obob$_{c}ondor$ $Documentation$

Release 0.0.9

Thomas Hartmann

Jun 02, 2022

CONTENTS

1 What is this? 1

1.1 An introduction to Cluster Computing . . . 1
1.2 How to create a python environment for you cluster jobs . . . 4
1.3 How to use obob_condor . . . 5
1.4 Automatically generating filenames for your jobs . . . 7
1.5 Reference . . . 10

2 Indices and tables 13

Python Module Index 15

Index 17

CHAPTER

ONE

WHAT IS THIS?

You were probably directed to this python package / website because you are going to use the cluster of the group of Prof. Weisz at the University of Salzburg.

Lots of the analyses we run uses lots of RAM and/or takes a long time to compute on a workstation or laptop. A cluster (in general, not specifically this one) provides access to powerful computing resources that are shared between users.

In order to ensure a fair allocation of the resources, you cannot access them directly but instead define so-called `obob_condor.Job` together with how much RAM and how many CPUs your job is going to need and submit that information to the cluster.

The cluster then tries to allocate these resources for you and runs your job.

The software we use here is called HTCondor. It is a very powerful and complex piece of software. This package makes it much easier for you to use the cluster by hiding a lot of the complexities.

If you are already familiar with our cluster infrastructure (because you are migrating from Matlab), you can skip the first section and go directly to *How to use obob_condor*. Otherwise, get *An introduction to Cluster Computing*.

1.1 An introduction to Cluster Computing

1.1.1 Introduction

Welcome to the introduction of the HNC Condor. In order to use it efficiently, you have to be familiar with a few concepts. I try to be as simple and short as possible and provide you with examples. But please bear in mind that the information you find on this page is the absolute minimum! So be sure to understand it or drop by and ask. The introduction will cover the following aspects:

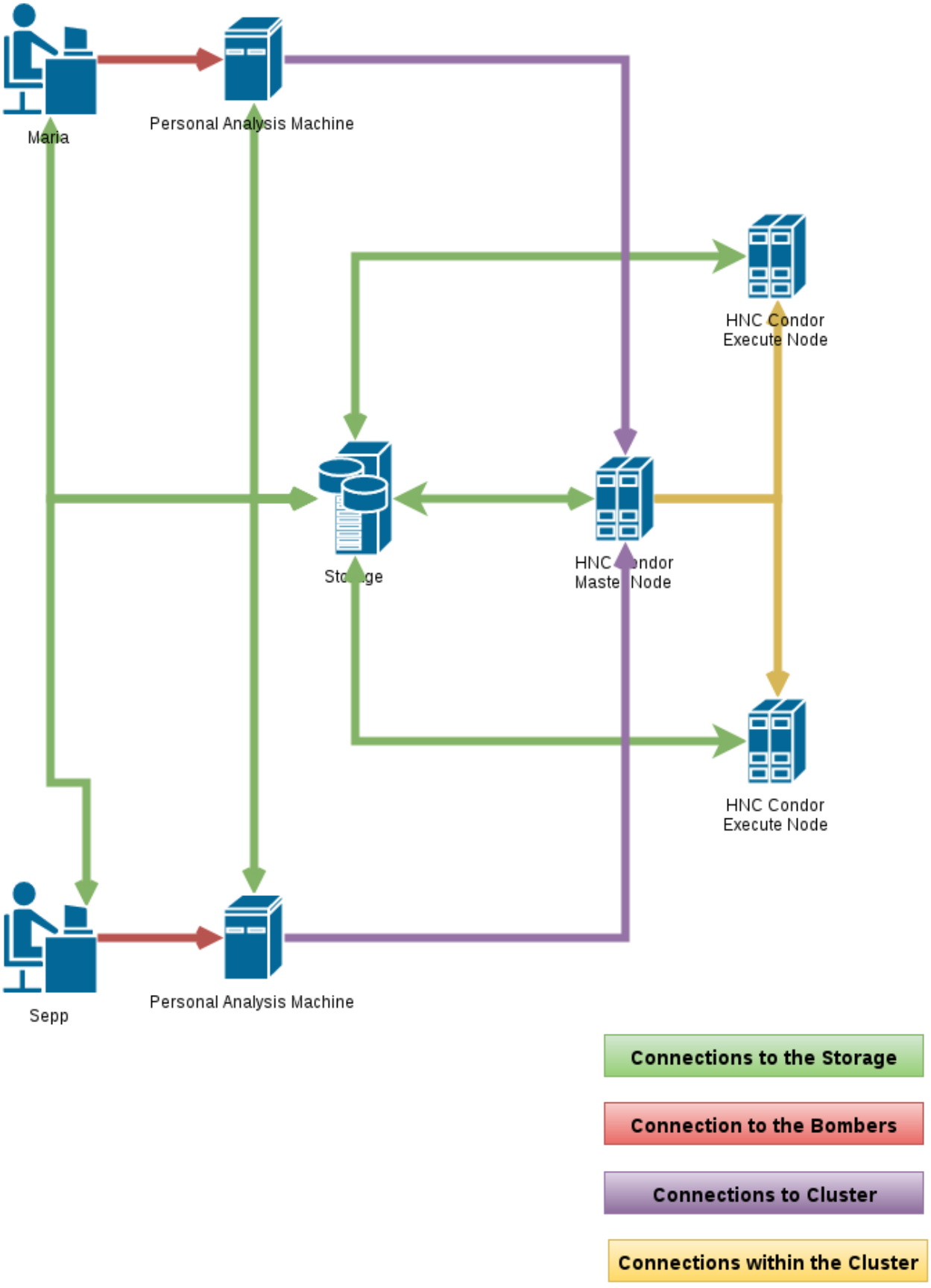
1. What is a cluster?
2. How do I use the cluster?

1.1.2 What is a cluster?

General Remarks

In one short sentence: a cluster is a bunch of computers that are used by a bunch of people. These computers provide resources, in our case CPU power and memory, to all users. These resources have to be distributed as optimal and fair as possible between the users.

Here is a picture of how the system looks like:


Maria
Personal Analysis Machine
HNC Condor
Execute Node
Sto ge
HNC ndor
Maste Node
HNC Condor
Execute Node
Sepp
Personal Analysis Machine
Connections to the Storage
Connection to the Bombers
Connections to Cluster
Connections within the Cluster

As you see, your computer is connected to your Personal Analysis Machine (a.k.a. Bomber). All the Bombers are connected to the HNC Condor Master Node. Through this connection, your Bomber tells the HNC Condor Master Node that it wants it to do a job. You can think of a job as a python function or method that takes some parameters. The Master Node also needs some more information, for instance, how much RAM your job will need. The Master Node then collects all your jobs and the jobs of everybody else who wants to compute things on the cluster. It then asks the HNC Condor Execute Nodes whether they have the resources for the jobs (i.e., enough RAM and CPU). If one of them says yes, he will get one of the jobs and execute it.

Sounds complicated? Don't worry! This `obob_condor` package makes it super easy for you!

About Fairness

Please always bear in mind that the Cluster is a resource that you share with all your colleagues. There are, of course, ways to use the system in your advantage while putting everyone else at a disadvantage. Please just do not! This system works best when everybody has everybody else in mind. And it also increases your Karma™.

As I wrote before, the HNC Condor Master Node collects all the jobs by all the users who want to use the Cluster and then distributes it to the Execute Nodes. It tries to be as fair as possible in the distribution of the jobs. For example, if two people are submitting jobs at the same time, it will make sure that both get half of the resources. However, the Master Node cannot guess how many resources like your jobs need. So, you need to tell him and try to be as exact as possible.

At the moment, the only thing you need to tell the Cluster is how much RAM your job will need. If your job consumes more RAM, it will be put on hold, which means that it will stop being executed. If you specify too much RAM, less of your jobs will run.

1.1.3 How do I use the Cluster?

If you want to use the cluster, all you need to do is to connect to your Bomber.

What is going on on the cluster?

To be able to monitor what is going on on the cluster, you first need to open a terminal. To do this, click on the "Applications Menu" and then on "Terminal Emulation". In the new window enter this command:

```
watch -n4 'condor_q -global'
```

You will see something like this:

```
-- Schedd: cf000016.sbg.ac.at : <141.201.106.7:9618?... @ 06/01/22 08:46:49
OWNER    BATCH_NAME    SUBMITTED   DONE   RUN    IDLE   HOLD  TOTAL JOB_IDS
bAAAAAAA ID: 46449    6/1  08:29      8      4      _      _     12 46449.0-3
bBBBBBBB ID: 46450    6/1  08:32      _      1      _      _      1 46450.0
bCCCCCCC ID: 46451    6/1  08:46      _     30      _      _     30 46451.0-29

Total for query: 35 jobs; 0 completed, 0 removed, 0 idle, 35 running, 0 held, 0 suspended
Total for all users: 35 jobs; 0 completed, 0 removed, 0 idle, 35 running, 0 held, 0␣
↪suspended
```

As you can see, currently 3 Job Clusters are running by 3 different users. Here is what some of the individual columns mean:

Column	Description
OWNER	The username of the person who submitted the job
BATCH_NAME	Every Job Cluster gets an ID which is show here
SUBMITTED	Date and time on which the job was submitted
DONE	How many of the jobs have completed. Sucessfully or not!
RUN	How many jobs are currently running
IDLE	How many jobs are waiting for resources to become available
HOLD	How many jobs are on hold. If something went wrong, HTCondor might put jobs in this condition. You can use `condor_q -hold` to find out what happened.
TOTAL	Total number of jobs in the Job Cluster

How do I submit my first job?

Go to *How to use obob_condor* and find out!

1.2 How to create a python environment for you cluster jobs

1.2.1 Create a new python environment and make sure `obob_condor` gets installed in it

Open a terminal and navigate to your folder on `/mnt/obob`:

```
cd /mnt/obob/staff/thartmann
```

Now create a new folder where you want to store the code and the environment:

```
mkdir my_obob_condor_test
cd my_obob_condor_test
```

Create a file called `environment.yml`. This file lists all the conda and pypi packages your project needs. Take a look here for a description. For this tutorial, it should look like this:

```
channels:
  - defaults
  - conda-forge

dependencies:
  - pip
  - spyder
  - pip:
      - obob_condor
```

You can use any text editor you like, but if you are connected via X2go or remote desktop, the easiest way is to start a nice graphical text editor like this:

```
gedit environment.yml
```

If you wanted to analyze actual data with it, you would probably need to add `mne`, `pandas`, etc. to it. But we can leave it like this.

Copy-paste the content in there, save and exit and go back to the terminal.

You can now create the environment like this:

```
mamba env create -p ./.venv
```

When this is done, you must activate the environment:

```
conda activate ./.venv
```

And then start Spyder (a Python editor) like this:

```
spyder
```

Before you go on, make sure that:

1. The folder that is shown in the box on the top right, just below the menu bar is the folder you just created. If not, click the folder button to the right and navigate to the correct one.
2. Just below, you find a pane which has four different tab options: "Help", "Variable Explorer", "Plots" and "Files". Make sure "File" is selected.

You should now see the `environment.yml` file that we created before.

1.3 How to use obob_condor

Submitting Jobs to the OBOB cluster is quite simple using the obob_condor package.

1.3.1 Make sure you are on your bomber

Accessing the cluster requires the code to be run on the bomber as only those are connected to the cluster.

All the code also needs to be on `/mnt/obob` because the nodes also need access to them.

You also need to be in a python environment that has `obob_condor` in its dependencies. If you do not now what this means, please take a look at *How to create a python environment for you cluster jobs*.

1.3.2 Define a simple job

The first thing you need to do is define what your job should do. You therefore write a class that derives from *obob_condor.Job*. The only things you have to do is to supply a run method.

```
import obob_condor

class MyJob(obob_condor.Job):
    def run(self):
        print('Hello World!')
```

The job class can be anywhere in your sourcecode tree. It can be defined in the script you are running, in a python module or package. As long as you can import it, it is ok.

1.3.3 Define a job that takes arguments

Your job can also take any kind of arguments:

```
import obob_condor

class JobWithArgs(obob_condor.Job):
    def run(self, normal_arg, key_arg='my_default'):
        print('normal_arg=%s\nkey_arg=%s' % (str(normal_arg), str(key_arg)))
```

1.3.4 Getting and configuring the JobCluster

In order to submit your job, you need to get an instance of *obob_condor.JobCluster*. The constructor of this class has a lot of keyword arguments. You can set none, some or all of them. They all have quite sensible defaults:

```
import obob_condor

my_jobs = obob_condor.JobCluster(required_ram='6G')
```

Now we have a JobCluster that asks for 6GB of RAM per Job.

1.3.5 Adding jobs to the JobCluster

In order to add the jobs, use the *obob_condor.JobCluster.add_job()* method:

```
my_jobs.add_job(MyJob)
my_jobs.add_job(JobWithArgs, 'this_is_the_normal_arg', key_arg='and this the key arg')
```

1.3.6 Adding multiple jobs with just one call

A common use case of job submission is that you want to run the same job on a number of different combinations of parameters.

Let's consider a job like this:

```
class AverageData(obob_condor.Job):
    def run(self, subject_id, condition, lp_filter_freq)
        ...
```

And you have a list of subject_ids and conditions:

```
subject_ids = [
    '19800908igdb',
    '19990909klkl',
    '17560127anpr']

conditions = [
    'visual',
    'auditory']
```

We want to run the jobs for all combinations of subject_ids and conditions. This is what *obob_condor.PermuteArgument* is for:

```
from obob_condor import PermuteArgument

my_jobs.add_job(AverageData, PermuteArgument(subject_ids), PermuteArgument(conditions),␣
↪30)
```

This call adds 6 jobs, one for every combination of subject_ids and conditions.

This works for all kinds of arguments (normal ones and keyword arguments).

1.3.7 Submitting the Job

Now, all you need to do is to call submit:

```
my_jobs.submit()
```

For more advanced uses take a look at the *Reference*.

1.4 Automatically generating filenames for your jobs

1.4.1 Introduction

A common pattern for cluster jobs is:

1. They accept a bunch of parameters like:
    1. subject_id
    2. condition
    3. etc…
2. They load the respective data
3. Do some computation
4. Save the data to the storage

A common analysis also has more than one cluster job. Normally, each job stores its output in a separate folder. A good practise is that each subject gets their sub_folder in this job folder. Furthermore, all the job folders are normally in one "meta" folder.

Let's assume our project is called "super_markov". This means, you would have a folder like: `/mnt/obob/staff/hmustermann/super_markov/data`. You might then have a cluster job called `Preprocess`. The data of the subject `19800908igdb` should thus be stored in `/mnt/obob/staff/hmustermann/super_markov/data/Preprocess/19800908igdb/...`

Let's take this one step further and assume, for every subject, we have two conditions, `ordered` and `random` and we run separate jobs for each of them.

So, the final filenames would then look like:

- `/mnt/obob/staff/hmustermann/super_markov/data/Preprocess/19800908igdb/19800908igdb__condition_ordered.dat`
- `/mnt/obob/staff/hmustermann/super_markov/data/Preprocess/19800908igdb/19800908igdb__condition_random.dat`

Does this not look like something we should automate?

And one further issue: A cluster job might just die… Imagine, you submit 100 jobs and 2 of them do not complete. You want to resubmit only the two jobs that failed.

Ok, let's automate that as well!

1.4.2 Say hello to obob_condor.AutomaticFilenameJob

obob_condor.AutomaticFilenameJob is a subclass of *obob_condor.Job*. This means that it basically does all the same things and you use it in a very similar way: You override the `obob_condor.AutomaticFilenameJob.run()` method with the commands that you want to run on the cluster and it will do just that.

However, it introduces some additional methods that automate the file naming process.

For this to work properly, however, you should do some preparations.

1.4.3 Derive a Job class for the project

In order to use a common data folder for the whole project, the easiest way is to create a project specific Job class:

```
from obob_condor import AutomaticFilenameJob


class MyProjectJobs(AutomaticFilenameJob):
    base_data_folder = '/mnt/obob/staff/hmustermann/super_markov/data'
```

You see that the code is really simple. We create the class, derive it from *obob_condor.AutomaticFilenameJob* and the only thing we add to this class is that we set `obob_condor.AutomaticFilenameJob.base_data_folder`.

1.4.4 Write your Job classes for the individual jobs

Writing the individual job classes is as straight forward as you already know with the only exception that we derive those classes from the project wide job class we just created.

```
import joblib # This is a very good library for saving python objects.
from obob_condor import JobCluster, PermuteArgument

class Preprocess(MyProjectJobs):
    job_data_folder = 'Preprocessing'

    def run(self, subject_id, condition):
        # here you load your data
        # now you do you processing

        # now we want to save the data...
        joblib.dump(result_data, self.full_output_path)

job_cluster = JobCluster()
job_cluster.add_job(
   Preprocess,
   subject_id=PermuteArgument(['19800908igdb', '19700809abcd']),
   condition=PermuteArgument(['ordered', 'random'])
```

(continues on next page)

(continued from previous page)

```
)

job_cluster.submit()
```

As you can see, the `Preprocess` job gets gets executed four times (once for every combination of the `subject_id` and `condition` arguments).

Additionally, it set the `job_data_folder` property to define, in which subfolder of `base_data_folder` (which we defined in `MyProjectJobs` above) the data would end up in.

`self.full_output_path` now automatically generate a filename like this:

```
'base_data_folder/job_data_folder/subject_id/subject_id__firstKwargName_value__
↪secondKwargName_value.dat'
```

So, in our case, this would be for the first job:

```
'/mnt/obob/staff/hmustermann/super_markov/data/Preprocessing/19800908igdb/19800908igdb__
↪condition_ordered.dat'
```

1.4.5 Things to keep in mind

All folders are created automatically

If you do not want this, you can set `obob_condor.AutomaticFilenameJob.create_folder` to `False`.

Only keyword arguments are used for filename creation

You must specify the arguments as keyword arguments to be used for the filename when you add the job.

The order of the keyword arguments in the filename is the order to the arguments in the add_job call.

So, keep it constant!

There are more advanced usages

Like you can exclude kwargs from going into the filename. In this case, it might make sense to add a hash value instead. For instance if you iterate over multiple time regions.

For all information, check out the *reference*.

1.5 Reference

class obob_condor.**JobCluster**(*required_ram='2G'*, *adjust_mem=True*, *request_cpus=1*, *jobs_dir='jobs'*, *inc_jobsdir=True*, *owner=None*, *python_bin=None*, *working_directory=None*, *singularity_image=None*)

This is the main class, the *controller* of obob_condor. It collects all the jobs and takes care of submitting them to the cluster. It also contains information about how much RAM the jobs need, how many CPUs are requested etc.

Parameters

- **required_ram** (`str`, `float`, `int`, optional) – The amount of RAM required to run one Job in megabytes. A string like "2G" or "200M" will be converted accordingly.
- **adjust_mem** (`bool`, optional) – If True, the job will be restarted automatically if it gets killed by condor because it uses too much RAM.
- **request_cpus** (`int`, optional) – The number of CPUs requested
- **jobs_dir** (`str`, optional) – Folder to put all the jobs in. This one needs to be on the shared filesystem (so somewhere under /mnt/obob)
- **inc_jobsdir** (`str`, optional) – If this is set to True (default), jobs_dir is the parent folder for all the jobs folders. Each time a job is submitted, a new folder is created in the jobs_dir folder that contains all the necessary files and a folder called "log" containing the log files. If jobs_dir is set to False, the respective files are put directly under jobs_dir. In this case, jobs_dir must either be empty or not exist at all to avoid any side effects.
- **owner** (`str`, optional) – Username the job should run under. If you submit your jobs from one of the bombers, you do not need to set this. If you have set up your local machine to submit jobs and your local username is different from your username on the cluster, set owner to that username.
- **python_bin** (`str`, optional) – The path to the python interpreter that should run the jobs. If you do not set it, it gets chosen automatically. If the python interpreter you are using when submitting the jobs is on /mnt/obob/ that one will be used. If the interpreter you are using is **not** on /mnt/obob/ the default one at /mnt/obob/obob_mne will be used.
- **working_directory** (`str`, optional) – The working directory when the jobs run.
- **singularity_image** (`str`, optional) – Set this to a singularity image to have the jobs execute in it. Can be a link to a local file or to some online repository.

add_job(*job*, **args*, ***kwargs*)

Add one job to the JobCluster. All further arguments will be passed on to the Job.

Parameters

- **job** (child of *obob_condor.Job*) – The job class to be added.
- ***args** – Variable length argument list.
- ****kwargs** – Arbitrary keyword arguments.

run_local()

Runs the added jobs locally.

submit(*do_submit=True*)

Runs the added jobs on the cluster.

Parameters **do_submit** (bool, optional) – Set this to false to not actually submit but prepare all files.

class obob_condor.**Job**(**args*, ***kwargs*)

Abstract class for Jobs. This means, in order to define you own jobs, they need to be a subclass of this one.

You **must** implement (i.e. define in your subclass) the *run()* method. The run method can take as many arguments as you like. Only the types of arguments are restricted because they need to be saved to disk. In general, strings, numbers, lists and dictionaries are fine.

You **can** implement *shall_run()*. This can be used to see whether some output file already exists and restrict job submission to missing files.

run(**args*, ***kwargs*)

Implement this method to do the job.

shall_run(**args*, ***kwargs*)

This is an optional method. It gets called with the same arguments as the *run()* method, **before** the job is submitted. If it returns True, the job is submitted, if it returns False, it is not.

class obob_condor.**AutomaticFilenameJob**(**args*, ***kwargs*)

Bases: *obob_condor.job.Job*

Abstract class for Jobs providing automatic filename generation.

In order for this to work, you need to:

1. Set `base_data_folder` and `job_data_folder` as a class attribute.
2. If you use *shall_run()*, you need to do the super call.

This class then automatically creates the filename for each job using all the keyword arguments supplied.

Please take a look at *Automatically generating filenames for your jobs* for detailed examples.

Variables

- **base_data_folder** (*str or pathlib.Path*) – The base folder for the data. Is normally set once for all jobs of a project.
- **job_data_folder** (*str or pathlib.Path*) – The folder where the data for this job should be saved.
- **exclude_kwargs_from_filename** (*list*) – Normally, all keyword arguments are used to build the filename. if you want to exclude some of them, put the key in the list here.
- **include_hash_in_fname** (*bool*) – Include a hash of all arguments in the filename. This is helpful if you excluded some keyword arguments from filename creation but still need to get distinct filename.
- **run_only_when_not_existing** (*bool*) – If true, this job will only run if the file does not already exist.
- **create_folder** (*bool*) – If true, calling folders are created automatically
- **data_file_suffix** (*str, optional*) – The extension of the file. Defaults to *.dat*

classmethod get_full_data_folder()

Return the data folder for this job (i.e. `base_data_folder` plus `job_data_folder`).

shall_run(**args*, ***kwargs*)

This is an optional method. It gets called with the same arguments as the `run()` method, **before** the job is submitted. If it returns True, the job is submitted, if it returns False, it is not.

property full_output_path

The full path to the output file.

Type pathlib.Path

property output_filename

The filename for this subject.

Type str

property output_folder

The output folder for this subject.

Type pathlib.Path

class obob_condor.**PermuteArgument**(*args*)

This is a container for to-be-permuted arguments. See the example in the introductions for details.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

O

obob_condor, 10

INDEX

A

add_job() (*obob_condor.JobCluster method*), 10
AutomaticFilenameJob (*class in obob_condor*), 11

F

full_output_path (*obob_condor.AutomaticFilenameJob property*), 11

G

get_full_data_folder() (*obob_condor.AutomaticFilenameJob class method*), 11

J

Job (*class in obob_condor*), 11
JobCluster (*class in obob_condor*), 10

M

module
    obob_condor, 10

O

obob_condor
    module, 10
output_filename (*obob_condor.AutomaticFilenameJob property*), 12
output_folder (*obob_condor.AutomaticFilenameJob property*), 12

P

PermuteArgument (*class in obob_condor*), 12

R

run() (*obob_condor.Job method*), 11
run_local() (*obob_condor.JobCluster method*), 10

S

shall_run() (*obob_condor.AutomaticFilenameJob method*), 11
shall_run() (*obob_condor.Job method*), 11
submit() (*obob_condor.JobCluster method*), 10