specific subclass of DbMultipleValueAttributeBaseClass that has a dbnode associated to it, pass here the PK integer. This is used only in case the wrong number of elements is found in the raw data, to print a more meaningful message

**Returns** a dictionary, where for each entry the corresponding value is returned, deserialized back to lists, dictionaries, etc. Example: if data = {'a': {'datatype': 'list', 'ival': 2, ...}, 'a.0': {'datatype': 'int', 'ival': 2, ...}, 'a.1': {'datatype': 'txt', 'tval': 'yy'}], it will return {'a': [2, "yy"]}

### 4.1.7 ORM documentation: generic aiida.orm

This section describes the aiida/django object-relational mapping.

Some generic methods of the module aiida.orm

```

aiida.orm.CalculationFactory(module, from_abstract=False)
    Return a suitable JobCalculation subclass.

```

**Parameters**

- **module** – a valid string recognized as a Calculation plugin
- **from\_abstract** – A boolean. If False (default), actually look only to subclasses to JobCalculation, not to the base Calculation class. If True, check for valid strings for plugins of the Calculation base class.

`aiida.orm.DataFactory(module)`

Return a suitable Data subclass.

`aiida.orm.WorkflowFactory(module)`

Return a suitable Workflow subclass.

`aiida.orm.load_node(node_id=None, pk=None, uuid=None, parent_class=None)`

Return an AiiDA node given PK or UUID.

**Parameters**

- **node\_id** – PK (integer) or UUID (string) or a node
- **pk** – PK of a node
- **uuid** – UUID of a node
- **parent\_class** – if specified, checks whether the node loaded is a subclass of parent\_class

**Returns** an AiiDA node

**Raises**

- **ValueError** – if none or more than one of parameters is supplied or type of node\_id is neither string nor integer.
- **NotExistent** – if the parent\_class is specified and no matching Node is found.

`aiida.orm.load_workflow(wf_id=None, pk=None, uuid=None)`

Return an AiiDA workflow given PK or UUID.

**Parameters**

- **wf\_id** – PK (integer) or UUID (string) or a workflow
- **pk** – PK of a workflow
- **uuid** – UUID of a workflow

**Returns** an AiiDA workflow

**Raises** ValueError if none or more than one of parameters is supplied or type of wf\_id is neither string nor integer

## Computer

`class aiida.orm.computer.Computer(**kwargs)`

Base class to map a node in the DB + its permanent repository counterpart.

Stores attributes starting with an underscore.

Caches files and attributes before the first save, and saves everything only on store(). After the call to store(), in general attributes cannot be changed, except for those listed in the self.\_updatable\_attributes tuple (empty for this class, can be extended in a subclass).



Only after storing (or upon loading from uuid) metadata can be modified and in this case they are directly set on the db.

In the plugin, also set the `_plugin_type_string`, to be set in the DB in the 'type' field.

**copy()**

Return a copy of the current object to work with, not stored yet.

**full\_text\_info**

Return a (multiline) string with a human-readable detailed information on this computer.

**classmethod get(*computer*)**

Return a computer from its name (or from another Computer or DbComputer instance)

**get\_dbauthinfo(*user*)**

Return the `aiida.djsite.db.models.DbAuthInfo` instance for the given user on this computer, if the computer is not configured for the given user.

**Parameters** *user* – a DbUser instance.

**Returns** a `aiida.djsite.db.models.DbAuthInfo` instance

**Raises** `NotExist` if the computer is not configured for the given user.

**get\_default\_mpiprocs\_per\_machine()**

Return the default number of CPUs per machine (node) for this computer, or None if it was not set.

**get\_mpirun\_command()**

Return the mpirun command. Must be a list of strings, that will be then joined with spaces when submitting.

I also provide a sensible default that may be ok in many cases.

**is\_user\_configured(*user*)**

Return True if the computer is configured for the given user, False otherwise.

**Parameters** *user* – a DbUser instance.

**Returns** a boolean.

**is\_user\_enabled(*user*)**

Return True if the computer is enabled for the given user (looking only at the per-user setting: the computer could still be globally disabled).

**Note** Return False also if the user is not configured for the computer.

**Parameters** *user* – a DbUser instance.

**Returns** a boolean.

**classmethod list\_names()**

Return a list with all the names of the computers in the DB.

**logging** = <module 'logging' from '/usr/lib/python2.7/logging/\_\_init\_\_.pyc'>

**pk**

Return the principal key in the DB.

**set\_default\_mpiprocs\_per\_machine(*def\_cpus\_per\_machine*)**

Set the default number of CPUs per machine (node) for this computer. Accepts None if you do not want to set this value.

**set\_mpirun\_command(*val*)**

Set the mpirun command. It must be a list of strings (you can use `string.split()` if you have a single, space-separated string).

**store()**

Store the computer in the DB.

Differently from Nodes, a computer can be re-stored if its properties are to be changed (e.g. a new mpirun command, etc.)

**uuid**

Return the UUID in the DB.

**validate()**

Check if the attributes and files retrieved from the DB are valid. Raise a `ValidationError` if something is wrong.

Must be able to work even before storing: therefore, use the `get_attr` and similar methods that automatically read either from the DB or from the internal attribute cache.

For the base class, this is always valid. Subclasses will reimplement this. In the subclass, always call the `super().validate()` method first!

`aiida.orm.computer.delete_computer(computer)`

Delete a computer from the DB. It assumes that the DB backend does the proper checks and avoids to delete computers that have nodes attached to them.

Implemented as a function on purpose, otherwise complicated logic would be needed to set the internal state of the object after calling `computer.delete()`.

## Node

**class** `aiida.orm.node.AttributeManager(node)`

An object used internally to return the attributes as a dictionary.

**Note** Important! It cannot be used to change variables, just to read them. To change values (of unstored nodes), use the proper Node methods.

**\_\_init\_\_**(node)

**Parameters** `node` – the node object.

**class** `aiida.orm.node.Node(**kwargs)`

Base class to map a node in the DB + its permanent repository counterpart.

Stores attributes starting with an underscore.

Caches files and attributes before the first save, and saves everything only on `store()`. After the call to `store()`, in general attributes cannot be changed, except for those listed in the `self._updatable_attributes` tuple (empty for this class, can be extended in a subclass).

Only after storing (or upon loading from uuid) extras can be modified and in this case they are directly set on the db.

In the plugin, also set the `_plugin_type_string`, to be set in the DB in the ‘type’ field.

**\_\_init\_\_**(\*\*kwargs)

Initialize the object Node.

**Parameters** `uuid` (optional) – if present, the Node with given uuid is loaded from the database. (It is not possible to assign a uuid to a new Node.)

**add\_comment**(content, user=None)

Add a new comment.

**Parameters** `content` – string with comment

**add\_path** (*src\_abs*, *dst\_path*)

Copy a file or folder from a local file inside the repository directory. If there is a subpath, folders will be created.

Copy to a cache directory if the entry has not been saved yet.

**Parameters**

- **src\_abs** (*str*) – the absolute path of the file to copy.
- **dst\_filename** (*str*) – the (relative) path on which to copy.

**Todo** in the future, add an `add_attachment()` that has the same meaning of a extras file. Decide also how to store. If in two separate subfolders, remember to reset the limit.

**attrs** ()

Returns the keys of the attributes.

**Returns** a list of strings

**copy** ()

Return a copy of the current object to work with, not stored yet.

This is a completely new entry in the DB, with its own UUID. Works both on stored instances and with not-stored ones.

Copies files and attributes, but not the extras. Does not store the Node to allow modification of attributes.

**Returns** an object copy

**ctime**

Return the creation time of the node.

**dbnode**

**Returns** the corresponding Django DbNode object.

**del\_extra** (*key*)

Delete a extra, acting directly on the DB! The action is immediately performed on the DB. Since extras can be added only after storing the node, this function is meaningful to be called only after the `.store()` method.

**Parameters** **key** (*str*) – key name

**Raise** `AttributeError`: if key starts with underscore

**Raise** `ModificationNotAllowed`: if the node is not stored yet

**description**

Get the description of the node.

**Returns** a string

**extras** ()

Get the keys of the extras.

**Returns** a list of strings

**folder**

Get the folder associated with the node, whether it is in the temporary or the permanent repository.

**Returns** the `RepositoryFolder` object.

**get\_abs\_path** (*path=None*, *section=None*)

Get the absolute path to the folder associated with the Node in the AiiDA repository.

**Parameters**

- **path** (*str*) – the name of the subfolder inside the section. If None returns the abspath of the folder. Default = None.
- **section** – the name of the subfolder ('path' by default).

**Returns** a string with the absolute path

For the moment works only for one kind of files, 'path' (internal files)

**get\_attr** (*key, \*args*)

Get the attribute.

**Parameters**

- **key** – name of the attribute
- **value** (*optional*) – if no attribute key is found, returns value

**Returns** attribute value

**Raises**

- **IndexError** – If no attribute is found and there is no default
- **ValueError** – If more than two arguments are passed to get\_attr

**get\_attrs** ()

Return a dictionary with all attributes of this node.

**get\_comments** (*pk=None*)

Return a sorted list of comment values, one for each comment associated to the node.

**Parameters** **pk** – integer or list of integers. If it is specified, returns the comment values with desired pks. (pk refers to DbComment.pk)

**Returns** the list of comments, sorted by pk; each element of the list is a dictionary, containing (pk, email, ctime, mtime, content)

**get\_computer** ()

Get the computer associated to the node.

**Returns** the Computer object or None.

**get\_extra** (*key, \*args*)

Get the value of a extras, reading directly from the DB! Since extras can be added only after storing the node, this function is meaningful to be called only after the .store() method.

**Parameters**

- **key** (*str*) – key name
- **value** (*optional*) – if no attribute key is found, returns value

**Returns** the key value

**Raises ValueError** If more than two arguments are passed to get\_extra

**get\_extras** ()

Get the value of extras.

**Returns** the dictionary of extras ({} if no extras)

**get\_folder\_list** (*subfolder='.'*)

Get the the list of files/directory in the repository of the object.

**Parameters** **subfolder** (*str, optional*) – get the list of a subfolder

**Returns** a list of strings.

**get\_inputs** (*type=None, also\_labels=False, only\_in\_db=False*)

Return a list of nodes that enter (directly) in this node

**Parameters**

- **type** – If specified, should be a class, and it filters only elements of that specific type (or a subclass of ‘type’)
- **also\_labels** – If False (default) only return a list of input nodes. If True, return a list of tuples, where each tuple has the following format: (‘label’, Node), with ‘label’ the link label, and Node a Node instance or subclass
- **only\_in\_db** – Return only the inputs that are in the database, ignoring those that are in the local cache. Otherwise, return all links.

**get\_inputs\_dict** (*only\_in\_db=False*)

Return a dictionary where the key is the label of the input link, and the value is the input node.

**Returns** a dictionary {label:object}

**get\_outputs** (*type=None, also\_labels=False*)

Return a list of nodes that exit (directly) from this node

**Parameters**

- **type** – if specified, should be a class, and it filters only elements of that specific type (or a subclass of ‘type’)
- **also\_labels** – if False (default) only return a list of input nodes. If True, return a list of tuples, where each tuple has the following format: (‘label’, Node), with ‘label’ the link label, and Node a Node instance or subclass

**get\_outputs\_dict** ()

Return a dictionary where the key is the label of the output link, and the value is the input node. As some Nodes (Datas in particular) can have more than one output with the same label, all keys have the name of the link with appended the pk of the node in output. The key without pk appended corresponds to the oldest node.

**Returns** a dictionary {linkname:object}

**classmethod get\_subclass\_from\_pk** (*pk*)

Get a node object from the pk, with the proper subclass of Node. (integer primary key used in this database), but loading the proper subclass where appropriate.

**Parameters** **pk** – a string with the pk of the object to be loaded.

**Returns** the object of the proper subclass.

**Raise** **NotExistent**: if there is no entry of the desired object kind with the given pk.

**classmethod get\_subclass\_from\_uuid** (*uuid*)

Get a node object from the uuid, with the proper subclass of Node. (if Node(uuid=...) is called, only the Node class is loaded).

**Parameters** **uuid** – a string with the uuid of the object to be loaded.

**Returns** the object of the proper subclass.

**Raise** **NotExistent**: if there is no entry of the desired object kind with the given uuid.

**get\_user** ()

Get the user.

**Returns** a Django DbUser model object

**has\_children**

Property to understand if children are attached to the node :return: a boolean

**has\_parents**

Property to understand if parents are attached to the node :return: a boolean

**inp**

Traverse the graph of the database. Returns a databaseobject, linked to the current node, by means of the linkname. Example: B = A.inp.parameters: returns the object (B), with link from B to A, with linkname parameters C= A.inp: returns an InputManager, an object that is meant to be accessed as the previous example

**iterattrs** (*also\_updatable=True*)

Iterator over the attributes, returning tuples (key, value)

**Todo** optimize! At the moment, the call is very slow because it is also calling attr.getvalue() for each attribute, that has to perform complicated queries to rebuild the object.

**Parameters** **also\_updatable** (*bool*) – if False, does not iterate over attributes that are updatable

**iterextras** ()

Iterator over the extras, returning tuples (key, value)

**Todo** verify that I am not creating a list internally

**label**

Get the label of the node.

**Returns** a string.

**logger**

Get the logger of the Node object.

**Returns** Logger object

**mtime**

Return the modification time of the node.

**out**

Traverse the graph of the database. Returns a databaseobject, linked to the current node, by means of the linkname. Example: B = A.out.results: Returns the object B, with link from A to B, with linkname parameters

**pk**

**Returns** the principal key (the ID) as an integer, or None if the node was not stored yet

**classmethod query** (*\*args, \*\*kwargs*)

Map to the aiidaobjects manager of the DbNode, that returns Node objects (or their subclasses) instead of DbNode entities.

# TODO: VERY IMPORTANT: the recognition of a subclass from the type # does not work if the modules defining the subclasses are not # put in subfolders. # In the future, fix it either to make a cache and to store the # full dependency tree, or save also the path.

**remove\_path** (*path*)

Remove a file or directory from the repository directory. Can be called only before storing.

**Parameters** **path** (*str*) – relative path to file/directory.

**set** (*\*\*kwargs*)

For each k=v pair passed as kwargs, call the corresponding set\_k(v) method (e.g., calling

`self.set(property=5, mass=2)` will call `self.set_property(5)` and `self.set_mass(2)`. Useful especially in the `__init__`.

**Note** it uses the `_set_incompatibilities` list of the class to check that we are not setting methods that cannot be set at the same time. `_set_incompatibilities` must be a list of tuples, and each tuple specifies the elements that cannot be set at the same time. For instance, if `_set_incompatibilities = [('property', 'mass')]`, then the call `self.set(property=5, mass=2)` will raise a `ValueError`. If a tuple has more than two values, it raises `ValueError` if *all* keys are provided at the same time, but it does not give any error if at least one of the keys is not present.

**Note** If one element of `_set_incompatibilities` is a tuple with only one element, this element will not be settable using this function (and in particular,

**Raises `ValueError`** if the corresponding `set_k` method does not exist in `self`, or if the methods cannot be set at the same time.

**`set_computer`** (*computer*)

Set the computer to be used by the node.

Note that the computer makes sense only for some nodes: `Calculation`, `RemoteData`, ...

**Parameters** **computer** – the computer object

**`set_extra`** (*key, value, exclusive=False*)

Immediately sets an extra of a calculation, in the DB! No `.store()` to be called. Can be used *only* after saving.

**Parameters**

- **key** (*string*) – key name
- **value** – key value
- **exclusive** – (default=False). If `exclusive` is `True`, it raises a `UniquenessError` if an Extra with the same name already exists in the DB (useful e.g. to “lock” a node and avoid to run multiple times the same computation on it).

**Raises `UniquenessError`** if extra already exists and `exclusive` is `True`.

**`set_extras`** (*the\_dict*)

Immediately sets several extras of a calculation, in the DB! No `.store()` to be called. Can be used *only* after saving.

**Parameters** **the\_dict** – a dictionary of key:value to be set as extras

**`store`** (*with\_transaction=True*)

Store a new node in the DB, also saving its repository directory and attributes.

Can be called only once. Afterwards, attributes cannot be changed anymore! Instead, extras can be changed only AFTER calling this `store()` function.

**Note** After successful storage, those links that are in the cache, and for which also the parent node is already stored, will be automatically stored. The others will remain unstored.

**Parameters** **with\_transaction** – if `False`, no transaction is used. This is meant to be used ONLY if the outer calling function has already a transaction open!

**`store_all`** (*with\_transaction=True*)

Store the node, together with all input links, if cached, and also the linked nodes, if they were not stored yet.

**Parameters** **with\_transaction** – if `False`, no transaction is used. This is meant to be used ONLY if the outer calling function has already a transaction open!

**uuid**

**Returns** a string with the uuid

**class** `aiida.orm.node.NodeInputManager` (*node*)

To document

**\_\_init\_\_** (*node*)

**Parameters** *node* – the node object.

**class** `aiida.orm.node.NodeOutputManager` (*node*)

To document

**\_\_init\_\_** (*node*)

**Parameters** *node* – the node object.

`aiida.orm.node.from_type_to_pluginclassname` (*typestr*)

Return the string to pass to the `load_plugin` function, starting from the ‘type’ field of a Node.

## Workflow

**class** `aiida.orm.workflow.Workflow` (*\*\*kwargs*)

Base class to represent a workflow. This is the superclass of any workflow implementations, and provides all the methods necessary to interact with the database.

The typical use case are workflow stored in the `aiida.workflow` packages, that are initiated either by the user in the shell or by some scripts, and that are monitored by the aiida daemon.

Workflow can have steps, and each step must contain some calculations to be executed. At the end of the step’s calculations the workflow is reloaded in memory and the next methods is called.

**add\_attribute** (*\_name*, *\_value*)

Add one attributes to the Workflow. If another attribute is present with the same name it will be overwritten.  
:param name: a string with the attribute name to store :param value: a storable object to store

**add\_attributes** (*\_params*)

Add a set of attributes to the Workflow. If another attribute is present with the same name it will be overwritten.  
:param name: a string with the attribute name to store :param value: a storable object to store

**add\_path** (*src\_abs*, *dst\_path*)

Copy a file or folder from a local file inside the repository directory. If there is a subpath, folders will be created.

Copy to a cache directory if the entry has not been saved yet. *src\_abs*: the absolute path of the file to copy.  
*dst\_filename*: the (relative) path on which to copy.

**add\_result** (*\_name*, *\_value*)

Add one result to the Workflow. If another result is present with the same name it will be overwritten.  
:param name: a string with the result name to store :param value: a storable object to store

**add\_results** (*\_params*)

Add a set of results to the Workflow. If another result is present with the same name it will be overwritten.  
:param name: a string with the result name to store :param value: a storable object to store

**append\_to\_report** (*text*)

Adds text to the Workflow report.

**Note** Once, in case the workflow is a subworkflow of any other Workflow this method calls the `parent append_to_report` method; now instead this is not the case anymore



**attach\_calculation** (*calc*)

Adds a calculation to the caller step in the database. This is a lazy call, no calculations will be launched until the `next` method gets called. For a step to be completed all the calculations linked have to be in RETRIEVED state, after which the next method gets called from the workflow manager. :param calc: a JobCalculation object :raise: AiidaException: in case the input is not of JobCalculation type

**attach\_workflow** (*sub\_wf*)

Adds a workflow to the caller step in the database. This is a lazy call, no workflow will be started until the `next` method gets called. For a step to be completed all the workflows linked have to be in FINISHED state, after which the next method gets called from the workflow manager. :param next\_method: a Workflow object

**clear\_report** ()

Wipe the Workflow report. In case the workflow is a subworkflow of any other Workflow this method calls the parent `clear_report` method.

**current\_folder**

Get the current repository folder, whether the temporary or the permanent.

**Returns** the RepositoryFolder object.

**dbworkflowinstance**

Get the DbWorkflow object stored in the super class.

**Returns** DbWorkflow object from the database

**description**

Get the description of the workflow.

**Returns** a string

**exit** ()

This is the method to call in `next` to finish the Workflow. When exit is the next method, and no errors are found, the Workflow is set to FINISHED and removed from the execution manager duties.

**get\_abs\_path** (*path*, *section=None*)

TODO: For the moment works only for one kind of files, 'path' (internal files)

**get\_all\_calcs** (*calc\_class=<class 'aiida.orm.calculation.job.JobCalculation'>*, *calc\_state=None*, *depth=16*)

Get all calculations connected with this workflow and all its subworkflows up to a given depth. The list of calculations can be restricted to a given calculation type and state :param calc\_class: the calculation class to which the calculations should belong (default: JobCalculation)

**Parameters**

- **calc\_state** – a specific state to filter the calculations to retrieve
- **depth** – the maximum depth level the recursion on sub-workflows will try to reach (0 means we stay at the step level and don't go into sub-workflows, 1 means we go down to one step level of the sub-workflows, etc.)

**Returns** a list of JobCalculation objects

**get\_attribute** (*\_name*)

Get one Workflow attribute :param name: a string with the attribute name to retrieve :return: a dictionary of storable objects

**get\_attributes** ()

Get the Workflow attributes :return: a dictionary of storable objects

**get\_folder\_list** (*subfolder=''*)

Get the the list of files/directory in the repository of the object.

**Parameters** `subfolder` (*str, optional*) – get the list of a subfolder

**Returns** a list of strings.

**get\_parameter** (*\_name*)

Get one Workflow parameter :param name: a string with the parameters name to retrieve :return: a dictionary of storable objects

**get\_parameters** ()

Get the Workflow parameters :return: a dictionary of storable objects

**get\_report** ()

Return the Workflow report.

**Note** once, in case the workflow is a subworkflow of any other Workflow this method calls the parent `get_report` method. This is not the case anymore.

**Returns** a list of strings

**get\_result** (*\_name*)

Get one Workflow result :param name: a string with the result name to retrieve :return: a dictionary of storable objects

**get\_results** ()

Get the Workflow results :return: a dictionary of storable objects

**get\_state** ()

Get the Workflow's state :return: a state from `wf_states` in `aiida.common.datastructures`

**get\_step** (*step\_method*)

Retrieves by name a step from the Workflow. :param step\_method: a string with the name of the step to retrieve or a method :raise: `ObjectDoesNotExist`: if there is no step with the specific name. :return: a `DbWorkflowStep` object.

**get\_step\_calculations** (*step\_method, calc\_state=None*)

Retrieves all the calculations connected to a specific step in the database. If the step is not existent it returns `None`, useful for simpler grammatic in the workflow definition. :param next\_method: a Workflow step (decorated) method :param calc\_state: a specific state to filter the calculations to retrieve :return: a list of `JobCalculations` objects

**get\_step\_workflows** (*step\_method*)

Retrieves all the workflows connected to a specific step in the database. If the step is not existent it returns `None`, useful for simpler grammatic in the workflow definition. :param next\_method: a Workflow step (decorated) method

**get\_steps** (*state=None*)

Retrieves all the steps from a specific workflow Workflow with the possibility to limit the list to a specific step's state. :param state: a state from `wf_states` in `aiida.common.datastructures` :return: a list of `DbWorkflowStep` objects.

**classmethod** **get\_subclass\_from\_dbnode** (*wf\_db*)

Loads the workflow object and reoads the python script in memory with the `importlib` library, the main class is searched and then loaded. :param wf\_db: a specific `DbWorkflowNode` object representing the Workflow :return: a Workflow subclass from the specific source code

**classmethod** **get\_subclass\_from\_pk** (*pk*)

Calls the `get_subclass_from_dbnode` selecting the `DbWorkflowNode` from the input `pk`. :param pk: a primary key index for the `DbWorkflowNode` :return: a Workflow subclass from the specific source code

**classmethod `get_subclass_from_uuid`** (*uuid*)  
 Calls the `get_subclass_from_dbnode` selecting the `DbWorkflowNode` from the input *uuid*. :param *uuid*: a uuid for the `DbWorkflowNode` :return: a Workflow subclass from the specific source code

**`get_temp_folder`** ()  
 Get the folder of the Node in the temporary repository.  
**Returns** a `SandboxFolder` object mapping the node in the repository.

**`has_failed`** ()  
 Returns True is the Workflow's state is ERROR

**`has_finished_ok`** ()  
 Returns True is the Workflow's state is FINISHED

**`has_step`** (*step\_method*)  
 Return if the Workflow has a step with a specific name. :param *step\_method*: a string with the name of the step to retrieve or a method

**`info`** ()  
 Returns an array with all the informations about the modules, file, class to locate the workflow source code

**`is_new`** ()  
 Returns True is the Workflow's state is CREATED

**`is_running`** ()  
 Returns True is the Workflow's state is RUNNING

**`is_subworkflow`** ()  
 Return True is this is a subworkflow (i.e., if it has a parent), False otherwise.

**`kill`** (*verbose=False*)  
 Stop the Workflow execution and change its state to FINISHED.  
 This method calls the `kill` method for each Calculation and each subworkflow linked to each RUNNING step.  
**Parameters** *verbose* – True to print the pk of each subworkflow killed  
**Raises `InvalidOperation`** if some calculations cannot be killed (the workflow will be also put to SLEEP so that it can be killed later on)

**`kill_step_calculations`** (*step*)  
 Calls the `kill` method for each Calculation linked to the step method passed as argument. :param *step*: a Workflow step (decorated) method

**`label`**  
 Get the label of the workflow.  
**Returns** a string

**`logger`**  
 Get the logger of the Workflow object, so that it also logs to the DB.  
**Returns** `LoggerAdapter` object, that works like a logger, but also has the 'extra' embedded

**`next`** (*next\_method*)  
 Adds the a new step to be called after the completion of the caller method's calculations and subworkflows.  
 This method must be called inside a Workflow step, otherwise an error is thrown. The code finds the caller method and stores in the database the input *next\_method* as the next method to be called. At this point no execution in made, only configuration updates in the database.

If during the execution of the caller method the user launched calculations or subworkflows, this method will add them to the database, making them available to the workflow manager to be launched. In fact all the calculation and subworkflow submissions are lazy method, really executed by this call.

**Parameters** `next_method` – a Workflow step method to execute after the caller method

**Raise** `AiidaException`: in case the caller method cannot be found or validated

**Returns** the wrapped methods, decorated with the correct step name

**pk**

Returns the DbWorkflow pk

**classmethod** `query` (*\*args*, *\*\*kwargs*)

Map to the aiidaobjects manager of the DbWorkflow, that returns Workflow objects instead of DbWorkflow entities.

**remove\_path** (*path*)

Remove a file or directory from the repository directory.

Can be called only before storing.

**repo\_folder**

Get the permanent repository folder. Use preferentially the `current_folder` method.

**Returns** the permanent RepositoryFolder object

**set\_params** (*params*, *force=False*)

Adds parameters to the Workflow that are both stored and used every time the workflow engine re-initialize the specific workflow to launch the new methods.

**set\_state** (*state*)

Set the Workflow's state :param name: a state from `wf_states` in `aiida.common.datastructures`

**sleep** ()

Changes the workflow state to SLEEP, only possible to call from a Workflow step decorated method.

**classmethod** `step` (*fun*)

This method is used as a decorator for workflow steps, and handles the method's execution, the state updates and the eventual errors.

The decorator generates a wrapper around the input function to execute, adding with the correct step name and a utility variable to make it distinguishable from non-step methods.

When a step is launched, the wrapper tries to run the function in case of error the state of the workflow is moved to ERROR and the traceback is stored in the report. In general the input method is a step obtained from the Workflow object, and the decorator simply handles a controlled execution of the step allowing the code not to break in case of error in the step's source code.

The wrapper also tests not to run two times the same step, unless a Workflow is in ERROR state, in this case all the calculations and subworkflows of the step are killed and a new execution is allowed.

**Parameters** `fun` – a methods to wrap, making it a Workflow step

**Raise** `AiidaException`: in case the workflow state doesn't allow the execution

**Returns** the wrapped methods,

**store** ()

Stores the DbWorkflow object data in the database

**uuid**

Returns the DbWorkflow uuid

**exception** `aiida.orm.workflow.WorkflowKillError(*args, **kwargs)`

An exception raised when a workflow failed to be killed. The `error_message_list` attribute contains the error messages from all the subworkflows.

**exception** `aiida.orm.workflow.WorkflowUnkillable`

Raised when a workflow cannot be killed because it is in the FINISHED or ERROR state.

`aiida.orm.workflow.get_workflow_info(w, tab_size=2, short=False, pre_string='', depth=16)`

Return a string with all the information regarding the given workflow and all its calculations and subworkflows. This is a recursive function (to print all subworkflows info as well).

#### Parameters

- **w** – a `DbWorkflow` instance
- **tab\_size** – number of spaces to use for the indentation
- **short** – if True, provide a shorter output (only total number of calculations, rather than the state of each calculation)
- **pre\_string** – string appended at the beginning of each line
- **depth** – the maximum depth level the recursion on sub-workflows will try to reach (0 means we stay at the step level and don't go into sub-workflows, 1 means we go down to one step level of the sub-workflows, etc.)

**Return lines** list of lines to be outputted

`aiida.orm.workflow.kill_all()`

Kills all the workflows not in FINISHED state running the `kill_from_uuid` method in a loop.

**Parameters** **uuid** – the UUID of the workflow to kill

`aiida.orm.workflow.kill_from_pk(pk, verbose=False)`

Kills a workflow without loading the class, useful when there was a problem and the workflow definition module was changed/deleted (and the workflow cannot be reloaded).

#### Parameters

- **pk** – the principal key (id) of the workflow to kill
- **verbose** – True to print the pk of each subworkflow killed

`aiida.orm.workflow.kill_from_uuid(uuid)`

Kills a workflow without loading the class, useful when there was a problem and the workflow definition module was changed/deleted (and the workflow cannot be reloaded).

**Parameters** **uuid** – the UUID of the workflow to kill

## Code

**class** `aiida.orm.code.Code(**kwargs)`

A code entity. It can either be 'local', or 'remote'.

- **Local code:** it is a collection of files/dirs (added using the `add_path()` method), where one file is flagged as executable (using the `set_local_executable()` method).
- **Remote code:** it is a pair (`remotecomputer`, `remotepath_of_executable`) set using the `set_remote_computer_exec()` method.

For both codes, one can set some code to be executed right before or right after the execution of the code, using the `set_preexec_code()` and `set_postexec_code()` methods (e.g., the `set_preexec_code()` can be used to load specific modules required for the code to be run).

**can\_run\_on** (*computer*)

Return True if this code can run on the given computer, False otherwise.

Local codes can run on any machine; remote codes can run only on the machine on which they reside.

TODO: add filters to mask the remote machines on which a local code can run.

**full\_text\_info**

Return a (multiline) string with a human-readable detailed information on this computer.

**classmethod get** (*label*, *computername=None*, *useremail=None*)

Get a code from its label.

#### Parameters

- **label** – the code label
- **computername** – filter only codes on computers with this name
- **useremail** – filter only codes belonging to a user with this email

#### Raises

- **NotExistent** – if no matches are found
- **MultipleObjectsError** – if multiple matches are found. In this case you may want to pass the additional parameters to filter the codes, or relabel the codes.

**get\_append\_text** ()

Return the postexec\_code, or an empty string if no post-exec code was defined.

**get\_execname** ()

Return the executable string to be put in the script. For local codes, it is `./LOCAL_EXECUTABLE_NAME`. For remote codes, it is the absolute path to the executable.

**classmethod get\_from\_string** (*code\_string*)

Get a Computer object with given identifier string, that can either be the numeric ID (pk), or the label (if unique); the label can either be simply the label, or in the format `label@machinename`. See the note below for details on the string detection algorithm.

---

**Note:** If a string that can be converted to an integer is given, the numeric ID is verified first (therefore, is a code A with a label equal to the ID of another code B is present, code A cannot be referenced by label). Similarly, the (leftmost) '@' symbol is always used to split code and computername. Therefore do not use '@' in the code name if you want to use this function ('@' in the computer name are instead valid).

---

**Parameters** **code\_string** – the code string identifying the code to load

#### Raises

- **NotExistent** – if no code identified by the given string is found
- **MultipleObjectsError** – if the string cannot identify uniquely a code

**get\_input\_plugin\_name** ()

Return the name of the default input plugin (or None if no input plugin was set).

**get\_prepend\_text** ()

Return the code that will be put in the scheduler script before the execution, or an empty string if no pre-exec code was defined.

**is\_local** ()

Return True if the code is 'local', False if it is 'remote' (see also documentation of the `set_local` and `set_remote` functions).

**classmethod** `list_for_plugin(plugin, labels=True)`

Return a list of valid code strings for a given plugin.

**Parameters**

- **plugin** – The string of the plugin.
- **labels** – if True, return a list of code names, otherwise return the code PKs (integers).

**Returns** a list of string, with the code names if labels is True, otherwise a list of integers with the code PKs.

**new\_calc** (\*args, \*\*kwargs)

Create and return a new Calculation object (unstored) with the correct plugin subclass, as obtained by the `self.get_input_plugin_name()` method.

Parameters are passed to the calculation `__init__` method.

**Note** it also directly creates the link to this code (that will of course be cached, since the new node is not stored yet).

**Raises**

- **MissingPluginError** – if the specified plugin does not exist.
- **ValueError** – if no plugin was specified.

**set\_append\_text** (code)

Pass a string of code that will be put in the scheduler script after the execution of the code.

**set\_files** (files)

Given a list of filenames (or a single filename string), add it to the path (all at level zero, i.e. without folders). Therefore, be careful for files with the same name!

**Todo** decide whether to check if the Code must be a local executable to be able to call this function.

**set\_input\_plugin\_name** (input\_plugin)

Set the name of the default input plugin, to be used for the automatic generation of a new calculation.

**set\_local\_executable** (exec\_name)

Set the filename of the local executable. Implicitly set the code as local.

**set\_prepend\_text** (code)

Pass a string of code that will be put in the scheduler script before the execution of the code.

**set\_remote\_computer\_exec** (remote\_computer\_exec)

Set the code as remote, and pass the computer on which it resides and the absolute path on that computer.

**Args:**

**remote\_computer\_exec:** a tuple (computer, remote\_exec\_path), where computer is a `aiida.orm.Computer` or an `aiida.djsite.db.models.DbComputer` object, and remote\_exec\_path is the absolute path of the main executable on remote computer.

`aiida.orm.code.delete_code` (code)

Delete a code from the DB. Check before that there are no output nodes.

NOTE! Not thread safe... Do not use with many users accessing the DB at the same time.

Implemented as a function on purpose, otherwise complicated logic would be needed to set the internal state of the object after calling `computer.delete()`.

### 4.1.8 ORM documentation: Data

**class** `aiida.orm.data.Data` (*\*\*kwargs*)

This class is base class for all data objects.

**convert** (*object\_format=None, \*args*)

Convert the AiiDA StructureData into another python object

**Parameters** `object_format` – Specify the output format

**export** (*fname, fileformat=None*)

Save a Data object to a file.

**Parameters**

- **fname** – string with file name. Can be an absolute or relative path.
- **fileformat** – kind of format to use for the export. If not present, it will try to use the extension of the file name.

**importfile** (*fname, fileformat=None*)

Populate a Data object from a file.

**Parameters**

- **fname** – string with file name. Can be an absolute or relative path.
- **fileformat** – kind of format to use for the export. If not present, it will try to use the extension of the file name.

**importstring** (*inputstring, fileformat, \*\*kwargs*)

Converts a Data object to other text format.

**Parameters** `fileformat` – a string (the extension) to describe the file format.

**Returns** a string with the structure description.

**set\_source** (*source*)

Sets the dictionary describing the source of Data object.

**source**

Gets the dictionary describing the source of Data object. Possible fields:

- **db\_name**: name of the source database.
- **db\_uri**: URI of the source database.
- **uri**: URI of the object's source. Should be a permanent link.
- **id**: object's source identifier in the source database.
- **version**: version of the object's source.
- **extras**: a dictionary with other fields for source description.
- **source\_md5**: MD5 checksum of object's source.
- **description**: human-readable free form description of the object's source.
- **license**: a string with a type of license.

---

**Note:** some limitations for setting the data source exist, see `_validate()`.

---

**Returns** dictionary describing the source of Data object.



## Structure

This module defines the classes for structures and all related functions to operate on them.

**class** `aiida.orm.data.structure.Kind` (*\*\*kwargs*)

This class contains the information about the species (kinds) of the system.

It can be a single atom, or an alloy, or even contain vacancies.

**\_\_init\_\_** (*\*\*kwargs*)

Create a site. One can either pass:

### Parameters

- **raw** – the raw python dictionary that will be converted to a Kind object.
- **ase** – an ase Atom object
- **kind** – a Kind object (to get a copy)

Or alternatively the following parameters:

### Parameters

- **symbols** – a single string for the symbol of this site, or a list of symbol strings
- **(optional) (mass)** – the weights for each atomic species of this site. If only a single symbol is provided, then this value is optional and the weight is set to 1.
- **(optional)** – the mass for this site in atomic mass units. If not provided, the mass is set by the `self.reset_mass()` function.
- **name** – a string that uniquely identifies the kind, and that is used to identify the sites.

**compare\_with** (*other\_kind*)

Compare with another Kind object to check if they are different.

---

**Note:** This does NOT check the ‘type’ attribute. Instead, it compares (with reasonable thresholds, where applicable): the mass, and the list of symbols and of weights. Moreover, it compares the `_internal_tag`, if defined (at the moment, defined automatically only when importing the Kind from ASE, if the atom has a non-zero tag). Note that the `_internal_tag` is only used while the class is loaded, but is not persisted on the database.

---

**Returns** A tuple with two elements. The first one is True if the two sites are ‘equivalent’ (same mass, symbols and weights), False otherwise. The second element of the tuple is a string, which is either None (if the first element was True), or contains a ‘human-readable’ description of the first difference encountered between the two sites.

**get\_raw** ()

Return the raw version of the site, mapped to a suitable dictionary. This is the format that is actually used to store each kind of the structure in the DB.

**Returns** a python dictionary with the kind.

**get\_symbols\_string** ()

Return a string that tries to match as good as possible the symbols of this kind. If there is only one symbol (no alloy) with 100% occupancy, just returns the symbol name. Otherwise, groups the full string in curly brackets, and try to write also the composition (with 2 precision only).

---

**Note:** If there is a vacancy (sum of weights<1), we indicate it with the X symbol followed by 1-sum(weights) (still with 2 digits precision, so it can be 0.00)

---

---

**Note:** Note the difference with respect to the symbols and the symbol properties!

---

**has\_vacancies** ()

Returns True if the sum of the weights is less than one. It uses the internal variable `_sum_threshold` as a threshold.

**Returns** a boolean

**is\_alloy** ()

To understand if kind is an alloy.

**Returns** True if the kind has more than one element (i.e., `len(self.symbols) != 1`), False otherwise.

**mass**

The mass of this species kind.

**Returns** a float

**name**

Return the name of this kind. The name of a kind is used to identify the species of a site.

**Returns** a string

**reset\_mass** ()

Reset the mass to the automatic calculated value.

The mass can be set manually; by default, if not provided, it is the mass of the constituent atoms, weighted with their weight (after the weight has been normalized to one to take correctly into account vacancies).

This function uses the internal `_symbols` and `_weights` values and thus assumes that the values are validated.

It sets the mass to None if the sum of weights is zero.

**set\_automatic\_kind\_name** (*tag=None*)

Set the type to a string obtained with the symbols appended one after the other, without spaces, in alphabetical order; if the site has a vacancy, a X is appended at the end too.

**set\_symbols\_and\_weights** (*symbols, weights*)

Set the chemical symbols and the weights for the site.

---

**Note:** Note that the kind name remains unchanged.

---

**symbol**

If the kind has only one symbol, return it; otherwise, raise a `ValueError`.

**symbols**

List of symbols for this site. If the site is a single atom, pass a list of one element only, or simply the string for that atom. For alloys, a list of elements.

---

**Note:** Note that if you change the list of symbols, the kind name remains unchanged.

---

**weights**

Weights for this species kind. Refer also to `:func:validate_symbols_tuple` for the validation rules on the weights.

**class** `aiida.orm.data.structure.Site` (*\*\*kwargs*)

This class contains the information about a given site of the system.

It can be a single atom, or an alloy, or even contain vacancies.

**\_\_init\_\_** (*\*\*kwargs*)

Create a site.

#### Parameters

- **kind\_name** – a string that identifies the kind (species) of this site. This has to be found in the list of kinds of the StructureData object. Validation will be done at the StructureData level.
- **position** – the absolute position (three floats) in angstrom

**get\_ase** (*kinds*)

Return a ase.Atom object for this site.

**Parameters** **kinds** – the list of kinds from the StructureData object.

---

**Note:** If any site is an alloy or has vacancies, a ValueError is raised (from the site.get\_ase() routine).

---

**get\_raw** ()

Return the raw version of the site, mapped to a suitable dictionary. This is the format that is actually used to store each site of the structure in the DB.

**Returns** a python dictionary with the site.

**kind\_name**

Return the kind name of this site (a string).

The type of a site is used to decide whether two sites are identical (same mass, symbols, weights, ...) or not.

**position**

Return the position of this site in absolute coordinates, in angstrom.

**class** `aiida.orm.data.structure.StructureData` (*\*\*kwargs*)

This class contains the information about a given structure, i.e. a collection of sites together with a cell, the boundary conditions (whether they are periodic or not) and other related useful information.

**append\_atom** (*\*\*kwargs*)

Append an atom to the Structure, taking care of creating the corresponding kind.

#### Parameters

- **ase** – the ase Atom object from which we want to create a new atom (if present, this must be the only parameter)
- **position** – the position of the atom (three numbers in angstrom)
- **symbols, weights, name** (..) – any further parameter is passed to the constructor of the Kind object. For the ‘name’ parameter, see the note below.

---

**Note:** Note on the ‘name’ parameter (that is, the name of the kind):

- if specified, no checks are done on existing species. Simply, a new kind with that name is created. If there is a name clash, a check is done: if the kinds are identical, no error is issued; otherwise, an error is issued because you are trying to store two different kinds with the same name.
- if not specified, the name is automatically generated. Before adding the kind, a check is done. If other species with the same properties already exist, no new kinds are created, but the site is added to the existing (identical) kind. (Actually, the first kind that is encountered). Otherwise, the name is made unique first, by adding to the string containing the list of chemical symbols a number starting from 1, until an unique name is found

---

**Note:** checks of equality of species are done using the `compare_with()` method.

---

**append\_kind** (*kind*)

Append a kind to the *StructureData*. It makes a copy of the kind.

**Parameters** **kind** – the site to append, must be a Kind object.

**append\_site** (*site*)

Append a site to the *StructureData*. It makes a copy of the site.

**Parameters** **site** – the site to append. It must be a Site object.

**cell**

Returns the cell shape.

**Returns** a 3x3 list of lists.

**cell\_angles**

Get the angles between the cell lattice vectors in degrees.

**cell\_lengths**

Get the lengths of cell lattice vectors in angstroms.

**clear\_kinds** ()

Removes all kinds for the StructureData object.

---

**Note:** Also clear all sites!

---

**clear\_sites** ()

Removes all sites for the StructureData object.

**get\_ase** ()

Get the ASE object. Requires to be able to import ase.

**Returns** an ASE object corresponding to this *StructureData* object.

---

**Note:** If any site is an alloy or has vacancies, a ValueError is raised (from the site.get\_ase() routine).

---

**get\_cell\_volume** ()

Returns the cell volume in Angstrom<sup>3</sup>.

**Returns** a float.

**get\_formula** (*mode='hill', separator=''*)

Return a string with the chemical formula.

**Parameters**

- **mode** – a string to specify how to generate the formula, can assume one of the following values:
  - ‘hill’ (default): count the number of atoms of each species, then use Hill notation, i.e. alphabetical order with C and H first if one or several C atom(s) is (are) present, e.g. [ ‘C’, ‘H’, ‘H’, ‘H’, ‘O’, ‘C’, ‘H’, ‘H’, ‘H’ ] will return ‘C2H6O’ [ ‘S’, ‘O’, ‘O’, ‘H’, ‘O’, ‘H’, ‘O’ ] will return ‘H2O4S’ From E. A. Hill, J. Am. Chem. Soc., 22 (8), pp 478–494 (1900)
  - ‘hill\_compact’: same as hill but the number of atoms for each species is divided by the greatest common divisor of all of them, e.g.

- `['C', 'H', 'H', 'H', 'O', 'C', 'H', 'H', 'H', 'O', 'O', 'O']` will return `'CH3O2'`
- `'reduce'`: group repeated symbols e.g. `['Ba', 'Ti', 'O', 'O', 'O', 'Ba', 'Ti', 'Ti', 'O', 'O', 'O']` will return `'BaTiO3BaTiO3BaTi2O3'`
  - `'group'`: will try to group as much as possible parts of the formula e.g. `['Ba', 'Ti', 'O', 'O', 'O', 'Ba', 'Ti', 'O', 'O', 'O', 'Ba', 'Ti', 'Ti', 'O', 'O', 'O']` will return `'(BaTiO3)2BaTi2O3'`
  - `'count'`: same as hill (i.e. one just counts the number of atoms of each species) without the re-ordering (take the order of the atomic sites), e.g. `['Ba', 'Ti', 'O', 'O', 'O', 'Ba', 'Ti', 'O', 'O', 'O']` will return `'Ba2Ti2O6'`
  - `'count_compact'`: same as count but the number of atoms for each species is divided by the greatest common divisor of all of them, e.g. `['Ba', 'Ti', 'O', 'O', 'O', 'Ba', 'Ti', 'O', 'O', 'O']` will return `'BaTiO3'`
- **separator** – a string used to concatenate symbols. Default empty.

**Returns** a string with the formula

---

**Note:** in modes `reduce`, `group`, `count` and `count_compact`, the initial order in which the atoms were appended by the user is used to group and/or order the symbols in the formula

---

**get\_kind**(*kind\_name*)

Return the kind object associated with the given kind name.

**Parameters** **kind\_name** – String, the name of the kind you want to get

**Returns** The Kind object associated with the given `kind_name`, if a Kind with the given name is present in the structure.

**Raise** `ValueError` if the `kind_name` is not present.

**get\_kind\_names**()

Return a list of kind names (in the same order of the `self.kinds` property, but return the names rather than Kind objects)

---

**Note:** This is NOT necessarily a list of chemical symbols! Use `get_symbols_set` for chemical symbols

---

**Returns** a list of strings.

**get\_pymatgen**()

Get pymatgen object. Returns `Structure` for structures with periodic boundary conditions (in three dimensions) and `Molecule` otherwise.

---

**Note:** Requires the pymatgen module (version  $\geq 3.0.13$ , usage of earlier versions may cause errors).

---

**get\_pymatgen\_molecule**()

Get the pymatgen `Molecule` object.

---

**Note:** Requires the pymatgen module (version  $\geq 3.0.13$ , usage of earlier versions may cause errors).

---

**Returns** a pymatgen `Molecule` object corresponding to this `StructureData` object.

**get\_pymatgen\_structure()**

Get the pymatgen Structure object.

---

**Note:** Requires the pymatgen module (version  $\geq 3.0.13$ , usage of earlier versions may cause errors).

---

**Returns** a pymatgen Structure object corresponding to this *StructureData* object.

**Raises ValueError** if periodic boundary conditions do not hold in at least one dimension of real space.

**get\_site\_kindnames()**

Return a list with length equal to the number of sites of this structure, where each element of the list is the kind name of the corresponding site.

---

**Note:** This is NOT necessarily a list of chemical symbols! Use `[self.get_kind(s.kind_name).get_symbols_string() for s in self.sites]` for chemical symbols

---

**Returns** a list of strings

**get\_symbols\_set()**

Return a set containing the names of all elements involved in this structure (i.e., for it joins the list of symbols for each kind *k* in the structure).

**Returns** a set of strings of element names.

**has\_vacancies()**

To understand if there are vacancies in the structure.

**Returns** a boolean, True if at least one kind has a vacancy

**is\_alloy()**

To understand if there are alloys in the structure.

**Returns** a boolean, True if at least one kind is an alloy

**kinds**

Returns a list of kinds.

**pbc**

Get the periodic boundary conditions.

**Returns** a tuple of three booleans, each one tells if there are periodic boundary conditions for the *i*-th real-space direction (*i*=1,2,3)

**reset\_cell(new\_cell)**

Reset the cell of a structure not yet stored to a new value.

**Parameters** *new\_cell* – list specifying the cell vectors

**Raises** ModificationNotAllowed: if object is already stored

**reset\_sites\_positions(new\_positions, conserve\_particle=True)**

Replace all the Site positions attached to the Structure

**Parameters**

- **new\_positions** – list of (3D) positions for every sites.
- **conserve\_particle** – if True, allows the possibility of removing a site. currently not implemented.

**Raises**

- **ModificationNotAllowed** – if object is stored already
- **ValueError** – if positions are invalid

---

**Note:** it is assumed that the order of the new `_positions` is given in the same order of the one it's substituting, i.e. the kind of the site will not be checked.

---

**set\_ase** (*aseatoms*)

Load the structure from a ASE object

**set\_pymatgen** (*obj*, *\*\*kwargs*)

Load the structure from a pymatgen object.

---

**Note:** Requires the pymatgen module (version  $\geq 3.0.13$ , usage of earlier versions may cause errors).

---

**set\_pymatgen\_molecule** (*mol*, *margin=5*)

Load the structure from a pymatgen Molecule object.

**Parameters** *margin* – the margin to be added in all directions of the bounding box of the molecule.

---

**Note:** Requires the pymatgen module (version  $\geq 3.0.13$ , usage of earlier versions may cause errors).

---

**set\_pymatgen\_structure** (*struct*)

Load the structure from a pymatgen Structure object.

---

**Note:** periodic boundary conditions are set to True in all three directions.

---



---

**Note:** Requires the pymatgen module (version  $\geq 3.0.13$ , usage of earlier versions may cause errors).

---

**sites**

Returns a list of sites.

`aiida.orm.data.structure.ase_refine_cell` (*aseatoms*, *\*\*kwargs*)

Detect the symmetry of the structure, remove symmetric atoms and refine unit cell.

**Parameters**

- **aseatoms** – an ase.atoms.Atoms instance
- **symprec** – symmetry precision, used by pypsglib

**Return newase** refined cell with reduced set of atoms

**Return symmetry** a dictionary describing the symmetry space group

`aiida.orm.data.structure.calc_cell_volume` (*cell*)

Calculates the volume of a cell given the three lattice vectors.

It is calculated as  $\text{cell}[0] \cdot (\text{cell}[1] \times \text{cell}[2])$ , where  $\cdot$  represents a dot product and  $\times$  a cross product.

**Parameters** *cell* – the cell vectors; there must be a 3x3 list of lists of floats, no other checks are done.

**Returns** the cell volume.

`aiida.orm.data.structure.get_formula` (*symbol\_list*, *mode='hill'*, *separator=''*)

Return a string with the chemical formula.

**Parameters**

- **symbol\_list** – a list of symbols, e.g. `['H', 'H', 'O']`
- **mode** – a string to specify how to generate the formula, can assume one of the following values:
  - ‘hill’ (default): count the number of atoms of each species, then use Hill notation, i.e. alphabetical order with C and H first if one or several C atom(s) is (are) present, e.g. `['C', 'H', 'H', 'H', 'O', 'C', 'H', 'H', 'H']` will return `'C2H6O'` `['S', 'O', 'O', 'H', 'O', 'H', 'O']` will return `'H2O4S'` From E. A. Hill, J. Am. Chem. Soc., 22 (8), pp 478–494 (1900)
  - ‘hill\_compact’: same as hill but the number of atoms for each species is divided by the greatest common divisor of all of them, e.g. `['C', 'H', 'H', 'H', 'O', 'C', 'H', 'H', 'H', 'O', 'O', 'O']` will return `'CH3O2'`
  - ‘reduce’: group repeated symbols e.g. `['Ba', 'Ti', 'O', 'O', 'O', 'Ba', 'Ti', 'O', 'O', 'Ba', 'Ti', 'Ti', 'O', 'O', 'O']` will return `'BaTiO3BaTiO3BaTi2O3'`
  - ‘group’: will try to group as much as possible parts of the formula e.g. `['Ba', 'Ti', 'O', 'O', 'O', 'Ba', 'Ti', 'O', 'O', 'O', 'Ba', 'Ti', 'Ti', 'O', 'O', 'O']` will return `'(BaTiO3)2BaTi2O3'`
  - ‘count’: same as hill (i.e. one just counts the number of atoms of each species) without the re-ordering (take the order of the atomic sites), e.g. `['Ba', 'Ti', 'O', 'O', 'O', 'Ba', 'Ti', 'O', 'O', 'O']` will return `'Ba2Ti2O6'`
  - ‘count\_compact’: same as count but the number of atoms for each species is divided by the greatest common divisor of all of them, e.g. `['Ba', 'Ti', 'O', 'O', 'O', 'Ba', 'Ti', 'O', 'O', 'O']` will return `'BaTiO3'`
- **separator** – a string used to concatenate symbols. Default empty.

**Returns** a string with the formula

---

**Note:** in modes reduce, group, count and count\_compact, the initial order in which the atoms were appended by the user is used to group and/or order the symbols in the formula

---

```
aiida.orm.data.structure.get_formula_from_symbol_list(_list, separator='')
```

Return a string with the formula obtained from the list of symbols. Examples: \*  
`[[1, 'Ba'], [1, 'Ti'], [3, 'O']]` will return `'BaTiO3'` \* `[[2, [ [1, 'Ba'], [1, 'Ti'] ] ]]` will return `'(BaTi)2'`

**Parameters**

- **\_list** – a list of symbols and multiplicities as obtained from the function `group_symbols`
- **separator** – a string used to concatenate symbols. Default empty.

**Returns** a string

```
aiida.orm.data.structure.get_formula_group(symbol_list, separator='')
```

Return a string with the chemical formula from a list of chemical symbols. The formula is written in a compact way, i.e. trying to group as much as possible parts of the formula.

---

**Note:** it works for instance very well if structure was obtained from an ASE supercell.

---



Example of result: ['Ba', 'Ti', 'O', 'O', 'O', 'Ba', 'Ti', 'O', 'O', 'O', 'Ba', 'Ti', 'Ti', 'O', 'O', 'O'] will return '(BaTiO3)2BaTi2O3'.

#### Parameters

- **symbol\_list** – list of symbols (e.g. ['Ba','Ti','O','O','O'])
- **separator** – a string used to concatenate symbols. Default empty.

**Returns** a string with the chemical formula for the given structure.

`aiida.orm.data.structure.get_pymatgen_version()`

**Returns** string with pymatgen version, None if can not import.

`aiida.orm.data.structure.get_symbols_string(symbols, weights)`

Return a string that tries to match as good as possible the symbols and weights. If there is only one symbol (no alloy) with 100% occupancy, just returns the symbol name. Otherwise, groups the full string in curly brackets, and try to write also the composition (with 2 precision only). If (sum of weights<1), we indicate it with the X symbol followed by 1-sum(weights) (still with 2 digits precision, so it can be 0.00)

#### Parameters

- **symbols** – the symbols as obtained from <kind>.\_symbols
- **weights** – the weights as obtained from <kind>.\_weights

---

**Note:** Note the difference with respect to the symbols and the symbol properties!

---

`aiida.orm.data.structure.get_valid_pbc(inputpbc)`

Return a list of three booleans for the periodic boundary conditions, in a valid format from a generic input.

**Raises ValueError** if the format is not valid.

`aiida.orm.data.structure.group_symbols(_list)`

Group a list of symbols to a list containing the number of consecutive identical symbols, and the symbol itself.

Examples:

- ['Ba', 'Ti', 'O', 'O', 'O', 'Ba'] will return [[1, 'Ba'], [1, 'Ti'], [3, 'O'], [1, 'Ba']]
- [ [ [1, 'Ba'], [1, 'Ti'] ], [ [1, 'Ba'], [1, 'Ti'] ] ] will return [[2, [ [1, 'Ba'], [1, 'Ti'] ] ]]

**Parameters** **\_list** – a list of elements representing a chemical formula

**Returns** a list of length-2 lists of the form [ multiplicity , element ]

`aiida.orm.data.structure.has_ase()`

**Returns** True if the ase module can be imported, False otherwise.

`aiida.orm.data.structure.has_pymatgen()`

**Returns** True if the pymatgen module can be imported, False otherwise.

`aiida.orm.data.structure.has_pyspglib()`

**Returns** True if the pyspglib module can be imported, False otherwise.

`aiida.orm.data.structure.has_vacancies(weights)`

Returns True if the sum of the weights is less than one. It uses the internal variable `_sum_threshold` as a threshold. :param weights: the weights :return: a boolean

`aiida.orm.data.structure.is_ase_atoms(ase_atoms)`

Check if the `ase_atoms` parameter is actually a `ase.Atoms` object.

**Parameters** `ase_atoms` – an object, expected to be an `ase.Atoms`.

**Returns** a boolean.

Requires the ability to import `ase`, by doing ‘import `ase`’.

`aiida.orm.data.structure.is_valid_symbol(symbol)`

Validates the chemical symbol name.

**Returns** True if the symbol is a valid chemical symbol (with correct capitalization), False otherwise.

Recognized symbols are for elements from hydrogen (Z=1) to lawrencium (Z=103).

`aiida.orm.data.structure.symop_fract_from_ortho(cell)`

Creates a matrix for conversion from fractional to orthogonal coordinates.

Taken from `svn://www.crystallography.net/cod-tools/trunk/lib/perl5/Fractional.pm`, revision 850.

**Parameters** `cell` – array of cell parameters (three lengths and three angles)

`aiida.orm.data.structure.symop_ortho_from_fract(cell)`

Creates a matrix for conversion from orthogonal to fractional coordinates.

Taken from `svn://www.crystallography.net/cod-tools/trunk/lib/perl5/Fractional.pm`, revision 850.

**Parameters** `cell` – array of cell parameters (three lengths and three angles)

`aiida.orm.data.structure.validate_symbols_tuple(symbols_tuple)`

Used to validate whether the chemical species are valid.

**Parameters** `symbols_tuple` – a tuple (or list) with the chemical symbols name.

**Raises** `ValueError` if any symbol in the tuple is not a valid chemical symbols (with correct capitalization).

Refer also to the documentation of `:func:is_valid_symbol`

`aiida.orm.data.structure.validate_weights_tuple(weights_tuple, threshold)`

Validates the weight of the atomic kinds.

**Raise** `ValueError` if the `weights_tuple` is not valid.

**Parameters**

- **weights\_tuple** – the tuple to validate. It must be a tuple of floats (as created by `:func:_create_weights_tuple`).
- **threshold** – a float number used as a threshold to check that the sum of the weights is  $\leq 1$ .

If the sum is less than one, it means that there are vacancies. Each element of the list must be  $\geq 0$ , and the sum must be  $\leq 1$ .

## Folder

**class** `aiida.orm.data.folder.FolderData(**kwargs)`

Stores a folder with subfolders and files.

No special attributes are set.

**get\_file\_content(path)**

Return the content of a path stored inside the folder as a string.

**Raises `NotExist`** if the path does not exist.

**`replace_with_folder`** (*folder*, *overwrite=True*)

Replace the data with another folder, always copying and not moving the original files.

**Args:** *folder*: the folder to copy from *overwrite*: if to overwrite the current content or not

## Singlefile

Implement subclass for a single file in the permanent repository files = [one\_single\_file] jsons = { }

methods: \* `get_content` \* `get_path` \* `get_aiaurl` (?) \* `get_md5` \* ...

To discuss: do we also need a simple directory class for full directories in the perm repo?

**class** `aiida.orm.data.singlefile.SinglefileData` (\*\*kwargs)

Pass as input a file parameter with the (absolute) path of a file on the hard drive. It will get copied inside the node.

Internally must have a single file, and stores as internal attribute the filename in the 'filename' attribute.

**`add_path`** (*src\_abs*, *dst\_filename=None*)

Add a single file

**`del_file`** (*filename*)

Remove a file from SingleFileData :param filename: name of the file stored in the DB

**`filename`**

Returns the name of the file stored

**`get_file_abs_path`** ()

Return the absolute path to the file in the repository

**`set_file`** (*filename*)

Add a file to the singlefiledata :param filename: absolute path to the file

## Upf

This module manages the UPF pseudopotentials in the local repository.

**class** `aiida.orm.data.upf.UpfData` (\*\*kwargs)

Function not yet documented.

**classmethod** `from_md5` (*md5*)

Return a list of all UPF pseudopotentials that match a given MD5 hash.

Note that the hash has to be stored in a `_md5` attribute, otherwise the pseudo will not be found.

**classmethod** `get_or_create` (*filename*, *use\_first=False*, *store\_upf=True*)

Pass the same parameter of the init; if a file with the same md5 is found, that UpfData is returned.

### Parameters

- **`filename`** – an absolute filename on disk
- **`use_first`** – if False (default), raise an exception if more than one potential is found. If it is True, instead, use the first available pseudopotential.
- **`store_upf`** (*bool*) – If false, the UpfData objects are not stored in the database. default=True.

**Return (upf, created)** where upf is the UpfData object, and create is either True if the object was created, or False if the object was retrieved from the DB.

**get\_upf\_family\_names** ()

Get the list of all upf family names to which the pseudo belongs

**classmethod get\_upf\_group** (*group\_name*)

Return the UpfFamily group with the given name.

**classmethod get\_upf\_groups** (*filter\_elements=None, user=None*)

Return all names of groups of type UpfFamily, possibly with some filters.

#### Parameters

- **filter\_elements** – A string or a list of strings. If present, returns only the groups that contains one Upf for every element present in the list. Default=None, meaning that all families are returned.
- **user** – if None (default), return the groups for all users. If defined, it should be either a DbUser instance, or a string for the username (that is, the user email).

**set\_file** (*filename*)

I pre-parse the file to store the attributes.

**store** (*\*args, \*\*kwargs*)

Store the node, reparsing the file so that the md5 and the element are correctly reset.

`aiida.orm.data.upf.get_pseudos_from_structure` (*structure, family\_name*)

Given a family name (a UpfFamily group in the DB) and a AiiDA structure, return a dictionary associating each kind name with its UpfData object.

#### Raises

- **MultipleObjectsError** – if more than one UPF for the same element is found in the group.
- **NotExistent** – if no UPF for an element in the group is found in the group.

`aiida.orm.data.upf.parse_upf` (*fname, check\_filename=True*)

Try to get relevant information from the UPF. For the moment, only the element name. Note that even UPF v.2 cannot be parsed with the XML minidom! (e.g. due to the & characters in the human-readable section).

If `check_filename` is `True`, raise a `ParsingError` exception if the filename does not start with the element name.

`aiida.orm.data.upf.upload_upf_family` (*folder, group\_name, group\_description, stop\_if\_existing=True*)

Upload a set of UPF files in a given group.

#### Parameters

- **folder** – a path containing all UPF files to be added. Only files ending in `.UPF` (case-insensitive) are considered.
- **group\_name** – the name of the group to create. If it exists and is non-empty, a `UniquenessError` is raised.
- **group\_description** – a string to be set as the group description. Overwrites previous descriptions, if the group was existing.
- **stop\_if\_existing** – if `True`, check for the md5 of the files and, if the file already exists in the DB, raises a `MultipleObjectsError`. If `False`, simply adds the existing `UPFData` node to the group.

## Cif

**class** `aiida.orm.data.cif.CifData` (*\*\*kwargs*)  
 Wrapper for Crystallographic Interchange File (CIF)

---

**Note:** the file (physical) is held as the authoritative source of information, so all conversions are done through the physical file: when setting `ase` or `values`, a physical CIF file is generated first, the values are updated from the physical CIF file.

---

**ase**  
 ASE object, representing the CIF.

---

**Note:** requires ASE module.

---

**classmethod** `from_md5` (*md5*)  
 Return a list of all CIF files that match a given MD5 hash.

---

**Note:** the hash has to be stored in a `_md5` attribute, otherwise the CIF file will not be found.

---

**generate\_md5** ()  
 Generate MD5 hash of the file's contents on-the-fly.

**get\_ase** (*\*\*kwargs*)  
 Returns ASE object, representing the CIF. This function differs from the property `ase` by the possibility to pass the keyworded arguments (*kwargs*) to `ase.io.cif.read_cif()`.

---

**Note:** requires ASE module.

---

**get\_formulae** (*mode='sum'*)  
 Get the formula.

**classmethod** `get_or_create` (*filename, use\_first=False, store\_cif=True*)  
 Pass the same parameter of the init; if a file with the same md5 is found, that CifData is returned.

### Parameters

- **filename** – an absolute filename on disk
- **use\_first** – if False (default), raise an exception if more than one CIF file is found. If it is True, instead, use the first available CIF file.
- **store\_cif** (*bool*) – If false, the CifData objects are not stored in the database. default=True.

**Return (cif, created)** where `cif` is the CifData object, and `create` is either True if the object was created, or False if the object was retrieved from the DB.

**get\_spacegroup\_numbers** ()  
 Get the spacegroup international number.

**has\_attached\_hydrogens** ()  
 Check if there are hydrogens without coordinates, specified as attached to the atoms of the structure.  
 :return: True if there are attached hydrogens, False otherwise.

**has\_partial\_occupancies** ()  
 Check if there are float values in the atom occupancies. :return: True if there are partial occupancies, False otherwise.

**set\_file** (*filename*)

Set the file. If the source is set and the MD5 checksum of new file is different from the source, the source has to be deleted.

**store** (*\*args, \*\*kwargs*)

Store the node.

**values**

PyCifRW structure, representing the CIF datablocks.

---

**Note:** requires PyCifRW module.

---

`aiida.orm.data.cif.cif_from_ase` (*ase, full\_occupancies=False, add\_fake\_biso=False*)

Construct a CIF datablock from the ASE structure. The code is taken from [https://wiki.fysik.dtu.dk/ase/epydoc/ase.io.cif-pysrc.html#write\\_cif](https://wiki.fysik.dtu.dk/ase/epydoc/ase.io.cif-pysrc.html#write_cif), as the original ASE code contains a bug in printing the Hermann-Mauguin symmetry space group symbol.

**Parameters** *ase* – ASE “images”

**Returns** array of CIF datablocks

`aiida.orm.data.cif.has_pycifrw` ()

**Returns** True if the PyCifRW module can be imported, False otherwise.

`aiida.orm.data.cif.parse_formula` (*formula*)

Parses the Hill formulae, written with spaces for separators.

`aiida.orm.data.cif.pycifrw_from_cif` (*datablocks, loops={}, names=None*)

Constructs PyCifRW’s CifFile from an array of CIF datablocks.

**Parameters**

- **datablocks** – an array of CIF datablocks
- **loops** – optional list of lists of CIF tag loops.
- **names** – optional list of datablock names

**Returns** CifFile

`aiida.orm.data.cif.symop_string_from_symop_matrix_tr` (*matrix, tr=[0, 0, 0], eps=0*)

Construct a CIF representation of symmetry operator plus translation. See International Tables for Crystallography Vol. A. (2002) for definition.

**Parameters**

- **matrix** – 3x3 matrix, representing the symmetry operator
- **tr** – translation vector of length 3 (default [0, 0, 0])
- **eps** – epsilon parameter for fuzzy comparison  $x == 0$

**Returns** CIF representation of symmetry operator

## Parameter

**class** `aiida.orm.data.parameter.ParameterData` (*\*\*kwargs*)

Pass as input in the init a dictionary, and it will get stored as internal attributes.

Usual rules for attribute names apply (in particular, keys cannot start with an underscore). If this is the case, a `ValueError` will be raised.

You can then change/delete/add more attributes before storing with the usual methods of `aiida.orm.Node`

**dict**

To be used to get direct access to the underlying dictionary with the syntax `node.dict.key` or `node.dict['key']`.

**Returns** an instance of the `AttributeResultManager`.

**get\_dict()**

Return a dict with the parameters

**keys()**

Iterator of valid keys stored in the `ParameterData` object

**set\_dict(dict)**

Replace the current dictionary with another one.

**Parameters dict** – The dictionary to set.

**update\_dict(dict)**

Update the current dictionary with the keys provided in the dictionary.

**Parameters dict** – a dictionary with the keys to substitute. It works like `dict.update()`, adding new keys and overwriting existing keys.

**Remote**

**class** `aiida.orm.data.remote.RemoteData` (*\*\*kwargs*)

Store a link to a file or folder on a remote machine.

Remember to pass a computer!

**add\_path** (*src\_abs, dst\_filename=None*)

Disable adding files or directories to a `RemoteData`

**is\_empty()**

Check if remote folder is empty

**ArrayData**

**class** `aiida.orm.data.array.ArrayData` (*\*args, \*\*kwargs*)

Store a set of arrays on disk (rather than on the database) in an efficient way using `numpy.save()` (therefore, this class requires `numpy` to be installed).

Each array is stored within the Node folder as a different `.npy` file.

**Note** Before storing, no caching is done: if you perform a `get_array()` call, the array will be re-read from disk. If instead the `ArrayData` node has already been stored, the array is cached in memory after the first read, and the cached array is used thereafter. If too much RAM memory is used, you can clear the cache with the `clear_internal_cache()` method.

**arraynames()**

Return a list of all arrays stored in the node, listing the files (and not relying on the properties).

**clear\_internal\_cache()**

Clear the internal memory cache where the arrays are stored after being read from disk (used in order to reduce at minimum the readings from disk). This function is useful if you want to keep the node in memory, but you do not want to waste memory to cache the arrays in RAM.

**delete\_array(name)**

Delete an array from the node. Can only be called before storing.

**Parameters name** – The name of the array to delete from the node.

**get\_array** (*name*)

Return an array stored in the node

**Parameters** **name** – The name of the array to return.

**get\_shape** (*name*)

Return the shape of an array (read from the value cached in the properties for efficiency reasons).

**Parameters** **name** – The name of the array.

**iterarrays** ()

Iterator that returns tuples (name, array) for each array stored in the node.

**set\_array** (*name, array*)

Store a new numpy array inside the node. Possibly overwrite the array if it already existed.

Internally, it stores a name.npy file in numpy format.

**Parameters**

- **name** – The name of the array.
- **array** – The numpy array to store.

## ArrayData subclasses

The following are Data classes inheriting from ArrayData.

**KpointsData** This module defines the classes related to band structures or dispersions in a Brillouin zone, and how to operate on them.

**class** `aiida.orm.data.array.kpoints.KpointsData` (*\*args, \*\*kwargs*)

Class to handle array of kpoints in the Brillouin zone. Provide methods to generate either user-defined k-points or path of k-points along symmetry lines. Internally, all k-points are defined in terms of crystal (fractional) coordinates. Cell and lattice vector coordinates are in Angstroms, reciprocal lattice vectors in Angstrom<sup>-1</sup>. :note: The methods setting and using the Bravais lattice info assume the PRIMITIVE unit cell is provided in input to the `set_cell` or `set_cell_from_structure` methods.

**cell**

The crystal unit cell. Rows are the crystal vectors in Angstroms. :return: a 3x3 numpy.array

**get\_kpoints** (*also\_weights=False, cartesian=False*)

Return the list of kpoints

**Parameters**

- **also\_weights** – if True, returns also the list of weights. Default = False
- **cartesian** – if True, returns points in cartesian coordinates, otherwise, returns in crystal coordinates. Default = False.

**get\_kpoints\_mesh** (*print\_list=False*)

Get the mesh of kpoints.

**Parameters** **print\_list** – default=False. If True, prints the mesh of kpoints as a list

**Raises** **AttributeError** if no mesh has been set

**Return** **mesh,offset** (if `print_list=False`) a list of 3 integers and a list of three floats  $0 < x < 1$ , representing the mesh and the offset of kpoints



**Return kpoints** (if `print_list = True`) an explicit list of kpoints coordinates, similar to what returned by `get_kpoints()`

**labels**

Labels associated with the list of kpoints. List of tuples with kpoint index and kpoint name: `[(0,'G'),(13,'M'),...]`

**pbcs**

The periodic boundary conditions along the vectors `a1,a2,a3`.

**Returns** a tuple of three booleans, each one tells if there are periodic boundary conditions for the *i*-th real-space direction (*i*=1,2,3)

**set\_cell** (*cell*, *pbcs=None*)

Set a cell to be used for symmetry analysis. To set a cell from an AiiDA structure, use “`set_cell_from_structure`”.

**Parameters**

- **cell** – 3x3 matrix of cell vectors. Orientation: each row represent a lattice vector. Units are Angstroms.
- **pbcs** – list of 3 booleans, True if in the *n*th crystal direction the structure is periodic. Default = [True,True,True]

**set\_cell\_from\_structure** (*structuredata*)

Set a cell to be used for symmetry analysis from an AiiDA structure. Inherits both the cell and the pbcs. To set manually a cell, use “`set_cell`”

**Parameters** **structuredata** – an instance of `StructureData`

**set\_kpoints** (*kpoints*, *cartesian=False*, *labels=None*, *weights=None*, *fill\_values=0*)

Set the list of kpoints. If a mesh has already been stored, raise a `ModificationNotAllowed`

**Parameters**

- **kpoints** – a list of kpoints, each kpoint being a list of one, two or three coordinates, depending on `self.pbcs`: if structure is 1D (only one True in `self.pbcs`) one allows singletons or scalars for each k-point, if it's 2D it can be a length-2 list, and in all cases it can be a length-3 list. Examples:
  - `[[0.,0.,0.],[0.1,0.1,0.1],...]` for 1D, 2D or 3D
  - `[[0.,0.],[0.1,0.1],...]` for 1D or 2D
  - `[[0.],[0.1],...]` for 1D
  - `[0., 0.1, ...]` for 1D (list of scalars)

For 0D (all pbcs are False), the list can be any of the above or empty - then only Gamma point is set. The value of *k* for the non-periodic dimension(s) is set by `fill_values`

- **cartesian** – if True, the coordinates given in input are treated as in cartesian units. If False, the coordinates are crystal, i.e. in units of `b1,b2,b3`. Default = False
- **labels** – optional, the list of labels to be set for some of the kpoints. See labels for more info
- **weights** – optional, a list of floats with the weight associated to the kpoint list
- **fill\_values** – scalar to be set to all non-periodic dimensions (indicated by False in `self.pbcs`), or list of values for each of the non-periodic dimensions.

**set\_kpoints\_mesh** (*mesh*, *offset*=[0.0, 0.0, 0.0])

Set KpointsData to represent a uniformly spaced mesh of kpoints in the Brillouin zone. This excludes the possibility of set/get kpoints

#### Parameters

- **mesh** – a list of three integers, representing the size of the kpoint mesh along b1,b2,b3.
- **offset** (*optional*) – a list of three floats between 0 and 1. [0.,0.,0.] is Gamma centered mesh [0.5,0.5,0.5] is half shifted [1.,1.,1.] by periodicity should be equivalent to [0.,0.,0.] Default = [0.,0.,0.].

### TrajectoryData

**class** aiida.orm.data.array.trajectory.**TrajectoryData** (\*args, \*\*kwargs)

Stores a trajectory (a sequence of crystal structures with timestamps, and possibly with velocities).

**get\_cells** ()

Return the array of cells, if it has already been set.

**Raises KeyError** if the trajectory has not been set yet.

**get\_positions** ()

Return the array of positions, if it has already been set.

**Raises KeyError** if the trajectory has not been set yet.

**get\_step\_data** (*index*)

Return a tuple with all information concerning the step with given index (0 is the first step, 1 the second step and so on). If you know only the step value, use the `get_step_index()` method to get the corresponding index.

If no velocities were specified, None is returned as the last element.

**Returns** A tuple in the format (step, time, cell, symbols, positions, velocities), where step is an integer, time is a float, cell is a  $3 \times 3$  matrix, symbols is an array of length n, positions is a  $n \times 3$  array, and velocities is either None or a  $n \times 3$  array

**Parameters index** – The index of the step that you want to retrieve, from 0 to `self.numsteps - 1`.

#### Raises

- **IndexError** – if you require an index beyond the limits.
- **KeyError** – if you did not store the trajectory yet.

**get\_step\_index** (*step*)

Given a value for the step (i.e., a value among those of the `steps` array), return the array index of that step, that can be used in other methods such as `get_step_data()` or `step_to_structure()`.

---

**Note:** Note that this function returns the first index found (i.e. if multiple steps are present with the same value, only the index of the first one is returned).

---

**Raises ValueError** if no step with the given value is found.

**get\_steps** ()

Return the array of steps, if it has already been set.

**Raises KeyError** if the trajectory has not been set yet.

**get\_symbols()**

Return the array of symbols, if it has already been set.

**Raises KeyError** if the trajectory has not been set yet.

**get\_times()**

Return the array of times (in ps), if it has already been set.

**Raises KeyError** if the trajectory has not been set yet.

**get\_velocities()**

Return the array of velocities, if it has already been set.

---

**Note:** This function (differently from all other `get_*` functions, will not raise an exception if the velocities are not set, but rather return `None` (both if no trajectory was not set yet, and if it the trajectory was set but no velocities were specified).

---

**numsites**

Return the number of stored sites, or zero if nothing has been stored yet.

**numsteps**

Return the number of stored steps, or zero if nothing has been stored yet.

**set\_structurelist(*structurelist*)**

Create trajectory from the list of `aiida.orm.data.structure.StructureData` instances.

**Parameters structurelist** – a list of `aiida.orm.data.structure.StructureData` instances.

**Raises ValueError** if symbol lists of supplied structures are different

**set\_trajectory(*steps, cells, symbols, positions, times=None, velocities=None*)**

Store the whole trajectory, after checking that types and dimensions are correct. Velocities are optional, if they are not passed, nothing is stored.

**Parameters**

- **steps** – integer array with dimension  $s$ , where  $s$  is the number of steps. Typically represents an internal counter within the code. For instance, if you want to store a trajectory with one step every 10, starting from step 65, the array will be `[65, 75, 85, ...]`. No checks are done on duplicate elements or on the ordering, but anyway this array should be sorted in ascending order, without duplicate elements. If your code does not provide an internal counter, just provide for instance `arange(s)`.
- **cells** – float array with dimension  $s \times 3 \times 3$ , where  $s$  is the length of the `steps` array. Units are angstrom. In particular, `cells[i, j, k]` is the  $k$ -th component of the  $j$ -th cell vector at the time step with index  $i$  (identified by step number `step[i]` and with timestamp `times[i]`).
- **symbols** – string array with dimension  $n$ , where  $n$  is the number of atoms (i.e., sites) in the structure. The same array is used for each step. Normally, the string should be a valid chemical symbol, but actually any unique string works and can be used as the name of the atomic kind (see also the `step_to_structure()` method).
- **positions** – float array with dimension  $s \times n \times 3$ , where  $s$  is the length of the `steps` array and  $n$  is the length of the `symbols` array. Units are angstrom. In particular, `positions[i, j, k]` is the  $k$ -th component of the  $j$ -th atom (or site) in the structure at the time step with index  $i$  (identified by step number `step[i]` and with timestamp `times[i]`).

- **times** – if specified, float array with dimension  $s$ , where  $s$  is the length of the `steps` array. Contains the timestamp of each step in picoseconds (ps).
- **velocities** – if specified, must be a float array with the same dimensions of the `positions` array. The array contains the velocities in the atoms.

---

**Todo**

Choose suitable units for velocities

---

**step\_to\_structure** (*index*, *custom\_kinds=None*)

Return an AiiDA `aiida.orm.data.structure.StructureData` node (not stored yet!) with the coordinates of the given step, identified by its index. If you know only the step value, use the `get_step_index()` method to get the corresponding index.

---

**Note:** The periodic boundary conditions are always set to True.

---

**Parameters**

- **index** – The index of the step that you want to retrieve, from 0 to `self.numsteps-1`.
- **custom\_kinds** – (Optional) If passed must be a list of `aiida.orm.data.structure.Kind` objects. There must be one kind object for each different string in the `symbols` array, with `kind.name` set to this string. If this parameter is omitted, the automatic kind generation of AiiDA `aiida.orm.data.structure.StructureData` nodes is used, meaning that the strings in the `symbols` array must be valid chemical symbols.

## 4.1.9 ORM documentation: Calculations

**class** `aiida.orm.calculation.Calculation` (*\*\*kwargs*)

This class provides the definition of an “abstract” AiiDA calculation. A calculation in this sense is any computation that converts data into data.

You will typically use one of its subclasses, often a `JobCalculation` for calculations run via a scheduler.

**get\_code** ()

Return the code for this calculation, or None if the code was not set.

**get\_linkname** (*link*, *\*args*, *\*\*kwargs*)

Return the linkname used for a given input link

Pass as parameter “NAME” if you would call the `use_NAME` method. If the `use_NAME` method requires a further parameter, pass that parameter as the second parameter.

**logger**

Get the logger of the `Calculation` object, so that it also logs to the DB.

**Returns** `LoggerAdapter` object, that works like a logger, but also has the ‘extra’ embedded

**class** `aiida.orm.calculation.inline.InlineCalculation` (*\*\*kwargs*)

Subclass used for calculations that are automatically generated using the `make_inline` wrapper/decorator.

This is used to automatically create a calculation node for a simple calculation

**get\_function\_name** ()

Get the function name.

**Returns** a string

`aiida.orm.calculation.inline.make_inline(func)`

This `make_inline` wrapper/decorator takes a function with specific requirements, runs it and stores the result as an `InlineCalculation` node. It will also store all other nodes, including any possibly unstored input node! The return value of the wrapped calculation will also be slightly changed, see below.

The wrapper:

- checks that the function name ends with the string `'_inline'`
- checks that each input parameter is a valid Data node (can be stored or unstored)
- runs the actual function
- gets the result values
- checks that the result value is a dictionary, where the key are all strings and the values are all **unstored** data nodes
- creates an `InlineCalculation` node, links all the kwargs as inputs and the returned nodes as outputs, using the keys as link labels
- stores all the nodes (including, possibly, unstored input nodes given as kwargs)
- returns a length-two tuple, where the first element is the `InlineCalculation` node, and the second is the dictionary returned by the wrapped function

To use this function, you can use it as a decorator of a wrapped function:

```
@make_inline
def copy_inline(source):
    return {copy: source.copy() }
```

In this way, every time you call `copy_inline`, the wrapped version is actually called, and the return value will be a tuple with the `InlineCalculation` instance, and the returned dictionary. For instance, if `s` is a valid Data node, with the following lines:

```
c, s_copy_dict = copy_inline(source=s)
s_copy = s_copy_dict['copy']
```

`c` will contain the new `InlineCalculation` instance, `s_copy` the (stored) copy of `s` (with the side effect that, if `s` was not stored, after the function call it will be automatically stored).

**Note** If you use a wrapper, make sure to write explicitly in the docstrings that the function is going to store the nodes.

The second possibility, if you want that by default the function does not store anything, but can be wrapped when it is necessary, is the following. You simply define the function you want to wrap (`copy_inline` in the example above) without decorator:

```
def copy_inline(source):
    return {copy: source.copy() }
```

This is a normal function, so to call it you will normally do:

```
s_copy_dict = copy_inline(s)
```

while if you want to wrap it, so that an `InlineCalculation` is created, and everything is stored, you will run:

```
c, s_copy_dict = make_inline(f)(s=s)
```

Note that, with the wrapper, all the parameters to `f()` have to be passed as keyworded arguments. Moreover, the return value is different, i.e. `(c, s_copy_dict)` instead of simply `s_copy_dict`.

---

**Note:** EXTREMELY IMPORTANT! The wrapped function MUST have the following requirements in order to be reproducible. These requirements cannot be enforced, but must be followed when writing the wrapped function.

- The function MUST NOT USE information that is not passed in the kwargs. In particular, it cannot read files from the hard-drive (that will not be present in another user's computer), it cannot connect to external databases and retrieve the current entries in that database (that could change over time), etc.
- The only exception to the above rule is the access to the AiiDA database for the *parents* of the input nodes. That is, you can take the input nodes passed as kwargs, and use also the data given in their inputs, the inputs of their inputs, ... but you CANNOT use any output of any of the above-mentioned nodes (that could change over time).
- The function MUST NOT have side effects (creating files on the disk, adding entries to an external database, ...).

---

**Note:** The function will also store:

- the source of the function in an attribute "source\_code", and the first line at which the function appears (attribute "first\_line\_source\_code"), as returned by `inspect.getsourcelines`;
- the full source file in "source\_file", if it is possible to retrieve it (this will be set to `None` otherwise, e.g. if the function was defined in the interactive shell).

For this reason, try to keep, if possible, all the code to be run within the same file, so that it is possible to keep the provenance of the functions that were run (if you instead call a function in a different file, you will never know in the future what that function did). If you call external modules and you matter about provenance, it would be good to also return in a suitable dictionary the version of these modules (e.g., after importing a module `XXX`, you can check if the module defines a variable `XXX.__version__` or `XXX.VERSION` or something similar, and store it in an output node).

---

**Todo** For the time being, I am storing the function source code and the full source code file in the attributes of the calculation. To be moved to an input Code node!

**Note** All nodes will be stored, including unstored input nodes!!

**Parameters** **kwargs** – all kwargs are passed to the wrapped function

**Returns** a length-two tuple, where the first element is the `InlineCalculation` node, and the second is the dictionary returned by the wrapped function. All nodes are stored.

**Raises**

- **TypeError** – if the return value is not a dictionary, the keys are not strings, or the values are not data nodes. Raise also if the input values are not data nodes.
- **ModificationNotAllowed** – if the returned Data nodes are already stored.
- **Exception** – All other exceptions from the wrapped function are not caught.

`aiida.orm.calculation.inline.optional_inline(func)`

`optional_inline` wrapper/decorator takes a function, which can be called either as wrapped in `InlineCalculation` or a simple function, depending on 'store' keyworded argument (`True` stands for `InlineCalculation`, `False` for simple function). The wrapped function has to adhere to the requirements by `make_inline` wrapper/decorator.

Usage example:

```
@optional_inline
def copy_inline(source=None):
    return {'copy': source.copy() }
```

Function `copy_inline` will be wrapped in `InlineCalculation` when invoked in following way:

```
copy_inline(source=node, store=True)
```

while it will be called as a simple function when invoked:

```
copy_inline(source=node)
```

In any way the `copy_inline` will return the same results.

**class** `aiida.orm.calculation.job.CalculationResultManager` (*calc*)

An object used internally to interface the calculation object with the Parser and consequentially with the ParameterData object result. It shouldn't be used explicitly by a user.

`__init__` (*calc*)

**Parameters** *calc* – the calculation object.

**class** `aiida.orm.calculation.job.JobCalculation` (\*\*kwargs)

This class provides the definition of an AiiDA calculation that is run remotely on a job scheduler.

`get_append_text` ()

Get the calculation-specific append text, which is going to be appended in the scheduler-job script, just after the code execution.

`get_custom_scheduler_commands` ()

Return a (possibly multiline) string with the commands that the user wants to manually set for the scheduler. See also the documentation of the corresponding `set_` method.

**Returns** the custom scheduler command, or an empty string if no custom command was defined.

`get_environment_variables` ()

Return a dictionary of the environment variables that are set for this calculation.

Return an empty dictionary if no special environment variables have to be set for this calculation.

`get_import_sys_environment` ()

To check if it's loading the system environment on the submission script.

**Returns** a boolean. If True the system environment will be load.

`get_job_id` ()

Get the scheduler job id of the calculation.

**Returns** a string

`get_max_memory_kb` ()

Get the memory (in KiloBytes) requested to the scheduler.

**Returns** an integer

`get_max_wallclock_seconds` ()

Get the max wallclock time in seconds requested to the scheduler.

**Returns** an integer

`get_mpirun_extra_params` ()

Return a list of strings, that are the extra params to pass to the mpirun (or equivalent) command after the one provided in `computer.mpirun_command`. Example: `mpirun -np 8 extra_params[0] extra_params[1] ... exec.x`

Return an empty list if no parameters have been defined.

**get\_parser\_name()**

Return a string locating the module that contains the output parser of this calculation, that will be searched in the 'aiida/parsers/plugins' directory. None if no parser is needed/set.

**Returns** a string.

**get\_parserclass()**

Return the output parser object for this calculation, or None if no parser is set.

**Returns** a Parser class.

**Raise** MissingPluginError from ParserFactory no plugin is found.

**get\_prepend\_text()**

Get the calculation-specific prepend text, which is going to be prepended in the scheduler-job script, just before the code execution.

**get\_priority()**

Get the priority, if set, of the job on the cluster.

**Returns** a string or None

**get\_queue\_name()**

Get the name of the queue on cluster.

**Returns** a string or None.

**get\_resources(full=False)**

Returns the dictionary of the job resources set.

**Parameters** **full** – if True, also add the default values, e.g.  
default\_mpiproc\_per\_machine

**Returns** a dictionary

**get\_retrieved\_node()**

Return the retrieved data folder, if present.

**Returns** the retrieved data folder object, or None if no such output node is found.

**Raises** MultipleObjectsError if more than one output node is found.

**get\_scheduler\_error()**

Return the output of the scheduler error (a string) if the calculation has finished, and output node is present, and the output of the scheduler was retrieved.

Return None otherwise.

**get\_scheduler\_output()**

Return the output of the scheduler output (a string) if the calculation has finished, and output node is present, and the output of the scheduler was retrieved.

Return None otherwise.

**get\_scheduler\_state()**

Return the status of the calculation according to the cluster scheduler.

**Returns** a string.

**get\_state(from\_attribute=False)**

Get the state of the calculation.

---

**Note:** the 'most recent' state is obtained using the logic in the `aiida.common.datastructures.sort_states` function.



---

**Todo**

Understand if the state returned when no state entry is found in the DB is the best choice.

---

**Parameters** `from_attribute` – if set to True, read it from the attributes (the attribute is also set with `set_state`, unless the state is set to IMPORTED; in this way we can also see the state before storing).

**Returns** a string. If `from_attribute` is True and no attribute is found, return None. If `from_attribute` is False and no entry is found in the DB, also return None.

**get\_withmpi ()**

Get whether the job is set with mpi execution.

**Returns** a boolean. Default=True.

**has\_failed ()**

Get whether the calculation is in a failed status, i.e. SUBMISSIONFAILED, RETRIEVALFAILED, PARSINGFAILED or FAILED.

**Returns** a boolean

**has\_finished\_ok ()**

Get whether the calculation is in the FINISHED status.

**Returns** a boolean

**kill ()**

Kill a calculation on the cluster.

Can only be called if the calculation is in status WITHSCHEDULER.

The command tries to run the kill command as provided by the scheduler, and raises an exception if something goes wrong. No changes of calculation status are done (they will be done later by the calculation manager).

**res**

To be used to get direct access to the parsed parameters.

**Returns** an instance of the CalculationResultManager.

**Note** a practical example on how it is meant to be used: let's say that there is a key 'energy' in the dictionary of the parsed results which contains a list of floats. The command `calc.res.energy` will return such a list.

**set\_append\_text (val)**

Set the calculation-specific append text, which is going to be appended in the scheduler-job script, just after the code execution.

**Parameters** `val` – a (possibly multiline) string

**set\_custom\_scheduler\_commands (val)**

Set a (possibly multiline) string with the commands that the user wants to manually set for the scheduler.

The difference of this method with respect to the `set_prepend_text` is the position in the scheduler submission file where such text is inserted: with this method, the string is inserted before any non-scheduler command.

**set\_environment\_variables (env\_vars\_dict)**

Set a dictionary of custom environment variables for this calculation.

Both keys and values must be strings.

In the remote-computer submission script, it's going to export variables as `export 'keys'='values'`

**set\_import\_sys\_environment** (*val*)

If set to true, the submission script will load the system environment variables.

**Parameters** **val** (*bool*) – load the environment if True

**set\_max\_memory\_kb** (*val*)

Set the maximum memory (in KiloBytes) to be asked to the scheduler.

**Parameters** **val** – an integer. Default=None

**set\_max\_wallclock\_seconds** (*val*)

Set the wallclock in seconds asked to the scheduler.

**Parameters** **val** – An integer. Default=None

**set\_mpirun\_extra\_params** (*extra\_params*)

Set the extra params to pass to the mpirun (or equivalent) command after the one provided in `computer.mpirun_command`. Example: `mpirun -np 8 extra_params[0] extra_params[1] ... exec.x`

**Parameters** **extra\_params** – must be a list of strings, one for each extra parameter

**set\_parser\_name** (*parser*)

Set a string for the output parser Can be None if no output plugin is available or needed.

**Parameters** **parser** – a string identifying the module of the parser. Such module must be located within the folder 'aiida/parsers/plugins'

**set\_prepend\_text** (*val*)

Set the calculation-specific prepend text, which is going to be prepended in the scheduler-job script, just before the code execution.

See also `set_custom_scheduler_commands`

**Parameters** **val** – a (possibly multiline) string

**set\_priority** (*val*)

Set the priority of the job to be queued.

**Parameters** **val** – the values of priority as accepted by the cluster scheduler.

**set\_queue\_name** (*val*)

Set the name of the queue on the remote computer.

**Parameters** **val** (*str*) – the queue name

**set\_resources** (*resources\_dict*)

Set the dictionary of resources to be used by the scheduler plugin, like the number of nodes, cpus, ... This dictionary is scheduler-plugin dependent. Look at the documentation of the scheduler. (scheduler type can be found with `calc.get_computer().get_scheduler_type()`)

**set\_withmpi** (*val*)

Set the calculation to use mpi.

**Parameters** **val** – A boolean. Default=True

**store** (*\*args, \*\*kwargs*)

Override the `store()` method to store also the calculation in the NEW state as soon as this is stored for the first time.

**submit ()**

Puts the calculation in the TOSUBMIT status.

Actual submission is performed by the daemon.

**submit\_test (folder=None, subfolder\_name=None)**

Test submission, creating the files in a local folder.

**Note** this submit\_test function does not require any node (neither the calculation nor the input links) to be stored yet.

#### Parameters

- **folder** – A Folder object, within which each calculation files are created; if not passed, a subfolder ‘submit\_test’ of the current folder is used.
- **subfolder\_name** – the name of the subfolder to use for this calculation (within Folder). If not passed, a unique string starting with the date and time in the format `yymmdd-HHMMSS-` is used.

## Quantum Espresso - PW

Plugin to create a Quantum Espresso pw.x file.

**class** `aiida.orm.calculation.job.quantumespresso.pw.PwCalculation (**kwargs)`

Main DFT code (PWscf, pw.x) of the Quantum ESPRESSO distribution. For more information, refer to <http://www.quantum-espresso.org/>

## Quantum Espresso - PW immigrant

Plugin to immigrate a Quantum Espresso pw.x job that was not run using AiiDa.

**class** `aiida.orm.calculation.job.quantumespresso.pwimmigrant.PwimmigrantCalculation (**kwargs)`

Create a PwCalculation object that can be used to import old jobs.

This is a subclass of `aiida.orm.calculation.quantumespresso.PwCalculation` with slight modifications to some of the class variables and additional methods that

1. parse the job’s input file to create the calculation’s input nodes that would exist if the calculation were submitted using AiiDa,
2. bypass the functions of the daemon, and prepare the node’s attributes such that all the processes (copying of the files to the repository, results parsing, ect.) can be performed

---

**Note:** The keyword arguments of PwCalculation are also available.

---

#### Parameters

- **remote\_workdir (str)** – Absolute path to the directory where the job was run. The transport of the computer you link ask input to the calculation is the transport that will be used to retrieve the calculation’s files. Therefore, `remote_workdir` should be the absolute path to the job’s directory on that computer.
- **input\_file\_name (str)** – The file name of the job’s input file.
- **output\_file\_name (str)** – The file name of the job’s output file (i.e. the file containing the stdout of QE).

**create\_input\_nodes** (*open\_transport*, *input\_file\_name=None*, *output\_file\_name=None*, *remote\_workdir=None*)

Create calculation input nodes based on the job's files.

**Parameters** *open\_transport* (*aiida.transport.plugins.local.LocalTransport* | *aiida.transport.plugins.ssh.SshTransport*) – An open instance of the transport class of the calculation's computer. See the tutorial for more information.

This method parses the files in the job's remote working directory to create the input nodes that would exist if the calculation were submitted using AiiDa. These nodes are

- a 'parameters' ParameterData node, based on the namelists and their variable-value pairs;
- a 'kpoints' KpointsData node, based on the *K\_POINTS* card;
- a 'structure' StructureData node, based on the *ATOMIC\_POSITIONS* and *CELL\_PARAMETERS* cards;
- one 'pseudo\_X' UpfData node for the pseudopotential used for the atomic species with name X, as specified in the *ATOMIC\_SPECIES* card;
- a 'settings' ParameterData node, if there are any fixed coordinates, or if the gamma kpoint is used;

and can be retrieved as a dictionary using the `get_inputs_dict()` method. *These input links are cached-links; nothing is stored by this method (including the calculation node itself).*

---

**Note:** QE stores the calculation's pseudopotential files in the `<outdir>/<prefix>.save/` subfolder of the job's working directory, where `outdir` and `prefix` are QE *CONTROL* variables (see [pw input file description](#)). This method uses these files to either get—if the a node already exists for the pseudo—or create a UpfData node for each pseudopotential.

---

### Keyword arguments

---

**Note:** These keyword arguments can also be set when instantiating the class or using the `set_` methods (e.g. `set_remote_workdir`). Offering to set them here simply offers the user an additional place to set their values. *Only the values that have not yet been set need to be specified.*

---

### Parameters

- **input\_file\_name** (*str*) – The file name of the job's input file.
- **output\_file\_name** (*str*) – The file name of the job's output file (i.e. the file containing the stdout of QE).
- **remote\_workdir** (*str*) – Absolute path to the directory where the job was run. The transport of the computer you link ask input to the calculation is the transport that will be used to retrieve the calculation's files. Therefore, `remote_workdir` should be the absolute path to the job's directory on that computer.

### Raises

- [`aiida.common.exceptions.InputValidationError`](#) – if `open_transport` is a different type of transport than the computer's.
- [`aiida.common.exceptions.InvalidOperation`](#) – if `open_transport` is not open.
- [`aiida.common.exceptions.InputValidationError`](#) – if `remote_workdir`, `input_file_name`, and/or `output_file_name` are not set prior to or during the call of this method.

- `aiida.common.exceptions.FeatureNotAvailable` – if the input file uses anything other than `ibrav = 0`, which is not currently implemented in aiiida.
- `aiida.common.exceptions.ParsingError` – if there are issues parsing the input file.
- `IOError` – if there are issues reading the input file.

#### **prepare\_for\_retrieval\_and\_parsing** (*open\_transport*)

Tell the daemon that the calculation is computed and ready to be parsed.

**Parameters** `open_transport` (*aiida.transport.plugins.local.LocalTransport* | *aiida.transport.plugins.ssh.SshTransport*) – An open instance of the transport class of the calculation’s computer. See the tutorial for more information.

The next time the daemon updates the status of calculations, it will see this job is in the ‘COMPUTED’ state and will retrieve its output files and parse the results.

If the daemon is not currently running, nothing will happen until it is started again.

This method also stores the calculation and all input nodes. It also copies the original input file to the calculation’s repository folder.

#### **Raises**

- `aiida.common.exceptions.InputValidationError` – if `open_transport` is a different type of transport than the computer’s.
- `aiida.common.exceptions.InvalidOperation` – if `open_transport` is not open.

#### **set\_input\_file\_name** (*input\_file\_name*)

Set the file name of the job’s input file (e.g. ‘pw.in’).

**Parameters** `input_file_name` (*str*) – The file name of the job’s input file.

#### **set\_output\_file\_name** (*output\_file\_name*)

Set the file name of the job’s output file (e.g. ‘pw.out’).

**Parameters** `output_file_name` (*str*) – The file name of file containing the job’s stdout.

#### **set\_output\_subfolder** (*output\_subfolder*)

Manually set the job’s `outdir` variable (e.g. ‘./out/’).

---

**Note:** The `outdir` variable is normally set automatically by

1. looking for the `outdir` CONTROL namelist variable
2. looking for the `$ESPRESSO_TMPDIR` environment variable on the calculation’s computer (using the transport)
3. using the QE default, the calculation’s `remote_workdir`

but this method is made available to the user, in the event that they wish to set it manually.

---

**Parameters** `output_subfolder` (*str*) – The job’s `outdir` variable.

#### **set\_prefix** (*prefix*)

Manually set the job’s `prefix` variable (e.g. ‘pwscf’).

---

**Note:** The `prefix` variable is normally set automatically by

1. looking for the `prefix` CONTROL namelist variable

2.using the QE default, 'pwsf'

but this method is made available to the user, in the event that they wish to set it manually.

---

**Parameters** `prefix` (*str*) – The job's prefix variable.

**set\_remote\_workdir** (*remote\_workdir*)

Set the job's remote working directory.

**Parameters** `remote_workdir` (*str*) – Absolute path of the job's remote working directory.

## TemplateReplacer

This is a simple plugin that takes two node inputs, both of type `ParameterData`, with the following labels: `template` and `parameters`. You can also add other `SinglefileData` nodes as input, that will be copied according to what is written in 'template' (see below).

- `parameters`: a set of parameters that will be used for substitution.
- `template`: can contain the following parameters:
  - `input_file_template`: a string with substitutions to be managed with the `format()` function of python, i.e. if you want to substitute a variable called 'varname', you write `{varname}` in the text. See <http://www.python.org/dev/peps/pep-3101/> for more details. The replaced file will be the input file.
  - `input_file_name`: a string with the file name for the input. If it is not provided, no file will be created.
  - `output_file_name`: a string with the file name for the output. If it is not provided, no redirection will be done and the output will go in the scheduler output file.
  - `cmdline_params`: a list of strings, to be passed as command line parameters. Each one is substituted with the same rule of `input_file_template`. Optional
  - `input_through_stdin`: if `True`, the input file name is passed via `stdin`. Default is `False` if missing.
  - **files\_to\_copy**: if defined, a list of tuple pairs, with format ('link\_name', 'dest\_rel\_path'); for each tuple, an input link and with file type 'Singlefile', and the content is copied to a remote file named 'dest\_rel\_path' Errors are raised in the input links are non-existent, or of the wrong type, or if there are unused input files.

TODO: probably use Python's Template strings instead?? TODO: catch exceptions

```
class aiiada.orm.calculation.job.simpleplugins.templatereplacer.TemplatereplacerCalculation (***)
    Simple stub of a plugin that can be used to replace some text in a given template. Can be used for many different
    codes, or as a starting point to develop a new plugin.
```

### 4.1.10 Calculation parsers

This section describes the different parsers classes for calculations.

#### Quantum ESPRESSO parsers

```
aiida.parsers.plugins.quantumespresso.convert_qe2aiida_structure (output_dict,
                                                                    in-
                                                                    put_structure=None)
```

Receives the dictionary cell parsed from quantum espresso Convert it into an AiiDA structure object

## Basic Raw Cp Parser

`aiida.parsers.plugins.quantumespresso.basic_raw_parser_cp.parse_cp_text_output` (*data*, *xml\_data*)  
 data must be a list of strings, one for each lines, as returned by `readlines()`. On output, a dictionary with parsed values

`aiida.parsers.plugins.quantumespresso.basic_raw_parser_cp.parse_cp_traj_stanzas` (*num\_elements*, *split\_lines*, *prepend\_name*, *rescale=1.0*)

*num\_elements*: Number of lines (with three elements) between lines with two only elements (containing step number and time in ps). *num\_elements* is 3 for cell, and the number of atoms for coordinates and positions.

*splitlines*: a list of lines of the file, already split in pieces using `string.split`

*prepend\_name*: a string to be prepended to the name of keys returned in the return dictionary.

*rescale*: the values in each stanza are multiplied by this factor, for units conversion

`aiida.parsers.plugins.quantumespresso.basic_raw_parser_cp.parse_cp_xml_counter_output` (*data*)  
 Parse xml file `print_counter.xml` data must be a single string, as returned by `file.read()` (notice the difference with `parse_text_output`!) On output, a dictionary with parsed values.

`aiida.parsers.plugins.quantumespresso.basic_raw_parser_cp.parse_cp_xml_output` (*data*)  
 Parse xml data data must be a single string, as returned by `file.read()` (notice the difference with `parse_text_output`!) On output, a dictionary with parsed values. Democratically, we have decided to use picoseconds as units of time, eV for energies, Angstrom for lengths.

## Basic Raw Pw Parser

A collection of function that are used to parse the output of Quantum Espresso PW. The function that needs to be called from outside is `parse_raw_output()`. The functions mostly work without aiida specific functionalities. The parsing will try to convert whatever it can in some dictionary, which by operative decision doesn't have much structure encoded, [the values are simple]

`aiida.parsers.plugins.quantumespresso.basic_raw_parser_pw.cell_volume` (*a1*, *a2*, *a3*)

returns the volume of the primitive cell: `la1.(a2xa3)`

`aiida.parsers.plugins.quantumespresso.basic_raw_parser_pw.convert_list_to_matrix` (*in\_matrix*, *n\_rows*, *n\_columns*)

converts a list into a list of lists (a matrix like) with *n\_rows* and *n\_columns*

`aiida.parsers.plugins.quantumespresso.basic_raw_parser_pw.convert_qe_time_to_sec` (*timestr*)  
 Given the walltime string of Quantum Espresso, converts it in a number of seconds (float).

`aiida.parsers.plugins.quantumespresso.basic_raw_parser_pw.parse_QE_errors` (*lines*, *count*, *warnings*)

Parse QE errors messages (those appearing between some lines with ' %%%%%%%%%%')

### Parameters

- **lines** – list of strings, the output text file as read by `readlines()` or as obtained by `data.split('n')` when data is the text file read by `read()`

- **count** – the line at which we identified some ' %%%%%%%%% '
- **warnings** – the warnings already parsed in the file

**Return messages** a list of QE error messages

```
aiida.parsers.plugins.quantumespresso.basic_raw_parser_pw.parse_pw_text_output (data,
                                                                                   xml_data=None,
                                                                                   struc-
                                                                                   ture_data=None,
                                                                                   in-
                                                                                   put_dict=None)
```

Parses the text output of QE-PWscf.

#### Parameters

- **data** – a string, the file as read by read()
- **xml\_data** – the dictionary with the keys read from xml.
- **structure\_data** – dictionary, coming from the xml, with info on the structure

**Return parsed\_data** dictionary with key values, referring to quantities at the last scf step.

**Return trajectory\_data** key,values referring to intermediate scf steps, as in the case of vc-relax.  
Empty dictionary if no value is present.

**Return critical\_messages** a list with critical messages. If any is found in parsed\_data['warnings'],  
the calculation is FAILED!

```
aiida.parsers.plugins.quantumespresso.basic_raw_parser_pw.parse_pw_xml_output (data,
                                                                                   dir_with_bands=None)

Parse the xml data of QE v5.0.x Input data must be a single string, as returned by file.read() Returns a dictionary
with parsed values
```

```
aiida.parsers.plugins.quantumespresso.basic_raw_parser_pw.parse_raw_output (out_file,
                                                                              in-
                                                                              put_dict,
                                                                              parser_opts=None,
                                                                              xml_file=None,
                                                                              dir_with_bands=None)

Parses the output of a calculation Receives in input the paths to the output file and the xml file.
```

#### Parameters

- **out\_file** – path to pw std output
- **input\_dict** – not used
- **parser\_opts** – not used
- **dir\_with\_bands** – path to directory with all k-points (Kxxxxx) folders
- **xml\_file** – path to QE data-file.xml

**Returns out\_dict** a dictionary with parsed data

**Return successful** a boolean that is False in case of failed calculations

#### Raises

- **QEOutputParsingError** – for errors in the parsing,
- **AssertionError** – if two keys in the parsed dicts are found to be qual

3 different keys to check in output: parser\_warnings, xml\_warnings and warnings. On an upper level, these flags MUST be checked. The first two are expected to be empty unless QE failures or unfinished jobs.



## Basic Pw Parser

**class** `aiida.parsers.plugins.quantumespresso.basicpw.BasicpwParser` (*calc*)

This class is the implementation of the Parser class for PWscf.

**get\_linkname\_out\_kpoints** ()

Returns the name of the link to the output\_kpoints Node exists if cell has changed and no bands are stored.

**get\_linkname\_outarray** ()

Returns the name of the link to the output\_array Node may exist in case of calculation='scf'

**get\_linkname\_outstructure** ()

Returns the name of the link to the output\_structure Node exists if positions or cell changed.

**get\_linkname\_outtrajectory** ()

Returns the name of the link to the output\_trajectory. Node exists in case of calculation='md', 'vc-md', 'relax', 'vc-relax'

**get\_parser\_settings\_key** ()

Return the name of the key to be used in the calculation settings, that contains the dictionary with the parser\_options

**parse\_with\_retrieved** (*retrieved*)

Receives in input a dictionary of retrieved nodes. Does all the logic here.

## Constants

Physical or mathematical constants. Since every code has its own conversion units, this module defines what QE understands as for an eV or other quantities. Whenever possible, we try to use the constants defined in :py:mod:aiida.common.constants:, but if some constants are slightly different among different codes (e.g., different standard definition), we define the constants in this file.

## Cp Parser

**class** `aiida.parsers.plugins.quantumespresso.cp.CpParser` (*calc*)

This class is the implementation of the Parser class for Cp.

**get\_linkname\_trajectory** ()

Returns the name of the link to the output\_structure (None if not present)

**parse\_with\_retrieved** (*retrieved*)

Receives in input a dictionary of retrieved nodes. Does all the logic here.

## 4.1.11 QueryTool documentation

This section describes the querytool class for querying nodes with an easy Python interface.

**class** `aiida.orm.querytool.QueryTool`

Class to make easy queries without extensive knowledge of SQL, Django and/or the internal storage mechanism of AiiDA.

---

**Note:** This feature is under constant development, so the name of the methods may change in future versions to allow for increased querying capabilities.

---



---

**Todo**

---

missing features:

- add `__in` filter
- allow `__in` filter to accept other querytool objects to perform a single query
- implement searches through the TC table
- document the methods
- allow to get attributes of queried data via a single query with suitable methods
- add checks to verify whether filters as `<=`, `==`, etc are valid for the specified data type (e.g., `__gt` only with numbers and dates, ...)
- probably many other things...

---

**add\_attr\_filter** (*key, filtername, value, negate=False, relnode=None, relnodeclass=None*)

Add a new filter on the value of attributes of the nodes you want to query.

#### Parameters

- **key** – the value of the key
- **filtername** – the type of filter to apply. Multiple filters are supported (depending on the type of value), like `'<='`, `'<'`, `'>'`, `'>='`, `'='`, `'contains'`, `'iexact'`, `'startswith'`, `'endswith'`, `'istartswith'`, `'iendswith'`, ... (the prefix `'i'` means “case-insensitive”, in the case of strings).
- **value** – the value of the attribute
- **negate** – if True, add the negation of the current filter
- **relnode** – if specified, asks to apply the filter not on the node that is currently being queried, but rather on a node linked to it. Can be `“res”` for output results, `“inp.LINKNAME”` for input nodes with a given link name, `“out.LINKNAME”` for output nodes with a given link name.
- **relnodeclass** – if `relnode` is specified, you can here add a further filter on the type of linked node for which you are executing the query (e.g., if you want to filter for outputs whose `'energy'` value is lower than zero, but only if `'energy'` is in a `ParameterData` node).

**add\_extra\_filter** (*key, filtername, value, negate=False, relnode=None, relnodeclass=None*)

Add a new filter on the value of extras of the nodes you want to query.

#### Parameters

- **key** – the value of the key
- **filtername** – the type of filter to apply. Multiple filters are supported (depending on the type of value), like `'<='`, `'<'`, `'>'`, `'>='`, `'='`, `'contains'`, `'iexact'`, `'startswith'`, `'endswith'`, `'istartswith'`, `'iendswith'`, ... (the prefix `'i'` means “case-insensitive”, in the case of strings).
- **value** – the value of the extra
- **negate** – if True, add the negation of the current filter
- **relnode** – if specified, asks to apply the filter not on the node that is currently being queried, but rather on a node linked to it. Can be `“res”` for output results, `“inp.LINKNAME”` for input nodes with a given link name, `“out.LINKNAME”` for output nodes with a given link name.
- **relnodeclass** – if `relnode` is specified, you can here add a further filter on the type of linked node for which you are executing the query (e.g., if you want to filter for outputs whose `'energy'` value is lower than zero, but only if `'energy'` is in a `ParameterData` node).

**create\_attrs\_dict** ()

Return a dictionary of the raw data from the attributes associated to the queried nodes.

**create\_extras\_dict** ()

Return a dictionary of the raw data from the extras associated to the queried nodes.

**get\_attributes** ()

Get the raw values of all the attributes of the queried nodes.

**limit\_pks** (*pk\_list*)

Limit the query to a given list of pks.

**Parameters** **pk\_list** – the list of pks you want to limit your query to.

**run\_query** (*with\_data=False, order\_by=None*)

Run the query using the filters that have been pre-set on this class, and return a generator of the obtained Node (sub)classes.

**Parameters** **order\_by** – if specified, order by the given field

**set\_class** (*the\_class*)

Pass a class to filter results only of a specific Node (sub)class, and its subclasses.

**set\_group** (*group, exclude=False*)

Filter calculations only within a given node. This can be called multiple times for an AND query.

---

#### Todo

Add the possibility of specifying the group as an object rather than with its name, so that one can also query special groups, etc.

---

#### Todo

Add the possibility of specifying “OR” type queries on multiple groups, and any combination of AND, OR, NOT.

---

#### Parameters

- **group** – the name of the group
- **exclude** – if True, excude results

## 4.1.12 DbImporter documentation

### Generic database importer class

This section describes the base class for the import of data from external databases.

`aiida.tools.dbimporters.DbImporterFactory` (*pluginname*)

This function loads the correct DbImporter plugin class

**class** `aiida.tools.dbimporters.baseclasses.CifEntry` (*db\_name=None, db\_uri=None, id=None, version=None, extras={}, uri=None*)

Represents an entry from the structure database (COD, ICSD, ...).

**cif**

Returns raw contents of a CIF file as string.

**get\_aiida\_structure()**

Returns AiiDA-compatible structure, representing the crystal structure from the CIF file.

**get\_ase\_structure()**

Returns ASE representation of the CIF.

---

**Note:** To be removed, as it is duplicated in `aiida.orm.data.cif.CifData`.

---

**get\_cif\_node(store=False)**

Creates a CIF node, that can be used in AiiDA workflow.

**Returns** `aiida.orm.data.cif.CifData` object

**get\_parsed\_cif()**

Returns data structure, representing the CIF file. Can be created using PyCIFRW or any other open-source parser.

**Returns** list of lists

**get\_raw\_cif()**

Returns raw contents of a CIF file as string.

**Returns** contents of a file as string

**class** `aiida.tools.dbimporters.baseclasses.DbEntry` (*db\_name=None, db\_uri=None, id=None, version=None, extras={}, uri=None*)

Represents an entry from external database.

**contents**

Returns raw contents of a file as string.

**class** `aiida.tools.dbimporters.baseclasses.DbImporter`

Base class for database importers.

**get\_supported\_keywords()**

Returns the list of all supported query keywords.

**Returns** list of strings

**query(\*\*kwargs)**

Method to query the database.

**Parameters**

- **id** – database-specific entry identifier
- **element** – element name from periodic table of elements
- **number\_of\_elements** – number of different elements
- **mineral\_name** – name of mineral
- **chemical\_name** – chemical name of substance
- **formula** – chemical formula
- **volume** – volume of the unit cell in cubic angstroms
- **spacegroup** – symmetry space group symbol in Hermann-Mauguin notation
- **spacegroup\_hall** – symmetry space group symbol in Hall notation
- **a** – length of lattice vector in angstroms
- **b** – length of lattice vector in angstroms

- **c** – length of lattice vector in angstroms
- **alpha** – angles between lattice vectors in degrees
- **beta** – angles between lattice vectors in degrees
- **gamma** – angles between lattice vectors in degrees
- **z** – number of the formula units in the unit cell
- **measurement\_temp** – temperature in kelvins at which the unit-cell parameters were measured
- **measurement\_pressure** – pressure in kPa at which the unit-cell parameters were measured
- **diffraction\_temp** – mean temperature in kelvins at which the intensities were measured
- **diffraction\_pressure** – mean pressure in kPa at which the intensities were measured
- **authors** – authors of the publication
- **journal** – name of the journal
- **title** – title of the publication
- **year** – year of the publication
- **journal\_volume** – journal volume of the publication
- **journal\_issue** – journal issue of the publication
- **first\_page** – first page of the publication
- **last\_page** – last page of the publication
- **doi** – digital object identifier (DOI), referring to the publication

**Raises `NotImplementedError`** if search using given keyword is not implemented.

**setup\_db** (*\*\*kwargs*)

Sets the database parameters. The method should reconnect to the database using updated parameters, if already connected.

**class** `aiida.tools.dbimporters.baseclasses.DbSearchResults` (*results*)

Base class for database results.

All classes, inheriting this one and overriding `at()`, are able to benefit from having functions `__iter__`, `__len__` and `__getitem__`.

**class** `DbSearchResultsIterator` (*results, increment=1*)

Iterator for search results

`DbSearchResults.__iter__()`

Instances of `aiida.tools.dbimporters.baseclasses.DbSearchResults` can be used as iterators.

`DbSearchResults.at` (*position*)

Returns position-th result as `aiida.tools.dbimporters.baseclasses.DbEntry`.

**Parameters** *position* – zero-based index of a result.

**Raises `IndexError`** if *position* is out of bounds.

`DbSearchResults.fetch_all()`

Returns all query results as an array of `aiida.tools.dbimporters.baseclasses.DbEntry`.

`DbSearchResults.next()`

Returns the next result of the query (instance of `aiida.tools.dbimporters.baseclasses.DbEntry`).

**Raises `StopIteration`** when the end of result array is reached.

**class** `aiida.tools.dbimporters.baseclasses.UpfEntry` (`db_name=None`, `db_uri=None`,  
`id=None`, `version=None`, `extras={}`,  
`uri=None`)

Represents an entry from the pseudopotential database.

**get\_upf\_node** (`store=False`)

Creates an UPF node, that can be used in AiiDA workflow.

**Returns** `aiida.orm.data.upf.UpfData` object

## Structural databases

### COD database importer

**class** `aiida.tools.dbimporters.plugins.cod.CodDbImporter` (`**kwargs`)

Database importer for Crystallography Open Database.

**get\_supported\_keywords** ()

Returns the list of all supported query keywords.

**Returns** list of strings

**query** (`**kwargs`)

Performs a query on the COD database using keyword = value pairs, specified in kwargs.

**Returns** an instance of `aiida.tools.dbimporters.plugins.cod.CodSearchResults`.

**query\_sql** (`**kwargs`)

Forms a SQL query for querying the COD database using keyword = value pairs, specified in kwargs.

**Returns** string containing a SQL statement.

**setup\_db** (`**kwargs`)

Changes the database connection details.

**class** `aiida.tools.dbimporters.plugins.cod.CodEntry` (`uri`, `db_name='Crystallography`  
`Open` `Database'`,  
`db_uri='http://www.crystallography.net'`,  
`**kwargs`)

Represents an entry from COD.

**class** `aiida.tools.dbimporters.plugins.cod.CodSearchResults` (`results`)

Results of the search, performed on COD.

### ICSD database importer

**exception** `aiida.tools.dbimporters.plugins.icsd.CifFileErrorExp`

Raised when the author loop is missing in a CIF file.

**class** `aiida.tools.dbimporters.plugins.icsd.IcsddbImporter` (*\*\*kwargs*)

Importer for the Inorganic Crystal Structure Database, short ICSD, provided by FIZ Karlsruhe. It allows to run queries and analyse all the results. See the [DbImporter documentation and tutorial page](#) for more information.

#### Parameters

- **server** – Server URL, the web page of the database. It is required in order to have access to the full database. It should contain both the protocol and the domain name and end with a slash, as in:

```
server = "http://ICSDSERVER.com/"
```

- **urladd** – part of URL which is added between query and the server URL (default: `index.php?`). only needed for web page query
- **querydb** – boolean, decides whether the mysql database is queried (default: `True`). If `False`, the query results are obtained through the web page query, which is restricted to a maximum of 1000 results per query.
- **dl\_db** – icsd comes with a full (default: `icsd`) and a demo database (`icsdd`). This parameter allows the user to switch to the demo database for testing purposes, if the access rights to the full database are not granted.
- **host** – MySQL database host. If the MySQL database is hosted on a different machine, use “127.0.0.1” as host, and open a SSH tunnel to the host using:

```
ssh -L 3306:localhost:3306 username@hostname.com
```

or (if e.g. you get an `URLError` with `Errno 111` (Connection refused) upon querying):

```
ssh -L 3306:localhost:3306 -L 8010:localhost:80 username@hostname.com
```

- **user** – mysql database username (default: `dba`)
- **passwd** – mysql database password (default: `sql`)
- **db** – name of the database (default: `icsd`)
- **port** – Port to access the mysql database (default: `3306`)

**get\_supported\_keywords** ()

**Returns** List of all supported query keywords.

**query** (*\*\*kwargs*)

Depending on the `db_parameters`, the mysql database or the web page are queried. Valid parameters are found using `IcsddbImporter.get_supported_keywords()`.

**Parameters** *kwargs* – A list of “keyword = [values]” pairs.

**setup\_db** (*\*\*kwargs*)

Change the database connection details. At least the host server has to be defined.

**Parameters** *kwargs* – `db_parameters` for the mysql database connection (host, user, passwd, db, port)

**class** `aiida.tools.dbimporters.plugins.icsd.IcsdEntry` (*uri*, *\*\*kwargs*)

Represent an entry from Icsd.

#### Note

- Before July 2nd 2015, `source['id']` contained `icsd.IDNUM` (internal icsd id number) and `source['extras']['cif_nr']` the cif number (`icsd.COLL_CODE`).

- After July 2nd 2015, `source['id']` has been replaced by the cif number and `source['extras']['idnum']` is `icsd.IDNUM`.

**cif**

**Returns** cif file of Icsd entry.

**get\_aiida\_structure()**

**Returns** AiiDA structure corresponding to the CIF file.

**get\_ase\_structure()**

**Returns** ASE structure corresponding to the cif file.

**get\_cif\_node()**

Create a CIF node, that can be used in AiiDA workflow.

**Returns** `aiida.orm.data.cif.CifData` object

**get\_corrected\_cif()**

Add quotes to the lines in the author loop if missing.

**Note** ase raises an AssertionError if the quotes in the author loop are missing.

**class** `aiida.tools.dbimporters.plugins.icsd.IcsdSearchResults` (*query*,  
*db\_parameters*)

Result manager for the query performed on ICSD.

**Parameters**

- **query** – mysql query or webpage query
- **db\_parameters** – database parameter setup during the initialisation of the IcsdDbImporter.

**at** (*position*)

Return *position*-th result as IcsdEntry.

**next** ()

Return next result as IcsdEntry.

**query\_db\_version** ()

Query the version of the icsd database (last row of RELEASE\_TAGS).

**query\_page** ()

Query the mysql or web page database, depending on the `db_parameters`. Store the `number_of_results`, cif file number and the corresponding icsd number.

**Note** Icsd uses its own number system, different from the CIF file numbers.

**exception** `aiida.tools.dbimporters.plugins.icsd.NoResultsWebExp`

Raised when a webpage query returns no results.

`aiida.tools.dbimporters.plugins.icsd.correct_cif` (*cif*)

Correct the format of the CIF files. At the moment, it only fixes missing quotes in the authors field (`ase.read.io` only works if the author names are quoted, if not an AssertionError is raised).

**Parameters** **cif** – A string containing the content of the CIF file.

**Returns** a string containing the corrected CIF file.



**MPOD database importer**

```
class aiiida.tools.dbimporters.plugins.mpod.MpodDbImporter (**kwargs)
    Database importer for Material Properties Open Database.

    get_supported_keywords ()
        Returns the list of all supported query keywords.

        Returns list of strings

    query (**kwargs)
        Performs a query on the MPOD database using keyword = value pairs, specified in kwargs.

        Returns an instance of aiida.tools.dbimporters.plugins.mpod.MpodSearchResults.

    query_get (**kwargs)
        Forms a HTTP GET query for querying the MPOD database. May return more than one query in case an
        intersection is needed.

        Returns a list containing strings for HTTP GET statement.

    setup_db (query_url=None, **kwargs)
        Changes the database connection details.

class aiiida.tools.dbimporters.plugins.mpod.MpodEntry (uri, **kwargs)
    Represents an entry from MPOD.

class aiiida.tools.dbimporters.plugins.mpod.MpodSearchResults (results)
    Results of the search, performed on MPOD.
```

**OQMD database importer**

```
class aiiida.tools.dbimporters.plugins.oqmd.OqmdDbImporter (**kwargs)
    Database importer for Open Quantum Materials Database.

    get_supported_keywords ()
        Returns the list of all supported query keywords.

        Returns list of strings

    query (**kwargs)
        Performs a query on the OQMD database using keyword = value pairs, specified in kwargs.

        Returns an instance of aiida.tools.dbimporters.plugins.oqmd.OqmdSearchResults.

    query_get (**kwargs)
        Forms a HTTP GET query for querying the OQMD database.

        Returns a strings for HTTP GET statement.

    setup_db (query_url=None, **kwargs)
        Changes the database connection details.

class aiiida.tools.dbimporters.plugins.oqmd.OqmdEntry (uri, **kwargs)
    Represents an entry from OQMD.

class aiiida.tools.dbimporters.plugins.oqmd.OqmdSearchResults (results)
    Results of the search, performed on OQMD.
```

### PCOD database importer

```
class aiiida.tools.dbimporters.plugins.pcod.PcodDbImporter (**kwargs)
    Database importer for Predicted Crystallography Open Database.

    query (**kwargs)
        Performs a query on the PCOD database using keyword = value pairs, specified in kwargs.

        Returns an instance of aiida.tools.dbimporters.plugins.pcod.PcodSearchResults.

    query_sql (**kwargs)
        Forms a SQL query for querying the PCOD database using keyword = value pairs, specified in
        kwargs.

        Returns string containing a SQL statement.

class aiiida.tools.dbimporters.plugins.pcod.PcodEntry (uri, db_name='Predicted Crys-
    tallography Open Database',
    db_uri='http://www.crystallography.net/pcod',
    **kwargs)

    Represents an entry from PCOD.

class aiiida.tools.dbimporters.plugins.pcod.PcodSearchResults (results)
    Results of the search, performed on PCOD.
```

### TCOD database importer

```
class aiiida.tools.dbimporters.plugins.tcod.TcodDbImporter (**kwargs)
    Database importer for Theoretical Crystallography Open Database.

    query (**kwargs)
        Performs a query on the TCOD database using keyword = value pairs, specified in kwargs.

        Returns an instance of aiida.tools.dbimporters.plugins.tcod.TcodSearchResults.

class aiiida.tools.dbimporters.plugins.tcod.TcodEntry (uri, db_name='Theoretical
    Crystallography Open Database',
    db_uri='http://www.crystallography.net/tcod',
    **kwargs)

    Represents an entry from TCOD.

class aiiida.tools.dbimporters.plugins.tcod.TcodSearchResults (results)
    Results of the search, performed on TCOD.
```

### Other databases

#### NNINC database importer

```
class aiiida.tools.dbimporters.plugins.nninc.NnincDbImporter (**kwargs)
    Database importer for NNIN/C Pseudopotential Virtual Vault.

    get_supported_keywords ()
        Returns the list of all supported query keywords.

        Returns list of strings

    query (**kwargs)
        Performs a query on the NNIN/C Pseudopotential Virtual Vault using keyword = value pairs, speci-
        fied in kwargs.
```

**Returns** an instance of `aiida.tools.dbimporters.plugins.nninc.NnincSearchResults`.

**query\_get** (*\*\*kwargs*)

Forms a HTTP GET query for querying the NNIN/C Pseudopotential Virtual Vault.

**Returns** a string with HTTP GET statement.

**setup\_db** (*query\_url=None, \*\*kwargs*)

Changes the database connection details.

**class** `aiida.tools.dbimporters.plugins.nninc.NnincEntry` (*uri, \*\*kwargs*)

Represents an entry from NNIN/C Pseudopotential Virtual Vault.

**class** `aiida.tools.dbimporters.plugins.nninc.NnincSearchResults` (*results*)

Results of the search, performed on NNIN/C Pseudopotential Virtual Vault.

### 4.1.13 DbExporter documentation

#### TCOD database exporter

`aiida.tools.dbexporters.tcod.cif_encode_contents` (*content*, *gzip=False*, *gzip\_threshold=1024*)

Encodes data for usage in CIF text field in a *best possible* way: binary data is encoded using Base64 encoding; text with non-ASCII symbols, too long lines or lines starting with semicolons (;) is encoded using Quoted-printable encoding.

**Parameters** **content** – the content to be encoded

**Return content** encoded content

**Return encoding** a string specifying used encoding (None, 'base64', 'ncr', 'quoted-printable', 'gzip+base64')

`aiida.tools.dbexporters.tcod.decode_textfield` (*content, method*)

Decodes the contents of encoded CIF textfield.

**Parameters**

- **content** – the content to be decoded
- **method** – method, which was used for encoding the contents (None, 'base64', 'ncr', 'quoted-printable', 'gzip+base64')

**Returns** decoded content

**Raises ValueError** if the encoding method is unknown

`aiida.tools.dbexporters.tcod.decode_textfield_base64` (*content*)

Decodes the contents for CIF textfield from Base64 using standard Python implementation (`base64.standard_b64decode()`)

**Parameters** **content** – a string with contents

**Returns** decoded string

`aiida.tools.dbexporters.tcod.decode_textfield_gzip_base64` (*content*)

Decodes the contents for CIF textfield from Base64 and decompresses them with gzip.

**Parameters** **content** – a string with contents

**Returns** decoded string

`aiida.tools.dbexporters.tcod.decode_textfield_ncr` (*content*)

Decodes the contents for CIF textfield from Numeric Character Reference.

**Parameters** *content* – a string with contents

**Returns** decoded string

```
aiida.tools.dbexporters.tcod.decode_textfield_quoted_printable(content)
```

Decodes the contents for CIF textfield from quoted-printable encoding.

**Parameters** *content* – a string with contents

**Returns** decoded string

```
aiida.tools.dbexporters.tcod.deposit(what, type, author_name=None, author_email=None, url=None, title=None, user_name=None, password=False, user_email=None, code_label='cif_cod_deposit', computer_name=None, replace=None, message=None, **kwargs)
```

Launches a *aiida.orm.calculation.job.JobCalculation* to deposit data node to \*COD-type database.

**Returns** launched *aiida.orm.calculation.job.JobCalculation* instance.

**Raises** **ValueError** if any of the required parameters are not given.

```
aiida.tools.dbexporters.tcod.deposition_cmdline_parameters(parser, expclass='Data')
```

Provides descriptions of command line options, that are used to control the process of deposition to TCOD.

**Parameters**

- **parser** – an argparse.Parser instance
- **expclass** – name of the exported class to be shown in help string for the command line options

---

**Note:** This method must not set any default values for command line options in order not to clash with any other data deposition plugins.

---

```
aiida.tools.dbexporters.tcod.encode_textfield_base64(content, foldwidth=76)
```

Encodes the contents for CIF textfield in Base64 using standard Python implementation (`base64.standard_b64encode()`).

**Parameters**

- **content** – a string with contents
- **foldwidth** – maximum width of line (default is 76)

**Returns** encoded string

```
aiida.tools.dbexporters.tcod.encode_textfield_gzip_base64(content, **kwargs)
```

Gzips the given string and encodes it in Base64.

**Parameters** *content* – a string with contents

**Returns** encoded string

```
aiida.tools.dbexporters.tcod.encode_textfield_ncr(content)
```

Encodes the contents for CIF textfield in Numeric Character Reference. Encoded characters:

- `\x09`, `\x0A`, `\x0D`, `\x20–\x7E`;
- `‘`; `’`, if encountered on the beginning of the line;
- `‘\t‘`
- `‘.‘` and `‘?’`, if comprise the entire textfield.

**Parameters** `content` – a string with contents

**Returns** encoded string

`aiida.tools.dbexporters.tcod.encode_textfield_quoted_printable(content)`

Encodes the contents for CIF textfield in quoted-printable encoding. In addition to non-ASCII characters, that are encoded by Python function `quopri.encodestring()`, following characters are encoded:

- `;`, if encountered on the beginning of the line;
- `\t`
- `‘`, `’` and `?`, if comprise the entire textfield.

**Parameters** `content` – a string with contents

**Returns** encoded string

`aiida.tools.dbexporters.tcod.export_cif(what, **kwargs)`

Exports given coordinate-containing \*Data node to string of CIF format.

**Returns** string with contents of CIF file.

`aiida.tools.dbexporters.tcod.export_cifnode(what, parameters=None, trajectory_index=None, store=False, reduce_symmetry=True, **kwargs)`

The main exporter function. Exports given coordinate-containing \*Data node to `aiida.orm.data.cif.CifData` node, ready to be exported to TCOD. All \*Data types, having method `_get_cif()`, are supported in addition to `aiida.orm.data.cif.CifData`.

**Parameters**

- **what** – data node to be exported.
- **parameters** – a `aiida.orm.data.parameter.ParameterData` instance, produced by the same calculation as the original exported node.
- **trajectory\_index** – a step to be converted and exported in case a `aiida.orm.data.array.trajectory.TrajectoryData` is exported.
- **store** – boolean indicating whether to store intermediate nodes or not. Default False.
- **dump\_aiida\_database** – boolean indicating whether to include the dump of AiiDA database (containing only transitive closure of the exported node). Default True.
- **exclude\_external\_contents** – boolean indicating whether to exclude nodes from AiiDA database dump, that are taken from external repositories and have a URL link allowing to refetch their contents. Default False.
- **gzip** – boolean indicating whether to Gzip large CIF text fields. Default False.
- **gzip\_threshold** – integer indicating the maximum size (in bytes) of uncompressed CIF text fields when the **gzip** option is in action. Default 1024.

**Returns** a `aiida.orm.data.cif.CifData` node.

`aiida.tools.dbexporters.tcod.export_values(what, **kwargs)`

Exports given coordinate-containing \*Data node to PyCIFRW CIF data structure.

**Returns** CIF data structure.

---

**Note:** Requires PyCIFRW.

---

`aiida.tools.dbexporters.tcod.extend_with_cmdline_parameters` (*parser*, *exp-  
class='Data'*)

Provides descriptions of command line options, that are used to control the process of exporting data to TCOD CIF files.

#### Parameters

- **parser** – an `argparse.Parser` instance
- **expclass** – name of the exported class to be shown in help string for the command line options

---

**Note:** This method must not set any default values for command line options in order not to clash with any other data export plugins.

---

`aiida.tools.dbexporters.tcod.translate_calculation_specific_values` (*calc*, *translator*, *\*\*kwargs*)

Translates calculation-specific values from `aiida.orm.calculation.job.JobCalculation` subclass to appropriate TCOD CIF tags.

#### Parameters

- **calc** – an instance of `aiida.orm.calculation.job.JobCalculation` subclass.
- **translator** – class, derived from `aiida.tools.dbexporters.tcod_plugins.BaseTcodtranslator`

**Raises ValueError** if **translator** is not derived from proper class.

## TCOD parameter translator documentation

### Base class

**class** `aiida.tools.dbexporters.tcod_plugins.BaseTcodtranslator`

Base translator from calculation-specific input and output parameters to TCOD CIF dictionary tags.

**classmethod** `get_BZ_integration_grid_X` (*calc*, *\*\*kwargs*)

Returns a number of points in the Brillouin zone along reciprocal lattice vector X.

**classmethod** `get_BZ_integration_grid_Y` (*calc*, *\*\*kwargs*)

Returns a number of points in the Brillouin zone along reciprocal lattice vector Y.

**classmethod** `get_BZ_integration_grid_Z` (*calc*, *\*\*kwargs*)

Returns a number of points in the Brillouin zone along reciprocal lattice vector Z.

**classmethod** `get_BZ_integration_grid_shift_X` (*calc*, *\*\*kwargs*)

Returns the shift of the Brillouin zone points along reciprocal lattice vector X.

**classmethod** `get_BZ_integration_grid_shift_Y` (*calc*, *\*\*kwargs*)

Returns the shift of the Brillouin zone points along reciprocal lattice vector Y.

**classmethod** `get_BZ_integration_grid_shift_Z` (*calc*, *\*\*kwargs*)

Returns the shift of the Brillouin zone points along reciprocal lattice vector Z.

**classmethod** `get_atom_site_residual_force_Cartesian_x` (*calc*, *\*\*kwargs*)

Returns a list of x components for Cartesian coordinates of residual force for atom. The list order MUST be the same as in the resulting structure.

**classmethod `get_atom_site_residual_force_Cartesian_y`** (*calc*, *\*\*kwargs*)  
Returns a list of y components for Cartesian coordinates of residual force for atom. The list order MUST be the same as in the resulting structure.

**classmethod `get_atom_site_residual_force_Cartesian_z`** (*calc*, *\*\*kwargs*)  
Returns a list of z components for Cartesian coordinates of residual force for atom. The list order MUST be the same as in the resulting structure.

**classmethod `get_atom_type_basisset`** (*calc*, *\*\*kwargs*)  
Returns a list of basisset names for each atom type. The list order MUST be the same as of `get_atom_type_symbol()`.

**classmethod `get_atom_type_symbol`** (*calc*, *\*\*kwargs*)  
Returns a list of atom types. Each atom site MUST occur only once in this list. List MUST be sorted.

**classmethod `get_atom_type_valence_configuration`** (*calc*, *\*\*kwargs*)  
Returns valence configuration of each atom type. The list order MUST be the same as of `get_atom_type_symbol()`.

**classmethod `get_computation_wallclock_time`** (*calc*, *\*\*kwargs*)  
Returns the computation wallclock time in seconds.

**classmethod `get_ewald_energy`** (*calc*, *\*\*kwargs*)  
Returns Ewald energy in eV.

**classmethod `get_exchange_correlation_energy`** (*calc*, *\*\*kwargs*)  
Returns exchange correlation (XC) energy in eV.

**classmethod `get_fermi_energy`** (*calc*, *\*\*kwargs*)  
Returns Fermi energy in eV.

**classmethod `get_hartree_energy`** (*calc*, *\*\*kwargs*)  
Returns Hartree energy in eV.

**classmethod `get_integration_Methfessel_Paxton_order`** (*calc*, *\*\*kwargs*)  
Returns the order of Methfessel-Paxton approximation if used.

**classmethod `get_integration_smearing_method`** (*calc*, *\*\*kwargs*)  
Returns the smearing method name as string.

**classmethod `get_integration_smearing_method_other`** (*calc*, *\*\*kwargs*)  
Returns the smearing method name as string if the name is different from specified in `cif_dft.dic`.

**classmethod `get_kinetic_energy_cutoff_EEX`** (*calc*, *\*\*kwargs*)  
Returns kinetic energy cutoff for exact exchange (EEX) operator in eV.

**classmethod `get_kinetic_energy_cutoff_charge_density`** (*calc*, *\*\*kwargs*)  
Returns kinetic energy cutoff for charge density in eV.

**classmethod `get_kinetic_energy_cutoff_wavefunctions`** (*calc*, *\*\*kwargs*)  
Returns kinetic energy cutoff for wavefunctions in eV.

**classmethod `get_number_of_electrons`** (*calc*, *\*\*kwargs*)  
Returns the number of electrons.

**classmethod `get_one_electron_energy`** (*calc*, *\*\*kwargs*)  
Returns one electron energy in eV.

**classmethod `get_software_executable_path`** (*calc*, *\*\*kwargs*)  
Returns the file-system path to the executable that was run for this computation.

**classmethod** `get_software_package` (*calc*, *\*\*kwargs*)

Returns the package or program name that was used to produce the structure. Only package or program name should be used, e.g. 'VASP', 'psi3', 'Abinit', etc.

**classmethod** `get_software_package_compilation_timestamp` (*calc*, *\*\*kwargs*)

Returns the timestamp of package/program compilation in ISO 8601 format.

**classmethod** `get_software_package_version` (*calc*, *\*\*kwargs*)

Returns software package version used to compute and produce the computed structure file. Only version designator should be used, e.g. '3.4.0', '2.1rc3'.

**classmethod** `get_total_energy` (*calc*, *\*\*kwargs*)

Returns the total energy in eV.

## CP

**class** `aiida.tools.dbexporters.tcod_plugins.cp.CpTcodtranslator`

Quantum ESPRESSO's CP-specific input and output parameter translator to TCOD CIF dictionary tags.

**classmethod** `get_computation_wallclock_time` (*calc*, *\*\*kwargs*)

Returns the computation wallclock time in seconds.

**classmethod** `get_number_of_electrons` (*calc*, *\*\*kwargs*)

Returns the number of electrons.

**classmethod** `get_software_package` (*calc*, *\*\*kwargs*)

Returns the package or program name that was used to produce the structure. Only package or program name should be used, e.g. 'VASP', 'psi3', 'Abinit', etc.

## NWChem (pymatgen-based)

**class** `aiida.tools.dbexporters.tcod_plugins.nwcpymatgen.NwcpymatgenTcodtranslator`

NWChem's input and output parameter translator to TCOD CIF dictionary tags.

**classmethod** `get_atom_type_basisset` (*calc*, *\*\*kwargs*)

Returns a list of basisset names for each atom type. The list order MUST be the same as of `get_atom_type_symbol()`.

**classmethod** `get_atom_type_symbol` (*calc*, *\*\*kwargs*)

Returns a list of atom types. Each atom site MUST occur only once in this list. List MUST be sorted.

**classmethod** `get_atom_type_valence_configuration` (*calc*, *\*\*kwargs*)

Returns valence configuration of each atom type. The list order MUST be the same as of `get_atom_type_symbol()`.

**classmethod** `get_software_package` (*calc*, *\*\*kwargs*)

Returns the package or program name that was used to produce the structure. Only package or program name should be used, e.g. 'VASP', 'psi3', 'Abinit', etc.

**classmethod** `get_software_package_compilation_timestamp` (*calc*, *\*\*kwargs*)

Returns the timestamp of package/program compilation in ISO 8601 format.

**classmethod** `get_software_package_version` (*calc*, *\*\*kwargs*)

Returns software package version used to compute and produce the computed structure file. Only version designator should be used, e.g. '3.4.0', '2.1rc3'.



**PW**

**class** `aiida.tools.dbexporters.tcod_plugins.pw.PwTcodtranslator`

Quantum ESPRESSO's PW-specific input and output parameter translator to TCOD CIF dictionary tags.

**classmethod** `get_BZ_integration_grid_X` (*calc*, *\*\*kwargs*)

Returns a number of points in the Brillouin zone along reciprocal lattice vector X.

**classmethod** `get_BZ_integration_grid_Y` (*calc*, *\*\*kwargs*)

Returns a number of points in the Brillouin zone along reciprocal lattice vector Y.

**classmethod** `get_BZ_integration_grid_Z` (*calc*, *\*\*kwargs*)

Returns a number of points in the Brillouin zone along reciprocal lattice vector Z.

**classmethod** `get_BZ_integration_grid_shift_X` (*calc*, *\*\*kwargs*)

Returns the shift of the Brillouin zone points along reciprocal lattice vector X.

**classmethod** `get_BZ_integration_grid_shift_Y` (*calc*, *\*\*kwargs*)

Returns the shift of the Brillouin zone points along reciprocal lattice vector Y.

**classmethod** `get_BZ_integration_grid_shift_Z` (*calc*, *\*\*kwargs*)

Returns the shift of the Brillouin zone points along reciprocal lattice vector Z.

**classmethod** `get_atom_site_residual_force_Cartesian_x` (*calc*, *\*\*kwargs*)

Returns a list of x components for Cartesian coordinates of residual force for atom. The list order MUST be the same as in the resulting structure.

**classmethod** `get_atom_site_residual_force_Cartesian_y` (*calc*, *\*\*kwargs*)

Returns a list of y components for Cartesian coordinates of residual force for atom. The list order MUST be the same as in the resulting structure.

**classmethod** `get_atom_site_residual_force_Cartesian_z` (*calc*, *\*\*kwargs*)

Returns a list of z components for Cartesian coordinates of residual force for atom. The list order MUST be the same as in the resulting structure.

**classmethod** `get_computation_wallclock_time` (*calc*, *\*\*kwargs*)

Returns the computation wallclock time in seconds.

**classmethod** `get_ewald_energy` (*calc*, *\*\*kwargs*)

Returns Ewald energy in eV.

**classmethod** `get_exchange_correlation_energy` (*calc*, *\*\*kwargs*)

Returns exchange correlation (XC) energy in eV.

**classmethod** `get_fermi_energy` (*calc*, *\*\*kwargs*)

Returns Fermi energy in eV.

**classmethod** `get_hartree_energy` (*calc*, *\*\*kwargs*)

Returns Hartree energy in eV.

**classmethod** `get_integration_Methfessel_Paxton_order` (*calc*, *\*\*kwargs*)

Returns the order of Methfessel-Paxton approximation if used.

**classmethod** `get_integration_smearing_method` (*calc*, *\*\*kwargs*)

Returns the smearing method name as string.

**classmethod** `get_integration_smearing_method_other` (*calc*, *\*\*kwargs*)

Returns the smearing method name as string if the name is different from specified in `cif_dft.dic`.

**classmethod** `get_kinetic_energy_cutoff_EEX` (*calc*, *\*\*kwargs*)

Returns kinetic energy cutoff for exact exchange (EEX) operator in eV.

---

**Note:** by default returns `ecutrho`, as indicated in <http://www.quantum-espresso.org/wp->

[content/uploads/Doc/INPUT\\_PW.html](http://www.quantum-espresso.org/wp-content/uploads/Doc/INPUT_PW.html)

---

**classmethod** `get_kinetic_energy_cutoff_charge_density` (*calc*, **\*\*kwargs**)

Returns kinetic energy cutoff for charge density in eV.

---

**Note:** by default returns  $4 * \text{ecutwfc}$ , as indicated in [http://www.quantum-espresso.org/wp-content/uploads/Doc/INPUT\\_PW.html](http://www.quantum-espresso.org/wp-content/uploads/Doc/INPUT_PW.html)

---

**classmethod** `get_kinetic_energy_cutoff_wavefunctions` (*calc*, **\*\*kwargs**)

Returns kinetic energy cutoff for wavefunctions in eV.

**classmethod** `get_number_of_electrons` (*calc*, **\*\*kwargs**)

Returns the number of electrons.

**classmethod** `get_one_electron_energy` (*calc*, **\*\*kwargs**)

Returns one electron energy in eV.

**classmethod** `get_software_package` (*calc*, **\*\*kwargs**)

Returns the package or program name that was used to produce the structure. Only package or program name should be used, e.g. 'VASP', 'psi3', 'Abinit', etc.

**classmethod** `get_total_energy` (*calc*, **\*\*kwargs**)

Returns the total energy in eV.

## 4.1.14 aiida.tools documentation

### Tools

#### pw input parser

Tools for parsing QE PW input files and creating AiiDa Node objects based on them.

**TODO: Parse CONSTRAINTS, OCCUPATIONS, ATOMIC\_FORCES once they are implemented** in AiiDa

**class** `aiida.tools.codespecific.quantum.espresso.pwinputparser.PwInputFile` (*pwininput*)

Class used for parsing Quantum Espresso pw.x input files and using the info.

#### Variables

- **namelists** – A nested dictionary of the namelists and their key-value pairs. The namelists will always be upper-case keys, while the parameter keys will always be lower-case.

For example:

```
{ "CONTROL": { "calculation": "bands",
               "prefix": "al",
               "pseudo_dir": "./pseudo",
               "outdir": "./out"},
  "ELECTRONS": { "diagonalization": "cg"},
  "SYSTEM": { "nbnd": 8,
              "ecutwfc": 15.0,
              "celldm(1)": 7.5,
              "ibrav": 2,
              "nat": 1,
              "ntyp": 1}
}
```

- **atomic\_positions** – A dictionary with

- units: the units of the positions (always lower-case) or None
- names: list of the atom names (e.g. 'Si', 'Si0', 'Si\_0')
- positions: list of the [x, y, z] positions
- fixed\_coords: list of [x, y, z] (bools) of the force modifications (**Note:** True <=> Fixed, as defined in the `BasePwCpInputGenerator._if_pos` method)

For example:

```
{'units': 'bohr',
 'names': ['C', 'O'],
 'positions': [[0.0, 0.0, 0.0],
               [0.0, 0.0, 2.5]]
 'fixed_coords': [[False, False, False],
                  [True, True, True]]}
```

- **cell\_parameters** – A dictionary (if CELL\_PARAMETERS is present; else: None) with

- units: the units of the lattice vectors (always lower-case) or None
- cell: 3x3 list with lattice vectors as rows

For example:

```
{'units': 'angstrom',
 'cell': [[16.9, 0.0, 0.0],
          [-2.6, 8.0, 0.0],
          [-2.6, -3.5, 7.2]]}
```

- **k\_points** – A dictionary containing

- type: the type of kpoints (always lower-case)
- points: an Nx3 list of the kpoints (will not be present if type = 'gamma' or type = 'automatic')
- weights: a 1xN list of the kpoint weights (will not be present if type = 'gamma' or type = 'automatic')
- mesh: a 1x3 list of the number of equally-spaced points in each direction of the Brillouin zone, as in Monkhorst-Pack grids (only present if type = 'automatic')
- offset: a 1x3 list of the grid offsets in each direction of the Brillouin zone (only present if type = 'automatic') (**Note:** The offset value for each direction will be *one of* 0.0 [no offset] *or* 0.5 [offset by half a grid step]. This differs from the Quantum Espresso convention, where an offset value of 1 corresponds to a half-grid-step offset, but adheres to the current AiiDa convention.

Examples:

```
{'type': 'crystal',
 'points': [[0.125, 0.125, 0.0],
            [0.125, 0.375, 0.0],
            [0.375, 0.375, 0.0]],
 'weights': [1.0, 2.0, 1.0]}

{'type': 'automatic',
 'points': [8, 8, 8],
 'offset': [0.0, 0.5, 0.0]}
```

```
{'type': 'gamma'}
```

- **atomic\_species** – A dictionary with
  - names: list of the atom names (e.g. 'Si', 'Si0', 'Si\_0') (case as-is)
  - masses: list of the masses of the atoms in 'names'
  - pseudo\_file\_names: list of the pseudopotential file names for the atoms in 'names' (case as-is)

Example:

```
{'names': ['Li', 'O', 'Al', 'Si'],
 'masses': [6.941, 15.9994, 26.98154, 28.0855],
 'pseudo_file_names': ['Li.pbe-sl-rrkjus_psl.1.0.0.UPF',
                       'O.pbe-nl-rrkjus_psl.1.0.0.UPF',
                       'Al.pbe-nl-rrkjus_psl.1.0.0.UPF',
                       'Si3 28.0855 Si.pbe-nl-rrkjus_psl.1.0.0.UPF']}
```

**\_\_init\_\_**(*pwinput*)

Parse inputs's namelist and cards to create attributes of the info.

**Parameters** *pwinput* – Any one of the following

- A string of the (existing) absolute path to the pwinput file.
- A single string containing the pwinput file's text.
- A list of strings, with the lines of the file as the elements.
- A file object. (It will be opened, if it isn't already.)

**Raises**

- **IOError** – if *pwinput* is a file and there is a problem reading the file.
- **TypeError** – if *pwinput* is a list containing any non-string element(s).
- ***aiida.common.exceptions.ParsingError*** – if there are issues parsing the pwinput.

**get\_kpointsdata**()

Return a KpointsData object based on the data in the input file.

This uses all of the data in the input file to do the necessary unit conversion, ect. and then creates an AiiDa KpointsData object.

**Note:** If the calculation uses only the gamma k-point (if *self.k\_points['type'] == 'gamma'*), it is necessary to also attach a settings node to the calculation with *gamma\_only = True*.

**Returns** KpointsData object of the kpoints in the input file

**Return type** *aiida.orm.data.array.kpoints.KpointsData*

**Raises** ***aiida.common.exceptions.NotImplimentedError*** if the kpoints are in a format not yet supported.

**get\_structuredata**()

Return a StructureData object based on the data in the input file.

This uses all of the data in the input file to do the necessary unit conversion, ect. and then creates an AiiDa StructureData object.

All of the names corresponding of the Kind objects composing the StructureData object will match those found in the ATOMIC\_SPECIES block, so the pseudopotentials can be linked to the calculation using the kind.name for each specific type of atom (in the event that you wish to use different pseudo's for two or more of the same atom).

**Returns** StructureData object of the structure in the input file

**Return type** *aiida.orm.data.structure.StructureData*

**Raises** *aiida.common.exceptions.ParsingError* if there are issues parsing the input.

`aiida.tools.codespecific.quantumespresso.pwinputparser.parse_atomic_positions(txt)`  
Return dict containing info from the ATOMIC\_POSITIONS card block in txt.

---

**Note:** If the units are unspecified, they will be returned as None.

---

**Parameters** `txt` (*str*) – A single string containing the QE input text to be parsed.

**Returns**

A dictionary with

- units: the units of the positions (always lower-case) or None
- names: list of the atom names (e.g. 'Si', 'Si0', 'Si\_0')
- positions: list of the [x, y, z] positions
- fixed\_coords: list of [x, y, z] (bools) of the force modifications (**Note:** True <=> Fixed, as defined in the `BasePwCpInputGenerator._if_pos` method)

For example:

```
{'units': 'bohr',
 'names': ['C', 'O'],
 'positions': [[0.0, 0.0, 0.0],
               [0.0, 0.0, 2.5]]
 'fixed_coords': [[False, False, False],
                  [True, True, True]]}
```

**Return type** *dict*

**Raises** *aiida.common.exceptions.ParsingError* if there are issues parsing the input.

`aiida.tools.codespecific.quantumespresso.pwinputparser.parse_atomic_species(txt)`  
Return dict containing info from the ATOMIC\_SPECIES card block in txt.

**Parameters** `txt` (*str*) – A single string containing the QE input text to be parsed.

**Returns**

A dictionary with

- names: list of the atom names (e.g. 'Si', 'Si0', 'Si\_0') (case as-is)
- masses: list of the masses of the atoms in 'names'
- pseudo\_file\_names: list of the pseudopotential file names for the atoms in 'names' (case as-is)

Example:

```
{'names': ['Li', 'O', 'Al', 'Si'],
 'masses': [6.941, 15.9994, 26.98154, 28.0855],
 'pseudo_file_names': ['Li.pbe-sl-rrkjus_psl.1.0.0.UPF',
                       'O.pbe-nl-rrkjus_psl.1.0.0.UPF',
                       'Al.pbe-nl-rrkjus_psl.1.0.0.UPF',
                       'Si3 28.0855 Si.pbe-nl-rrkjus_psl.1.0.0.UPF']}
```

**Return type** *dict*

**Raises** *aiida.common.exceptions.ParsingError* if there are issues parsing the input.

`aiida.tools.codespecific.quantum Espresso.pwinputparser.parse_cell_parameters(txt)`  
Return dict containing info from the CELL\_PARAMETERS card block in txt.

---

**Note:** This card is only needed if `ibrav = 0`. Therefore, if the card is not present, the function will return `None` and not raise an error.

---



---

**Note:** If the units are unspecified, they will be returned as `None`. The units interpreted by QE depend on whether or not one of ‘`celldm(1)`’ or ‘`a`’ is set in `&SYSTEM`.

---

**Parameters** `txt` (*str*) – A single string containing the QE input text to be parsed.

#### Returns

A dictionary (if CELL\_PARAMETERS is present; else: `None`) with

- `units`: the units of the lattice vectors (always lower-case) or `None`
- `cell`: 3x3 list with lattice vectors as rows

For example:

```
{'units': 'angstrom',
 'cell': [[16.9, 0.0, 0.0],
          [-2.6, 8.0, 0.0],
          [-2.6, -3.5, 7.2]]}
```

**Return type** *dict* or `None`

**Raises** *aiida.common.exceptions.ParsingError* if there are issues parsing the input.

`aiida.tools.codespecific.quantum Espresso.pwinputparser.parse_k_points(txt)`  
Return dict containing info from the K\_POINTS card block in txt.

---

**Note:** If the type of kpoints (where type = x in the card header, “K\_POINTS x”) is not present, type will be returned as ‘`tpiba`’, the QE default.

---

**Parameters** `txt` (*str*) – A single string containing the QE input text to be parsed.

#### Returns

A dictionary containing

- `type`: the type of kpoints (always lower-case)
- `points`: an Nx3 list of the kpoints (will not be present if type = ‘`gamma`’ or type = ‘`automatic`’)

- **weights**: a 1xN list of the kpoint weights (will not be present if type = 'gamma' or type = 'automatic')
- **mesh**: a 1x3 list of the number of equally-spaced points in each direction of the Brillouin zone, as in Monkhorst-Pack grids (only present if type = 'automatic')
- **offset**: a 1x3 list of the grid offsets in each direction of the Brillouin zone (only present if type = 'automatic') (**Note:** The offset value for each direction will be *one of* 0.0 [no offset] *or* 0.5 [offset by half a grid step]. This differs from the Quantum Espresso convention, where an offset value of 1 corresponds to a half-grid-step offset, but adheres to the current AiiDa convention.

Examples:

```
{'type': 'crystal',
 'points': [[0.125, 0.125, 0.0],
            [0.125, 0.375, 0.0],
            [0.375, 0.375, 0.0]],
 'weights': [1.0, 2.0, 1.0]}

{'type': 'automatic',
 'points': [8, 8, 8],
 'offset': [0.0, 0.5, 0.0]}

{'type': 'gamma'}
```

**Return type** *dict*

**Raises** *aiida.common.exceptions.ParsingError* if there are issues parsing the input.

`aiida.tools.codespecific.quantum.espresso.pwinputparser.parse_namelist(txt)`

Parse txt to extract a dictionary of the namelist info.

**Parameters** `txt` (*str*) – A single string containing the QE input text to be parsed.

**Returns**

A nested dictionary of the namelists and their key-value pairs. The namelists will always be upper-case keys, while the parameter keys will always be lower-case.

For example:

```
{"CONTROL": {"calculation": "bands",
             "prefix": "al",
             "pseudo_dir": "./pseudo",
             "outdir": "./out"},
 "ELECTRONS": {"diagonalization": "cg"},
 "SYSTEM": {"nbnd": 8,
            "ecutwfc": 15.0,
            "celldm(1)": 7.5,
            "ibrav": 2,
            "nat": 1,
            "ntyp": 1}
}
```

**Return type** *dict*

**Raises** *aiida.common.exceptions.ParsingError* if there are issues parsing the input.

`aiida.tools.codespecific.quantum.espresso.pwinputparser.str2val(valstr)`

Return a python value by converting valstr according to f90 syntax.

**Parameters** `valstr` (*str*) – String representation of the variable to be converted. (e.g. `‘.true.’`)

**Returns** A python variable corresponding to `valstr`.

**Return type** `bool` or `float` or `int` or `str`

**Raises** `ValueError`: if a suitable conversion of `valstr` cannot be found.



---

## Indices and tables

---

- `genindex`
- `modindex`
- `search`



**a**

`aiida.cmdline`, 256  
`aiida.cmdline.baseclass`, 256  
`aiida.cmdline.commands.daemon`, 258  
`aiida.cmdline.commands.data`, 259  
`aiida.cmdline.verdilib`, 257  
`aiida.common`, 229  
`aiida.common.datastructures`, 229  
`aiida.common.exceptions`, 230  
`aiida.common.extendeddicts`, 231  
`aiida.common.folders`, 233  
`aiida.common.pluginloader`, 235  
`aiida.common.utils`, 237  
`aiida.djsite.db.models`, 262  
`aiida.execmanager`, 261  
`aiida.orm`, 267  
`aiida.orm.calculation`, 304  
`aiida.orm.calculation.inline`, 304  
`aiida.orm.calculation.job`, 307  
`aiida.orm.calculation.job.quantumesspresso.pw`, 311  
`aiida.orm.calculation.job.quantumesspresso.pwimmigrant`, 311  
`aiida.orm.calculation.job.simpleplugins.template_replacer`, 314  
`aiida.orm.code`, 281  
`aiida.orm.computer`, 268  
`aiida.orm.data`, 284  
`aiida.orm.data.array`, 299  
`aiida.orm.data.array.kpoints`, 300  
`aiida.orm.data.array.trajectory`, 302  
`aiida.orm.data.cif`, 297  
`aiida.orm.data.folder`, 294  
`aiida.orm.data.parameter`, 298  
`aiida.orm.data.remote`, 299  
`aiida.orm.data.singlefile`, 295  
`aiida.orm.data.structure`, 285  
`aiida.orm.data.upf`, 295  
`aiida.orm.node`, 270  
`aiida.orm.querytool`, 317  
`aiida.orm.workflow`, 276  
`aiida.parsers.plugins.quantumesspresso`, 314  
`aiida.parsers.plugins.quantumesspresso.basic_raw_parser`, 315  
`aiida.parsers.plugins.quantumesspresso.basic_raw_parser`, 315  
`aiida.parsers.plugins.quantumesspresso.basicpw`, 317  
`aiida.parsers.plugins.quantumesspresso.constants`, 317  
`aiida.parsers.plugins.quantumesspresso.cp`, 317  
`aiida.scheduler.__init__`, 251  
`aiida.scheduler.datastructures`, 252  
`aiida.tools.codespecific.quantumesspresso.pwinputparser`, 334  
`aiida.tools.dbexporters.tcod`, 327  
`aiida.tools.dbexporters.tcod_plugins`, 330  
`aiida.tools.dbexporters.tcod_plugins.cp`, 332  
`aiida.tools.dbexporters.tcod_plugins.nwcpymatgen`, 332  
`aiida.tools.dbexporters.tcod_plugins.pw`, 333  
`aiida.tools.dbimporters`, 319  
`aiida.tools.dbimporters.baseclasses`, 319  
`aiida.tools.dbimporters.plugins.cod`, 322  
`aiida.tools.dbimporters.plugins.icsd`, 322  
`aiida.tools.dbimporters.plugins.mpod`, 325  
`aiida.tools.dbimporters.plugins.nninc`, 326  
`aiida.tools.dbimporters.plugins.oqmd`, 325  
`aiida.tools.dbimporters.plugins.pcod`, 326  
`aiida.tools.dbimporters.plugins.tcod`, 326

`aiida.transport.__init__`, [240](#)

## Symbols

- `__enter__()` (aiida.transport.\_\_init\_\_.Transport method), 240
- `__exit__()` (aiida.transport.\_\_init\_\_.Transport method), 240
- `__init__()` (aiida.cmdline.commands.daemon.Daemon method), 258
- `__init__()` (aiida.cmdline.commands.data.Data method), 259
- `__init__()` (aiida.orm.calculation.job.CalculationResultManager method), 213, 307
- `__init__()` (aiida.orm.data.structure.Kind method), 191, 285
- `__init__()` (aiida.orm.data.structure.Site method), 193, 287
- `__init__()` (aiida.orm.node.AttributeManager method), 177, 270
- `__init__()` (aiida.orm.node.Node method), 177, 270
- `__init__()` (aiida.orm.node.NodeInputManager method), 182, 276
- `__init__()` (aiida.orm.node.NodeOutputManager method), 182, 276
- `__init__()` (aiida.tools.codespecific.quantumpresso.pwinputparser.PwInputFile method), 336
- `__iter__()` (aiida.tools.dbimporters.baseclasses.DbSearchResults method), 321
- `add_attr_filter()` (aiida.orm.querytool.QueryTool method), 318
- `add_attribute()` (aiida.orm.workflow.Workflow method), 182, 276
- `add_attributes()` (aiida.orm.workflow.Workflow method), 183, 276
- `add_comment()` (aiida.orm.node.Node method), 177, 270
- `add_extra_filter()` (aiida.orm.querytool.QueryTool method), 318
- `add_from_logrecord()` (aiida.djsite.db.models.DbLog class method), 264
- `add_path()` (aiida.orm.data.remote.RemoteData method), 205, 299
- `add_path()` (aiida.orm.data.singlefile.SinglefileData method), 201, 295
- `add_path()` (aiida.orm.node.Node method), 177, 270
- `add_path()` (aiida.orm.workflow.Workflow method), 183, 276
- `add_result()` (aiida.orm.workflow.Workflow method), 183, 276
- `add_results()` (aiida.orm.workflow.Workflow method), 183, 276
- aiida.cmdline (module), 256
- aiida.cmdline.baseclass (module), 256
- aiida.cmdline.commands.daemon (module), 258
- aiida.cmdline.commands.data (module), 259
- aiida.cmdline.verdilib (module), 257
- aiida.common (module), 229
- aiida.common.datastructures (module), 229
- aiida.common.exceptions (module), 230
- aiida.common.extendeddicts (module), 231
- aiida.common.folders (module), 233
- aiida.common.pluginloader (module), 235
- aiida.common.utils (module), 237
- aiida.djsite.db.models (module), 262
- aiida.execmanager (module), 261
- aiida.orm (module), 174, 267
- aiida.orm.calculation (module), 210, 304
- aiida.orm.calculation.inline (module), 210, 304
- aiida.orm.calculation.job (module), 213, 307
- aiida.orm.calculation.job.quantumpresso.pw (module), 217, 311

## A

- `abspath` (aiida.common.folders.Folder attribute), 233
- `accepts_default_mpi_procs_per_machine()` (aiida.scheduler.datastructures.JobResource class method), 254
- `accepts_default_mpi_procs_per_machine()` (aiida.scheduler.datastructures.NodeNumberJobResource class method), 255
- `accepts_default_mpi_procs_per_machine()` (aiida.scheduler.datastructures.ParEnvJobResource class method), 255

- aiida.orm.calculation.job.quantumpresso.pwimmigrant (module), 217, 311
  - aiida.orm.calculation.job.simpleplugins.templatereplacer (module), 220, 314
  - aiida.orm.code (module), 188, 281
  - aiida.orm.computer (module), 175, 268
  - aiida.orm.data (module), 190, 284
  - aiida.orm.data.array (module), 205, 299
  - aiida.orm.data.array.kpoints (module), 206, 300
  - aiida.orm.data.array.trajectory (module), 208, 302
  - aiida.orm.data.cif (module), 203, 297
  - aiida.orm.data.folder (module), 201, 294
  - aiida.orm.data.parameter (module), 205, 298
  - aiida.orm.data.remote (module), 205, 299
  - aiida.orm.data.singlefile (module), 201, 295
  - aiida.orm.data.structure (module), 191, 285
  - aiida.orm.data.upf (module), 201, 295
  - aiida.orm.node (module), 176, 270
  - aiida.orm.querytool (module), 317
  - aiida.orm.workflow (module), 182, 276
  - aiida.parsers.plugins.quantumpresso (module), 220, 314
  - aiida.parsers.plugins.quantumpresso.basic\_raw\_parser\_cp (module), 220, 315
  - aiida.parsers.plugins.quantumpresso.basic\_raw\_parser\_pw (module), 221, 315
  - aiida.parsers.plugins.quantumpresso.basicpw (module), 223, 317
  - aiida.parsers.plugins.quantumpresso.constants (module), 223, 317
  - aiida.parsers.plugins.quantumpresso.cp (module), 223, 317
  - aiida.scheduler.\_\_init\_\_ (module), 251
  - aiida.scheduler.datastructures (module), 252
  - aiida.tools.codespecific.quantumpresso.pwinputparser (module), 334
  - aiida.tools.dbexporters.tcod (module), 327
  - aiida.tools.dbexporters.tcod\_plugins (module), 330
  - aiida.tools.dbexporters.tcod\_plugins.cp (module), 332
  - aiida.tools.dbexporters.tcod\_plugins.nwcpymatgen (module), 332
  - aiida.tools.dbexporters.tcod\_plugins.pw (module), 333
  - aiida.tools.dbimporters (module), 319
  - aiida.tools.dbimporters.baseclasses (module), 319
  - aiida.tools.dbimporters.plugins.cod (module), 322
  - aiida.tools.dbimporters.plugins.icsd (module), 322
  - aiida.tools.dbimporters.plugins.mpod (module), 325
  - aiida.tools.dbimporters.plugins.nninc (module), 326
  - aiida.tools.dbimporters.plugins.oqmd (module), 325
  - aiida.tools.dbimporters.plugins.pcod (module), 326
  - aiida.tools.dbimporters.plugins.tcod (module), 326
  - aiida.transport.\_\_init\_\_ (module), 240
  - AiidaException, 230
  - append\_atom() (aiida.orm.data.structure.StructureData method), 193, 287
  - append\_kind() (aiida.orm.data.structure.StructureData method), 194, 288
  - append\_list\_cmdline\_arguments() (aiida.cmdline.commands.data.Listable method), 260
  - append\_site() (aiida.orm.data.structure.StructureData method), 194, 288
  - append\_to\_report() (aiida.orm.workflow.Workflow method), 183, 276
  - ArrayData (class in aiida.orm.data.array), 205, 299
  - arraynames() (aiida.orm.data.array.ArrayData method), 205, 299
  - ase (aiida.orm.data.cif.CifData attribute), 203, 297
  - ase\_refine\_cell() (in module aiida.orm.data.structure), 197, 291
  - at() (aiida.tools.dbimporters.baseclasses.DbSearchResults method), 321
  - at() (aiida.tools.dbimporters.plugins.icsd.IcsdSearchResults method), 324
  - attach\_calculation() (aiida.orm.workflow.Workflow method), 183, 276
  - attach\_workflow() (aiida.orm.workflow.Workflow method), 183, 277
  - AttributeDict (class in aiida.common.extendeddicts), 231
  - AttributeManager (class in aiida.orm.node), 176, 270
  - attributes (aiida.djsite.db.models.DbNode attribute), 266
  - attrs() (aiida.orm.node.Node method), 177, 271
  - AuthenticationError, 230
- ## B
- BaseFactory() (in module aiida.common.pluginloader), 235
  - BaseTcodtranslator (class in aiida.tools.dbexporters.tcod\_plugins), 330
  - BasicpwParser (class in aiida.parsers.plugins.quantumpresso.basicpw), 223, 317
- ## C
- calc\_cell\_volume() (in module aiida.orm.data.structure), 198, 291
  - CalcInfo (class in aiida.common.datastructures), 229
  - Calculation (class in aiida.orm.calculation), 210, 304
  - CalculationFactory() (in module aiida.orm), 174, 267
  - CalculationResultManager (class in aiida.orm.calculation.job), 213, 307
  - can\_run\_on() (aiida.orm.code.Code method), 188, 281
  - cell (aiida.orm.data.array.kpoints.KpointsData attribute), 206, 300
  - cell (aiida.orm.data.structure.StructureData attribute), 194, 288

- `cell_angles` (aiida.orm.data.structure.StructureData attribute), 194, 288
- `cell_lengths` (aiida.orm.data.structure.StructureData attribute), 194, 288
- `cell_volume()` (in module aiida.parsers.plugins.quantumespresso.basic\_raw\_parser\_pw), 221, 315
- `chdir()` (aiida.transport.\_\_init\_\_.Transport method), 240
- `chmod()` (aiida.transport.\_\_init\_\_.Transport method), 240
- `chown()` (aiida.transport.\_\_init\_\_.Transport method), 240
- `cif` (aiida.tools.dbimporters.baseclasses.CifEntry attribute), 319
- `cif` (aiida.tools.dbimporters.plugins.icsd.IcsdEntry attribute), 324
- `cif_encode_contents()` (in module aiida.tools.dbexporters.tcod), 327
- `cif_from_ase()` (in module aiida.orm.data.cif), 204, 298
- `CifData` (class in aiida.orm.data.cif), 203, 297
- `CifEntry` (class in aiida.tools.dbimporters.baseclasses), 319
- `CifFileErrorExp`, 322
- `classproperty` (class in aiida.common.utils), 237
- `clear_internal_cache()` (aiida.orm.data.array.ArrayData method), 205, 299
- `clear_kinds()` (aiida.orm.data.structure.StructureData method), 194, 288
- `clear_report()` (aiida.orm.workflow.Workflow method), 183, 277
- `clear_sites()` (aiida.orm.data.structure.StructureData method), 194, 288
- `close()` (aiida.transport.\_\_init\_\_.Transport method), 240
- `CodDbImporter` (class in aiida.tools.dbimporters.plugins.cod), 322
- `Code` (class in aiida.orm.code), 188, 281
- `CodeInfo` (class in aiida.common.datastructures), 229
- `CodEntry` (class in aiida.tools.dbimporters.plugins.cod), 322
- `CodSearchResults` (class in aiida.tools.dbimporters.plugins.cod), 322
- `compare_with()` (aiida.orm.data.structure.Kind method), 191, 285
- `complete()` (aiida.cmdline.baseclass.VerdiCommand method), 256
- `complete()` (aiida.cmdline.verdilib.Install method), 257
- `Completion` (class in aiida.cmdline.verdilib), 257
- `CompletionCommand` (class in aiida.cmdline.verdilib), 257
- `Computer` (class in aiida.orm.computer), 175, 268
- `ConfigurationError`, 230
- `configure_user()` (aiida.cmdline.commands.daemon.DaemonCreate method), 258
- `ContentNotExistent`, 230
- `contents` (aiida.tools.dbimporters.baseclasses.DbEntry attribute), 320
- `conv_to_fortran()` (in module aiida.common.utils), 237
- `convert()` (aiida.orm.data.Data method), 190, 284
- `convert_list_to_matrix()` (in module aiida.parsers.plugins.quantumespresso.basic\_raw\_parser\_pw), 221, 315
- `create_aiida_structure()` (in module aiida.parsers.plugins.quantumespresso), 220, 314
- `convert_qe_time_to_sec()` (in module aiida.parsers.plugins.quantumespresso.basic\_raw\_parser\_pw), 221, 315
- `copy()` (aiida.common.extendeddicts.AttributeDict method), 231
- `copy()` (aiida.orm.computer.Computer method), 175, 269
- `copy()` (aiida.orm.node.Node method), 177, 271
- `copy()` (aiida.transport.\_\_init\_\_.Transport method), 240
- `copy_from_remote_to_remote()` (aiida.transport.\_\_init\_\_.Transport method), 240
- `copy_from_remote_to_remote()` (in module aiida.transport.\_\_init\_\_), 245
- `copyfile()` (aiida.transport.\_\_init\_\_.Transport method), 241
- `copytree()` (aiida.transport.\_\_init\_\_.Transport method), 241
- `correct_cif()` (in module aiida.tools.dbimporters.plugins.icsd), 324
- `CpParser` (class in aiida.parsers.plugins.quantumespresso.cp), 223, 317
- `CpTcodtranslator` (class in aiida.tools.dbexporters.tcod\_plugins.cp), 332
- `create()` (aiida.common.folders.Folder method), 233
- `create_attrs_dict()` (aiida.orm.querytool.QueryTool method), 318
- `create_display_name()` (in module aiida.common.utils), 237
- `create_extras_dict()` (aiida.orm.querytool.QueryTool method), 319
- `create_file_from_filelike()` (aiida.common.folders.Folder method), 233
- `create_input_nodes()` (aiida.orm.calculation.job.quantumespresso.pwimmigrant.Pwimmigrant method), 217, 311
- `create_job_resource()` (aiida.scheduler.\_\_init\_\_.Scheduler class method), 251
- `create_symlink()` (aiida.common.folders.Folder method), 233
- `create_value()` (aiida.djsite.db.models.DbMultipleValueAttributeBaseClass class method), 264
- `ctime` (aiida.orm.node.Node attribute), 177, 271
- `current_folder` (aiida.orm.workflow.Workflow attribute), 183, 277

## D

- Daemon (class in aiida.cmdline.commands.daemon), 258
- daemon\_logshow() (aiida.cmdline.commands.daemon.Daemon method), 259
- daemon\_restart() (aiida.cmdline.commands.daemon.Daemon method), 259
- daemon\_start() (aiida.cmdline.commands.daemon.Daemon method), 259
- daemon\_status() (aiida.cmdline.commands.daemon.Daemon method), 259
- daemon\_stop() (aiida.cmdline.commands.daemon.Daemon method), 259
- Data (class in aiida.cmdline.commands.data), 259
- Data (class in aiida.orm.data), 190, 284
- DataFactory() (in module aiida.orm), 174, 268
- DbAttribute (class in aiida.djsite.db.models), 262
- DbAttributeBaseClass (class in aiida.djsite.db.models), 262
- DbAuthInfo (class in aiida.djsite.db.models), 263
- DbCalcState (class in aiida.djsite.db.models), 263
- DbComment (class in aiida.djsite.db.models), 263
- DbComputer (class in aiida.djsite.db.models), 263
- DbContentError, 230
- DbEntry (class in aiida.tools.dbimporters.baseclasses), 320
- DbExtra (class in aiida.djsite.db.models), 263
- DbGroup (class in aiida.djsite.db.models), 264
- DbImporter (class in aiida.tools.dbimporters.baseclasses), 320
- DbImporterFactory() (in module aiida.tools.dbimporters), 319
- DbLink (class in aiida.djsite.db.models), 264
- DbLock (class in aiida.djsite.db.models), 264
- DbLog (class in aiida.djsite.db.models), 264
- DbMultipleValueAttributeBaseClass (class in aiida.djsite.db.models), 264
- dbnode (aiida.orm.node.Node attribute), 177, 271
- DbNode (class in aiida.djsite.db.models), 266
- DbPath (class in aiida.djsite.db.models), 266
- DbSearchResults (class in aiida.tools.dbimporters.baseclasses), 321
- DbSearchResults.DbSearchResultsIterator (class in aiida.tools.dbimporters.baseclasses), 321
- DbSetting (class in aiida.djsite.db.models), 266
- DbUser (class in aiida.djsite.db.models), 267
- DbWorkflow (class in aiida.djsite.db.models), 267
- DbWorkflowData (class in aiida.djsite.db.models), 267
- dbworkflowinstance (aiida.orm.workflow.Workflow attribute), 183, 277
- DbWorkflowStep (class in aiida.djsite.db.models), 267
- decode\_textfield() (in module aiida.tools.dbexporters.tcod), 327
- decode\_textfield\_base64() (in module aiida.tools.dbexporters.tcod), 327
- decode\_textfield\_gzip\_base64() (in module aiida.tools.dbexporters.tcod), 327
- decode\_textfield\_ncr() (in module aiida.tools.dbexporters.tcod), 327
- decode\_textfield\_quoted\_printable() (in module aiida.tools.dbexporters.tcod), 328
- DefaultFieldsAttributeDict (class in aiida.common.extendeddicts), 231
- defaultkeys() (aiida.common.extendeddicts.DefaultFieldsAttributeDict method), 232
- del\_extra() (aiida.orm.node.Node method), 178, 271
- del\_file() (aiida.orm.data.singlefile.SinglefileData method), 201, 295
- del\_value() (aiida.djsite.db.models.DbMultipleValueAttributeBaseClass class method), 264
- del\_value\_for\_node() (aiida.djsite.db.models.DbAttributeBaseClass class method), 262
- delete\_array() (aiida.orm.data.array.ArrayData method), 206, 299
- delete\_code() (in module aiida.orm.code), 190, 283
- delete\_computer() (in module aiida.orm.computer), 176, 270
- deposit() (aiida.cmdline.commands.data.Depositable method), 259
- deposit() (in module aiida.tools.dbexporters.tcod), 328
- Depositable (class in aiida.cmdline.commands.data), 259
- deposition\_cmdline\_parameters() (in module aiida.tools.dbexporters.tcod), 328
- description (aiida.orm.node.Node attribute), 178, 271
- description (aiida.orm.workflow.Workflow attribute), 183, 277
- deserialize\_attributes() (in module aiida.djsite.db.models), 267
- dict (aiida.orm.data.parameter.ParameterData attribute), 205, 298

## E

- encode\_textfield\_base64() (in module aiida.tools.dbexporters.tcod), 328
- encode\_textfield\_gzip\_base64() (in module aiida.tools.dbexporters.tcod), 328
- encode\_textfield\_ncr() (in module aiida.tools.dbexporters.tcod), 328
- encode\_textfield\_quoted\_printable() (in module aiida.tools.dbexporters.tcod), 329
- erase() (aiida.common.folders.Folder method), 233
- escape\_for\_bash() (in module aiida.common.utils), 237
- exec\_command\_wait() (aiida.transport.\_\_init\_\_.Transport method), 241



- exec\_from\_cmdline() (in module aiida.cmdline.verdilib), 258  
 existing\_plugins() (in module aiida.common.pluginloader), 236  
 exists() (aiida.common.folders.Folder method), 233  
 exit() (aiida.orm.workflow.Workflow method), 183, 277  
 expand() (aiida.djsite.db.models.DbPath method), 266  
 export() (aiida.cmdline.commands.data.Exportable method), 260  
 export() (aiida.orm.data.Data method), 190, 284  
 export\_cif() (in module aiida.tools.dbexporters.tcod), 329  
 export\_cifnode() (in module aiida.tools.dbexporters.tcod), 329  
 export\_shard\_uuid() (in module aiida.common.utils), 237  
 export\_values() (in module aiida.tools.dbexporters.tcod), 329  
 Exportable (class in aiida.cmdline.commands.data), 259  
 extend\_with\_cmdline\_parameters() (in module aiida.tools.dbexporters.tcod), 329  
 extrakeys() (aiida.common.extendeddicts.DefaultFieldsAttributeDict method), 232  
 extras (aiida.djsite.db.models.DbNode attribute), 266  
 extras() (aiida.orm.node.Node method), 178, 271
- ## F
- FailedError, 230  
 FeatureDisabled, 230  
 FeatureNotAvailable, 230  
 fetch\_all() (aiida.tools.dbimporters.baseclasses.DbSearchResults method), 321  
 FileAttribute (class in aiida.transport.\_\_init\_\_), 240  
 filename (aiida.orm.data.singlefile.SinglefileData attribute), 201, 295  
 FixedFieldsAttributeDict (class in aiida.common.extendeddicts), 232  
 folder (aiida.orm.node.Node attribute), 178, 271  
 Folder (class in aiida.common.folders), 233  
 folder\_limit (aiida.common.folders.Folder attribute), 233  
 FolderData (class in aiida.orm.data.folder), 201, 294  
 from\_md5() (aiida.orm.data.cif.CifData class method), 203, 297  
 from\_md5() (aiida.orm.data.upf.UpfData class method), 201, 295  
 from\_type\_to\_pluginclassname() (in module aiida.orm.node), 182, 276  
 full\_text\_info (aiida.orm.code.Code attribute), 188, 282  
 full\_text\_info (aiida.orm.computer.Computer attribute), 175, 269
- ## G
- generate\_md5() (aiida.orm.data.cif.CifData method), 203, 297  
 get() (aiida.orm.code.Code class method), 188, 282  
 get() (aiida.orm.computer.Computer class method), 175, 269  
 get() (aiida.transport.\_\_init\_\_.Transport method), 241  
 get\_abs\_path() (aiida.common.folders.Folder method), 233  
 get\_abs\_path() (aiida.orm.node.Node method), 178, 271  
 get\_abs\_path() (aiida.orm.workflow.Workflow method), 184, 277  
 get\_aiida\_class() (aiida.djsite.db.models.DbNode method), 266  
 get\_aiida\_class() (aiida.djsite.db.models.DbWorkflow method), 267  
 get\_aiida\_structure() (aiida.tools.dbimporters.baseclasses.CifEntry method), 319  
 get\_aiida\_structure() (aiida.tools.dbimporters.plugins.icsd.IcsdEntry method), 324  
 get\_all\_calcs() (aiida.orm.workflow.Workflow method), 184, 277  
 get\_all\_values\_for\_node() (aiida.djsite.db.models.DbAttributeBaseClass class method), 262  
 get\_append\_text() (aiida.orm.calculation.job.JobCalculation method), 213, 307  
 get\_append\_text() (aiida.orm.code.Code method), 188, 282  
 get\_array() (aiida.orm.data.array.ArrayData method), 206, 300  
 get\_ase() (aiida.orm.data.cif.CifData method), 203, 297  
 get\_ase() (aiida.orm.data.structure.Site method), 193, 287  
 get\_ase() (aiida.orm.data.structure.StructureData method), 194, 288  
 get\_ase\_structure() (aiida.tools.dbimporters.baseclasses.CifEntry method), 320  
 get\_ase\_structure() (aiida.tools.dbimporters.plugins.icsd.IcsdEntry method), 324  
 get\_atom\_site\_residual\_force\_Cartesian\_x() (aiida.tools.dbexporters.tcod\_plugins.BaseTcodtranslator class method), 330  
 get\_atom\_site\_residual\_force\_Cartesian\_x() (aiida.tools.dbexporters.tcod\_plugins.pw.PwTcodtranslator class method), 333  
 get\_atom\_site\_residual\_force\_Cartesian\_y() (aiida.tools.dbexporters.tcod\_plugins.BaseTcodtranslator class method), 330  
 get\_atom\_site\_residual\_force\_Cartesian\_y() (aiida.tools.dbexporters.tcod\_plugins.pw.PwTcodtranslator class method), 333  
 get\_atom\_site\_residual\_force\_Cartesian\_z() (aiida.tools.dbexporters.tcod\_plugins.BaseTcodtranslator class method), 331

<code>get_atom_site_residual_force_Cartesian_z()</code> (aiida.tools.dbexporters.tcod_plugins.pw.PwTcodtranslator class method), 333	<code>get_BZ_integration_grid_X()</code> (aiida.tools.dbexporters.tcod_plugins.pw.PwTcodtranslator class method), 333	<code>get_BZ_integration_grid_X()</code> (aiida.tools.dbexporters.tcod_plugins.pw.PwTcodtranslator class method), 333
<code>get_atom_type_basisset()</code> (aiida.tools.dbexporters.tcod_plugins.BaseTcodtranslator class method), 331	<code>get_BZ_integration_grid_Y()</code> (aiida.tools.dbexporters.tcod_plugins.BaseTcodtranslator class method), 330	<code>get_BZ_integration_grid_Y()</code> (aiida.tools.dbexporters.tcod_plugins.BaseTcodtranslator class method), 330
<code>get_atom_type_basisset()</code> (aiida.tools.dbexporters.tcod_plugins.nwcpymatgen.NwcpymatgenTcodtranslator class method), 332	<code>get_BZ_integration_grid_Y()</code> (aiida.tools.dbexporters.tcod_plugins.pw.PwTcodtranslator class method), 333	<code>get_BZ_integration_grid_Y()</code> (aiida.tools.dbexporters.tcod_plugins.pw.PwTcodtranslator class method), 333
<code>get_atom_type_symbol()</code> (aiida.tools.dbexporters.tcod_plugins.BaseTcodtranslator class method), 331	<code>get_BZ_integration_grid_Z()</code> (aiida.tools.dbexporters.tcod_plugins.BaseTcodtranslator class method), 330	<code>get_BZ_integration_grid_Z()</code> (aiida.tools.dbexporters.tcod_plugins.BaseTcodtranslator class method), 330
<code>get_atom_type_symbol()</code> (aiida.tools.dbexporters.tcod_plugins.nwcpymatgen.NwcpymatgenTcodtranslator class method), 332	<code>get_BZ_integration_grid_Z()</code> (aiida.tools.dbexporters.tcod_plugins.pw.PwTcodtranslator class method), 333	<code>get_BZ_integration_grid_Z()</code> (aiida.tools.dbexporters.tcod_plugins.pw.PwTcodtranslator class method), 333
<code>get_atom_type_valence_configuration()</code> (aiida.tools.dbexporters.tcod_plugins.BaseTcodtranslator class method), 331	<code>get_cell_volume()</code> (aiida.orm.data.structure.StructureData method), 194, 288	<code>get_cell_volume()</code> (aiida.orm.data.structure.StructureData method), 194, 288
<code>get_atom_type_valence_configuration()</code> (aiida.tools.dbexporters.tcod_plugins.nwcpymatgen.NwcpymatgenTcodtranslator class method), 332	<code>get_cells()</code> (aiida.orm.data.array.trajectory.TrajectoryData method), 208, 302	<code>get_cells()</code> (aiida.orm.data.array.trajectory.TrajectoryData method), 208, 302
<code>get_attr()</code> (aiida.orm.node.Node method), 178, 272	<code>get_cif_node()</code> (aiida.tools.dbimporters.plugins.icsd.IcsdEntry method), 324	<code>get_cif_node()</code> (aiida.tools.dbimporters.plugins.icsd.IcsdEntry method), 324
<code>get_attribute()</code> (aiida.orm.workflow.Workflow method), 184, 277	<code>get_class_string()</code> (in module aiida.common.utils), 237	<code>get_class_string()</code> (in module aiida.common.utils), 237
<code>get_attribute()</code> (aiida.transport.__init__.Transport method), 241	<code>get_class_typestring()</code> (in module aiida.common.pluginloader), 236	<code>get_class_typestring()</code> (in module aiida.common.pluginloader), 236
<code>get_attributes()</code> (aiida.orm.querytool.QueryTool method), 319	<code>get_code()</code> (aiida.orm.calculation.Calculation method), 210, 304	<code>get_code()</code> (aiida.orm.calculation.Calculation method), 210, 304
<code>get_attributes()</code> (aiida.orm.workflow.Workflow method), 184, 277	<code>get_column_names()</code> (aiida.cmdline.commands.data.Listable method), 260	<code>get_column_names()</code> (aiida.cmdline.commands.data.Listable method), 260
<code>get_attrs()</code> (aiida.orm.node.Node method), 178, 272	<code>get_command_name()</code> (aiida.cmdline.baseclass.VerdiCommand class method), 256	<code>get_command_name()</code> (aiida.cmdline.baseclass.VerdiCommand class method), 256
<code>get_authinfo()</code> (in module aiida.execmanager), 261	<code>get_command_suggestion()</code> (in module aiida.cmdline.verdilib), 258	<code>get_command_suggestion()</code> (in module aiida.cmdline.verdilib), 258
<code>get_BZ_integration_grid_shift_X()</code> (aiida.tools.dbexporters.tcod_plugins.BaseTcodtranslator class method), 330	<code>get_comments()</code> (aiida.orm.node.Node method), 178, 272	<code>get_comments()</code> (aiida.orm.node.Node method), 178, 272
<code>get_BZ_integration_grid_shift_X()</code> (aiida.tools.dbexporters.tcod_plugins.pw.PwTcodtranslator class method), 333	<code>get_computation_wallclock_time()</code> (aiida.tools.dbexporters.tcod_plugins.BaseTcodtranslator class method), 331	<code>get_computation_wallclock_time()</code> (aiida.tools.dbexporters.tcod_plugins.BaseTcodtranslator class method), 331
<code>get_BZ_integration_grid_shift_Y()</code> (aiida.tools.dbexporters.tcod_plugins.BaseTcodtranslator class method), 330	<code>get_computation_wallclock_time()</code> (aiida.tools.dbexporters.tcod_plugins.cp.CpTcodtranslator class method), 332	<code>get_computation_wallclock_time()</code> (aiida.tools.dbexporters.tcod_plugins.cp.CpTcodtranslator class method), 332
<code>get_BZ_integration_grid_shift_Y()</code> (aiida.tools.dbexporters.tcod_plugins.pw.PwTcodtranslator class method), 333	<code>get_computation_wallclock_time()</code> (aiida.tools.dbexporters.tcod_plugins.pw.PwTcodtranslator class method), 333	<code>get_computation_wallclock_time()</code> (aiida.tools.dbexporters.tcod_plugins.pw.PwTcodtranslator class method), 333
<code>get_BZ_integration_grid_shift_Z()</code> (aiida.tools.dbexporters.tcod_plugins.BaseTcodtranslator class method), 330	<code>get_computer()</code> (aiida.orm.node.Node method), 179, 272	<code>get_computer()</code> (aiida.orm.node.Node method), 179, 272
<code>get_BZ_integration_grid_shift_Z()</code> (aiida.tools.dbexporters.tcod_plugins.pw.PwTcodtranslator class method), 333	<code>get_content_list()</code> (aiida.common.folders.Folder method), 234	<code>get_content_list()</code> (aiida.common.folders.Folder method), 234
<code>get_BZ_integration_grid_X()</code> (aiida.tools.dbexporters.tcod_plugins.BaseTcodtranslator class method), 330	<code>get_corrected_cif()</code> (aiida.tools.dbimporters.plugins.icsd.IcsdEntry method), 324	<code>get_corrected_cif()</code> (aiida.tools.dbimporters.plugins.icsd.IcsdEntry method), 324
	<code>get_custom_scheduler_commands()</code> (aiida.orm.calculation.job.JobCalculation method), 324	<code>get_custom_scheduler_commands()</code> (aiida.orm.calculation.job.JobCalculation method), 324

- method), 213, 307
- get\_daemon\_pid() (aiida.cmdline.commands.daemon.Daemon method), 259
- get\_dbauthinfo() (aiida.orm.computer.Computer method), 175, 269
- get\_dbcomputer() (aiida.djsite.db.models.DbComputer class method), 263
- get\_default\_fields() (aiida.common.extendeddicts.DefaultFieldsAttributeDict class method), 232
- get\_default\_mpiprocs\_per\_machine() (aiida.orm.computer.Computer method), 175, 269
- get\_deposit\_plugins() (aiida.cmdline.commands.data.Depositable method), 259
- get\_detailed\_jobinfo() (aiida.scheduler.\_\_init\_\_.Scheduler method), 251
- get\_dict() (aiida.orm.data.parameter.ParameterData method), 205, 299
- get\_environment\_variables() (aiida.orm.calculation.job.JobCalculation method), 213, 307
- get\_ewald\_energy() (aiida.tools.dbexporters.tcod\_plugins.BaseTcodtranslator class method), 331
- get\_ewald\_energy() (aiida.tools.dbexporters.tcod\_plugins.pw.PwTcodtranslator class method), 333
- get\_exchange\_correlation\_energy() (aiida.tools.dbexporters.tcod\_plugins.BaseTcodtranslator class method), 331
- get\_exchange\_correlation\_energy() (aiida.tools.dbexporters.tcod\_plugins.pw.PwTcodtranslator class method), 333
- get\_execname() (aiida.orm.code.Code method), 188, 282
- get\_export\_plugins() (aiida.cmdline.commands.data.Exportable method), 260
- get\_extra() (aiida.orm.node.Node method), 179, 272
- get\_extras() (aiida.orm.node.Node method), 179, 272
- get\_extremes\_from\_positions() (in module aiida.common.utils), 237
- get\_fermi\_energy() (aiida.tools.dbexporters.tcod\_plugins.BaseTcodtranslator class method), 331
- get\_fermi\_energy() (aiida.tools.dbexporters.tcod\_plugins.pw.PwTcodtranslator class method), 333
- get\_file\_abs\_path() (aiida.orm.data.singlefile.SinglefileData method), 201, 295
- get\_file\_content() (aiida.orm.data.folder.FolderData method), 201, 294
- get\_folder\_list() (aiida.orm.node.Node method), 179, 272
- get\_folder\_list() (aiida.orm.workflow.Workflow method), 184, 277
- get\_formula() (aiida.orm.data.structure.StructureData method), 195, 288
- get\_formula() (in module aiida.orm.data.structure), 198, 291
- get\_formula\_from\_symbol\_list() (in module aiida.orm.data.structure), 198, 292
- get\_formula\_group() (in module aiida.orm.data.structure), 199, 292
- get\_formulae() (aiida.orm.data.cif.CifData method), 203, 297
- get\_from\_string() (aiida.orm.code.Code class method), 188, 282
- get\_full\_command\_name() (aiida.cmdline.baseclass.VerdiCommand method), 256
- get\_full\_command\_name() (aiida.cmdline.baseclass.VerdiCommandWithSubcommands method), 256
- get\_function\_name() (aiida.orm.calculation.inline.InlineCalculation method), 210, 304
- get\_hartree\_energy() (aiida.tools.dbexporters.tcod\_plugins.BaseTcodtranslator class method), 331
- get\_hartree\_energy() (aiida.tools.dbexporters.tcod\_plugins.pw.PwTcodtranslator class method), 333
- get\_import\_plugins() (aiida.cmdline.commands.data.Importable method), 260
- get\_import\_sys\_environment() (aiida.orm.calculation.job.JobCalculation method), 213, 307
- get\_input\_plugin\_name() (aiida.orm.code.Code method), 189, 282
- get\_inputs() (aiida.orm.node.Node method), 179, 272
- get\_inputs\_dict() (aiida.orm.node.Node method), 179, 273
- get\_integration\_Methfessel\_Paxton\_order() (aiida.tools.dbexporters.tcod\_plugins.BaseTcodtranslator class method), 331
- get\_integration\_Methfessel\_Paxton\_order() (aiida.tools.dbexporters.tcod\_plugins.pw.PwTcodtranslator class method), 333
- get\_integration\_smearing\_method() (aiida.tools.dbexporters.tcod\_plugins.BaseTcodtranslator class method), 331
- get\_integration\_smearing\_method() (aiida.tools.dbexporters.tcod\_plugins.pw.PwTcodtranslator

- class method), 333
- get\_integration\_smearing\_method\_other() (aiida.tools.dbexporters.tcod\_plugins.BaseTcodTranslator class method), 331
- get\_integration\_smearing\_method\_other() (aiida.tools.dbexporters.tcod\_plugins.pw.PwTcodTranslator class method), 333
- get\_job\_id() (aiida.orm.calculation.job.JobCalculation method), 213, 307
- get\_kind() (aiida.orm.data.structure.StructureData method), 195, 289
- get\_kind\_names() (aiida.orm.data.structure.StructureData method), 195, 289
- get\_kinetic\_energy\_cutoff\_charge\_density() (aiida.tools.dbexporters.tcod\_plugins.BaseTcodTranslator class method), 331
- get\_kinetic\_energy\_cutoff\_charge\_density() (aiida.tools.dbexporters.tcod\_plugins.pw.PwTcodTranslator class method), 334
- get\_kinetic\_energy\_cutoff\_EEX() (aiida.tools.dbexporters.tcod\_plugins.BaseTcodTranslator class method), 331
- get\_kinetic\_energy\_cutoff\_EEX() (aiida.tools.dbexporters.tcod\_plugins.pw.PwTcodTranslator class method), 333
- get\_kinetic\_energy\_cutoff\_wavefunctions() (aiida.tools.dbexporters.tcod\_plugins.BaseTcodTranslator class method), 331
- get\_kinetic\_energy\_cutoff\_wavefunctions() (aiida.tools.dbexporters.tcod\_plugins.pw.PwTcodTranslator class method), 334
- get\_kpoints() (aiida.orm.data.array.kpoints.KpointsData method), 206, 300
- get\_kpoints\_mesh() (aiida.orm.data.array.kpoints.KpointsData method), 206, 300
- get\_kpointsdata() (aiida.tools.codespecific.quantumpresso.pwinputparser.PwInputParser method), 336
- get\_linkname() (aiida.orm.calculation.Calculation method), 210, 304
- get\_linkname\_out\_kpoints() (aiida.parsers.plugins.quantumpresso.basicpw.BasicpwParser method), 223, 317
- get\_linkname\_outarray() (aiida.parsers.plugins.quantumpresso.basicpw.BasicpwParser method), 223, 317
- get\_linkname\_outstructure() (aiida.parsers.plugins.quantumpresso.basicpw.BasicpwParser method), 223, 317
- get\_linkname\_outtrajectory() (aiida.parsers.plugins.quantumpresso.basicpw.BasicpwParser method), 223, 317
- get\_linkname\_trajectory() (aiida.parsers.plugins.quantumpresso.cp.CpParser method), 223, 317
- get\_listparams() (in module aiida.cmdline.verdilib), 258
- get\_max\_memory\_kb() (aiida.orm.calculation.job.JobCalculation method), 213, 307
- get\_max\_wallclock\_seconds() (aiida.orm.calculation.job.JobCalculation method), 213, 307
- get\_mode() (aiida.transport.\_\_init\_\_.Transport method), 242
- get\_mpirun\_command() (aiida.orm.computer.Computer method), 175, 269
- get\_mpirun\_extra\_params() (aiida.orm.calculation.job.JobCalculation method), 213, 307
- get\_new\_uuid() (in module aiida.common.utils), 237
- get\_number\_of\_electrons() (aiida.tools.dbexporters.tcod\_plugins.BaseTcodTranslator class method), 331
- get\_number\_of\_electrons() (aiida.tools.dbexporters.tcod\_plugins.cp.CpTcodTranslator class method), 332
- get\_number\_of\_electrons() (aiida.tools.dbexporters.tcod\_plugins.pw.PwTcodTranslator class method), 334
- get\_object\_from\_string() (in module aiida.common.utils), 237
- get\_one\_electron\_energy() (aiida.tools.dbexporters.tcod\_plugins.BaseTcodTranslator class method), 331
- get\_one\_electron\_energy() (aiida.tools.dbexporters.tcod\_plugins.pw.PwTcodTranslator class method), 334
- get\_or\_create() (aiida.orm.data.cif.CifData class method), 203, 297
- get\_or\_create() (aiida.orm.data.upf.UpfData class method), 203, 297
- get\_outputs() (aiida.orm.node.Node method), 179, 273
- get\_outputs\_dict() (aiida.orm.node.Node method), 179, 273
- get\_parameter() (aiida.orm.workflow.Workflow method), 184, 278
- get\_parameters() (aiida.orm.workflow.Workflow method), 184, 278
- get\_parsed\_cif() (aiida.tools.dbimporters.baseclasses.CifEntry method), 320
- get\_parser\_name() (aiida.orm.calculation.job.JobCalculation method), 214, 308
- get\_parser\_settings\_key() (aiida.parsers.plugins.quantumpresso.basicpw.BasicpwParser method), 223, 317
- get\_parserclass() (aiida.orm.calculation.job.JobCalculation method), 214, 308

- get\_positions() (aiida.orm.data.array.trajectory.TrajectoryData method), 208, 302
- get\_prepend\_text() (aiida.orm.calculation.job.JobCalculation method), 214, 308
- get\_prepend\_text() (aiida.orm.code.Code method), 189, 282
- get\_priority() (aiida.orm.calculation.job.JobCalculation method), 214, 308
- get\_pseudos\_from\_structure() (in module aiida.orm.data.upf), 202, 296
- get\_pymatgen() (aiida.orm.data.structure.StructureData method), 196, 289
- get\_pymatgen\_molecule() (aiida.orm.data.structure.StructureData method), 196, 289
- get\_pymatgen\_structure() (aiida.orm.data.structure.StructureData method), 196, 289
- get\_pymatgen\_version() (in module aiida.orm.data.structure), 199, 293
- get\_query\_dict() (aiida.djsite.db.models.DbMultipleValueAttributeBase class method), 265
- get\_queue\_name() (aiida.orm.calculation.job.JobCalculation method), 214, 308
- get\_raw() (aiida.orm.data.structure.Kind method), 192, 285
- get\_raw() (aiida.orm.data.structure.Site method), 193, 287
- get\_raw\_cif() (aiida.tools.dbimporters.baseclasses.CifEntry method), 320
- get\_report() (aiida.orm.workflow.Workflow method), 184, 278
- get\_repository\_folder() (in module aiida.common.utils), 237
- get\_resources() (aiida.orm.calculation.job.JobCalculation method), 214, 308
- get\_result() (aiida.orm.workflow.Workflow method), 184, 278
- get\_results() (aiida.orm.workflow.Workflow method), 184, 278
- get\_retrieved\_node() (aiida.orm.calculation.job.JobCalculation method), 214, 308
- get\_scheduler\_error() (aiida.orm.calculation.job.JobCalculation method), 214, 308
- get\_scheduler\_output() (aiida.orm.calculation.job.JobCalculation method), 214, 308
- get\_scheduler\_state() (aiida.orm.calculation.job.JobCalculation method), 214, 308
- get\_shape() (aiida.orm.data.array.ArrayData method), 206, 300
- get\_short\_doc() (aiida.scheduler.\_\_init\_\_.Scheduler class method), 252
- get\_short\_doc() (aiida.transport.\_\_init\_\_.Transport class method), 242
- get\_show\_plugins() (aiida.cmdline.commands.data.Visualizable method), 261
- get\_simple\_name() (aiida.djsite.db.models.DbNode method), 266
- get\_site\_kindnames() (aiida.orm.data.structure.StructureData method), 196, 290
- get\_software\_executable\_path() (aiida.tools.dbexporters.tcod\_plugins.BaseTcodtranslator class method), 331
- get\_software\_package() (aiida.tools.dbexporters.tcod\_plugins.BaseTcodtranslator class method), 331
- get\_software\_package() (aiida.tools.dbexporters.tcod\_plugins.cp.CpTcodtranslator class method), 332
- get\_software\_package() (aiida.tools.dbexporters.tcod\_plugins.nwcpymatgen.NwcpymatgenT class method), 332
- get\_software\_package() (aiida.tools.dbexporters.tcod\_plugins.pw.PwTcodtranslator class method), 334
- get\_software\_package\_compilation\_timestamp() (aiida.tools.dbexporters.tcod\_plugins.BaseTcodtranslator class method), 332
- get\_software\_package\_compilation\_timestamp() (aiida.tools.dbexporters.tcod\_plugins.nwcpymatgen.NwcpymatgenT class method), 332
- get\_software\_package\_version() (aiida.tools.dbexporters.tcod\_plugins.BaseTcodtranslator class method), 332
- get\_software\_package\_version() (aiida.tools.dbexporters.tcod\_plugins.nwcpymatgen.NwcpymatgenT class method), 332
- get\_spacegroup\_numbers() (aiida.orm.data.cif.CifData method), 203, 297
- get\_state() (aiida.orm.calculation.job.JobCalculation method), 214, 308
- get\_state() (aiida.orm.workflow.Workflow method), 184, 278
- get\_step() (aiida.orm.workflow.Workflow method), 184, 278
- get\_step\_calculations() (aiida.orm.workflow.Workflow method), 185, 278
- get\_step\_data() (aiida.orm.data.array.trajectory.TrajectoryData method), 208, 302
- get\_step\_index() (aiida.orm.data.array.trajectory.TrajectoryData method), 208, 302



<code>get_step_workflows()</code> (aiida.orm.workflow.Workflow method), 185, 278	<code>get_topdir()</code> (aiida.common.folders.RepositoryFolder method), 235
<code>get_steps()</code> (aiida.orm.data.array.trajectory.TrajectoryData method), 208, 302	<code>get_tot_num_mpiprocs()</code> (aiida.scheduler.datastructures.JobResource method), 254
<code>get_steps()</code> (aiida.orm.workflow.Workflow method), 185, 278	<code>get_tot_num_mpiprocs()</code> (aiida.scheduler.datastructures.NodeNumberJobResource method), 255
<code>get_structuredata()</code> (aiida.tools.codespecific.quantumespresso.pwinputparser.DbImporter method), 336	<code>get_tot_num_mpiprocs()</code> (aiida.scheduler.datastructures.ParEnvJobResource method), 256
<code>get_subclass_from_dbnode()</code> (aiida.orm.workflow.Workflow class method), 185, 278	<code>get_total_energy()</code> (aiida.tools.dbexporters.tcod_plugins.BaseTcodtranslator class method), 332
<code>get_subclass_from_pk()</code> (aiida.orm.node.Node class method), 180, 273	<code>get_total_energy()</code> (aiida.tools.dbexporters.tcod_plugins.pw.PwTcodtranslator class method), 334
<code>get_subclass_from_pk()</code> (aiida.orm.workflow.Workflow class method), 185, 278	<code>get_transport()</code> (aiida.djsite.db.models.DbAuthInfo method), 263
<code>get_subclass_from_uuid()</code> (aiida.orm.node.Node class method), 180, 273	<code>get_unique_filename()</code> (in module aiida.common.utils), 238
<code>get_subclass_from_uuid()</code> (aiida.orm.workflow.Workflow class method), 185, 278	<code>get_upf_family_names()</code> (aiida.orm.data.upf.UpfData method), 202, 296
<code>get_subfolder()</code> (aiida.common.folders.Folder method), 234	<code>get_upf_group()</code> (aiida.orm.data.upf.UpfData class method), 202, 296
<code>get_submit_script()</code> (aiida.scheduler.__init__.Scheduler method), 252	<code>get_upf_groups()</code> (aiida.orm.data.upf.UpfData class method), 202, 296
<code>get_suggestion()</code> (in module aiida.common.utils), 238	<code>get_upf_node()</code> (aiida.tools.dbimporters.baseclasses.UpfEntry method), 322
<code>get_supported_keywords()</code> (aiida.tools.dbimporters.baseclasses.DbImporter method), 320	<code>get_user()</code> (aiida.orm.node.Node method), 180, 273
<code>get_supported_keywords()</code> (aiida.tools.dbimporters.plugins.cod.CodDbImporter method), 322	<code>get_valid_auth_params()</code> (aiida.transport.__init__.Transport class method), 242
<code>get_supported_keywords()</code> (aiida.tools.dbimporters.plugins.icsd.IcsdDbImporter method), 323	<code>get_valid_fields()</code> (aiida.common.extendeddicts.FixedFieldsAttributeDict class method), 232
<code>get_supported_keywords()</code> (aiida.tools.dbimporters.plugins.mpod.MpodDbImporter method), 325	<code>get_valid_keys()</code> (aiida.scheduler.datastructures.JobResource class method), 254
<code>get_supported_keywords()</code> (aiida.tools.dbimporters.plugins.nninc.NnincDbImporter method), 326	<code>get_valid_keys()</code> (aiida.scheduler.datastructures.NodeNumberJobResource class method), 255
<code>get_supported_keywords()</code> (aiida.tools.dbimporters.plugins.oqmd.OqmdDbImporter method), 325	<code>get_valid_pbc()</code> (in module aiida.orm.data.structure), 199, 293
<code>get_symbols()</code> (aiida.orm.data.array.trajectory.TrajectoryData method), 209, 302	<code>get_valid_transports()</code> (aiida.transport.__init__.Transport class method), 242
<code>get_symbols_set()</code> (aiida.orm.data.structure.StructureData method), 196, 290	<code>get_value_for_node()</code> (aiida.djsite.db.models.DbAttributeBaseClass class method), 262
<code>get_symbols_string()</code> (aiida.orm.data.structure.Kind method), 192, 285	<code>get_velocities()</code> (aiida.orm.data.array.trajectory.TrajectoryData method), 209, 303
<code>get_symbols_string()</code> (in module aiida.orm.data.structure), 199, 293	<code>get_withmpi()</code> (aiida.orm.calculation.job.JobCalculation method), 215, 309
<code>get_temp_folder()</code> (aiida.orm.workflow.Workflow method), 185, 279	<code>get_workflow_info()</code> (in module aiida.orm.workflow), 187, 281
<code>get_times()</code> (aiida.orm.data.array.trajectory.TrajectoryData method), 209, 303	<code>getcwd()</code> (aiida.transport.__init__.Transport method), 242
	<code>getfile()</code> (aiida.transport.__init__.Transport method), 242

- getJobs() (aiida.scheduler.\_\_init\_\_.Scheduler method), 251
- gettree() (aiida.transport.\_\_init\_\_.Transport method), 242
- getvalue() (aiida.djsite.db.models.DbMultipleValueAttributeBaseClass method), 265
- glob() (aiida.transport.\_\_init\_\_.Transport method), 242
- gotocomputer\_command() (aiida.transport.\_\_init\_\_.Transport method), 242
- group\_symbols() (in module aiida.orm.data.structure), 199, 293
- grouper() (in module aiida.common.utils), 238
- gunzip\_string() (in module aiida.common.utils), 238
- gzip\_string() (in module aiida.common.utils), 238
- ## H
- has\_ase() (in module aiida.orm.data.structure), 200, 293
- has\_attached\_hydrogens() (aiida.orm.data.cif.CifData method), 203, 297
- has\_children (aiida.orm.node.Node attribute), 180, 273
- has\_failed() (aiida.orm.calculation.job.JobCalculation method), 215, 309
- has\_failed() (aiida.orm.workflow.Workflow method), 185, 279
- has\_finished\_ok() (aiida.orm.calculation.job.JobCalculation method), 215, 309
- has\_finished\_ok() (aiida.orm.workflow.Workflow method), 185, 279
- has\_key() (aiida.djsite.db.models.DbAttributeBaseClass class method), 262
- has\_parents (aiida.orm.node.Node attribute), 180, 274
- has\_partial\_occupancies() (aiida.orm.data.cif.CifData method), 204, 297
- has\_pycifrw() (in module aiida.orm.data.cif), 204, 298
- has\_pymatgen() (in module aiida.orm.data.structure), 200, 293
- has\_pyspglib() (in module aiida.orm.data.structure), 200, 293
- has\_step() (aiida.orm.workflow.Workflow method), 185, 279
- has\_vacancies() (aiida.orm.data.structure.Kind method), 192, 286
- has\_vacancies() (aiida.orm.data.structure.StructureData method), 196, 290
- has\_vacancies() (in module aiida.orm.data.structure), 200, 293
- Help (class in aiida.cmdline.verdilib), 257
- ## I
- IcsdDbImporter (class in aiida.tools.dbimporters.plugins.icsd), 322
- IcsdEntry (class in aiida.tools.dbimporters.plugins.icsd), 323
- IcsdSearchResults (class in aiida.tools.dbimporters.plugins.icsd), 324
- iglob() (aiida.transport.\_\_init\_\_.Transport method), 242
- Importable (class in aiida.cmdline.commands.data), 260
- importfile() (aiida.orm.data.Data method), 190, 284
- importstring() (aiida.orm.data.Data method), 190, 284
- info() (aiida.orm.workflow.Workflow method), 185, 279
- InlineCalculation (class in aiida.orm.calculation.inline), 210, 304
- inp (aiida.orm.node.Node attribute), 180, 274
- InputValidationError, 230
- insert\_path() (aiida.common.folders.Folder method), 234
- Install (class in aiida.cmdline.verdilib), 257
- InternalError, 230
- InvalidOperation, 230
- is\_alloy() (aiida.orm.data.structure.Kind method), 192, 286
- is\_alloy() (aiida.orm.data.structure.StructureData method), 196, 290
- is\_ase\_atoms() (in module aiida.orm.data.structure), 200, 293
- is\_daemon\_user() (in module aiida.cmdline.commands.daemon), 259
- is\_empty() (aiida.orm.data.remote.RemoteData method), 205, 299
- is\_local() (aiida.orm.code.Code method), 189, 282
- is\_new() (aiida.orm.workflow.Workflow method), 185, 279
- is\_running() (aiida.orm.workflow.Workflow method), 185, 279
- is\_subworkflow() (aiida.djsite.db.models.DbWorkflow method), 267
- is\_subworkflow() (aiida.orm.workflow.Workflow method), 185, 279
- is\_user\_configured() (aiida.orm.computer.Computer method), 175, 269
- is\_user\_enabled() (aiida.orm.computer.Computer method), 176, 269
- is\_valid\_symbol() (in module aiida.orm.data.structure), 200, 294
- isdir() (aiida.common.folders.Folder method), 234
- isdir() (aiida.transport.\_\_init\_\_.Transport method), 243
- isfile() (aiida.common.folders.Folder method), 234
- isfile() (aiida.transport.\_\_init\_\_.Transport method), 243
- iterarrays() (aiida.orm.data.array.ArrayData method), 206, 300
- iterattrs() (aiida.orm.node.Node method), 180, 274
- iterextras() (aiida.orm.node.Node method), 180, 274
- ## J
- JobCalculation (class in aiida.orm.calculation.job), 213, 307
- JobInfo (class in aiida.scheduler.datastructures), 252
- JobResource (class in aiida.scheduler.datastructures), 253

JobTemplate (class in aiida.scheduler.datastructures), 254

## K

keys() (aiida.orm.data.parameter.ParameterData method), 205, 299

kill() (aiida.orm.calculation.job.JobCalculation method), 215, 309

kill() (aiida.orm.workflow.Workflow method), 185, 279

kill() (aiida.scheduler.\_\_init\_\_.Scheduler method), 252

kill\_all() (in module aiida.orm.workflow), 187, 281

kill\_daemon() (aiida.cmdline.commands.daemon.Daemon method), 259

kill\_from\_pk() (in module aiida.orm.workflow), 187, 281

kill\_from\_uuid() (in module aiida.orm.workflow), 188, 281

kill\_step\_calculations() (aiida.orm.workflow.Workflow method), 186, 279

Kind (class in aiida.orm.data.structure), 191, 285

kind\_name (aiida.orm.data.structure.Site attribute), 193, 287

kinds (aiida.orm.data.structure.StructureData attribute), 196, 290

KpointsData (class in aiida.orm.data.array.kpoints), 206, 300

## L

label (aiida.orm.node.Node attribute), 180, 274

label (aiida.orm.workflow.Workflow attribute), 186, 279

labels (aiida.orm.data.array.kpoints.KpointsData attribute), 207, 301

LicensingException, 231

limit\_pk() (aiida.orm.querytool.QueryTool method), 319

list() (aiida.cmdline.commands.data.Listable method), 260

list\_all\_node\_elements() (aiida.djsite.db.models.DbAttributeBaseClass class method), 262

list\_for\_plugin() (aiida.orm.code.Code class method), 189, 282

list\_names() (aiida.orm.computer.Computer class method), 176, 269

Listable (class in aiida.cmdline.commands.data), 260

listdir() (aiida.transport.\_\_init\_\_.Transport method), 243

ListParams (class in aiida.cmdline.verdilib), 257

load\_node() (in module aiida.orm), 174, 268

load\_plugin() (in module aiida.common.pluginloader), 236

load\_workflow() (in module aiida.orm), 175, 268

LockPresent, 231

logger (aiida.orm.calculation.Calculation attribute), 210, 304

logger (aiida.orm.node.Node attribute), 180, 274

logger (aiida.orm.workflow.Workflow attribute), 186, 279

logger (aiida.scheduler.\_\_init\_\_.Scheduler attribute), 252

logger (aiida.transport.\_\_init\_\_.Transport attribute), 243

logging (aiida.orm.computer.Computer attribute), 176, 269

long\_field\_length() (aiida.djsite.db.models.DbMultipleValueAttributeBaseClass method), 265

## M

MachineInfo (class in aiida.scheduler.datastructures), 255

make\_inline() (in module aiida.orm.calculation.inline), 210, 305

makedirs() (aiida.transport.\_\_init\_\_.Transport method), 243

mass (aiida.orm.data.structure.Kind attribute), 192, 286

md5\_file() (in module aiida.common.utils), 238

MissingPluginError, 231

mkdir() (aiida.transport.\_\_init\_\_.Transport method), 243

mode\_dir (aiida.common.folders.Folder attribute), 234

mode\_file (aiida.common.folders.Folder attribute), 234

ModificationNotAllowed, 231

MpodDbImporter (class in aiida.tools.dbimporters.plugins.mpod), 325

MpodEntry (class in aiida.tools.dbimporters.plugins.mpod), 325

MpodSearchResults (class in aiida.tools.dbimporters.plugins.mpod), 325

mtime (aiida.orm.node.Node attribute), 181, 274

MultipleObjectsError, 231

## N

name (aiida.orm.data.structure.Kind attribute), 192, 286

new\_calc() (aiida.orm.code.Code method), 189, 283

next() (aiida.orm.workflow.Workflow method), 186, 279

next() (aiida.tools.dbimporters.baseclasses.DbSearchResults method), 322

next() (aiida.tools.dbimporters.plugins.icsd.IcsdSearchResults method), 324

NnincDbImporter (class in aiida.tools.dbimporters.plugins.nninc), 326

NnincEntry (class in aiida.tools.dbimporters.plugins.nninc), 327

NnincSearchResults (class in aiida.tools.dbimporters.plugins.nninc), 327

Node (class in aiida.orm.node), 177, 270

NodeInputManager (class in aiida.orm.node), 182, 276

NodeNumberJobResource (class in aiida.scheduler.datastructures), 255

NodeOutputManager (class in aiida.orm.node), 182, 276

NoResultsWebExp, 324

normalize() (aiida.transport.\_\_init\_\_.Transport method), 243

NotExistent, 231

numsites (aiida.orm.data.array.trajectory.TrajectoryData attribute), 209, 303



[numsteps](#) ([aiida.orm.data.array.trajectory.TrajectoryData](#) attribute), 209, 303  
[NwcpymatgenTcodtranslator](#) (class in [aiida.tools.dbexporters.tcod\\_plugins.nwcpymatgen](#)), 332  
**O**  
[open\(\)](#) ([aiida.common.folders.Folder](#) method), 235  
[open\(\)](#) ([aiida.transport.\\_\\_init\\_\\_.Transport](#) method), 244  
[optional\\_inline\(\)](#) (in module [aiida.orm.calculation.inline](#)), 212, 306  
[OqmdDbImporter](#) (class in [aiida.tools.dbimporters.plugins.oqmd](#)), 325  
[OqmdEntry](#) (class in [aiida.tools.dbimporters.plugins.oqmd](#)), 325  
[OqmdSearchResults](#) (class in [aiida.tools.dbimporters.plugins.oqmd](#)), 325  
[out](#) ([aiida.orm.node.Node](#) attribute), 181, 274  
**P**  
[ParameterData](#) (class in [aiida.orm.data.parameter](#)), 205, 298  
[ParEnvJobResource](#) (class in [aiida.scheduler.datastructures](#)), 255  
[parse\\_atomic\\_positions\(\)](#) (in module [aiida.tools.codespecific.quantumespresso.pwinputparser](#)), 337  
[parse\\_atomic\\_species\(\)](#) (in module [aiida.tools.codespecific.quantumespresso.pwinputparser](#)), 337  
[parse\\_cell\\_parameters\(\)](#) (in module [aiida.tools.codespecific.quantumespresso.pwinputparser](#)), 338  
[parse\\_cp\\_text\\_output\(\)](#) (in module [aiida.parsers.plugins.quantumespresso.basic\\_raw\\_parser\\_cp](#)), 220, 315  
[parse\\_cp\\_traj\\_stanzas\(\)](#) (in module [aiida.parsers.plugins.quantumespresso.basic\\_raw\\_parser\\_cp](#)), 221, 315  
[parse\\_cp\\_xml\\_counter\\_output\(\)](#) (in module [aiida.parsers.plugins.quantumespresso.basic\\_raw\\_parser\\_cp](#)), 221, 315  
[parse\\_cp\\_xml\\_output\(\)](#) (in module [aiida.parsers.plugins.quantumespresso.basic\\_raw\\_parser\\_cp](#)), 221, 315  
[parse\\_formula\(\)](#) (in module [aiida.orm.data.cif](#)), 204, 298  
[parse\\_k\\_points\(\)](#) (in module [aiida.tools.codespecific.quantumespresso.pwinputparser](#)), 338  
[parse\\_namelists\(\)](#) (in module [aiida.tools.codespecific.quantumespresso.pwinputparser](#)), 339  
[parse\\_profile\(\)](#) (in module [aiida.cmdline.verdilib](#)), 258  
[parse\\_pw\\_text\\_output\(\)](#) (in module [aiida.parsers.plugins.quantumespresso.basic\\_raw\\_parser\\_pw](#)), 221, 316  
[parse\\_pw\\_xml\\_output\(\)](#) (in module [aiida.parsers.plugins.quantumespresso.basic\\_raw\\_parser\\_pw](#)), 222, 316  
[parse\\_QE\\_errors\(\)](#) (in module [aiida.parsers.plugins.quantumespresso.basic\\_raw\\_parser\\_pw](#)), 221, 315  
[parse\\_raw\\_output\(\)](#) (in module [aiida.parsers.plugins.quantumespresso.basic\\_raw\\_parser\\_pw](#)), 222, 316  
[parse\\_upf\(\)](#) (in module [aiida.orm.data.upf](#)), 202, 296  
[parse\\_with\\_retrieved\(\)](#) ([aiida.parsers.plugins.quantumespresso.basicpw.BasicpwParser](#) method), 223, 317  
[parse\\_with\\_retrieved\(\)](#) ([aiida.parsers.plugins.quantumespresso.cp.CpParser](#) method), 223, 317  
[ParsingError](#), 231  
[path\\_exists\(\)](#) ([aiida.transport.\\_\\_init\\_\\_.Transport](#) method), 244  
[pbc](#) ([aiida.orm.data.array.kpoints.KpointsData](#) attribute), 207, 301  
[pbc](#) ([aiida.orm.data.structure.StructureData](#) attribute), 196, 290  
[PcodDbImporter](#) (class in [aiida.tools.dbimporters.plugins.pcod](#)), 326  
[PcodEntry](#) (class in [aiida.tools.dbimporters.plugins.pcod](#)), 326  
[PcodSearchResults](#) (class in [aiida.tools.dbimporters.plugins.pcod](#)), 326  
[pk](#) ([aiida.orm.computer.Computer](#) attribute), 176, 269  
[pk](#) ([aiida.orm.node.Node](#) attribute), 181, 274  
[pk](#) ([aiida.orm.workflow.Workflow](#) attribute), 186, 280  
[PluginInternalError](#), 231  
[position](#) ([aiida.orm.data.structure.Site](#) attribute), 193, 287  
[prepare\\_for\\_retrieval\\_and\\_parsing\(\)](#) ([aiida.orm.calculation.job.quantumespresso.pwimmigrant.Pwimmigrant](#) method), 219, 313  
[ProfileConfigurationError](#), 231  
[ProfileParsingException](#), 257  
[put\(\)](#) ([aiida.transport.\\_\\_init\\_\\_.Transport](#) method), 244  
[putfile\(\)](#) ([aiida.transport.\\_\\_init\\_\\_.Transport](#) method), 244  
[puttree\(\)](#) ([aiida.transport.\\_\\_init\\_\\_.Transport](#) method), 244  
[PwCalculation](#) (class in [aiida.orm.calculation.job.quantumespresso.pw](#)), 217, 311  
[PwimmigrantCalculation](#) (class in [aiida.orm.calculation.job.quantumespresso.pwimmigrant](#)), 217, 311  
[PwInputFile](#) (class in [aiida.tools.codespecific.quantumespresso.pwinputparser](#)), 334

PwTcodtranslator (class in aiida.tools.dbexporters.tcod\_plugins.pw), 333  
 pycifrw\_from\_cif() (in module aiida.orm.data.cif), 204, 298

## Q

query() (aiida.cmdline.commands.data.Listable method), 260  
 query() (aiida.orm.node.Node class method), 181, 274  
 query() (aiida.orm.workflow.Workflow class method), 186, 280  
 query() (aiida.tools.dbimporters.baseclasses.DbImporter method), 320  
 query() (aiida.tools.dbimporters.plugins.cod.CodDbImporter method), 322  
 query() (aiida.tools.dbimporters.plugins.icsd.IcsdDbImporter method), 323  
 query() (aiida.tools.dbimporters.plugins.mpod.MpodDbImporter method), 325  
 query() (aiida.tools.dbimporters.plugins.nninc.NnincDbImporter method), 326  
 query() (aiida.tools.dbimporters.plugins.oqmd.OqmdDbImporter method), 325  
 query() (aiida.tools.dbimporters.plugins.pcod.PcodDbImporter method), 326  
 query() (aiida.tools.dbimporters.plugins.tcod.TcodDbImporter method), 326  
 query\_db\_version() (aiida.tools.dbimporters.plugins.icsd.IcsdSearchResults method), 324  
 query\_get() (aiida.tools.dbimporters.plugins.mpod.MpodDbImporter method), 325  
 query\_get() (aiida.tools.dbimporters.plugins.nninc.NnincDbImporter method), 327  
 query\_get() (aiida.tools.dbimporters.plugins.oqmd.OqmdDbImporter method), 325  
 query\_group() (aiida.cmdline.commands.data.Listable method), 260  
 query\_page() (aiida.tools.dbimporters.plugins.icsd.IcsdSearchResults method), 324  
 query\_past\_days() (aiida.cmdline.commands.data.Listable method), 260  
 query\_sql() (aiida.tools.dbimporters.plugins.cod.CodDbImporter method), 322  
 query\_sql() (aiida.tools.dbimporters.plugins.pcod.PcodDbImporter method), 326  
 QueryTool (class in aiida.orm.querytool), 317

## R

RemoteData (class in aiida.orm.data.remote), 205, 299  
 RemoteOperationError, 231  
 remove() (aiida.transport.\_\_init\_\_.Transport method), 244

remove\_path() (aiida.common.folders.Folder method), 235  
 remove\_path() (aiida.orm.node.Node method), 181, 274  
 remove\_path() (aiida.orm.workflow.Workflow method), 186, 280  
 rename() (aiida.transport.\_\_init\_\_.Transport method), 244  
 replace\_with\_folder() (aiida.common.folders.Folder method), 235  
 replace\_with\_folder() (aiida.orm.data.folder.FolderData method), 201, 295  
 repo\_folder (aiida.orm.workflow.Workflow attribute), 186, 280  
 RepositoryFolder (class in aiida.common.folders), 235  
 res (aiida.orm.calculation.job.JobCalculation attribute), 215, 309  
 reset\_cell() (aiida.orm.data.structure.StructureData method), 197, 290  
 reset\_mass() (aiida.orm.data.structure.Kind method), 192, 286  
 reset\_sites\_positions() (aiida.orm.data.structure.StructureData method), 197, 290  
 retrieve\_computed\_for\_authinfo() (in module aiida.execmanager), 261  
 retrieve\_jobs() (in module aiida.execmanager), 261  
 rmdir() (aiida.transport.\_\_init\_\_.Transport method), 244  
 rmtree() (aiida.transport.\_\_init\_\_.Transport method), 244  
 Run (class in aiida.cmdline.verdilib), 258  
 run() (aiida.cmdline.baseclass.VerdiCommand method), 256  
 run() (aiida.cmdline.verdilib.CompletionCommand method), 257  
 run\_query() (aiida.orm.querytool.QueryTool method), 319  
 Runserver (class in aiida.cmdline.verdilib), 258

## S

SearchResultsFolder (class in aiida.common.folders), 235  
 Scheduler (class in aiida.scheduler.\_\_init\_\_), 251  
 SchedulerFactory() (in module aiida.scheduler.\_\_init\_\_), 252  
 set\_location (aiida.common.folders.RepositoryFolder attribute), 235  
 set\_location (aiida.orm.node.Node method), 181, 274  
 set\_append\_text() (aiida.orm.calculation.job.JobCalculation method), 215, 309  
 set\_append\_text() (aiida.orm.code.Code method), 189, 283  
 set\_array() (aiida.orm.data.array.ArrayData method), 206, 300  
 set\_ase() (aiida.orm.data.structure.StructureData method), 197, 291

<code>set_automatic_kind_name()</code> (aiida.orm.data.structure.Kind method), 192, 286	<code>set_mpirun_extra_params()</code> (aiida.orm.calculation.job.JobCalculation method), 216, 310
<code>set_cell()</code> (aiida.orm.data.array.kpoints.KpointsData method), 207, 301	<code>set_output_file_name()</code> (aiida.orm.calculation.job.quantumpresso.pwimmigrant.PwimmigrantCalculation method), 219, 313
<code>set_cell_from_structure()</code> (aiida.orm.data.array.kpoints.KpointsData method), 207, 301	<code>set_output_subfolder()</code> (aiida.orm.calculation.job.quantumpresso.pwimmigrant.PwimmigrantCalculation method), 219, 313
<code>set_class()</code> (aiida.orm.querytool.QueryTool method), 319	<code>set_params()</code> (aiida.orm.workflow.Workflow method), 186, 280
<code>set_computer()</code> (aiida.orm.node.Node method), 181, 275	<code>set_parser_name()</code> (aiida.orm.calculation.job.JobCalculation method), 216, 310
<code>set_custom_scheduler_commands()</code> (aiida.orm.calculation.job.JobCalculation method), 215, 309	<code>set_prefix()</code> (aiida.orm.calculation.job.quantumpresso.pwimmigrant.PwimmigrantCalculation method), 219, 313
<code>set_default_mpi_procs_per_machine()</code> (aiida.orm.computer.Computer method), 176, 269	<code>set_prepend_text()</code> (aiida.orm.calculation.job.JobCalculation method), 216, 310
<code>set_dict()</code> (aiida.orm.data.parameter.ParameterData method), 205, 299	<code>set_prepend_text()</code> (aiida.orm.code.Code method), 190, 283
<code>set_environment_variables()</code> (aiida.orm.calculation.job.JobCalculation method), 215, 309	<code>set_priority()</code> (aiida.orm.calculation.job.JobCalculation method), 216, 310
<code>set_extra()</code> (aiida.orm.node.Node method), 181, 275	<code>set_pymatgen()</code> (aiida.orm.data.structure.StructureData method), 197, 291
<code>set_extras()</code> (aiida.orm.node.Node method), 182, 275	<code>set_pymatgen_molecule()</code> (aiida.orm.data.structure.StructureData method), 197, 291
<code>set_file()</code> (aiida.orm.data.cif.CifData method), 204, 297	<code>set_pymatgen_structure()</code> (aiida.orm.data.structure.StructureData method), 197, 291
<code>set_file()</code> (aiida.orm.data.singlefile.SinglefileData method), 201, 295	<code>set_queue_name()</code> (aiida.orm.calculation.job.JobCalculation method), 216, 310
<code>set_file()</code> (aiida.orm.data.upf.UpfData method), 202, 296	<code>set_remote_computer_exec()</code> (aiida.orm.code.Code method), 190, 283
<code>set_files()</code> (aiida.orm.code.Code method), 189, 283	<code>set_remote_workdir()</code> (aiida.orm.calculation.job.JobCalculation method), 220, 314
<code>set_group()</code> (aiida.orm.querytool.QueryTool method), 319	<code>set_resources()</code> (aiida.orm.calculation.job.JobCalculation method), 216, 310
<code>set_import_sys_environment()</code> (aiida.orm.calculation.job.JobCalculation method), 216, 310	<code>set_source()</code> (aiida.orm.data.Data method), 190, 284
<code>set_input_file_name()</code> (aiida.orm.calculation.job.quantumpresso.pwimmigrant.PwimmigrantCalculation method), 219, 313	<code>set_state()</code> (aiida.orm.workflow.Workflow method), 186, 280
<code>set_input_plugin_name()</code> (aiida.orm.code.Code method), 189, 283	<code>set_structurereadlist()</code> (aiida.orm.data.array.trajectory.TrajectoryData method), 209, 303
<code>set_kpoints()</code> (aiida.orm.data.array.kpoints.KpointsData method), 207, 301	<code>set_symbols_and_weights()</code> (aiida.orm.data.structure.Kind method), 192, 286
<code>set_kpoints_mesh()</code> (aiida.orm.data.array.kpoints.KpointsData method), 208, 301	<code>set_trajectory()</code> (aiida.orm.data.array.trajectory.TrajectoryData method), 209, 303
<code>set_local_executable()</code> (aiida.orm.code.Code method), 189, 283	<code>set_transport()</code> (aiida.scheduler.__init__.Scheduler method), 252
<code>set_max_memory_kb()</code> (aiida.orm.calculation.job.JobCalculation method), 216, 310	<code>set_value()</code> (aiida.djsite.db.models.DbMultipleValueAttributeBaseClass class method), 265
<code>set_max_wallclock_seconds()</code> (aiida.orm.calculation.job.JobCalculation method), 216, 310	<code>set_value_for_node()</code> (aiida.djsite.db.models.DbAttributeBaseClass method), 265
<code>set_mpirun_command()</code> (aiida.orm.computer.Computer method), 176, 269	

- class method), 262
  - set\_withmpi() (aiida.orm.calculation.job.JobCalculation method), 216, 310
  - setup\_db() (aiida.tools.dbimporters.baseclasses.DbImporter method), 321
  - setup\_db() (aiida.tools.dbimporters.plugins.cod.CodDbImporter method), 322
  - setup\_db() (aiida.tools.dbimporters.plugins.icsd.IcsdDbImporter method), 323
  - setup\_db() (aiida.tools.dbimporters.plugins.mpod.MpodDbImporter method), 325
  - setup\_db() (aiida.tools.dbimporters.plugins.nninc.NnincDbImporter method), 327
  - setup\_db() (aiida.tools.dbimporters.plugins.oqmd.OqmdDbImporter method), 325
  - sha1\_file() (in module aiida.common.utils), 238
  - Shell (class in aiida.cmdline.verdilib), 258
  - show() (aiida.cmdline.commands.data.Visualizable method), 261
  - SinglefileData (class in aiida.orm.data.singlefile), 201, 295
  - Site (class in aiida.orm.data.structure), 193, 286
  - sites (aiida.orm.data.structure.StructureData attribute), 197, 291
  - sleep() (aiida.orm.workflow.Workflow method), 186, 280
  - sort\_states() (in module aiida.common.datastructures), 230
  - source (aiida.orm.data.Data attribute), 190, 284
  - step() (aiida.orm.workflow.Workflow class method), 186, 280
  - step\_to\_structure() (aiida.orm.data.array.trajectory.TrajectoryData method), 210, 304
  - store() (aiida.orm.calculation.job.JobCalculation method), 216, 310
  - store() (aiida.orm.computer.Computer method), 176, 269
  - store() (aiida.orm.data.cif.CifData method), 204, 298
  - store() (aiida.orm.data.upf.UpfData method), 202, 296
  - store() (aiida.orm.node.Node method), 182, 275
  - store() (aiida.orm.workflow.Workflow method), 187, 280
  - store\_all() (aiida.orm.node.Node method), 182, 275
  - str2val() (in module aiida.tools.codespecific.quantumespresso.pwinputparser), 339
  - str\_timedelta() (in module aiida.common.utils), 239
  - StructureData (class in aiida.orm.data.structure), 193, 287
  - subfolder (aiida.common.folders.RepositoryFolder attribute), 235
  - submit() (aiida.orm.calculation.job.JobCalculation method), 216, 310
  - submit\_calc() (in module aiida.execmanager), 261
  - submit\_from\_script() (aiida.scheduler.\_\_init\_\_.Scheduler method), 252
  - submit\_jobs() (in module aiida.execmanager), 261
  - submit\_jobs\_with\_authinfo() (in module aiida.execmanager), 261
  - submit\_test() (aiida.orm.calculation.job.JobCalculation method), 217, 311
  - subspecifier\_pk (aiida.djsite.db.models.DbMultipleValueAttributeBaseClass attribute), 265
  - subspecifiers\_dict (aiida.djsite.db.models.DbMultipleValueAttributeBaseClass attribute), 266
  - symbol (aiida.orm.data.structure.Kind attribute), 193, 286
  - symbols (aiida.orm.data.structure.Kind attribute), 193, 286
  - sync\_link() (aiida.transport.\_\_init\_\_.Transport method), 244
  - symop\_fract\_from\_ortho() (in module aiida.orm.data.structure), 200, 294
  - symop\_ortho\_from\_fract() (in module aiida.orm.data.structure), 200, 294
  - symop\_string\_from\_symop\_matrix\_tr() (in module aiida.orm.data.cif), 204, 298
- ## T
- TcodDbImporter (class in aiida.tools.dbimporters.plugins.tcod), 326
  - TcodEntry (class in aiida.tools.dbimporters.plugins.tcod), 326
  - TcodSearchResults (class in aiida.tools.dbimporters.plugins.tcod), 326
  - TemplaterplacerCalculation (class in aiida.orm.calculation.job.simpleplugins.templaterplacer), 220, 314
  - TrajectoryData (class in aiida.orm.data.array.trajectory), 208, 302
  - translate\_calculation\_specific\_values() (in module aiida.tools.dbexporters.tcod), 330
  - transport (aiida.scheduler.\_\_init\_\_.Scheduler attribute), 252
  - Transport (class in aiida.transport.\_\_init\_\_), 240
  - TransportFactory() (in module aiida.transport.\_\_init\_\_), 245
  - TransportInternalError, 245
- ## U
- UniquenessError, 231
  - update\_dict() (aiida.orm.data.parameter.ParameterData method), 205, 299
  - update\_environment() (in module aiida.cmdline.verdilib), 258
  - update\_jobs() (in module aiida.execmanager), 261
  - update\_running\_calcs\_status() (in module aiida.execmanager), 262
  - UpfData (class in aiida.orm.data.upf), 201, 295
  - UpfEntry (class in aiida.tools.dbimporters.baseclasses), 322

upload\_upf\_family() (in module aiida.orm.data.upf), 202, 296

uuid (aiida.common.folders.RepositoryFolder attribute), 235

uuid (aiida.orm.computer.Computer attribute), 176, 270

uuid (aiida.orm.node.Node attribute), 182, 275

uuid (aiida.orm.workflow.Workflow attribute), 187, 280

## V

validate() (aiida.common.extendeddicts.DefaultFieldsAttributeDict method), 232

validate() (aiida.orm.computer.Computer method), 176, 270

validate\_key() (aiida.djsite.db.models.DbMultipleValueAttributeBaseClass class method), 266

validate\_list\_of\_string\_tuples() (in module aiida.common.utils), 239

validate\_symbols\_tuple() (in module aiida.orm.data.structure), 200, 294

validate\_weights\_tuple() (in module aiida.orm.data.structure), 200, 294

ValidationError, 231

values (aiida.orm.data.cif.CifData attribute), 204, 298

VerdiCommand (class in aiida.cmdline.baseclass), 256

VerdiCommandWithSubcommands (class in aiida.cmdline.baseclass), 256

Visualizable (class in aiida.cmdline.commands.data), 261

## W

weights (aiida.orm.data.structure.Kind attribute), 193, 286

whoami() (aiida.transport.\_\_init\_\_.Transport method), 245

Workflow (class in aiida.orm.workflow), 182, 276

WorkflowFactory() (in module aiida.orm), 174, 268

WorkflowInputValidationError, 231

WorkflowKillError, 187, 280

WorkflowUnkillable, 187, 281

## X

xyz\_parser\_iterator() (in module aiida.common.utils), 239