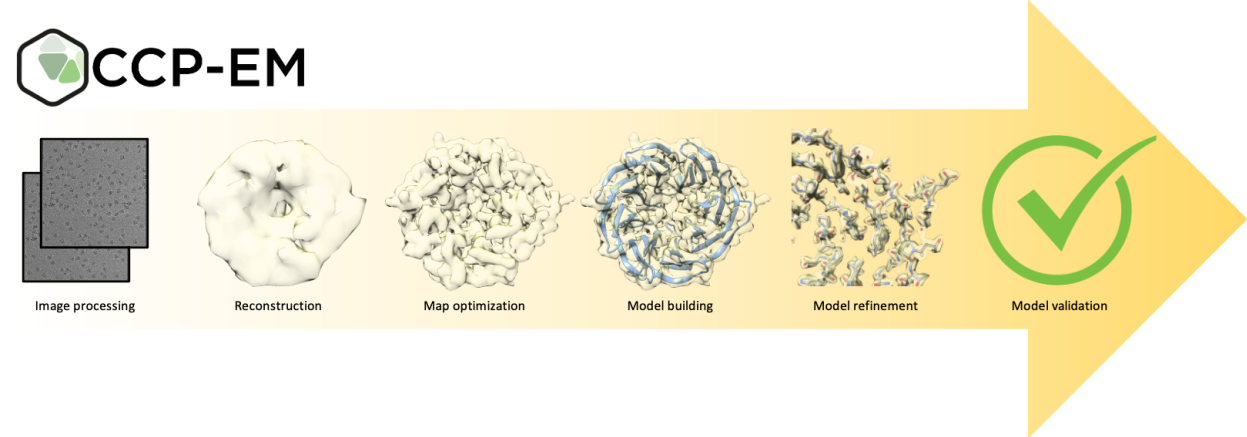
CCPEM-pipeliner

Matthew G Iadanza, Colin M Palmer, Jola Mirecka

Apr 25, 2024

CONTENTS



The CCPEM-pipeliner is an integrated suite of software tools for processing single particle cryoEM data, from preprocessing raw image data through building and fitting atomic models.

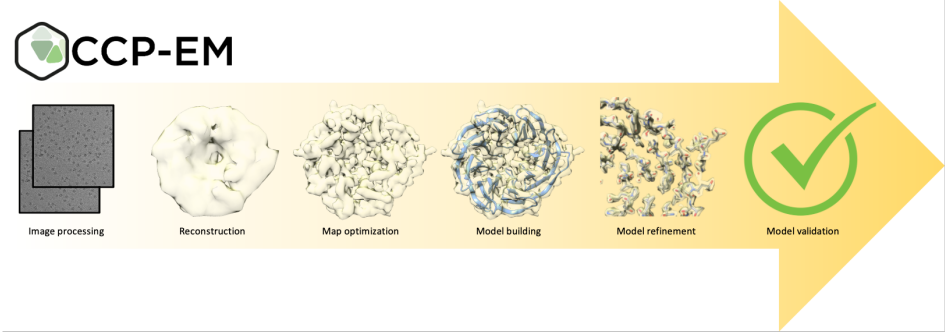
The pipeliner brings together a variety of 3rd party software in a single unified framework for seamless integration of the different programs, along with tools for management and analysis of the project.

CHAPTER

ONE

GENERAL INFO AND INSTALLATION

1.1 CCPEM-pipeliner



1.1.1 Installation

Once the package has been downloaded navigate into the `ccpem-pipeliner` directory and install the pipeliner with the command:

```
pip install -e .
```

1.1.2 Check the installation

Once the pipeliner is installed use the command `check_setup.py` to check that the setup is complete and the pipeliner can find the Relion programs it needs to run.

1.1.3 Documentation

Documentation is available online at: ccpem-pipeliner.readthedocs.io/en/latest/

To build the documentation yourself, install the documentation build requirements as follows:

```
pip install -r requirements-docs.txt
```

Then navigate to the `ccpem-pipeliner/docs` directory and run the command:

```
make html
```

Then open the file `ccpem-pipeliner/docs/_build/html/index.html` in a web browser to access the documentation.

1.1.4 Adding plugins

To add additional job plugins the plugin must first be written as a `PipelinerJob` object.
See the pipeliner documentation for a description of the format.

The plugin must then be placed in the `Pipeliner/jobs/` folder or one of its subfolders, or create a new subfolder.

Once the plugin file is in place add it to the list of entry points in `setup.cfg`.

If `setup.cfg` has been modified update the pipeliner by running `pip install -e .` for the changes to take effect.

1.1.5 For Developers

It's a good idea to work in a virtual environment for this project. Set one up as follows:

```
python3 -m venv venv/
source venv/bin/activate
pip install -r requirements.txt -r requirements-dev.txt
pre-commit install
```

This project uses pre-commit to run Black for code formatting, flake8 for linting and some other simple checks, and mypy for type checks.

You might want to install Black separately yourself too, so you can run it from your IDE.

According to the flake8 documentation, flake8 should not stop a git commit from going ahead unless `flake8.strict` is set in the git config. That doesn't actually seem to work: with the current configuration, commits fail if flake8 finds any problems. There are some flake8 warnings that have not been fixed yet, so to get around this, flake8 checks can be disabled: `SKIP=flake8 git commit ...`

1.1.6 Unit tests

Run the tests with `pytest`.

Some of the tests are quite slow or require some interaction so these tests are skipped by default. Set the environment variable `PIPELINER_TEST_SLOW` to a non-empty string to run the slower tests as well. Set the environment variable `PIPELINER_TEST_INTERACTIVE` to a non-empty string to also run the tests that require interaction.

CHAPTER

TWO

GETTING STARTED

2.1 Getting started

The CCPEM-Pipeliner provides easy access to a variety of software tools for all steps of processing cryoEM data from data preprocessing through model building and validation. The workflow is tracked and tools for visualising the full project and analysing the results are provided.

The ccpem-pipeliner serves the back end for its companion software Doppio which provides a full graphical user interface.

2.1.1 Start with a *Project*

The **Project** is contained in a single project directory.

Note: Paths used in the pipeliner, such as paths to files in job parameters are generally relative to this project directory.

To create a project or access an existing project using the API:

```
from pipeliner.api.manage_project import PipelinerProject

my_project = PipelinerProject()
```

To start a new project from the command line

```
$ pipeliner --start_new_project
```

2.1.2 A *Project* is made up of *Jobs*

The project is made of up **Jobs**. Each job is one type of operation on the data, although jobs can have several steps. Jobs are defined by their **jobtype**. The format of the jobtype is:

```
<program>.<function>.<keywords>
```

with:

<*program*>: the main piece of external software used by the job
<*function*>: the task being performed
<*keywords*>: serve to further differentiate the jobtype; an unlimited number are allowed

To get information about a specific job type from the command line:

```
$ pipeliner --job_info <job type>
```

Jobs are written in their own job directories with the format:

```
<function>/job<nnn>/
```

with the job number automatically incremented as the project progresses.

Note: A job's directory is also its name, which is used to identify it. The job's name EX: *AutoPick/job004/* **requires** the trailing slash at the end.

2.1.3 Jobs are created from parameter files

Jobs can be created by reading from either or two types of **Parameter Files**: *run.job* or *job.star*

Both files define the jobtype, if the job is new or a continuation of an old job, and the parameters or **JobOptions**.

run.job files are more verbose and easier to manually edit:

```
job_type == relion.autopick.log
is_continue == false
Pixel size in micrographs (A) == 1.02
Pixel size in references (A) == 3.54
...
```

job.star files have a more complicated format but have the advantage that the Pipeliner has functions to dynamically edit them:

```
data_job

_rlnJobTypeLabel          relion.autopick.log
_rlnJobIsContinue    0

data_joboptions_values
loop_
_rlnJobOptionVariable #1
_rlnJobOptionValue #2
angpix         1.02
angpix_ref     3.54
...
```

Note: *job.star* and *run.job* files can be used interchangeably for almost all applications in a project.

2.1.4 Getting a *run.job* or *job.star* file

A parameter file with the default values for any job can be generated with *write_default_runjob()* or *write_default_jobstar()*

API:

```
from pipeliner.api.api_utils import default_runjob, default_jobstar

default_runjob("relion.autopick.log")
default_jobstar("relion.autopick.log")
```

Command line:

```
$ pipeliner --default_runjob relion.autopick.log
$ pipeliner --default_jobstar relion.autopick.log
```

This will create the files *relion_autopick_log_job.star* and *relion_autopick_log_run.job*

2.1.5 Running a job

With the paramter file created the job can now be run with *run_job()*:

API:

```
my_project.run_job("relion_autopick_log_job.star")
```

Command line:

```
$ pipeliner --run_job relion_autopick_log_job.star
```

This will create and run the job *AutoPick/job001/*

Alternatively a job can be run from a `dict` containing its parameters which can be generated with the function *pipeliner.api.api_utils.job_default_parameters_dict()* This dict can be edited in place before using it to run a job.

```
from pipeliner.api.api_utils import job_default_parameters_dict
from pipeliner.api.manage_project import run_job

params = job_default_parameters_dict("relion.autopick.log")
params["fn_input"] = "Path/to/new/input_file.mrc"
run_job(params)
```

2.1.6 Continuing a job

Some jobs can be continued from where they finished. When a job is run a file *continue_job.star* is written in its job directory. This file contains only the parameters that are allowed to be modified when the job is continued. Edit this file if any parameters need to be changed and then continue the job with:

API:

```
my_project.continue_job("AutoPick/job001/")
```

Command line:

```
$ pipeliner --continue_job AutoPick/job001/
```

Note: The job's full name was used to continue it, *not* the name of the *continue_job.star* file

2.1.7 Submitting jobs to a queue

Pipeliner jobs can be submitted to a queuing system using a submission script template that incorporates values from the job's JobOptions.

The pipeliner will update variables bracketed by *XXX* in the submission template from the job's JobOptions and then run the submission script using the command specified in the job's *qsub* JobOption.

Table 1: Template variables updated from JobOptions

Script Variable	JobOption	GUI Field
XXXmpinodesXXX see note	nr_mpi	Number of MPI procs:
XXXthreadsXXX see note	nr_threads	Number of threads:
XXXdedicatedXXX	min_dedicated	Minimum dedicated cores per node:
XXXqueueXXX	queuename	Queue name:
XXXextra1XXX	qsub_extra_1	Set from environment variable PIPELINER_QSUB_EXTRA1
XXXextra2XXX	qsub_extra_2	Set from environment variable PIPELINER_QSUB_EXTRA2
XXXextra3XXX	qsub_extra_3	Set from environment variable PIPELINER_QSUB_EXTRA3
XXXextra4XXX	qsub_extra_4	Set from environment variable PIPELINER_QSUB_EXTRA4

There are some additional variables available for submission scripts that are not drawn from the JobOptions:

Table 2: Additional template variables

Script Variable	Substitution
XXXnameXXX	The job's name; the same as its output directory
XXXcoresXXX	The number of mpi processes multiplied by the number of threads
XXXerrfileXXX	Path to the job's run.err file
XXXoutfileXXX	Path to the job's run.out file
XXXcommandXXX see note	The full commands list for the job.

Note: The variable *XXXcommandXXX* will already have the mpirun command specified by the *mpirun_com* JobOption prepended to commands where necessary. It generally does NOT need to be included in the submission script. The default for *mpirun_com* is *mpi_run -n XXXmpinodesXXX* meaning the *XXXmpinodesXXX* variable generally also does not need to be included in the commands section of the submission script template. Similarly, the number of threads used by a job is usually set in the commands, so the *XXXthreadsXXX* variable rarely needs to be used.

The submission script template must be written for your specific system. Here is an example submission script template for a cluster running SLURM:

```
#!/bin/bash
#SBATCH --ntasks=XXXmpinodesXXX
#SBATCH --partition=XXXqueueXXX
#SBATCH --cpus-per-task=XXXthreadsXXX
```

(continues on next page)

(continued from previous page)

```
#SBATCH --error=XXXerrfileXXX
#SBATCH --output=XXXoutfileXXX
#SBATCH --gres=gpu:2

XXXcommandXXX
```

2.1.8 Modifying parameters

The python API can modify *job.star* parameter files on-the-fly using *edit_jobstar()*. This avoids manual editing of the parameter files when stringing together multiple jobs:

```
from pipeliner.api.api_utils import edit_jobstar

movie_jobstar = my_project.write_default_jobstar("relion.import.movies")
edit_jobstar(movie_jobstar, {"fn_in_raw": "Movies/*.mrcs"})
movie_job = my_project.run_job(movie_jobstar).output_name

mocorr_jobstar = my_project.write_default_jobstar("relion.motioncorr.own")
edit_jobstar(mocorr_jobstar, {"fn_in": movie_job.output_name + "movies.star"})
mocorr_job = my_project.run_job(mocorr_jobstar).output_name
```

alternatively this can be done solely with dicts:

```
from pipeliner.api.api_utils import job_default_parameters_dict

import_params = job_default_parameters_dict("relion.import.movies")
import_params["fn_in_raw"] = "Movies/*.mrcs"
movie_job = my_project.run_job(import_params).output_name

mocorr_params = job_default_parameters_dict("relion.motioncorr.own")
mocorr_params["fn_in"] = movie_job.output_name + "movies.star"
mocorr_job = my_project.run_job(mocorr_params).output_name
```

2.1.9 Running schedules

Scheduling allows for sets of jobs to be run multiple times via *schedule_job()* and *run_schedule()*

Note: When a job is scheduled placeholder files are created for all of its outputs so these files can be used as if they already exist.

Here is running the same jobs as above, except using the scheduling functions to run the set of import and motion correction jobs 10 times:

API:

```
movie_jobstar = my_project.write_default_jobstar("relion.import.movies")
edit_jobstar(movie_jobstar, {"fn_in_raw": "Movies/*.mrcs"})
movie_job = my_project.schedule_job(movie_jobstar)
```

(continues on next page)

(continued from previous page)

```
mocorr_jobstar = my_project.write_default_jobstar("relion.motioncorr.own")
edit_jobstar(mocorr_jobstar, {"fn_in": movie_job.output_name + "movies.star"})
mocorr_job = my_project.schedule_job(mocorr_jobstar)

my_project.run_schedule(
    fn_sched="my_schedule",
    job_ids=[movie_job.putput_name, mocorr_job.output_name],
    nr_repeat=10,
    )
```

To accomplish this from the command line the parameter files for the Import and MotionCorr jobs must already have been created with the correct file names as inputs

```
$ pipeliner --schedule_job <import job param file>
$ pipeliner --schedule_job <motion corr job param file>
$ pipeliner --run_schedule --name my_schedule --jobs job001 job002 --nr_repeat 10
```

Note: The command line tool intelligently parses job names, so for the job named *Import/job001/* it would accept *job001* or *1* as well as the full job name

2.1.10 Other job tools

A variety of other tool exist for modifying jobs in the project. See the api documentation for how to use these functions:

- *set_alias* - Give a job an more descriptive name
- *run_cleanup* - Move intermediate files from jobs into the trash to save disk space
- *delete_job* - Move a job to the trash
- *undelete_job* - Remove a job from the trash and restore it to the project
- *empty_trash* - Permanently delete files in the trash
- *prepare_metadata_report* - Get metadata about an entire project

CHAPTER

THREE

WRITING NEW JOBS

3.1 Writing new jobs

New software can be added to the ccpem-pipeliner by writing a *PipelinerJob*.

A *PipelinerJob* is completely self contained and has everything needed to run a new piece of software in the pipeliner framework. It must accomplish the following tasks:

1) Define the job's input parameters and how they are displayed in the GUI
2) Define how the parameters will be validated
3) Generate the commands that need to be run
4) Define the job's output nodes (i.e. the output files that are displayed in the GUI and made available as inputs to other jobs in the pipeline)
5) Perform any final tasks that need to happen after the commands have been executed
6) Make objects that allow the GUI to display the job's results graphically
7) Gather metadata about the job
8) Define how to clean up the job
9) Create the objects necessary to create a PDB/EMDB/EMPIAR deposition

Note: A minimal *PipelinerJob* only needs to define input parameters, define output nodes, and generate commands. Adding the additional methods will add functionality to the job and integrate it more fully into Doppio, the GUI built on top of the Pipeliner.

3.1.1 Making a new `PipelinerJob`

Required imports:

```
from pipeliner.pipeliner_job import PipelinerJob, Ref, ExternalProgram
from pipeliner.job_options import StringJobOption # and other types as necessary
from pipeliner.node_factory import create_node
from pipeliner.display_tools import create_results_display_object
```

Make a class for the new job:

```
class MyJob(PipelinerJob):
    PROCESS_NAME = "software.function.keywords"
    OUT_DIR = "MyJob"

    def __init__(self):
        super().__init__()  # don't forget to initialize the PipelinerJob superclass first
```

PROCESS_NAME is the name the pipeliner will use to identify the job, and should match the job type name that is used when the job is added to the `setup.cfg` file (see *"Adding the new job to the pipeliner"* below).

OUT_DIR is both the directory where the job's output directory (`jobNNN/`) will be written and is used to group jobs together in the GUI.

Some examples:

```
class RelionLogAutopick(PipelinerJob):
    PROCESS_NAME = "relion.autopick.log"
    OUT_DIR = "AutoPick"

class RelionTrainTopaz(PipelinerJob):
    PROCESS_NAME = "relion.autopick.topaz.train"
    OUT_DIR = "AutoPick"

class CryoloAutopick(PipelinerJob):
    PROCESS_NAME = "cryolo.autopick"
    OUT_DIR = "AutoPick"
```

3.1.2 `__init__` method

The job's `__init__` needs to do the following things:

- define information about the job
- define the parameters for the job

Define information about the job

Information about the job and the programs it needs to run are stored in its *JobInfo* object. For example:

```
self.jobinfo.display_name = "RELION initial model generation
self.jobinfo.version = "0.1"
self.jobinfo.programs = [ExternalProgram(command="relion_refine")]
self.jobinfo.short_desc = "Create a de novo 3D initial model"
self.jobinfo.long_desc = "Relion 4.0 uses a gradient-driven algorithm to generate a de␣
↪novo 3D"
self.jobinfo.references = [
    Ref(
        authors=["Scheres SHW"],
        title="RELION: implementation of a Bayesian approach to cryo-EM structure",
        journal="J Struct Biol.",
        year="2012",
        volume="180",
        issue="3",
```

(continues on next page)

(continued from previous page)

```
        pages="519-30",
        doi="10.1016/j.jsb.2012.09.006",
    )
]
```

Note: Each program in *programs* should be entered with an *ExternalProgram* object. These programs will be checked for availability, if they are not found in the system PATH the GUI will mark the job as unavailable.

Define the job's input parameters

Input parameters are defined by adding *JobOption* objects to the job's *joboptions* attribute. There are 8 types of *JobOption* :

Table 1: JobOption Types

Type	Data type	Appearance in GUI	Usage notes
StringJobOption	str	Text entry field	
FloatJobOption	float	Number entry field	
IntJobOption	int	Number entry field	
BooleanJobOption	bool	Checkbox	
MultipleChoiceJobO	str	Drawdown list	
InputNodeJobOptior	str	Text entry field (with options) and a file browser	Files that are part of the project
MultiInputNodeJobO	str	Text entry field (with options) and a file browser for entering multiple files	Files that are part of the project
ExternalFileJobOpt	str	Text entry field	Files that are outside the project
DirPathJobOption	str	Text entry field	A path to a directory
SearchStringJobOpt	str	Text entry field	A search string for multiple files

Note: The job option types and GUI behaviour for input nodes and files are currently under review and will probably change in the near future. See also *File paths in pipeliner jobs* below.

Special predefined job options

There are two methods in *PipelinerJob* that create special predefined JobOptions.

make_additional_args():

This adds a special *StringJobOption* called other_args.

This can be used in calling *parse_additional_args()* in the job's *get_commands()* method. It returns a list of the additional args with the quotation structure preserved.

For example, if the value for self.joboptions["other_args"] is 'arg1 arg2 "arg3 arg4"', it would return: ['arg1', 'arg2', '"arg3 arg4"']

get_runtab_options():

This adds the job options in RELION's 'Run' tab:

Table 2: JobOption Types

Name	JobOption type	Description
nr_mpi	*IntJobOption*	Number of MPIs to use only; added if `mpi=True`
nr_threads	*StringJobOption*	Number of threads to use; only added if `threads=True`
do_queue	*BooleanJobOption*	Should the job be submitted to the queue?
queuename	*StringJobOption*	Name of the queue
qsub	*StringJobOption*	Queue submission command to use
qsubscript	*ExternalFileJobOption*	Path to template for the submission script
min_dedicated	*IntJobOption*	Minimum dedicated cores per node

3.1.3 `create_output_nodes` method

The *create_output_nodes()* method is used to define the job's outputs, in the form of *Node* objects added to the job's *output_nodes* list.

The simplest way to do this is to use the *add_output_node()* helper method, which creates a node for a file in the job's output directory and adds it to the *output_nodes*. For example, for a job that makes a cryo-EM density map called `map.mrc` as output:

```
from pipeliner.nodes import NODE_DENSITYMAP
self.add_output_node("map.mrc", NODE_DENSITYMAP, ["keyword1", "keyword2"]))
```

Information about the different node types can be found *below*.

Note that *create_output_nodes()* is called *before* the job is run, so it defines the outputs that the job is expected to produce. It's helpful to define the outputs early because then other jobs can be scheduled to run afterwards using this job's outputs as their inputs. However, in some cases it's not possible to know all of the job's outputs until the commands have actually been run, in which case additional output nodes can be added at the end of the job using the *create_post_run_output_nodes method*.

3.1.4 `get_commands` method

The *get_commands()* method defines what the job will actually do, by creating a list of commands that will be run. It is called by the job runner when the job is started.

The commands must be returned as a list of *PipelinerCommand* objects, each of which contains a list of strings (i.e. a command and arguments, suitable for passing to `subprocess.run()`).

```
[
    PipelinerCommand(["first", "command", "to", "run"]),
    PipelinerCommand(["next", "command", "to", "run"]),
    PipelinerCommand(["final", "command", "to", "run"]),
]
```

The commands can be assembled in any manner you see fit. A variety of attributes of the *PipelinerJob* are available to get pieces of data during this process:

Table 3: JobOption Types

Attribute	Description	Data type
output_dir	The directory where the job should put its output files. (This is also used as the job's name in the pipeline.)	`str`
joboptions	A dict containing all of the JobOptions defined in `__init__`	`dict{str: JobOption}`
is_continu	Is the job being continued?	`bool`
input_node	List of the job's input nodes	`list[Node]`
output_noc	List of the job's output nodes	`list[Node]`

Note: The *get_commands()* method is run in a different process from the one where the job was created and submitted, and for jobs sent to a queue, it will also be on a different computer. Any state that was saved in attributes of the *PipelinerJob* before it was scheduled to run will be lost. You should make sure that your job is written such that it can be reliably recreated from its job option values alone.

Also note that *get_commands()* should run quickly and not produce any output files itself. Any actions that write files or take some time to run should be moved into scripts or executables that can be run as one of the job's commands.

File paths in pipeliner jobs

The project directory is the root for all the pipeliner's file handling: input and output files are stored as relative paths from the project directory, and jobs are run with the project directory as their working directory.

Jobs should be self-contained, i.e. they should write files only into their own job directory, *output_dir* (which will have the form `JobType/jobNNN/`, for example `Import/job001/`). The job's commands should take care to direct their output files only to this directory (or subdirectories within it) and not elsewhere in the project.

For example:

```
commands = [
    ["touch", os.path.join(self.output_dir, "output_file_1")],  # this is fine
    ["cp", input_file_1, input_file_2, self.output_dir],        # this is also fine

    # The next command is NOT fine - it would write the file directly in the project␣
↪directory
    ["touch", "output_file_2"],
]
```

The only exception to the rule that all paths should be relative and within the project is where a job needs to access some centrally-installed file or disk using an absolute path, for example a program executable, queue submission script template or local scratch disk. In these cases, absolute paths can be used but the files should be treated as external to the project and not added as input or output nodes.

Some uncooperative programs do not have the ability to specify where their results are written. In these cases the location from which the commands are executed can be changed by setting the *working_dir* attribute. The most common use case is programs that write their outputs directly into the working directory, in which case *working_dir* should be set to *output_dir*. The best place to do this is usually at the start of the *get_commands()* method.

If *working_dir* has been set then it is important that any file paths passed as arguments to the job's commands are given relative to the working directory rather than the project directory. The easiest way to accomplish this is by using `os.path.relpath()`. For example, in a pipeliner job the path to an input file is typically found using something like this: `input_file = self.joboptions["input_file"].get_string()`. Normally `input_file` can then

be used directly, but in a job where *working_dir* is set `os.path.relpath(input_file, self.working_dir)` should be used instead.

Note: Only the job's commands themselves will be run in the job's *working_dir*. The pipeliner code (including all *PipelinerJob* methods) is still run in the project directory.

3.1.5 `additional_joboption_validation` method

The *additional_joboption_validation()* method serves to do advanced validation of the `JobOptions` before a job is run. Simple validation is done automatically:

- parameters that are required have non-empty values
- parameter values are of the right type
- parameter values are within their specified ranges

additional_joboption_validation() is used for more specific validation tasks that take into account more than one job option such as:

- parameter A must be > parameter B
- parameters C, D, and E cannot all be equal

The method should return a list of *JobOptionValidationResult* objects.

Some examples:

```
def additional_joboption_validation(self):
  errors = []

  jobop_a = self.joboptions["param_a"]
  jobop_b = self.joboptions["param_b"]
  if jobop_a.get_number() < jobop_b.get_number():
    errors.append(
      JobOptionValidationResult(
        type="error",
        raised_by=[jobop_a, jobop_b],
        message="A must be greater than B",
      )
    )

  jobop_c = self.joboptions["param_c"].get_string()
  jobop_d = self.joboptions["param_d"].get_string()
  jobop_e = self.joboptions["param_d"].get_string()

  if jobop_c == jobop_d == jobop_e:
    errors.append(
      JobOptionValidationResult(
        type="error",
        raised_by=[jobop_c, jobop_d],
        message="C, D, and E cannot all be the same!",
      )
    )
```

(continues on next page)

(continued from previous page)

```
  return errors
```

3.1.6 create_post_run_output_nodes method

This function creates output nodes after the job has run. This is only necessary when the names of the output nodes are not known beforehand.

Example of a create_post_run_output_nodes function: In this case the job doesn't know the number of outputs it will produce until after it has executed the commands so they must be created *ex post facto* by `create_post_run_output_nodes`

```
def create_run_output_nodes(self):
  outputs = glob(self.output_name + "result_*.mrc")
    for f in outputs:
      self.output_nodes.append(create_node(f, "DensityMap", ["node", "keywords"]))
```

3.1.7 create_results_display method

The *create_results_display()* method generates result display objects that allow the Doppio GUI to display results from the pipeliner.

There are currently 16 types of *ResultsDisplayObject*:

Table 4: ResultsDisplayObject Types

Type	Description
ResultsDisplayTe	A single line of text
ResultsDisplayMo	An array of thumbnail images
ResultsDisplayGr	A plot with points, lines, or both
ResultsDisplayIm	A 2D Image
ResultsDisplayHi	A histogram
ResultsDisplayTa	Multiple text containing cells with a header row
ResultsDisplayMa	Integrated 3D viewer for mrc, map, cif, and pdb files
ResultsDisplayRv	A general Object that displays a webpage
ResultsDisplayPe	A special class of `ResultsDisplayObject` generated when there has been an error. Doppio attempts to update any `ResultsDisplayPending` objects when a job is viewed.
ResultsDisplayPl	A class that allows a Plotly object to be displayed; allows creation of more complex displays
ResultsDisplayPl	Allows a Plotly object to be passed directly to the display; for creation of more complex displays
ResultsDisplayPl	Allows for creation of more complex histograms with Plotly express histogram
ResultsDisplayPl	Allows for creation of more complex scatter plots with Plotly express scatter
ResultsDisplayTe	Displays the contents of a text file
ResultsDisplayPd	Displays the contents of a pdf file
ResultsDisplayHt	Displays html formatted results
ResultsDisplayJs	Displays JSON formatted results

They should all be created with the *pipeliner.display_tools.create_results_display_object()* function. This function safely creates the object, returning a *ResultsDisplayPending* object with an explanation if any errors are encountered.

3.1.8 gather_metadata method

The *gather_metadata()* method returns a dict of metadata about the results of the job. It doesn't need to gather any information about the parameters used, this will be done automatically.

Do this in any way you see fit, just return a `dict`.

3.1.9 prepare_clean_up_lists method

The *prepare_clean_up_lists()* method returns a `list` with two items; a list of files to delete and a list of directories that should be removed when the job is cleaned up. The purpose of cleanup is to free up disk space by removing unneeded files.

> **Warning:** Don't delete anything yet! This method should prepare lists of files for deletion, but they might not actually be deleted (e.g. if the user decides to cancel the clean up) so don't do anything irreversible here.

There are two levels of cleanup:

- Standard: delete files that are not necessary, such as intermediate iterations or tmp files
- Harsh: delete more, such as output files that can be reproduced easily

Make sure *prepare_clean_up_lists()* doesn't delete anything important or used by Doppio for results display.

Example *prepare_clean_up_lists()*:

```
def prepare_clean_up_lists(self, do_harsh):
    files_to_remove, dirs_to_remove = [], []

    tmp_files = glob(self.output_name + "*.tmp")
    tmp_dir = self.output_name + "tmpfiles"

    files_to_remove.extend(tmp_files)
    dirs_to_remove.extend(tmp_dir)

    if do_harsh:
        extra_files = glob(self.output_name + "*.extra")
        extra_dir = self.output_name + "extrafiles"
        files_to_remove.extend(extra_files)
        dirs_to_remove.extend(extra_dir)

    return [files_to_remove, dirs_to_remove]
```

3.1.10 prepare_deposition_data method

The *prepare_deposition_data()* method returns a list of deposition objects which are used to prepare data for deposition in to the PDB, EMDB, and EMPIAR.

This method must be implemented in any *PipelinerJob* that produces data included in a database deposition. For the PDB and EMDB these are defined by the published schema.

This feature is currently in development...

```
def onedep_deposition(self):
  sym = this_function_gets_symmetry()
  reso = this_function_gets_the_resolution()

  sp_filter = spatial_filtering_type_entry(
      high_frequency_cutoff=reso,
      software_list=("relion_refine",),
  )

  rec_filter = reconstruction_filtering_type_entry(spatial_filtering=sp_filter)

  recdep = final_reconstruction_type_entry(
      applied_symmetry=sym,
      algorithm="FOURIER SPACE",
      resolution=reso,
      resolution_method="FSC 0.143 CUT-OFF",
      reconstruction_filtering=rec_filter,
  )

  return [recdep]
```

3.1.11 Adding the new job to the pipeliner

Now you have a file `my_new_job.py`. It contains a class `MyJob` that describes a job of the type `mysoftware.function`.

Well done!

1) Put your job file into `ccpem-pipeliner/pipeliner/jobs/other`
2) Add an entry point definition for your new job in `ccpem-pipeliner/setup.cfg`: Go to the `ccpem_pipeliner.jobs` section Add your job in the format:

   `jobtype = package_name.module_name:ClassName`

   e.g. `mysoftware.function = pipeliner.jobs.other.my_new_job:MyJob`
3) Update the pipeliner installation with: `pip install -e .`
4) Check for your job by running `pipeliner --job_info mysoftware.function` from the command line

3.2 Node type names

These top-level node types are already in use in the pipeliner. It is good practice to use the constants found in *pipeliner.nodes* rather than typing node types to maintain continuity.

(scroll the table right to see the constant names)

Table 5: Top-Level

Node type	Description
AtomCoords	An atomic model file
AtomCoordsGroupMetadata	Metadata about a set of atomic coordinates files, a list of atomic models for example

Table 5 – continued fr

Node type	Description
DensityMap	A 3D cryoEM density map (could be half map, full map, sharpened etc)
DensityMapMetadata	Metadata about a single density map
DensityMapGroupMetadata	Metadata about multiple density maps
EulerAngles	Data about Euler angles
EvaluationMetric	A file containing evaluation metrics
Image2D	A single 2D image
Image2DStack	A single file containing a stack of 2D images
Image2DMetadata	Metadata about a single 2D image or stack
Image2DGroupMetadata	Metadata about a group of 2D images or stacks
Image3D	Any 3D image that is not a density map or mask, for example a local resolution map, 3D FSC or
Image3DMetadata	Metadata about a single 3D image, except density maps, which have their own specific node type
Image3DGroupMetadata	Metadata about a group of 3D images (but not masks or density maps, which have their own spec
LigandDescription	The stereochemical description of a ligand molecule
LogFile	A log file from a process. Could be PDF, text or another format
Mask2D	A mask for use with 2D images
Mask3D	A mask for use with 3D volumes
MicrographCoords	A file containing coordinate info for a single micrograph, e.g. a .star or .box file produced from p
MicrographCoordsGroup	A file containing coordinate info for multiple micrographs, e.g. a STAR file with a list of coordin
Micrograph	A single micrograph
MicrographMetadata	Metadata about a single micrograph
MicrographGroupMetadata	Metadata about a set of micrographs, for example a RELION corrected_micrographs.star file
MicrographMovie	A single multi-frame micrograph movie
MicrographMovieMetadata	Metadata about a single multi-frame micrograph movie
MicrographMovieGroupMetadata	Metadata about multiple micrograph movies, e.g. movies.star
MicroscopeData	Data about the microscope, such as collection parameters, defect files or MTF curves
MlModel	A machine learning model
OptimiserData	Specific type for RELION optimiser data from a refinement or classification job
ParamsData	Contains parameters for running an external program
ParticleGroupMetadata	Metadata for a set of particles, e.g particles.star from RELION
ProcessData	Other data resulting from a process that might be of use for other processes, for example an optim
Restraints	Distance and angle restraints for atomic model refinement
RigidBodies	A description of rigid bodies in an atomic model
Sequence	A protein or nucleic acid sequence file
SequenceGroup	A group of protein or nucleic acid sequences
SequenceAlignment	Files that contain or are used to generate multi sequence alignments
StructureFactors	A set of structure factors in reciprocal space, usually corresponding to a real space density map
TiltSeriesMetadata	Metadata about a single tomographic tilt series
TiltSeriesGroupMetadata	Metadata about a group of multiple tomographic tilt series
TomogramMetadata	Metadata about a single tomogram
TomogramGroupMetadata	Metadata about multiple tomograms
TomoOptimisationSet	Data about a RELION tomography optimisation set
TomoTrajectoryData	Data about a RELION tomography trajectory
TomoManifoldData	Data about a RELION tomography manifold

CHAPTER

FOUR

PIPELINER API

4.1 CCPEM-Pipeliner API

The pipeliner api provides access to all of the main functions of the pipeliner

4.1.1 PipelinerProject

To interact with a pipeliner project it must be created as a *PipelinerProject* object

class pipeliner.api.manage_project.**PipelinerProject**(*pipeline_name: str = 'default'*, *project_name: str | None = None*, *description: str | None = None*, *make_new_project: bool = False*)

Bases: object

This class forms the basis for a project.

pipeline_name

The name of the pipeline. Defaults to *default* if not set. There is really no good reason to give the pipeline any other name.

Type
str

abort_job(*job_name: str*) → None

Abort a running job.

This function signals to the job that it should abort, but does not wait for the job to respond.

Parameters
job_name – The job name. This must be exact, use `parse_procname` to find the job if you need to find it from a partial name, number or alias.

Raises

- **ValueError** – If there is no job with the given name
- **RuntimeError** – If the job is in any state except Running

cleanup_all(*harsh: bool = False*)

Runs cleanup on all jobs in a project

Parameters
harsh (*bool*) – Should harsh cleaning be performed?

compare_job_parameters(*jobs_list: List[str]*) → dict

Compare the running parameters of multiple jobs

Parameters
jobs_list (*list*) – The jobs to compare

Returns
{parameter: [value, value, value]}

Return type
dict

Raises

- **ValueError** – If any of the jobs is not found
- **ValueError** – If the jobs being compared are not of the same type

continue_job(*job_to_continue: str*, *comment: str | None = None*, *run_in_foreground=False*) → *PipelinerJob*

Continue a job that has already been run

To change the parameters in a continuation the user needs to edit the continue_job.star file in the job's directory

Parameters

- **job_to_continue** (*str*) – The name of the job to continue
- **comment** (*str*) – Comments for the job's jobinfo file

Returns

The
PipelinerJob object for the created job

Return type
PipelinerJob

Raises

- **ValueError** – If the continue_job.star file is not found and there is no job.star file in the job's directory to use as a backup
- **ValueError** – If the job is of a type that needs a optimizer file to continue and this file is not found
- **ValueError** – The job has iterations but the parameters specified would result in no additional iterations being run

create_archive(*job: str*, *full: bool = False*, *tar: bool = True*) → str

Creates an archive

Archives can be full or simple. Simple archives contain the directory structure of the project, the parameter files for each job and a script to rerun the project through the terminal job. The full archive contains the full job dirs for the terminal job and all of its children

Parameters

- **job** (*str*) – The name of the terminal job in the workflow
- **full** (*bool*) – If `True` a full archive is written else a simple archive is written
- **tar** (*bool*) – Should the newly written archive be compressed?

Returns
A message telling the type of archive and its name

Return type
str

create_reference_report(*terminal_job: str*) → Tuple[str, int]

Create a report on all the references used in the project

Parameters
terminal_job (*str*) – use this job and all its parents

Returns
(The name of the report created, number of jobs in report)

Return type
tuple

delete_job(*job: str*) → bool

Delete a job

Removes the job from the main project and moves it and its children it to the Trash

Parameters
job (*str*) – The name of the job to be deleted

Returns
`True` If a job was deleted, `False` if no jobs were deleted

Return type
bool

edit_comment(*job_name: str*, *comment: str | None = None*, *overwrite: bool = False*, *new_rank: int | None = None*)

Edit the comment of a job

Parameters

- **job_name** (*str*) – The name of the job to eddit the comment for
- **comment** (*str*) – The comment to add/append
- **overwrite** (*bool*) – if *True* overwrites otiginal comment, otherwise appends it to the current comment
- **new_rank** (*int*) – New rank to assign to job, use -1 to revert the rank to *None*

Raises
ValueError – If the new rank is not *None* or an integer

static empty_trash()

Deletes all the files and dirs in the Trash directory

Returns
`True` if any files were deleted, `False` If no files were deleted

Return type
bool

find_job_by_comment(*contains: List[str] | None = None*, *not_contains: List[str] | None = None*, *job_type: str | None = None*, *command: bool = False*) → List[str]

Find Jobs by their comments or command

Parameters

- **contains** (*list*) – Find jobs that contain all of the strings in this list
- **not_contains** (*list*) – Find jobs that do not contain any of these strings
- **job_type** (*str*) – Only consider jobs who's type contain this string
- **command** (*bool*) – If *True* searches the job's command history rather than its comments

Returns
Names of all the jobs found

Return type
list

Raises
ValueError – If nothing is specified for contains and not_contains

find_job_by_rank(*equals: int | None = None*, *less_than: int | None = None*, *greater_than: int | None = None*, *job_type: str | None = None*) → List[str]

Find jobs by their rank

Ignores jobs that are unranked

Parameters

- **equals** (*int*) – Find jobs with this exact rank
- **less_than** (*int*) – Find jobs with ranks less then this number
- **greater_than** (*int*) – Find jobs with ranks higher than this number
- **job_type** (*str*) – Only consider jobs that contain this string in their job type

Returns
Names of the matching jobs

Return type
list

Raises

- **ValueError** – If nothing is specified to search for
- **ValueError** – If both equals and less_than/greater than are specified

get_job(*job_name: str*) → *PipelinerJob*

Get an existing job from the project.

Parameters
job_name (*str*) – The name of the job to get

Raises
ValueError – if the named job cannot be found

parse_proclist(*list_o_procs: list*, *search_trash: bool = False*) → list

Finds full process names for multiple processes

Returns full process names IE: *Import/job001/* from *job001* or *1*

Parameters

- **list_o_procs** (*list*) – A list of string process names
- **search_trash** (*bool*) – Should the trash also be search?

Returns
All of the full process names

Return type
list

parse_procname(*in_proc: str*, *search_trash: bool = False*) → str

Find process name with the ability for parse ambiguous input.

Returns full process names IE: *Import/job001/* from *job001* or *1* Can look in both active processes and the Trash Can, accepts inputs containing only job number and process type and alias IE *Import/my_alias*

Parameters

- **in_proc** (*str*) – The text that is being checked against the list of processes
- **search_trash** (*bool*) – Should it return the process name if the process is in the trash?

Returns
the process name

Return type
str

Raises

- **ValueError** – if the process was in the trash but search_trash is false
- **ValueError** – if the process name is not in the pipeliner format, jobxxx, or a number. IE: An unrelated string
- **ValueError** – if the process name is not found

static prepare_deposition(*terminal_job: str*, *depo_type: Literal['onedep', 'empiar']*, *depo_id: str | None = None*, *jobstar_file: str | None = None*, *empiar_do_mov: bool = True*, *empiar_do_mics: bool = True*, *empiar_do_parts: bool = True*, *empiar_do_rparts: bool = True*) → str

Prepare a deposition for EMPIAR, EMDB, or PDB databases

Parameters

- **terminal_job** (*str*) – This job and all its parents will be included in the deposition
- **depo_type** (*Literal["onedep", "empiar"]*) – 'ondep' is used for PDB and EMDB, 'empir for EMPIAR'
- **depo_id** (*Optional[str]*) – A name for the deposition
- **jobstar_file** – (Optional[str]): For EMPIAR; A job.star file that contains additional required information that cannot be gathered from the jobs themselves see ':class:~pipeliner.jobs.other.empiar_deposition_job.EmpiarDepositionJob' for the specifics
- **empiar_do_mov** (*bool*) – For EMPIAR; should raw movies be included?
- **empiar_do_mics** (*bool*) – For EMPIAR; Should corrected micrographs be included?
- **empiar_do_parts** (*bool*) – For EMPIAR; Should particles be included?
- **empiar_do_parts** – For EMPIAR; Should polished particles be included?

Returns
The name path of the created archive

Return type
str

prepare_metadata_report(*jobname: str*) → Tuple[str, int]

Returns a full metadata trace for a job and all upstream jobs

Parameters
jobname – The name of the job to run on

Returns

(dict: str: file written, int:
number of jobs in the report)

Return type
tuple

run_cleanup(*jobs: list*, *harsh: bool = False*) → bool

Run the cleanup function for multiple jobs

Each job defines its own method for cleanup and harsh cleanup

Parameters

- **jobs** (*list*) – List of string job names to operate on
- **harsh** (*bool*) – Should harsh cleaning be performed

Returns
`True` if cleanup is successful, otherwise `False`

Return type
bool

run_job(*jobinput: str | dict | PipelinerJob*, *overwrite: str | None = None*, *alias: str | None = None*, *ignore_invalid_joboptions=False*, *run_in_foreground=False*) → *PipelinerJob*

Run a new job in the project

If a file is specified the job will be created from the parameters in that file If a dict is input the job will be created with defaults for all options except those specified in the dict.

If a dict is used for input it MUST contain at minimum {"_rlnJobTypeLabel": <the jobtype>}

Parameters

- **jobinput** (str, dict, *PipelinerJob*) – The path to a run.job or job.star file that defines the parameters for the job or a dict specifying job parameters or a *PipelinerJob* object
- **overwrite** (*str*) – The name of a job to overwrite, if `None` a new job will be created. A job can only be overwritten by a job of the same type
- **alias** (*str*) – Alias to assign to the new job
- **ignore_invalid_joboptions** (*bool*) – Run the job anyway even if the job options appear to be invalid
- **run_in_foreground** (*bool*) – Run job in the main process blocking anything else from happening until it completes

Returns
The name of the job that was run

Return type
str

Raises

- **ValueError** – If this method is used to continue a job

- **ValueError** – If the job options are invalid and `ignore_invalid_joboptions` is not set

run_schedule(*fn_sched: str*, *job_ids: List[str]*, *nr_repeat: int = 1*, *minutes_wait: int = 0*, *minutes_wait_before: int = 0*, *seconds_wait_after: int = 5*) → str

Runs a list of scheduled jobs

Parameters

- **fn_sched** (*str*) – A name to assign to the schedule
- **job_ids** (*list*) – A list of string job names to run
- **nr_repeat** (*int*) – Number of times to repeat the entire schedule
- **minutes_wait** (*int*) – Minimum number of minutes to wait between running each subsequent job
- **minutes_wait_before** (*int*) – Initial number of minutes to wait before starting to run the schedules.
- **seconds_wait_after** (*int*) – Time to wait after running each job

Returns
The name of the schedule that is run

Return type
str

Raises
ValueError – If the schedule name is already in use

run_scheduled_job(*job:* PipelinerJob, *run_in_foreground: bool = False*)

Run a job that has been scheduled

Parameters

- **job** (PipelinerJob) – The job to run
- **run_in_foreground** (*bool*) – Should the job be run in the foreground instead of being spun off as a separate process, for testing

schedule_continue_job(*job_to_continue: str*, *params_dict: dict | None = None*, *comments: str | None = None*) → *PipelinerJob*

Schedule a job to run

Adds the job to the pipeline with scheduled status, does not run it

Parameters

- **job_to_continue** (*str*) – the name of the job to continue
- **params_dict** (*dict*) – Parameters to change in the continuation job.star file. {param name: value}
- **comments** (*str*) – comments to add to the job's jobinfo file

Returns

The
PipelinerJob object for the newly scheduled job

Return type
PipelinerJob

schedule_job(*job_input: str | Dict[str, str | float | int | bool]*, *comment: str | None = None*, *alias: str | None = None*) → *PipelinerJob*

Schedule a job to run

Adds the job to the pipeline with scheduled status, does not run it

Parameters

- **job_input** (*str*) – The path to a run.job or job.star file that defines the parameters for the job or a dictionary containing job parameters
- **comment** (*str*) – Comments to put in the job's jobinfo file
- **alias** (*str*) – Alias to give to the job

Returns
The *PipelinerJob* object for the scheduled job

set_alias(*job: str*, *new_alias: str | None*)

Set the alias for a job

Parameters

- **job** (*str*) – The name of the job to set the alias for
- **new_alias** (*str*) – The new alias

Raises
ValueError – If the alias could not be set for any reason.

stop_schedule(*schedule_name: str*) → bool

Stops a currently running schedule

Kills the process running the schedule and marks the currently running job as aborted. Works to stop schedules that were started using the RELION GUI or pipeliner.

Parameters
schedule_name (*str*) – The name of the schedule to stop

Returns
`True` If the schedule was stopped, `False` if the schedule could not be found to stop

Return type
bool

undelete_job(*job: str*) → bool

Restores a job from the Trash back into the project

Also restores the job's alias if one existed

Parameters
job (*str*) – The job to undelete

Returns
`True` If a job was restored, otherwise `False`

Return type
bool

update_job_status(*job: str*, *new_status: str*)

Mark a job as finished, failed or aborted

If is_failed and is_aborted are both `False` the job is marked as finished.

Parameters

- **job** (*str*) – The name of the job to update
- **new_status** (*str*) – The new status for the job; Choose from *"Running"*, , *"Scheduled"*, *"Succeeded"*, *"Failed* or *"Aborted"*. Status names are not case sensitive

Raises
ValueError – If the new status is not one of the options

pipeliner.api.manage_project.**convert_pipeline**(*pipeline_name: str = 'default'*) → bool

Converts a pipeline file from the RELION 2.0-3.1 format

This format has integer node, process, and status IDs. The pipeliner format uses string IDs

Parameters
pipeline_name (*str*) – The name of the pipeline to be converted

Returns

The result of the conversion

`True` if the pipeline was converted, `False` if the pipeline was already in pipeliner format

Return type
bool

pipeliner.api.manage_project.**delete_summary_data_archive**(*filename: str*)

Remove an archive from the summary data

Args: filename (str): The name of the archive zip file or dir

pipeliner.api.manage_project.**delete_summary_data_metadata_report**(*filename: str*)

Remove a metadata report from the summary data

Args: filename (str): The name of the report file

pipeliner.api.manage_project.**delete_summary_data_reference_report**(*filename: str*)

Remove a reference report from the summary data

Args: filename (str): The name of the report file

pipeliner.api.manage_project.**get_archives_list_from_summary_file**() → Tuple[List[str], List[List[str]]]

Get a list of the reference reports in the summary file

Returns
([column, headers], [[line, 1, data], [line, 2, data]]

Return type
tuple

pipeliner.api.manage_project.**get_commands_and_nodes**(*job_file: str*) → tuple

Tell what commands a job file would return and nodes that would be created

Parameters
job_file (*str*) – The path to a run.job or job.star file

Returns

- A list of commands. Each item in the commands list is a list of commands arguments. IE: `[[com1-arg1, com1-arg2],[com2-arg1]]`
- A list of input nodes that would be created. Each item in the list is a tuple: `[(name, type), (name, type)]`

- A list of output nodes that would be created. Each item in the list is a tuple: `[(name, type), (name, type)]`
- A list of any PipelinerWarning raised by joboption validation
- A list of the ExternalProgram objects used by the job

Return type
tuple

`pipeliner.api.manage_project.get_metadata_reports_from_summary_file() → Tuple[List[str], List[List[str]]]`

Get a list of the metadata reports in the summary file

Returns
([column, headers], [[line, 1, data], [line, 2, data]]

Return type
tuple

`pipeliner.api.manage_project.get_ref_reports_from_summary_file() → Tuple[List[str], List[List[str]]]`

Get a list of the reference reports in the summary file

Returns
([column, headers], [[line, 1, data], [line, 2, data]]

Return type
tuple

`pipeliner.api.manage_project.look_for_project(pipeline_name: str = 'default') → dict | None`

See if a pipeliner project exists in the current directory

Parameters
pipeline_name (*str*) – The name of the pipeline to look for. This is the same as the pipeline file name with "_pipeline.star" removed.

Returns
Info about the project, as a `dict`, or `None` if there is no existing project.

4.1.2 api_utilities

Utility functions do not require an existing project

`pipeliner.api.api_utils.edit_jobstar(fn_template: str, params_to_change: dict, out_fn: str) → str`

Modify one or more parameters in a job.star file

Parameters
- **fn_template** (*str*) – The name of the job.star file to use as a template
- **params_to_change** (*dict*) – The parameters to change in the format `{param_name: new_value}`
- **out_fn** (*str*) – Name for the new file to be written

Returns
The name of the output file written

Return type
str

pipeliner.api.api_utils.**get_job_info**(*job_type: str*) → *JobInfo* | None

Get information about a job

Parameters
job_type (*str*) – The type of job to return info on

Returns
JobInfo object with info about the job and it's references

Return type
JobInfo

Raises
ValueError – If the job type is not found

pipeliner.api.api_utils.**job_default_parameters_dict**(*jobtype: str*) → dict

Get dictionary of a job's parameters

Parameters
jobtype (*str*) – The type of job to get the dict for

Returns

The parameters dict. Suitable for running a job from
run_job()

Return type
dict

pipeliner.api.api_utils.**validate_starfile**(*fn_in: str*)

Checks for inappropriate use of reserved words in starfiles

Writes a corrected version with proper quotation if possible. The original file is saved with a '.orig' suffix added.

Parameters
fn_in (*str*) – The name of the file to check

pipeliner.api.api_utils.**write_default_jobstar**(*job_type: str*, *out_fn: str | None = None*, *relionstyle: bool = False*)

Write a job.star file for the specified type of job

The default jobstar contains all the job options with their values set as the defaults

Parameters

- **job_type** (*str*) – The type of job
- **out_fn** (*str*) – Name of the file to write the output to. If left blank defaults to *<job_type>_job.star*
- **relionstyle** (*bool*) – Should the job.star files be written in the relion format? Relion files are compatible with the pipeliner, but the pipeliner versions are not back compatible with Relion. If this option is selected a Relion job type should be used for job_type

Returns
The name of the output file written

Return type
str

pipeliner.api.api_utils.**write_default_runjob**(*job_type: str*, *out_fn: str | None = None*) → str

Write a run.job file for the specified type of job

The default runjob contains all the job option labels with their values set as the defaults

Parameters

- **job_type** (*str*) – The type of job
- **out_fn** (*str*) – Name of the file to write the output to. If left blank defaults to *<job_type>_run.job*

Returns
The name of the output file written

Return type
str

4.1.3 user_settings

User settings which can control the pipeliner's behaviour, environment and default job option values.

user_settings.py

Settings for the CCP-EM pipeliner.

Settings can be provided by the user in a JSON-formatted settings file or in environment variables. When the pipeliner is first run, a new settings file is created in a platform-specific directory (typically `~/.config/ccpem/pipeliner/` on Linux or `~/Library/Application Support/ccpem/pipeliner/` on Mac). This contains default settings values. These settings can be edited in that file, or overridden by setting environment variables before running the pipeliner. Both RELION- and Pipeliner-type environment variables are supported. For example, the setting for the default qsub script template can be set by editing the value for `"qsub_template"` in the settings file or by setting either the `PIPELINER_QSUB_TEMPLATE` or `RELION_QSUB_TEMPLATE` environment variable. If more than one of these is set, `PIPELINER_QSUB_TEMPLATE` will be used by preference, followed by `RELION_QSUB_TEMPLATE`, followed by the value from the settings file.

For programmatic access to settings values, use the helper functions which are provided at the module level (e.g. `user_settings.get_qsub_template()`. These allow easy access to the correctly-typed value of each individual setting.

This module also contains a Settings class and some internal functions, but there should be no need to access any of these directly from outside this module. To add a new setting, create a new `get_<setting name>()` function at the top of this module and add a new setting definition in `Settings.add_all_settings()`, making sure to use the correct types for the function calls and default value.

class pipeliner.user_settings.**BoolSettingDefinition**(*name: str*, *env_vars: List[str]*, *default: bool*)

Bases: *SettingDefinition*

Class for defining a setting with a bool value.

default: bool

static get_value_from_string(*value_str: str*) → bool

class pipeliner.user_settings.**IntSettingDefinition**(*name: str*, *env_vars: List[str]*, *default: int*)

Bases: *SettingDefinition*

Class for defining a setting with an int value.

default: int

static get_value_from_string(*value_str: str*) → int

class pipeliner.user_settings.**OptionalIntSettingDefinition**(*name: str*, *env_vars: List[str]*, *default: int | None*)

Bases: *SettingDefinition*

Class for defining a setting with an optional int value (i.e. int or None).

default: int | None

static get_value_from_string(*value_str: str*) → int | None

class pipeliner.user_settings.**OptionalStringSettingDefinition**(*name: str*, *env_vars: List[str]*, *default: str | None*)

Bases: *SettingDefinition*

Class for defining a setting with a string value.

default: str | None

static get_value_from_string(*value_str: str | None*) → str

class pipeliner.user_settings.**PathListSettingDefinition**(*name: str*, *env_vars: List[str]*, *default: List[str]*)

Bases: *SettingDefinition*

Class for defining a setting with a value which is a list of file paths.

In JSON format, the value should appear as a list of strings. When stored in an environment variable, the value should take the form of a single string with the individual paths separated by `os.pathsep` (typically `:`).

default: List[str]

static get_value_from_string(*value_str: str*) → List[str]

class pipeliner.user_settings.**SettingDefinition**(*name: str*, *env_vars: List[str]*, *default: Any*)

Bases: `object`

Base class for setting definitions. Not intended to be used directly.

default: Any

env_vars: List[str]

static get_value_from_string(*value_str: str*) → Any

name: str

class pipeliner.user_settings.**Settings**(*settings_file_override=None*)

Bases: `object`

Container class to hold settings definitions and the `get_<type>()` functions that fetch their values.

add_all_settings()

add_bool_setting(*, *name: str*, *env_vars: List[str]*, *default: bool*)

add_int_setting(*, *name: str*, *env_vars: List[str]*, *default: int*)

add_optional_int_setting(*, *name: str*, *env_vars: List[str]*, *default: int | None*)

add_optional_string_setting(*, *name: str*, *env_vars: List[str]*, *default: str | None*)

add_path_list_setting(*, *name: str*, *env_vars: List[str]*, *default: List[str]*)

add_string_setting(*, *name: str*, *env_vars: List[str]*, *default: str*)

check_for_extra_keys()

get_bool(*name: str*) → bool

get_int(*name: str*) → int

get_list(*name: str*) → List[str]

get_optional_int(*name: str*) → int | None

get_optional_string(*name: str*) → str

get_qsub_extras(*number: int*) → Dict[str, str]

Add definitions for the requested qsub extra settings and get their values

get_setting_value(*name: str*)

get_string(*name: str*) → str

class pipeliner.user_settings.**StringSettingDefinition**(*name: str*, *env_vars: List[str]*, *default: str*)

Bases: *SettingDefinition*

Class for defining a setting with a string value.

default: **str**

static get_value_from_string(*value_str: str*) → str

pipeliner.user_settings.**get_additional_program_paths**() → List[str]

Paths to prepend to the PATH environment variable before searching for program executables. Use this setting to make third-party software available to the pipeliner. Note that the paths should be to the directory containing the executable, not to the executable itself.

In JSON format, this setting is a list of strings. It can also be set using the PIPELINER_ADDITIONAL_PROGRAM_PATHS environment variable, which should be a PATH-style string containing individual paths separated by path separators (typically ':' on Linux and Mac).

pipeliner.user_settings.**get_ccpem_share_dir**() → str

Path to the share dir in an installation of ccpem for rvapi to use

pipeliner.user_settings.**get_ctffind_executable**() → str

The default CTFFIND-4.1+ executable.

pipeliner.user_settings.**get_default_nrmpi**() → int

The default for 'Number of MPI procs'.

pipeliner.user_settings.**get_default_nrthreads**() → int

The default for 'Number of threads'.

pipeliner.user_settings.**get_gctf_executable**() → str

The default Gctf executable.

pipeliner.user_settings.**get_minimum_dedicated**() → int

The default for 'Minimum dedicated cores per node'.

pipeliner.user_settings.**get_modelcraft_executable**() → str

The default Modelcraft executable.

pipeliner.user_settings.**get_motioncor2_executable**() → str

The default MotionCor2 executable.

pipeliner.user_settings.**get_mpi_max**() → int | None

The maximum number of MPI processes available from the GUI.

pipeliner.user_settings.**get_mpirun_command**() → str

The default command prepended to MPI jobs, including 'XXXmpinodesXXX' which will be substituted with the number of MPI nodes to use.

pipeliner.user_settings.**get_path_to_source_files**() → List[str]

Script files to set up the environment variables for the pipeliner. These files will be sourced when the pipeliner starts up and their environment read in and used to update the pipeliner's environment variables. This is a clunky mechanism intended mainly for setting up CCP4 and the old version of CCP-EM. Avoid using it for new packages.

In JSON format, this setting is a list of strings. It can also be set using the PIPELINER_PATH_TO_SOURCE_FILES environment variable, which should be a PATH-style string containing individual paths separated by path separators (typically ':' on Linux and Mac).

pipeliner.user_settings.**get_qsub_command**() → str

The default for 'Queue submit command'.

pipeliner.user_settings.**get_qsub_extra_count**() → int

The number of extra qsub template substitution variables to use.

pipeliner.user_settings.**get_qsub_extras**(*number: int*) → Dict[str, str]

Return a dictionary of values for a single numbered set of extra qsub settings.

For example:

```
get_qsub_extras(1) -> {
    "name": "Label for qsub_extra1",
    "default": "Default value for qsub_extra1",
    "help": "Help text for qsub_extra1",
}
```

pipeliner.user_settings.**get_qsub_template**() → str

The default queue submission script template.

pipeliner.user_settings.**get_queue_name**() → str

The default for 'Queue Name".

pipeliner.user_settings.**get_queue_use**() → bool

The default for 'Submit to queue?'.

pipeliner.user_settings.**get_resmap_executable**() → str

The default ResMap executable.

pipeliner.user_settings.**get_scratch_dir**() → str

The default scratch directory.

pipeliner.user_settings.**get_thread_max**() → int | None

The maximum number of threads per MPI process available from the GUI.

pipeliner.user_settings.**get_topaz_executable**() → str

The default Topaz executable.

pipeliner.user_settings.**get_warning_local_mpi**() → int | None

Warn if users try to submit local jobs with more than this many MPI nodes.

CHAPTER

FIVE

COMMAND LINE TOOLS

5.1 Command Line Tools

The `pipeliner` command allows pipeliner functions to be run from the command line.

CCPEM Pipeliner command line utility

```
usage: pipeliner [-h] [--new_project [project name]]
                 [--available_jobs [search string]]
                 [--check_jobs [search string]]
                 [--run_job [run.job or job.star file]]
                 [--overwrite [job name]] [--continue_job [job name]]
                 [--abort_job [job name]]
                 [--print_command [run.job or job.star file]]
                 [--schedule_job [run.job, job.star file or job name]]
                 [--run_schedule] [--name [NAME]]
                 [--jobs [job name [job name ...]]] [--min_between [n]]
                 [--nr_repeats [1]] [--wait_sec_after [2]]
                 [--wait_min_before [0]] [--stop_schedule [STOP_SCHEDULE]]
                 [--delete_job [job name]] [--undelete_job [job name]]
                 [--set_alias [job name] [new alias]]
                 [--clear_alias [job name]] [--set_status [job name]
                 {finished, failed, aborted}]
                 [--cleanup [job name [job name ...]]] [--harsh]
                 [--validate_starfile [star file]]
                 [--convert_pipeline_file [_pipeline.star file]]
                 [--default_jobstar [job type]] [--relionstyle]
                 [--default_runjob [job type]] [--job_info [job type]]
                 [--empty_trash] [--job_references [job type]]
                 [--references_report [job name]]
                 [--metadata_report [terminal job]]
                 [--full_archive [terminal job name]]
                 [--simple_archive [terminal job name]]
                 [--empiar_deposition [terminal job name]]
                 [--onedep_deposition [terminal job name]]
                 [--deposition_id [Deposition id Assigned by the database]]
                 [--empiar_jobstar_file [Template file to update]]
                 [--empiar_movies] [--empiar_mics] [--empiar_parts]
                 [--empiar_corr_parts]
```

5.1.1 Arguments

--new_project Initialize a new project in this directory. Project name is optional, if none is specified the project will be called 'default' (recommended)

--available_jobs Show a list of all available jobs, optionally add a search string to limit the search, Leave blank to show all available job types

--check_jobs Get a list of available and unavailable jobs with detailed info onthe programs they use and why they are unavailable (if applicable)

5.1.2 Running jobs

--run_job Create a job using a run.job or job.star file to get the parameters

--overwrite Use with –run_job to overwrite a current job rather than making a new job. Jobs can only be overwritten with the same job type

--continue_job Continue a job that has been previously run. Edit the job's continue_job.star file to modify parameters for the continuation

--abort_job Abort a running job

--print_command Read a job.star or run.job file and print the command(s) it would produce

--schedule_job Add a job to list of scheduled jobs using a run.job or job.star file to get the parameters. If followed by a job.star file it will create a new job if followed by a job name it will schedule a continuation of that job

5.1.3 Executing Schedules

--run_schedule Create a schedule choosing from the currently scheduled jobs and run it

Default: False

--name (required) Enter a name for the new schedule

--jobs (required) Enter the jobs that will be run. Make sure to list the jobs in the order which they should be run

--min_between (optional) Wait at least this many minutes between jobs

Default: 0

--nr_repeats (optional) Repeat the schedule this many times

Default: 1

--wait_sec_after (optional) Wait this many seconds after finishing before starting the next job

Default: 2

--wait_min_before (optional) Wait this many minutes before starting the schedule

Default: 0

--stop_schedule (required) Enter a name for the schedule to be stopped

5.1.4 Deleting jobs

--delete_job Remove job(s) and put in the trash, deletes the job and all of its child processes

5.1.5 Undeleting jobs

--undelete_job Restore a deleted job and any of its deleted parent processes from the trash

5.1.6 Modifying jobs

--set_alias Set the alias of a job

--clear_alias Clear the alias of a job

--set_status Set the status of a job; choose from 'finished, failed, or aborted

5.1.7 Cleaning Up Job(s)

--cleanup Delete intermediate files from these job(s) to save disk space; enter ALL to clean up all jobs

--harsh Add this argument to –cleanup to delete even more files

Default: False

5.1.8 Utilities

--validate_starfile Check a star file and make sure it is written in the correct format. If errors are found will attempt to fix them

--convert_pipeline_file Convert a pipeline file from Relion 3.1 to the CCPEM pipeliner format

--default_jobstar Make a job.star file with the default values for a specific job type for use with the ccpem-pipeliner

--relionstyle OPTIONAL: Add this argument to –default_jobstar to write job.star files which are compatible with RELION 4.0. RELIONstyle job.star files are fully compatible with the pipeliner, pipeliner job.star files may havedifferences that cause bugs in RELION

Default: False

--default_runjob Make a _run.job file with the default values for a specific job type. These files can also be used to run jobs and a more human readable

--job_info Get info about a specific job type, including any reference(s)

--empty_trash Delete the files in the trash. THIS CANNOT BE UNDONE!

Default: False

--job_references Get literature reference(s) for a specific job type

--references_report Get all the literature references for a project with the selected job as the termial job

5.1.9 Project Analysis

--metadata_report Prepares a report in .json format for the terminal job and allof its parent jobs

5.1.10 Project Archiving

--full_archive Create a full archive for a project. This will contain the entire job dirs for the stated terminal job and all of its parents

--simple_archive Create a simple archive for a project. This will contain just the directory structure and parameter files for the stated terminal job andall of it's parents along with a script to automatically re-run the project through the terminal job

5.1.11 Prepare EMDB/EMPIAR/PDB depositions

--empiar_deposition Prepare a deposition for upload to the EMPIAR database

--onedep_deposition Prepare a deposition for upload to the pdb/emdb databases

--deposition_id [Optional] The deposition ID. This must be assigned by EBI. If not specified, a draft deposition will be created

--empiar_jobstar_file (optional) A jobstar file from a pipeliner.deposition.empiar job. This contains data necessary from the deposition that cannot be automatically found. If not included, the deposition will only contain the data that can be found and will need to be manually completed

--empiar_movies (optional) Add this argument to deposition to include unprocessed movies in the EMPIAR deposition

Default: False

--empiar_mics (optional) Add this argument to deposition to include corrected micrographs in the EMPIAR deposition

Default: False

--empiar_parts (optional) Add this argument to deposition to include particles in the EMPIAR deposition

Default: False

--empiar_corr_parts (optional) Add this argument to deposition to include corrected (polished) particles in the EMPIAR deposition

Default: False

CHAPTER

SIX

CORE MODULES

6.1 Pipeline Tools

6.1.1 Nodes and Processes

6.1.2 ProjectGraph

The ProjectGraph handles all the management of the pipeline keeping track of which jobs have been run, their statuses, and what files are input and outputs to the various jobs

class pipeliner.project_graph.**ProjectGraph**(*name: str = 'default'*, *pipeline_dir: str | os.PathLike[str] = '.'*, *read_only: bool = True*, *create_new: bool = False*)

Bases: object

The main ProjectGraph object is used for manipulating the pipeline

node_list

A *Node* object for every file that is an input or output for a job in the project

Type
list

process_list

A *Process* object for each job in the project

Type
list

job_counter

The number of the *next* job in the project IE: If there are 10 jobs in a project job_counter is 11

Type
int

add_job(*job:* PipelinerJob, *as_status: str*, *do_overwrite: bool*, *alias: str | None = None*) → *Process*

Add a job to the pipeline

Adds the *Process* for the job, a *Node* for each of its input and output files, and writes a mini-pipeline containing just that job

Parameters

- **job** (*PipelinerJob*) – The job to add.
- **as_status** (*str*) – The status of the job in the pipeline

- **do_overwrite** (*bool*) – If the job already exists, should it be overwritten?
- **alias** (*str*) – Alias to assign to job

Returns
The *Process* for the new job

Return type
Process

add_new_input_edge(*node:* Node, *process:* Process) → None

Add a *Node* to a *Process* as in input

Parameters

- **node** (*Node*) – The node to add
- **process** (*Process*) – The Process to add the Node to

add_new_output_edge(*process:* Process, *node:* Node, *mini:* *bool* = *False*) → None

Add a *Node* to a *Process* as in output

Parameters

- **node** (*Node*) – The node to add
- **process** (*Process*) – The Process to add the Node to
- **mini** (*bool*) – Is the pipeline being operated on a mini pipeline written inside a job directory

add_node(*node:* Node, *touch_if_not_exists:* *bool* = *False*) → *Node* | None

Add a *Node* to the pipeline

A node is only added if it doesn't already exist in the pipeline, so if a node is used as an input the keywords from the process that wrote the node will overrule any added in the node's definition from the process that used it as input

Parameters

- **node** (*Node*) – The node to add
- **touch_if_not_exists** (*bool*) – If the file for the node does not exist should it be created?

Returns
The Node that was added. If this node already existed, returns the existing copy

Return type
Node

Raises
RuntimeError – If the node name is empty

add_process(*process:* Process, *do_overwrite:* *bool*) → *Process*

Add a *Process* to the pipeline

Parameters

- **process** (*Process*) – The *Process* to add
- **do_overwrite** (*bool*) – If the process already exists should it be overwritten?

Returns
The *Process* that was added or the existing *Process* if it already existed

Return type
(*Process*)

Raises
RuntimeError – If the *Process* already exists and overwrite is `False`

alias_checks(*process*, *new_alias*) → str

A set of checks for alias to make sure they fit the requirements

Parameters

- **process** (Process) – The Process object that is getting the new alias
- **new_alias** (*str*) – The alias that will be applied to the process

check_process_completion() → None

Check to see if any processes have finished running, update their status.

static check_process_status(*proc:* Process, *finished: List[*Process*]*, *failed: List[*Process*]*, *aborted: List[*Process*]*) → Tuple[List[*Process*], List[*Process*], List[*Process*]]

Check if a job has a completion status file and assign to the right list

Parameters

- **proc** (Process) – The process to check
- **finished** (*List[*Process*]*) – Process that are finished successfully
- **failed** (*List[*Process*]*) – Processes that have failed
- **aborted** (*List[*Process*]*) – Processes that were killed by the user

Returns
The updated input lists

Return type
Tuple[List[*Process*], List[*Process*], List[*Process*]]

clean_up_job(*process:* Process, *do_harsh: bool*) → bool

Cleans up a job by deleting intermediate files

Gets a list of files to delete from the specific job's cleanup function. First checks that none of the files that are slated for deletion are on the Node list or are of a few specific types that RELION needs. Then moves all the intermediate files to the trash

There are two tiers of clean up 'harsh' and normal. Each job defines what files are cleaned up by each cleanup type

Parameters

- **process** (*Process*) – The *Process* of the job to clean up
- **do_harsh** (*bool*) – Should harsh cleaning be performed?

Returns
`True` if cleaning was performed `False` if the specified job had no clean up method or it was protected from harsh cleaning

Return type
bool

cleanup_all_jobs(*do_harsh: bool*) → int

Clean up all jobs in the project

Parameters
do_harsh (*bool*) – Should harsh cleaning be performed?

Returns
The number of jobs that were successfully cleaned

Return type
int

close() → None

Close the pipeline by releasing the lock.

After closing, this object will be read-only.

Note that this does not write the pipeline to disk. If you have made changes that should be saved, call :meth:`_write` first or (preferably) open the pipeline in a context manager.

delete_job(*process:* Process) → None

Remove a job from the pipeline

:param process (*Process*:): The job to remove

delete_temp_node_file(*node:* Node) → None

Remove files associated with a *Node*

Also removes the directory if it is empty

Parameters
node (*Node*) – The node to remove the file for

delete_temp_node_files(*process:* Process) → None

Delete all the files for the nodes in a specific *Process*

Parameters
process (*Process*) – The Process to create the files for

find_immediate_child_processes(*process:* Process) → List[*Process*]

Find just the immediate child processes of a process

Parameters
process (*Process*) – The process to find children for

Returns
The *Process* object for each job connected to the input *Process*

Return type
list

find_node(*name: str*) → *Node* | None

Retrieve the *Node* object for a file

Parameters
name (*str*) – The name of the file to get the node for

Returns
The file's *Node* object. None if the file is not found.

Return type
Node

find_process(*name_or_alias: str*) → *Process* | None

Retrieve the *Process* object for a job in the pipeline

Parameters
name_or_alias (*str*) – The job name or its alias

Returns
The job's *Process*, or *None* if the job was not found.

Return type
pipeliner.process.Process

Raises
RuntimeError – If multiple processes with the same name are found

get_downstream_network(*process:* Process) → List[Tuple[*Node* | None, *Process* | None, *Process*]]

Gets data for drawing a network downstream from process

Parameters
process (*Process*) – The process to trace

Returns
Contains a tuple for each edge: (*Node*, parent *Process*, child *Process*). Each edge describes one file in the network.

Return type
list

static get_node_name(*node:* Node) → str

Get the relative path of a node file with its alias if it exists

This returns the relative path (which is the same as the file name) unless the job that created the node has an alias in which case it returns the file path with the alias instead of <jobtype>/jobxxx/

Parameters
node (*Node*) – The node to get the name for

Returns
The relative path of the file to node points to with the job's alias if applicable

Return type
str

get_pipeline_edges() → Tuple[List[Tuple[str, str]], List[Tuple[str, str]]]

Find the edges that track connections between processes

Returns
([input edges], [output edges]) input edges is a list of tuples (input file, process name) output edges is a list of tuples (process name, output file) Note that the order in the tuples is different in the input and output edges! This is done to match the order of the pipeline STAR file columns.

Return type
tuple

get_project_procs_list(*terminal_job: str*, *reverse: bool = True*) → List[*Process*]

make a list of all the process objects for a job and its parents

Parameters

- **terminal_job** (*str*) – The name of the final job in the workflow
- **reverse** (*bool*) – sort in reverse order (High to low)

Returns
The *Process* objects for the terminal job and its parents, in reverse order by job number

Return type
list

Raises
ValueError – If the terminal job was not found

get_upstream_network(*process:* Process) → List[Tuple[*Node* | None, *Process* | None, *Process*]]

Gets data for drawing a network upstream from process

Parameters
process (*Process*) – The process to trace

Returns
Contains a tuple for each edge: (*Node*, parent *Process*, child *Process*). Each edge describes one file in the network.

Return type
list

get_whole_project_network() → List[Tuple[*Node* | None, *Process* | None, *Process*]]

Get the edges and nodes for the entire project

Returns
Edges [nod type, parent_job, child_job, extra info] set: The names of all nodes

Return type
list

property name

The name of the pipeline, usually "default".

_pipeline.star is added to the name to generate the pipeline file name.

property pipeline_dir

The directory containing the pipeline file.

property read_only

If this object should be able to make changes to the pipeline.

remake_node_directory() → None

Erase and rewrite RELION's .Nodes directory

set_job_alias(*process:* Process | *None*, *new_alias:* *str* | *None*) → None

Set a job's alias

Sets the alias in the pipeline and creates the alias symlink that points to the job directory.

Parameters

- **process** (*Process*) – The *Process* to make an alias for
- **new_alias** (*str*) – The new alias for the job

Raises

- **ValueError** – If *process* is *None*
- **ValueError** – If the *Process* is not found
- **ValueError** – If the new alias is *None*, which is not allowed
- **ValueError** – If the new alias is shorter than 2 characters
- **ValueError** – If the new alias begins with "job", which would cause problems

- **ValueError** – If the new alias is not unique

property star_file

The name of the pipeline STAR file, usually "default_pipeline.star".

touch_temp_node_file(*node:* Node, *touch_if_not_exists:* *bool*) → None

Create a placeholder file for a node that will be created later

Parameters

- **node** (*Node*) – The node to create the file for
- **touch_if_not_exists** (*bool*) – Should the file be created if it does not already exist

touch_temp_node_files(*process:* Process) → None

Create placeholder files for all nodes in a *Process*

Parameters

process (*Process*) – The Process to create the files for

undelete_job(*del_job:* *str*) → None

Get job out of the trash and restore it to the pipeline

Also restores any alias the job may have had as long as it does not conflict wit the current aliases

Parameters

del_job (*str*) – The name of the deleted job

update_lock_message(*lock_message:* *str*) → None

Updates the contents of the lockfile for the pipeline

This enables the user to see which process has locked the pipeline

update_status(*the_proc:* Process, *new_status:* *str*) → None

Change the status of a job

The job can be marked as "Succeeded", "Failed", "Aborted", "Running" or "Scheduled"

Parameters

- **the_proc** (*Process*) – The *Process* to update the status for
- **new_status** (*str*) – The new status for the job

Returns

Was the status changed?

Return type

bool

Raises

- **ValueError** – If the new_status is not in the approved list
- **ValueError** – If a job with any other status than 'Running' is marked 'Aborted'
- **ValueError** – If a job's updated status is the same as it's current status

validate_pipeline_file() → Tuple[Document, str]

Check if the pipeline file is valid and what version it is

Returns

(cif.Document containing the pipeline data, pipeline version string)

Return type

Tuple

6.1.3 Metadata Tools

pipeliner.metadata_tools.**add_md_to_summary_data**(*filename: str*, *terminal_job: str*, *njobs: int*)

Add a metadata report to the summary file :param filename: The name of the report file :type filename: str :param terminal_job: The job the report was launched from :type terminal_job: str :param njobs: The number of jobs contained in the report :type njobs: int

pipeliner.metadata_tools.**format_for_metadata**(*joboption:* JobOption) → str | float | int | bool | None

Format data from relion starfiles for JSON.

Changes 'Yes'/'No" to True/False, None for blank vals and removes any quotation marks

Parameters
joboption (JobOption) – The joboption to format

Returns

A properly formatted str, float or
int or bool

Return type
str

Raises
ValueError – If a boolean job option doesn't have a value compatible with a class:*bool*"

pipeliner.metadata_tools.**get_job_metadata**(*this_job:* Process) → dict

Run a job's metadata gathering method

Parameters
this_job (*pipeliner.process.Process*) – The *Process* object for the job to gather metadata from

Returns
Metadata dict for the job Returns None if the job has no metadata gathering function

Return type
dict

pipeliner.metadata_tools.**get_metadata_chain**(*pipeline*, *this_job:* Process, *full: bool = False*) → Tuple[dict, int]

Get the metadata for a job and all its upstream jobs

Metadata is gathered by each individual job's gather_metadata function

Parameters

- **pipeline** (*ProjectGraph*) –
- **this_job** (*Process*) – The *Process* object for the terminal job
- **full** (*bool*) – Should the metadata report also contain information about continuations and multiple runs of the jobs or just the current one

Returns
(the metadata dict, number of jobs in it)

Return type
tuple

pipeliner.metadata_tools.**make_job_parameters_schema**(*job_type: str*) → dict

Write the json schema for the running parameters of a job

The metadata schema have two parts: the running parameters part is generated automatically and the results part is written by the user

Parameters
job_type (*str*) – The job type to write the schema for

Returns
The json schema ready to be dumped to json file

Return type
dict

pipeliner.metadata_tools.**make_job_results_schema**(*job_type: str*) → dict

pipeliner.metadata_tools.**remove_md_from_summary_data**(*filename: str*)

Remove a report from the summary file

Parameters
filename (*str*) – Name of the report file to remove

6.2 Tools for Running Jobs

The job factory, job manager and job runner are used to create, schedule and run jobs. These objects and the functions inside them generally do not need to be accessed directly, usually the desired action can be done more easily through the functions in the *pipeliner api*

6.2.1 The job factory

The job factory functions identify the available job types and return the correct type of job from the job type specified in a parameter file

pipeliner.job_factory.**active_job_from_proc**(*the_proc:* Process) → *PipelinerJob*

Create an active job from an existing process

Used when the functions inside a job subclass need to be called on an existing job

Parameters
the_proc (*Process*) – The process to create a job from

Returns

The job subclass
object for the process

Return type
PipelinerJob

pipeliner.job_factory.**get_job_from_entrypoint**(*type_name: str*) → *PipelinerJob* | None

pipeliner.job_factory.**get_job_types**(*search_term: str = ''*) → List[*PipelinerJob*]

Returns all the job types and info about them

Note that the returned list actually contains job instances, not classes.

Parameters
search_term (*str*) – Only return jobs with this string in their job type name

Returns

A list of the *PipelinerJob*-based job
objects for each job found

Return type
list

pipeliner.job_factory.**job_can_run**(*job_type: str*) → bool

Check that the programs are available to run a specific job type

Parameters
job_type (*str*) – The job type IE relion.class3d.helical

Returns
Are the programs needed to run that job available on this system

Return type
bool

pipeliner.job_factory.**job_from_dict**(*job_input: Mapping[str, str | int | float | bool]*) → *PipelinerJob*

Create a job from a dictionary

The dict must define the job type with a '_rlnJobTypeLabel' key. Any other keys will override the default options for that parameter

Parameters
job_input (*dict*) – The dict containing the params. At minimum, it must contain *{'_rlnJobTypeLabel': <jobtype>}*

Returns
The job subclass

Return type
PipelinerJob

Raises

- **ValueError** – If the '_rlnJobTypeLabel' key is missing from the dict or is not a string
- **ValueError** – If '_rlnJobIsContinue' is in the dict - this function is not for creating continuations of jobs
- **ValueError** – If the specified jobtype is not found
- **ValueError** – If any of the parameters in the dict are not in the jobtype returned

pipeliner.job_factory.**new_job_of_type**(*type_name: str*, *joboptions: Mapping[str, str | int | float | bool] | None = None*) → *PipelinerJob*

Creates a new object of the correct PipelinerJob sub-type

Parameters

- **type_name** (*str*) – The job process name
- **joboptions** (*dict*) – Dict of the job's joboptions, only necessary if converting from a RELION4.0 style jobname

Returns
The job subclass

Return type
PipelinerJob

Raises
ValueError – If the job type is not found

pipeliner.job_factory.**read_job**(*filename: str*) → *PipelinerJob*

Reads run.job and job.star files and returns the correct Pipeliner job class

Parameters
filename (*str*) – The run.job or job.star file

Returns
The job subclass

Return type
PipelinerJob

Raises

- ValueError – If the file name entered does not end with .job or .star
- RuntimeError – If the input file is in the RELION 3.1 format and conversion fails
- RuntimeError – If the job type specified in the file cannot be found

6.2.2 The job manager

pipeliner.job_manager.**abort_job**(*pipeline:* ProjectGraph, *job_name: str*) → None

Abort a running job.

This function writes a file to signal that the job should abort, but does not wait for the job to respond.

Parameters

- **pipeline** – A writable (i.e. non-read-only) ProjectGraph object
- **job_name** – The job name to abort (including a trailing slash "/")

Raises

- ValueError – If there is no job with the given name
- RuntimeError – If the job is in any state except Running

pipeliner.job_manager.**run_job**(*pipeline:* ProjectGraph, *job:* PipelinerJob, *alias: str | None = None*, *overwrite:* Process | *None = None*, *ignore_invalid_joboptions=False*, *run_in_foreground=False*) → *Process*

Run a job.

The job will first be scheduled by passing it to `schedule_job` and then run by calling `run_scheduled_job`. See those functions for more details.

Parameters

- **pipeline** – A writable (i.e. non-read-only) `ProjectGraph` object to add the new job to.
- **job** – The job to run.
- **alias** – An optional alias for the job, which can be used to refer to the job instead of its number.
- **overwrite** – An optional Process object representing an old job (of the same type) to be overwritten and replaced by the new job.
- **ignore_invalid_joboptions** (*bool*) – Run the job anyway even if the job options appear to be invalid.

- **run_in_foreground** – If True, run the job in the foreground and do not return until it has finished. If False, the job will be launched in the background and this function will return immediately. This option is mainly intended for testing of the pipeliner. Note that if `run_in_foreground` is True, the project will be kept locked by this process for the entire duration of the job, preventing any other actions happening in the project in the meantime.

Returns
The Process object representing the job.

Raises

- `ValueError` – If `overwrite` is given, but the old job is a different type or other processes in the pipeline have used its outputs.
- `ValueError` – If `job.is_continue` is True but the job does not already have an output directory assigned.
- `ValueError` – If `ignore_invalid_joboptions` is False and the job options appear to be invalid.
- `ValueError` – If `run_in_foreground` is True and the job is set to be submitted to a queue.
- `RuntimeError` – If `pipeline.read_only` is True.

pipeliner.job_manager.**run_schedule**(*fn_sched: str*, *job_ids: List[str] | None = None*, *nr_repeat: int = 1*, *minutes_wait: int = 0*, *minutes_wait_before: int = 0*, *seconds_wait_after: int = 0*, *pipeline_name: str = 'default'*) → None

Run the jobs in a schedule.

Parameters

- **fn_sched** (*str*) – The name to be assigned to the schedule
- **job_ids** (*list*) – A list of `str` job names
- **nr_repeat** (*int*) – Number of times to repeat the entire schedule
- **minutes_wait** (*int*) – Minimum time to wait between jobs in minutes. If this has been passed whilst the job is running the next job will start immediately
- **minutes_wait_before** (*int*) – Wait this amount of time before initially starting to run the schedule
- **seconds_wait_after** (*int*) – Wait this many seconds before starting each job this wait always occurs, even if the minimum time between jobs has already been surpassed
- **pipeline_name** (*str*) – The name of the pipeline to use

Raises

- `ValueError` – If a schedule lock file exists with the selected schedule name, suggesting another schedule with the same name is already running
- `ValueError` – If the job directory for a scheduled job could not be found
- `ValueError` – (If a job.star file is not found in one of the directories of a job to be run
- `ValueError` – If an input node for a job cannot be found
- `ValueError` – If a job in the schedule fails

pipeliner.job_manager.**run_scheduled_job**(*pipeline: ProjectGraph*, *job: PipelinerJob*, *process: Process*, *run_in_foreground=False*) → *Process*

Run a job that has already been scheduled.

By default, the job will be launched in the background in a detached process. If necessary (mainly for testing) the job can instead be run in the foreground by passing `run_in_foreground=True`. Note that this option overrides the job's queue submission settings.

Parameters

- **pipeline** – A writable (i.e. non-read-only) `ProjectGraph` pipeline containing the scheduled job.
- **job** – The job to run.
- **process** – The Process object representing the job's entry in the pipeline.
- **run_in_foreground** – If True, run the job in the foreground and do not return until it has finished. If False, the job will be launched in the background and this function will return immediately. This option is mainly intended for testing of the pipeliner. Note that if `run_in_foreground` is True, the project will be kept locked by this process for the entire duration of the job, preventing any other actions happening in the project in the meantime.

Returns

The Process object representing the job. Note that this might be a new and different object from the one passed in to the `process` argument.

Raises

- **ValueError** – If the job is not already present in the pipeline in the Scheduled state.
- **FileNotFoundError** – If the job's "job.star" file cannot be found in the job directory.
- **RuntimeError** – If `pipeline.read_only` is True.

`pipeliner.job_manager.`**schedule_job**(*pipeline:* ProjectGraph, *job:* PipelinerJob, *alias: str | None = None*, *overwrite:* Process | *None = None*, *ignore_invalid_joboptions=False*) → *Process*

Schedule a job to run later.

Note that jobs must be scheduled before they can be run.

The job will be added to the pipeline in the Scheduled state, and the job's options will be written to "run.job" and "job.star" files in the job directory.

If the job is new, it will be assigned a job number and a new output directory will be created for it. If the job is a continuation (i.e. `job.is_continue` is True) the previous job directory and number will be used. If `overwrite` is given, the job in `overwrite.name` will be deleted and replaced with the new job (as long as the old and new jobs are of the same type).

If any of the job's input files are in the project "DoppioUploads" directory, they will be moved into an "InputFiles" directory inside the job directory, and the relevant job options will be updated to point to the new file locations.

Parameters

- **pipeline** – A writable (i.e. non-read-only) `ProjectGraph` object to add the new job to.
- **job** – The job to schedule and add to the pipeline.
- **alias** – An optional alias for the job, which can be used to refer to the job instead of its number.
- **overwrite** – An optional Process object representing an old job (of the same type) to be overwritten and replaced by the new job.
- **ignore_invalid_joboptions** (*bool*) – Schedule the job anyway even if the job options appear to be invalid.

Returns
The Process object representing the newly-scheduled job.

Raises

- `ValueError` – If `overwrite` is given, but the old job is a different type or other processes in the pipeline have used its outputs.
- `ValueError` – If `job.is_continue` is True but the job does not already have an output directory assigned.
- `ValueError` – If `ignore_invalid_joboptions` is False and the job options appear to be invalid.
- `RuntimeError` – If `pipeline.read_only` is True.

`pipeliner.job_manager.`**`wait_for_job_to_finish`**(*job:* PipelinerJob, *ping:* *float* = *1.0*, *timeout:* *float* = *86400.0*, *error_on_fail:* *bool* = *False*, *error_on_abort:* *bool* = *False*) → str

Wait for the job to finish, with any status

Parameters

- **job** (`PipelinerJob`) – The job to wait for
- **ping** (*float*) – How long to wait before checking for the file again (in seconds)
- **timeout** (*float*) – Raise an error after this much time has elapsed, even if the job hasn't finished (in seconds). The default is 24 hours.
- **error_on_fail** (*bool*) – Raise an error if the job fails
- **error_on_abort** (*bool*) – Raise an error if the job is aborted

Returns
The final status of the job, Failed, Succeeded, or Aborted

Return type
str

Raises

- `RuntimeError` – upon the watched job failing, being aborted, or either if error_on_fail or error_on_abort are True
- `RuntimeError` – If the job still hasn't finished by the timeout time

6.2.3 The job runner

job_runner.py

This script is responsible for executing pipeliner jobs.

It is intended only for internal use by the pipeliner and should not normally be called by any other code.

Users should instead run jobs using the "–run_job" argument to the standard "pipeliner" command.

6.3 Star File Utilities

class pipeliner.starfile_handler.**BodyFile**(*fn_in: str*)

Bases: *StarFile*

A star file that lists the bodies in a multibody refinement

bodycount

Number bodies in the file

Type
int

class pipeliner.starfile_handler.**DataStarFile**(*fn_in: str | os.PathLike[str]*)

Bases: *StarFile*

A general class for starfiles that contain data, such as particles files

data

A gemmi cif object containing the data from the star file

Type
gemmi.cif.Document

column_as_list(*blockname: str | None*, *colname: str*) → List[str]

Return a single column from a block as a list

Parameters

- **blockname** (*str*) – The name of the block to use
- **colname** (*str*) – The name of the column to get

Returns
The values from that column

Return type
list

count_block(*blockname: str | None = None*) → int

Count the number of items in a block that only contains a single loop

This is the format in most relion data star files

Parameters
blockname (*str*) – The name of the block to count

Returns
The count

Return type
int

get_block(*blockname: str | None = None*) → Block

Get a block from the star file

Parameters
blockname (*str*) – The name of the block to get. Use `None` if the file has a single unnamed block

Returns
The desired block or None if not found

Return type
(gemmi.cif.Block)

class pipeliner.starfile_handler.**JobStar**(*fn_in: str*)

Bases: *StarFile*

A class for star files that define a pipeliner job parameters

jobtype

the job type, converted from the relion nomenclature if necessary

Type
str

joboptions

The joboptions {name: value} all values are strings regardless of joboption type

Type
dict

is_continue

is the job a continuation

Type
bool

is_tomo

is the job tomography (might be removed when relon compatobility is deprecated)

Type
bool

all_options_as_dict() → Dict[str, str]

Returns a dict of all the parameters of a jobstar file

The dict contains both the job options and the running options. All values in the dict are strings.

make_substitutions(*conversions: Dict[str, str]*)

Substitute one job name or file name for another wherever it appears

Parameters
conversions (*Dict[str, str]*) – {"old job name": "new job name"}

modify(*params_to_change: dict*, *allow_add=False*)

Change multiple values in a jobstar from a dict

Parameters

- **params_to_change** (*dict*) – The Parameters to change in the template in the format {param_name: new_value}
- **allow_add** (*bool*) – Can new joboptions be added?

Raises
ValueError – If an attempt is made to add new fields with allow_add=False

write(*outname: str | os.PathLike[str] = ''*)

Write a star file from the JobStar

Parameters
outname (str or Path) – name of the output file; will overwrite the original file if none provided

class pipeliner.starfile_handler.**StarFile**(*fn_in: str | os.PathLike[str]*)

Bases: object

A superclass for all types of starfiles used by the pipeliner

file_name

The star file

Type
str or Path

loop_as_list(*block: str = ''*, *columns: List[str] | None = None*, *headers: bool = False*) → List[List[str]]

Returns a set of columns from a starfile loop as a list

Parameters

- **block** (*str*) – The name of the block to get the data from
- **columns** (*list*) – Names of the columns to get, if None then all columns are returned
- **headers** (*bool*) – Should the 1st row of the returned list be the header names

Returns

The column data one row per sublist. If headers=True
the first sublist contains the headers

Return type
List[List[str]]

Raises

- **ValueError** – If the specified block is not found
- **ValueError** – If the specified block does not contain a loop
- **ValueError** – If any of the specified columns are not found

pairs_as_dict(*block: str = ''*) → Dict[str, str]

Returns paired values from a starfile as a dict

Parameters
block (*str*) – The name of the block to get the data from

Returns
{parameter: value}

Return type
dict

Raises

- **ValueError** – If the specified block is not found
- **ValueError** – If the specified block is a loop and not a pair-value

pipeliner.starfile_handler.**compare_starfiles**(*starfile1: str*, *starfile2: str*) → Tuple[bool, bool]

See if two starfiles contain the same information

Direct comparison can be difficult because the starfile columns or blocks can be in different orders

Parameters

- **starfile1** (*str*) – Name of the first file
- **starfile2** (*str*) – Name of the second file

Returns
(bool, bool) [0] True if the starfile structures are identical (Names of blocks and columns) [1] True if the data in the two files are identical

Return type
tuple

pipeliner.starfile_handler.**convert_old_relion_pipeline**(*fn_in: str*)

Convert a pipeline STAR file from RELION 3 or 4 to ccpem-pipeliner format.

The file is also checked and corrected if it contains any reserved words which would cause Gemmi's CIF parser to fail when the file is read.

Note that this function reads the whole file several times, but pipeline STAR files are never expected to be so large that this would take a significant amount of time.

Returns
`True` if the file was converted, or `False` otherwise.

Raises
ValueError – if the pipeline file is invalid or its version cannot be determined

pipeliner.starfile_handler.**fix_reserved_words**(*fn_in*)

Make sure the starfile doesn't contain any illegally used reserved words

Overwrites the original file if it is corrected. The old file is saved as `filename.orig`.

Returns
`True` if the file was corrected, or `False` if it contained no reserved words and did not need correcting.

pipeliner.starfile_handler.**interconvert_box_star_coords**(*fn_in: str*, *out_name: str | None = None*) → str

Interconvert .box and .star coordinate files

Parameters

- **fn_in** (*str*) – The file to convert
- **out_name** (*str*) – Dir to write the output file into, if None writes it the same place as the input, with the same name just switching the extension

Returns
Name of the file written

Return type
str

Raises
ValueError – If the file is not .box or .star

6.3.1 Starfile writing utilities

Writes star files in the same style as RELION

pipeliner.star_writer.**write**(*doc: Document*, *filename: str | os.PathLike[str]*)

Write a Gemmi CIF document to a RELION-style STAR file.

The file is written as an atomic operation. This ensures that any processes reading it will always see a valid version (old or new) and not a half-written new file.

The document is written to a temporary file first, then after the writing is complete and the file has been flushed to disk, the target file is replaced with the temporary one. The temporary file will always be removed even if the replacement is unsuccessful.

Note: it is still possible for data to be lost using this function, if two processes try to write to the file at the same time. In that case, one of the new versions of the file will be kept and the other will not.

Parameters

- **doc** (gemmi.cif.Document) – The data to write out
- **filename** (*str or pathlib.Path*) – The name of the file to write the data to

pipeliner.star_writer.**write_jobstar**(*in_dict: dict*, *out_fn: str | os.PathLike[str]*)

Write a job.star file from a dictionary of options

Parameters

- **in_dict** (*dict*) – Dict of job option keys and values
- **out_fn** (str or Path) – Name of the file to write to

pipeliner.star_writer.**write_to_stream**(*doc: Document*, *out_stream: IO[str]*)

Write a Gemmi CIF document to an output stream using RELION's output style.

Parameters

- **doc** (gemmi.cif.Document) – The data to write out
- **out_stream** (*str*) – The name of the file to write the data to

6.4 General Utilities

These utilities are used by the pipeliner for basic tasks such as nice looking on-screen display, checking file names, and getting directory and file names

class pipeliner.utils.**DirectoryBasedLock**(*dirname: str | os.PathLike[str] = '.relion_lock'*, *timeout=60.0*)

Bases: object

A lock based on the creation and existence of a directory on the file system.

The interface is almost the same as Python's standard multiprocessing.Lock, except for some changes related to timeout behaviour:

- There is a default timeout of 60 seconds when acquiring the lock (rather than the default None value, with corresponding infinite timeout, that is used by multiprocessing.Lock). This is for compatibility with previous RELION locking timeout behaviour.
- A timeout for use when entering a context manager can be set when the lock object is created. Note that this value is ignored if the acquire() method is called directly. If there is a timeout waiting to acquire the lock when entering a context manager, a TimeoutError is raised.

The principle of this lock is that directory creation is an atomic operation provided by the file system, even in (most, modern) networked file systems. If several processes try to create the same directory at the same time, only one will succeed and the rest will get an error. Therefore, we can use this as a locking primitive, acquiring the lock if we successfully create the directory and releasing it by deleting the directory afterwards.

The lock directory name can be set if required. For compatibility with RELION, the default directory name is ".relion_lock".

acquire(*block=True*, *timeout=60.0*)

Acquire a lock, blocking or non-blocking.

With the `block` argument set to `True` (the default), the method call will block until the lock is in an unlocked state, then set it to locked and return `True`.

With the `block` argument set to `False`, the method call does not block. If the lock is currently in a locked state, return `False`; otherwise set the lock to a locked state and return `True`.

When invoked with a positive, floating-point value for timeout, block for at most the number of seconds specified by timeout as long as the lock can not be acquired. The default is 60.0 seconds; note that this is different from the default timeout in `multiprocessing.Lock.acquire()`.

Invocations with a negative value for timeout are equivalent to a timeout of zero. Invocations with a timeout value of `None` set the timeout period to infinite. The timeout argument has no practical implications if the block argument is set to `False` and is thus ignored.

Returns
`True` if the lock has been acquired or `False` if the timeout period has elapsed.

:raises Various possible errors from `os.mkdir()` including: `FileNotFoundError` or `PermissionError`.

release()

Release the lock.

This can be called from any thread, not only the thread which has acquired the lock.

When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

When invoked on an unlocked lock, a RuntimeError is raised.

There is no return value.

pipeliner.utils.**check_for_illegal_symbols**(*check_string: str*, *string_name: str = 'input'*, *exclude: str = ''*) → str | None

Check a text string doesn't have any of the disallowed symbols.

Illegal symbols are !*?()^/#<>&%{}$.”’ and @.

Parameters

- **check_string** (*str*) – The string to be checked
- **string_name** (*str*) – The name of the string being checked; for more informative error messages
- **exclude** (*str*) – Any symbols that are normally in the illegal symbols list but should be allowed.

Returns
An error message if any illegal symbols are present

Return type
str

pipeliner.utils.**clean_job_dirname**(*dirname: str*) → str

Makes sure a pipeline job_dir name is valid and in the right format

Parameters
dirname (*str*) – The dirname to check

Returns
The correctly formatted dirname

Return type
str

Raises
ValueError – If the dir name connot be formatted correctly

pipeliner.utils.**clean_jobname**(*jobname: str*) → str

Makes sure job names are in the correct format

Job names must have a trailing slash, cannot begin with a slash, and have no illegal characters

Parameters
jobname (*str*) – The job name to be checked

Returns
The job name, with corrections in necessary

Return type
str

pipeliner.utils.**compare_nested_lists**(*a_list: list*, *e_list: list*, *tolerance: float = 0.0*)

Compare two nested lists, allow or a tolerance for float values

Parameters

- **a_list** (*list*) – the actual list
- **e_list** (*list*) – the expected list
- **tolerance** (*float*) – The tolerance for float values, 0 if they must match exactly

Returns
do they match within tolerance?

Return type
bool

pipeliner.utils.**convert_relative_filename**(*filename: str*) → str

Convert a filename that is relative to the project to just its name

IE: ../../my_dir/my_file.txt -> my_dir/my_file /my_dir/my_file.txt -> my_dir/my_file ~/my_dir/my_file.txt -> my_dir/my_file

Parameters
filename –

Returns
The part of the file path that is not relative to the project

Return type
str

pipeliner.utils.**count_file_lines**(*filename: str*) → int

Fast and efficient count of number of lines in a file

Parameters
filename (*str*) – Name of the file to count the lines in

Returns
Number of lines

Return type
int

pipeliner.utils.**date_time_tag**(*compact: bool = False*) → str

Get a current date and time tag

It can return a compact version or one that is easier to read

Parameters
compact (*bool*) – Should the returned tag be in the compact form

Returns

The datetime tag

compact format is: *YYYYMMDDHHMMSS*

verbose form is: *YYYY-MM-DD HH:MM:SS.MS*

Return type
str

pipeliner.utils.**decompose_pipeline_filename**(*fn_in: str*) → Tuple[str, int, str]

Breaks a job name into usable pieces

Returns everything before the job number, the job number as an int and everything after the job number setup for up to 20 dirs deep. The 20 directory limit is from the relion code but no really necessary anymore

Parameters
fn_in (*str*) – The job or file name to be broken down in the format: <job-type>/jobxxx/<filename>

Returns

The decomposed file name: (str, int, str)
[0] Everything before 'job' in the file name

[1] The job number

[2] Everything after the job number

Return type
tuple

Raises
ValueError – If the input file name is more than 20 directories deep

pipeliner.utils.**file_in_project**(*filename: str*) → bool

Check that a file is part of the project

Not done with os.path.abspath(file).startswith(project_dir) because this causes errors during testing

pipeliner.utils.**find_common_string**(*input_strings: List[str]*) → str

Find the common part of a list of strings starting from the beginning

Parameters
input_strings (*list*) – List of strings to compare

Returns
The common portion of the strings

Return type
str

Raises
ValueError – If input_list is shorter than 2

pipeliner.utils.**format_string_to_type_objs**(*in_str: str*) → str | int | float | bool | None

Returns Int, Float, Bool, and None Objects from strings

Any number with a decimal point, in scientific notation, or 'NaN' will return a float Any other number will retun an int 'False' or 'false' returns False 'True' or 'true' returns True 'None' returns a NoneType object

Parameters
in_str (*str*) – The input string

Returns
The appropriate object

Return type
Optional[Union[str, int, float, bool]]

pipeliner.utils.**get_file_size_mb**(*file: str | Path*) → float

Get the size of a file in MB, rounded to 2 decimal places

Parameters
file (*str*) – The file to check

pipeliner.utils.**get_job_number**(*job_name*)

Get the job number from a pipeliner job as an int

Parameters
job_name (*str*) – The job name in the pipeliner format

Returns
The job number

Return type
int

pipeliner.utils.**get_job_runner_command**() → List[str]

Get the full command to run the job_runner.py script.

pipeliner.utils.**get_job_script**(*name: str*) → str

Get the full path to a job script file.

Returns
The job script file, if it exists.

Raises
FileNotFoundError – if the named job script cannot be found.

pipeliner.utils.**get_pipeliner_root**() → Path

Get the directory of the main pipeliner module

Returns
The path of the pipeliner

Return type
Path

pipeliner.utils.**get_python_command**() → List[str]

Get the command to launch the current Python interpreter.

Note that the command is returned as a list and might include some arguments as well as the command itself.

pipeliner.utils.**get_regenerate_results_command**() → List[str]

Get the full command to run the regenerate_results.py script.

pipeliner.utils.**increment_file_basenames**(*files: List[str]*) → Dict[str, str]

Increment the base names of files if there are duplicates

e.g. Import/job001/myfile, Import/job002/myfile

Parameters
files (*List[str]*) – The files to operate on

Returns
The file name and its incremented basename

Return type
Dict[str, str]

pipeliner.utils.**is_uuid4**(*in_str: str*) → bool

Check that a string is a UID4

Parameters
in_str (*str*) – The string to test

Returns
Is the string a valid uid4

Return type
bool

pipeliner.utils.**launch_detached_process**(*command: List[str]*) → None

Run the given command as a detached process.

The process is started in a new session and with all file handles set to null, to ensure it keeps running in the background after the parent Python process exits.

Parameters
command (*List[str]*) – The commands to execute

pipeliner.utils.**make_pretty_header**(*text: str*, *char: str = '-='*, *top: bool = True*, *bottom: bool = True*)

Make nice looking headers for on-screen display

Parameters

- **text** (*str*) – The text to put in the header
- **char** (*str*) – What characters to use for the header

Returns
A nice looking header

Return type
str

pipeliner.utils.**print_nice_columns**(*list_in: List[str]*, *err_msg: str = 'ERROR: No items in input list'*)

Takes a list of items and makes three columns for nicer on-screen display

Parameters

- **list_in** (*str*) – The list to display in columns

- **err_msg** (*str*) – The message to display if the list is empty

pipeliner.utils.**run_subprocess**(**args*, ***kwargs*) → CompletedProcess

pipeliner.utils.**smart_strip_quotes**(*in_string: str*) → str

Strip the quotes from a string in an intelligent manner

Remove leading and ending ‘ and “ but don’t remove them internally

Parameters
in_string (*str*) – The input string

Returns
the string with leading and ending quotes removed

Return type
str

pipeliner.utils.**str_is_hex_colour**(*in_string*, *allow_0x: bool = False*) → bool

Test that a string is a hexadecimal colour code

Valid codes consist of a # symbol or ‘0x’ followed by exactly six hexadecimal digits (0-9 or a-f, lower or upper case).

Parameters

- **in_string** (*str*) – The string to test
- **allow_0x** (*bool*) – Also allow ‘0x’ style codes

Returns
is it a valid colour code?

Return type
bool

pipeliner.utils.**subprocess_popen**(**args*, ***kwargs*) → Popen

pipeliner.utils.**touch**(*filename: str*)

Create an empty file

Parameters
filename (*str*) – The name for the file to create

pipeliner.utils.**truncate_number**(*number: float*, *maxlength: int*) → str

Return a number with no more than x decimal places but no trailing 0s

This is used to format numbers in the exact same way that RELION does it. IE: with maxlength 3; 1.2000 = 1.2, 1.0 = 1, 1.23 = 1.23. RELION commands are happy to accept numbers with any number of decimal places or trailing 0s. This function is just to maintain continuity between RELION and pipeliner commands

Parameters

- **number** (*float*) – The number to be truncated
- **maxlength** (*int*) – The maximum number of decimal places

pipeliner.utils.**wrap_text**(*text_string: str*)

Produces <= 55 character wide wrapped text for on-screen display

Parameters
text_string (*str*) – The text to be displayed

CHAPTER

SEVEN

PIPELINER JOBS AND PLUGINS

7.1 Pipeliner Jobs

7.1.1 Pipeliner jobs

class pipeliner.pipeliner_job.**ExternalProgram**(*command: str*, *name: str | None = None*, *vers_com: List[str] | None = None*, *vers_lines: List[int] | None = None*)

Bases: object

Class to store info about external programs called by the pipeliner

command

The command that will be used to run the program

Type
str

name

The name for the program, command will be used unless this is specified

Type
str

exe_path

The path to the executable for the program

Type
str

vers_com

The command that needs to be run to get the version

Type
List[str]

vers_lines

The lines from the output of the version command that contain the version info

Type
List[int]

get_version() → str | None

class pipeliner.pipeliner_job.**JobInfo**(*display_name: str = 'Pipeliner job'*, *version: str = '0.0'*, *job_author: str | None = None*, *short_desc: str = 'No short description for this job'*, *long_desc: str = 'No long description for this job'*, *documentation: str = 'No online documentation available'*, *external_programs: List[ExternalProgram] | None = None*, *references: List[Ref] | None = None*)

Bases: `object`

Class for storing info about jobs.

This is used to generate documentation for the job within the pipeliner

display_name

A user-friendly name to describe the job in a GUI, this should not include the software used, because that info is pulled from the job type

Type
str

version

The version number of the pipeliner job

Type
str

job_author

Who wrote the pipeliner job

Type
str

short_desc

A one line "title" for the job

Type
str

long_desc

A detained description about what the job does

Type
str

documentation

A URL for online documentation

Type
str

programs

A list of 3rd party software used by the job. These are used by the pipeliner to determine if the job can be run, so they need to include all executables the job might call. If any program on this list cannot be found with *which* then the job will be marked as unable to run.

Type
List[*~pipeliner.pipeliner_job.ExternalProgram*]

references

A list of *Ref* objects used

Type
list

force_unavailable

This can be set to True if other checks for the job to be available (besides programs missing from the $PATH) have failed, e.g. a necessary library is missing

Type
bool

property is_available

Is the job available to run?

`True` if executables were found for all the job's programs or if `force_unavailable` has been set, or `False` otherwise.

class pipeliner.pipeliner_job.**PipelinerCommand**(*args: Sequence[str | float | int]*, *relion_control: bool = False*)

Bases: `object`

Holds a command that will be run by the pipeliner

com

The command that will be run each list item is one arg

Type
List[str]

relion_control

Does the command need the relion '–pipeline_control' argument appended before being run

Type
bool

add_pipeline_control(*outputdir: str*) → None

class pipeliner.pipeliner_job.**PipelinerJob**

Bases: `object`

Super-class for job objects.

Each job type has its own sub-class.

WARNING: do not instantiate this class directly, use the factory functions in this module.

jobinfo

Contains information about the job such as references

Type
JobInfo

output_dir

The path of the output directory created by this job

Type
str

alias

the alias for the job if one has been assigned

Type
str

is_continue
If this job is a continuation of an older job or a new one

Type
bool

input_nodes
A list of *Node* objects for each file used as in input

Type
list

output_nodes
A list of *Node* objects for files produced by the job

Type
list

joboptions
A dict of *JobOption* objects specifying the parameters for the job

Type
dict

is_tomo
Is the job a tomography job?

Type
bool

working_dir
The working directory to be used when running the job. This should normally be left as `None`, meaning the job will be run in the project directory. Jobs that write files in their working directory should instead work somewhere within the job's output directory, and take care to adjust the paths of input and output files accordingly.

Type
str

raw_options
A dict of all raw joboptions as they were read in

Type
dict

CATEGORY_LABEL = ''

OUT_DIR = ''

PROCESS_NAME = ''

add_compatibility_joboptions() → None
Write additional joboptions for back compatibility

Some JobOptions are needed by the original program (hey Relion 4), but not the pipeliner, they are added here so the files pipeliner writes will be back compatible with their original program.

add_output_node(*file_name: str*, *node_type: str*, *keywords: List[str] | None = None*) → None
Helper function to add a new Node for a file in the job's output directory.

This is a wrapper around `node_factory.create_node` which simply adds `self.output_dir` to the start of the file name before creating the node and adding it to `self.output_nodes`.

Parameters

- **file_name** – The name of the file that the new node will refer to. It is assumed that the file will be written to the job'a output directory. Note that the existence of the file is not checked, because this method will usually be called before the job has run.
- **node_type** – The top-level type for the new node. This should almost always be one of the constants defined in `pipeliner.nodes`.
- **keywords** – A list of keywords to append to the node type.

additional_joboption_validation() → List[*JobOptionValidationResult*]

Advanced validation of job parameters

This is a placeholder function for additional validation to be done by individual job subtypes, such as comparing JobOption values IE: JobOption A must be > JobOption B

Avoid using self.get_string or self.get_number in this function as they may raise an error if the JobOption is required and has no value. Use self.joboptions["jobopname"].value.

Returns

A list *JobOptionValidationResult* objects

Return type
list

check_joboption_is_now_deactivated(*jo: str*) → bool

Check if a joboption has become deactivated in relation to others

For example if job option A is False, job option B is now deactiavted

Parameters
jo (*str*) – The name of the JobOption to test

Returns
Has the JobOption been deactivated

Return type
bool

check_joboption_is_now_required(*jo: str*) → list

Check if a joboption has become required in relation to others

For example if job option A is True, job option B is now required

Parameters
jo (*str*) – The name of the joboption to test

Returns

pipeliner.job_options.JobOptionValidationResult:
for any errors found

Return type
list

create_input_nodes() → None

Automatically add the job's input nodes to its input node list.

Input nodes are created from each of the job's job options.

create_output_nodes() → None

Make the job's output nodes.

This method should be overridden by PipelinerJob subclasses.

The output nodes should be added to the list in the `output_nodes` attribute. The `add_output_node` function is helpful to create and add a new node in a single call.

If your job doesn't make any output nodes, or doesn't know what their names will be until the job has been run, you still need to override this method but your implementation can simply `pass` and do nothing. If you need to add output nodes at the end of the job, create them in `create_post_run_output_nodes`.

Note that this method is called by the job manager (via `PipelinerJob.prepare_to_run`) before the job is added to the pipeline. The job's output directory does exist when this method is called, but that could change in future versions of the pipeliner and jobs should avoid making any file system changes in this method.

create_post_run_output_nodes()

Placeholder function for post run node creation

Some jobs have output nodes that can only be created after the job has run because their names are not known until after they have been created. They can be added here. This function should ONLY add output nodes; any other work should be done in commands run by the job.

create_results_display() → Sequence[*ResultsDisplayObject*]

Create results display objects to be displayed by the GUI

This default implementation simply creates the default results display object for each of the job's output nodes. Subclasses that want customised results should override this method.

Returns
A list of *ResultsDisplayObject*

gather_metadata() → Dict[str, Any]

Placeholder function for metadata gathering

Each job class should define this individually

Returns
A placeholder "No metadata available" and the reason why

Return type
dict

get_additional_reference_info() → List[*Ref*]

A placeholder function for job that need to return additional references

This if for references that are not included in self.job info, such as ones pulled from the EMDB/PDB in fetch jobs

get_category_label() → str

Get a label for the category that this job belongs to.

If the job defines a `CATEGORY_LABEL` attribute, its value is simply returned. Otherwise, the second part of the process name is processed to produce a label by replacing underscores with spaces and converting to title case.

get_commands() → List[*PipelinerCommand*]

Get the commands to be run for a specific job.

This method should be overridden by PipelinerJob subclasses.

Jobs are normally run with the project directory as the working directory. If your job needs to run in a different working directory (for example if it calls a program which always writes files into the current directory), set the `self.working_dir` attribute in this method.

Note that this method should run quickly! Any long-running actions should be done in one of the job's commands instead. (If necessary, put Python code that needs to be run into a separate script in `pipeliner.scripts.job_scripts` and then call it as a command.)

Returns
The commands as a list of PipelinerCommand objects

`get_current_output_nodes()` → List[*Node*]

Get the current output nodes if the job was stopped prematurely

For most jobs there will not be any but for jobs with many iterations the most recent interation can be used if the job is aborted or failed and then later marked as successful

Returns
of *Node* objects

Return type
list

`get_default_params_dict()` → Dict[str, str]

Get a dict with the job's parameters and default values

Returns
All the job's parameters {parameter: default value}

Return type
dict

`get_extra_options()` → None

Get user specified extra queue submission options

`get_final_commands()` → List[List[str]]

Assemble the commands to be run for a job.

This function is intended to be called by the job runner just before the commands are run. Any setup required before the job starts should be done in `prepare_to_run()`.

Returns
The commands, in a lists of lists format. Each item in the main list is a single command composed of a list of strings (as used by `subprocess.run`, i.e. [com, arg1, arg2, ...])

`get_joboption_groups()` → Dict[str, List[str]]

Put the joboptions in groups according to their jobop_group attribute

Assumes that the joboptions have already been put in order of priority by self.set_joboption_order() or were in order to begin with.

Groups are ordered based on the highest priority joboption in that group from the order of the joboptions, except that "Main" is always the first group. Joboptions within the groups are ordered by priority.

Returns
The joboptions groupings {group: [jopbop, ... jobop]}

Return type
Dict[str, List[str]]

`get_mpi_command()` → List[int | float | str]

get_nr_mpi() → int

get_nr_threads() → int

get_runtab_options(*mpi: bool = False*, *threads: bool = False*, *addtl_args: bool = False*, *mpi_default_min: int = 1*, *mpi_must_be_odd: bool = False*) → None

Get the options found in the Run tab of the GUI, which are common to for all jobtypes

Adds entries to the joboptions dict for queueing, MPI, threading, and additional arguments. This method should be used when initialising a *PipelinerJob* subclass

Parameters

- **mpi** (*bool*) – Should MPI options be included?
- **threads** (*bool*) – Should multi-threading options be included
- **addtl_args** (*bool*) – Should and 'additional arguments' be added
- **mpi_default_min** (*int*) – The minimum for the default number of MPIs, will be used if mpi_default_min > user defined min number of MPI
- **mpi_must_be_odd** (*bool*) – Does the number of mpis have to be odd, like for relion refine_jobs.

handle_doppio_uploads(*dry_run=False*) → None

Tasks that have to be performed to deal with Doppio file uploads.

- **Move files from DoppioUploads to the job dir:**
  DoppioUploads/tmpdir/file -> JobType/jobNNN/InputFiles/file
- Update the job option values to point to the new file locations, so when the job input nodes are created they refer to the moved files

Parameters

dry_run – If True, do not actually try to move any files, just update the job option values. This option is only intended for use in testing.

is_submit_to_queue() → bool

load_results_display_files() → Sequence[*ResultsDisplayObject*]

Load the job's results display objects from files on disk.

This method must be fast because it is used by the GUI to load job results. Therefore, if a display object fails to load properly, no attempt is made to recalculate it and a `ResultsDisplayPending` object is returned instead.

If there are no results display files yet, an empty list is returned.

Returns

A list of *ResultsDisplayObject*

make_additional_args() → None

Get the additional arguments job option

make_queue_options() → None

Get options related to queueing and queue submission, which are common to for all jobtypes

parse_additional_args() → List[str]

Parse the additional arguments job option and return a list

Returns

A list ready to append to the command. Quotated strings are preserved
as quoted strings all others are split into individual items

Return type
list

prepare_clean_up_lists(*do_harsh: bool = False*) → Tuple[List[str], List[str]]

Placeholder function for preparation of list of files to clean up

Each job class should define this individually

Parameters
do_harsh (*bool*) – Should a harsh cleanup be performed

Returns
Two empty lists `([files, to, delete], [dirs, to, delete])`

Return type
tuple

prepare_deposition_data(*depo_type: str*) → Sequence[*EmpiarRefinedParticles* | *EmpiarParticles* | *EmpiarCorrectedMics* | *EmpiarMovieSet* | *OneDepData*]

Placeholder for function to return deposition data objects

The specific list returned should be defined by each jobtype

Parameters
depo_type (*str*) – EMPIAR or ONEDEP

Returns

The deposition object(s) returned by the specific job. These
need to be of the types defined in *pipeliner.deposition_tools.onedep_deposition* and *pipeliner.deposition_tools.empiar_deposition*

Return type
list

prepare_to_run(*ignore_invalid_joboptions=False*) → None

Prepare the job to run.

This function is intended to be called by the pipeliner before the job file is saved to disk. It does several things including: - Validate the job options - Make the job directory - Move uploaded Doppio user files into the job directory

Parameters
ignore_invalid_joboptions (*bool*) – Prepare the job to run anyway even if the job options appear to be invalid

Raises

- **ValueError** – If the job options appear to be invalid and `ignore_invalid_joboptions` is not set
- **RuntimeError** – If the job does not already have an output directory assigned

read(*filename: str*) → None

Reads parameters from a run.job or job.star file

Parameters
filename (*str*) – The file to read. Can be a run.job or job.star file

Raises
ValueError – If the file is a job.star file and job option from the *PipelinerJob* is missing from the input file

save_job_submission_script(*commands: list*) → str

Writes a submission script for jobs submitted to a queue

Parameters
commands (*list*) – The job's commands. In a list of lists format

Returns
The name of the submission script that was written

Return type
str

Raises

- **ValueError** – If no submission script template was specified in the job's joboptions
- **ValueError** – If the submission script template is not found
- **RuntimeError** – If the output script could not be written

save_results_display_files() → Sequence[*ResultsDisplayObject*]

Create new results display objects and save them to disk.

This method removes any existing results display files first, and returns the new display objects after they have been created and saved.

Returns
The newly-created results display objects.

set_joboption_order(*new_order=typing.List[str]*) → None

Replace the joboptions dict with an ordered dict

Use this to set the order the joboptions will appear in the GUI. If a joboption is not specified in the list it will be tagged on to the end of the list.

Parameters
new_order (*list[str]*) – A list of joboption keys, in the order they should appear

Raises
ValueError – If a nonexistent joboption is specified

set_option(*line: str*) → None

Sets a value in the joboptions dict from a run.job file

Parameters
line (*str*) – A line from a run.job file

Raises

- **RuntimeError** – If the line does not contain '=='
- **RuntimeError** – If the value of the line does not match any of the joboptions keys

validate_dynamically_required_joboptions() → List[*JobOptionValidationResult*]

Check all joboptions if they have become required because of if_required

For example if job option A is True, job option B is now required

Returns

pipeliner.job_options.JobOptionValidationResult:
for any errors found

Return type
list

validate_input_files() → List[*JobOptionValidationResult*]

Check that files specified as inputs actually exist

Returns

A list of *pipeliner.job_options.JobOptionValidationResult*
objects

Return type
list

validate_joboptions() → List[*JobOptionValidationResult*]

Make sure all the joboptions meet their validation criteria

Returns

A list *JobOptionValidationResult*
objects

Return type
list

write_jobstar(*output_dir: str*, *output_fn: str = 'job.star'*, *is_continue: bool = False*)

Write a job.star file.

Parameters

- **output_dir** (*str*) – The output directory.
- **output_fn** (*str*) – The name of the file to write. Defaults to job.star
- **is_continue** (*bool*) – Is the file for a continuation of a previously run job? If so only the parameters that can be changed on continuation are written. Overrules is_continue attribute of the job

write_runjob(*fn: str | None = None*) → None

Writes a run.job file

Parameters
fn (*str*) – The name of the file to write. Defaults to the file the pipeliner uses for storing GUI parameters. A directory can also be entered and it will add on the file name 'run.job'

class pipeliner.pipeliner_job.**Ref**(*authors: str | List[str] | None = None*, *title: str = ''*, *journal: str = ''*, *year: str = ''*, *volume: str = ''*, *issue: str = ''*, *pages: str = ''*, *doi: str = ''*, ***kwargs*)

Bases: object

Class to hold metadata about a citation or reference, typically a journal article.

authors

The authors of the reference.

Type
list

title
The reference's title.

Type
str

journal
The journal.

Type
str

year
The year of publication.

Type
str

volume
The volume number.

Type
str

issue
The issue number.

Type
str

pages
The page numbers.

Type
str

doi
The reference's Digital Object Identifier.

Type
str

other_metadata
Other metadata as needed. Gathered from kwargs

Type
dict

7.1.2 Display tools

Use these methods to create `ResultsDisplayObjects` used by the pipeliner GUI Doppio to create graphical outputs for each job.

`pipeliner.display_tools.`**`create_results_display_object`**(*dobj_type: str*, ***kwargs*) → *ResultsDisplayObject*

Safely create a results display object

Returns a ResultsDisplayPending if there are any problems. Give it the type of display object as the first argument followed by the kwargs for that specific type of ResultsDisplayObject

Parameters
dobj_type (*str*) – The type of DisplayObject to create

pipeliner.display_tools.**get_ordered_classes_arrays**(*model_file: str*, *ncols: int*, *boxsize: int*, *output_dir: str*, *output_filename: str*, *parts_file: str | None = None*, *title: str = '2D class averages'*, *start_collapsed: bool = False*, *flag: str = ''*) → *ResultsDisplayMontage | ResultsDisplayPending*

Return a 3D array of class averages from a Relion Class2D model file

Parameters

- **model_file** (*str*) – Name of the model file
- **ncols** (*int*) – number of columns desired in the file montage
- **boxsize** (*int*) – Size of the class averages in the final montage
- **output_dir** (*str*) – The output dir of the pipeliner job creating this object
- **output_filename** (*str*) – The name for the output montage file
- **parts_file** (*str*) – Path of the file containing the particles, for counting
- **title** (*str*) – A title for the DisplayObject
- **start_collapsed** (*bool*) – Should the display start out collapsed when displayed in the GUI
- **flag** (*str*) – If this display object contains scientificlly dubious results display this message

Returns
An object for the GUI to use to render the graph

Return type
ResultsDisplayMontage

pipeliner.display_tools.**graph_from_starfile_cols**(*title: str*, *starfile: str*, *block: str*, *ycols: list*, *xcols: list | None = None*, *xrange: list | None = None*, *yrange: list | None = None*, *data_series_labels: List[str] | None = None*, *xlabel: str = ''*, *ylabel: str = ''*, *assoc_data: List[str] | None = None*, *modes: List[str] | None = None*, *start_collapsed: bool = False*, *flag: str = ''*) → *ResultsDisplayGraph | ResultsDisplayPending*

Automatically generate a ResultsDisplayGraph object from a starfile

Can use one or two columns and third column for labels if desired

Parameters

- **title** (*str*) – The title of the final graph
- **starfile** (*str*) – Path to the star file ot use
- **block** (*str*) – The block to use in the starfile, use *None* for a starfile with only a single block
- **ycols** (*list*) – Column label(s) from the star file to use for the y data series
- **xcols** (*list*) – Column label(s) from the star file to use for the y data series if *None* a simple count from 1 will be used

- **xlabel** (*str*) – Label for the x axis, if no x data are specified the label will 'Count', if x data are specified and the xlabel is *None* the x axis label will be the name of the starfile column used
- **xrange** (*list*) – Range for x vlaues to be displayed, full range if *None*
- **yrange** (*list*) – Range for y vlaues to be displayed, full range if *None*
- **data_series_labels** (*list*) – Names for the data series
- **ylabel** (*str*) – Label for the y axis, if *None* the y axis label will be the name of the starfile column used
- **assoc_data** (*list*) – List of data file(s) associated with this graph
- **modes** (*list*) – Controls the appearance of each data series, choose from 'lines', 'markers' 'or lines+markers'
- **start_collapsed** (*bool*) – Should the display start out collapsed when displayed in the GUI
- **flag** (*str*) – If this display object contains scientificlly dubious results display this message

Returns
A ResultsDisplayGraph object for the created graph

Return type
ResultsDisplayGraph

pipeliner.display_tools.**histogram_from_starfile_col**(*title: str*, *starfile: str*, *block: str*, *data_col: str*, *xlabel: str = ''*, *ylabel: str = 'Count'*, *assoc_data: List[str] | None = None*, *start_collapsed: bool = False*, *flag: str = ''*) → *ResultsDisplayHistogram* | *ResultsDisplayPending*

Automatically generate a ResultsDisplayHistogram object from a starfile

Parameters

- **title** (*str*) – The title of the final graph
- **starfile** (*str*) – Path to the star file ot use
- **block** (*str*) – The block to use in the starfile, use *None* for a starfile with only a single block
- **data_col** (*str*) – Column label from the star file to use for the data series
- **xlabel** (*str*) – Label for the x axis, if no x data are specified the label will 'Count', if x data are specified and the xlabel is *None* the x axis label will be the name of the starfile column used
- **ylabel** (*str*) – Label for the y axis, if *None* the y axis label will be the name of the starfile column used
- **assoc_data** (*list*) – List of data file(s) associated with this graph
- **start_collapsed** (*bool*) – Should the display start out collapsed when displayed in the GUI
- **flag** (*str*) – If this display object contains scientificlly dubious results display this message

pipeliner.display_tools.**make_map_model_thumb_and_display**(*outputdir: str*, *maps: List[str] | None = None*, *maps_opacity: List[float] | None = None*, *maps_colours: List[str] | None = None*, *models: List[str] | None = None*, *models_colours: List[str] | None = None*, *title: str | None = None*, *maps_data: str = ''*, *models_data: str = ''*, *assoc_data: List | None = None*, *flag: str = ''*) → *ResultsDisplayMapModel* | *ResultsDisplayPending*

Make a display object for an atomic model overlaid over a map

Makes a binned map and a ResultsDisplayMapModel display object

Parameters

- **outputdir** (*str*) – Name of the job's output directory
- **maps** (*list*) – List of map files to use
- **models** (*list*) – List of model files to use
- **maps_opacity** (*list*) – List of opacity for the maps, from 0-1 if *None* 0.5 is used for all
- **maps_colours** (*list*) – Colors for the maps of specific ones are desired, otherwise mol* will assign them
- **title** (*str*) – The title for the ResultsDisplayMapModel object, if *None* the name of the map and model will be used
- **maps_data** (*str*) – Any additional data to be included about the map
- **models_data** (*str*) – Any additional data to be included about the map
- **models_colours** (*list*) – Colors for the models of specific ones are desired, otherwise mol* will assign them
- **assoc_data** (*list*) – List of associated data, if left as *None* then just uses the file itself
- **flag** (*str*) – If the results are considered scientifically dubious explain in this string

Returns
The DisplayObject for the map and model

Return type
ResultsDisplayMapModel

pipeliner.display_tools.**make_maps_slice_montage_and_3d_display**(*in_maps: Dict[str, str]*, *output_dir: str*, *combine_montages: bool = True*, *cmap: str = ''*) → List[*ResultsDisplayObject*]

Make a set of display objects for 3D maps

Returns separate 3D viewer display objects for each map and either a combined slices montage or a slices montage for each.

Parameters

- **in_maps** (*dict*) – {input file: label}. If the label is "", the filename will be used
- **output_dir** (*str*) – The job's output dir where the thumbnails dir will be created if necessary

- **combine_montages** (*bool*) – Should a single montage be made with slices for all maps, otherwise a separate montage is made for each
- **cmap** (*str*) – what color map to use for the montage, if any.

Returns

The display objects montage and then the 3D viewers if combine_montages is `False`, otherwise the montage followed by the 3D viewer for each map in the order they were given.

Return type
List

pipeliner.display_tools.**make_mrcs_central_slices_montage**(*in_files: Dict[str, str]*, *output_dir: str*, *cmap: str = ''*) → *ResultsDisplayMontage | ResultsDisplayPending*

Make a montage of x,y,z central slices of maps

Parameters

- **in_files** (*Dict[str, str]*) – {file name: label if different from file name}
- **output_dir** (*str*) – Where to make the Thumbnails dir (if necessary) and put the montage image
- **cmap** (*str*) – What colormap to use, if any
- **Returns** –
- **Union[ResultsDisplayMontage** – The montage ResultsDisplayObject or a ResultsDisplayPending if there was an error
- **ResultsDisplayPending]** – The montage ResultsDisplayObject or a ResultsDisplayPending if there was an error

pipeliner.display_tools.**make_particle_coords_thumb**(*in_mrc*, *in_coords*, *out_dir*, *thumb_size=640*, *pad=5*, *start_collapsed=False*, *title: str = 'Example picked particles'*, *flag: str = ''*, *markers: bool = False*) → *ResultsDisplayImage | ResultsDisplayPending*

Create a thumbnail of picked particle coords on their micrograph

Because the extraction box size is not known boxes will be a % of the total image size.

Parameters

- **in_mrc** (*str*) – Path to the merged micrograph mrc file
- **in_coords** (*str*) – Path to the .star coordinates file
- **out_dir** (*str*) – Name of the output directory
- **thumb_size** (*int*) – Size of the x dimension of the final thumbnail image
- **pad** (*int*) – Thickness of the particle box borders before binning in px
- **start_collapsed** (*bool*) – Should the display start out collapsed when displayed in the GUI
- **title** (*str*) – What title to use for the displayobj created
- **flag** (*str*) – If this display object contains scientificlly dubious results display this message
- **markers** (*bool*) – Instead of making boxes make markers

pipeliner.display_tools.**mini_montage_from_many_files**(*filelist: List[str]*, *outputdir: str*, *nimg: int = 5*, *montagesize: int = 640*, *title: str = ''*, *ncols: int = 5*, *associated_data: List[str] | None = None*, *labels: List[str] | None = None*, *cmap: str = ''*, *start_collapsed: bool = False*, *flag: str = ''*) → *ResultsDisplayMontage* | *ResultsDisplayPending*

Make a mini montage from a list of images

Merge and flatten image stacks

Parameters

- **filelist** (*list*) – A list of the files to use
- **outputdir** (*str*) – The output dir of the pipeliner job
- **nimg** (*int*) – Number of images to use in the montage
- **montagesize** (*int*) – Desired size of the final montage image
- **title** (*str*) – Title for the ResultsDisplay object that will be output
- **ncols** (*int*) – Number of columns to make in the montage
- **associated_data** (*list*) – Data files associated with these images, if *None* then all of the selected images
- **labels** (*list*) – The labels for the items in the montage
- **cmap** (*str*) – colormap to apply, if any
- **start_collapsed** (*bool*) – Should the display start out collapsed when displayed in the GUI
- **flag** (*str*) – If this display object contains scientificlly dubious results display this message

Returns
The DisplayObject for the map

Return type
ResultsDisplayImage

Raises
ValueError – If a non mrc or tiff image is used

pipeliner.display_tools.**mini_montage_from_stack**(*stack_file: str*, *outputdir: str*, *nimg: int = 40*, *ncols: int = 10*, *montagesize: int = 640*, *title: str = ''*, *labels: List[int | str] | None = None*, *cmap: str = ''*, *start_collapsed: bool = False*, *flag: str = ''*) → *ResultsDisplayMontage* | *ResultsDisplayPending*

Make a montage from a mrcs or tiff file

Parameters

- **stack_file** (*str*) – The path to the stack_file
- **outputdir** (*str*) – The output dir of the pipeliner job
- **nimg** (*int*) – Number of images to use in the montage, if < 1 uses all of them
- **ncols** (*int*) – Number of columns to use
- **montagesize** (*int*) – Desired size of the final montage image

- **title** (*str*) – Title for the ResultsDisplay object that will be output
- **labels** (*list*) – Labels for the images
- **cmap** (*str*) – colormap to apply, if any
- **start_collapsed** (*bool*) – Should the display start out collapsed when displayed in the GUI
- **flag** (*str*) – If this display object contains scientificlly dubious results display this message

Returns
The DisplayObject for the map

Return type
ResultsDisplayImage

Raises
ValueError – If a non mrc or tiff image is used

pipeliner.display_tools.**mini_montage_from_starfile**(*starfile: str*, *block: str*, *column: str*, *outputdir: str*, *title: str = ''*, *nimg: int = 20*, *montagesize: int = 640*, *ncols: int = 10*, *labels: List[str] | None = None*, *cmap: str = ''*, *start_collapsed: bool = False*, *flag: str = ''*) → *ResultsDisplayMontage* | *ResultsDisplayPending*

Make a montage from a list of images in a starfile column

Merge and flatten image stacks if they are encountered.

Parameters

- **starfile** (*str*) – The starfile to use
- **block** (*str*) – The name of the block with the images
- **column** (*str*) – The name of the column that has the images
- **outputdir** (*str*) – The output dir of the pipeliner job
- **title** (*str*) – The title for the object, automatically generated if “”
- **nimg** (*int*) – Number of images to use in the montage, uses all if < 1
- **montagesize** (*int*) – Desired size of the final montage image
- **ncols** (*int*) – number of columns to use
- **labels** (*list*) – Labels for the images in the montage, in order
- **cmap** (*str*) – colormap to apply, if any
- **start_collapsed** (*bool*) – Should the display start out collapsed when displayed in the GUI
- **flag** (*str*) – If this display object contains scientificlly dubious results display this message

Returns
The DisplayObject for the map

Return type
ResultsDisplayImage

Raises
ValueError – If a non mrc or tiff image is encountered

7.1.3 ResultsDisplay Objects

These objects generally should not be instantiated directly they should instead be created using the functions above.

class pipeliner.results_display_objects.**ResultsDisplayGraph**(*, *xvalues: list*, *yvalues: list*, *title: str*, *associated_data: list*, *data_series_labels: list*, *xaxis_label: str = ''*, *xrange: list | None = None*, *yaxis_label: str = ''*, *yrange: list | None = None*, *modes: list | None = None*, *start_collapsed: bool = False*, *flag: str = ''*)

Bases: *ResultsDisplayObject*

A simple graph for the GUI to display

It is best to not instantiate this class directly. Instead create it using *create_results_display_object*

title

The title of the object/graph

Type
str

xvalues

(list): list of x coordinate data series, can have multiple data series

xaxis_label

Label for the x axis if a graph

Type
str

xrange

Range of x to be displayed, displays the full range if *None*. If the x axis needs to be reveresd then enter the values backwards [max, min]

Type
list

yvalues

(list): List y coordinate data series can have multiple data series

yaxis_label

Label for the y axis if a graph

Type
str

yrange

Range of y to be displayed, displays the full range if *None*. If the y axis needs to be reveresd then enter the values backwards [max, min]

Type
list

data_series_labels

List of names of the different data series

Type
list

associated_data

A list of files that contributed the data used in the image/graph

Type
list

modes

Controls the appearance of each data series, choose from 'lines', 'markers' 'or lines+markers'

Type
list

start_collapsed

Should the object start out collapsed when displayed in the GUI

Type
bool

class pipeliner.results_display_objects.**ResultsDisplayHistogram**(*, *title: str*, *associated_data: list*, *data_to_bin: List[float] | None = None*, *xlabel: str = ''*, *ylabel: str = ''*, *bins: List[int] | None = None*, *bin_edges: List[float] | None = None*, *start_collapsed: bool = False*, *flag: str = ''*)

Bases: *ResultsDisplayObject*

A class for the GUI to display a histogram

It is best to not instantiate this class directly. Instead, create it using *create_results_display_object*

Parameters

- **title** (*str*) – The title of the histogram
- **data_to_bin** (*list*) – The data to bin
- **xlabel** (*str*) – Label for the x axis
- **ylabel** (*str*) – Label for the y axis
- **associated_data** (*list*) – List of data files associated with the histogram
- **bins** (*list*) – A list of bin counts, if they are known
- **bin_edges** (*list*) – A list of the bin edges, if they are already known
- **start_collapsed** (*bool*) – Should the object start out collapsed when displayed in the GUI

Raises

- **ValueError** – If no data or bins are specified
- **ValueError** – If an attempt is made to specify bins or bin edges when data to bin are being provided
- **ValueError** – If the associated data is not a list, or not provided

class pipeliner.results_display_objects.**ResultsDisplayHtml**(*, *title: str*, *associated_data: list*, *html_dir: str = ''*, *html_file: str = ''*, *html_str: str = ''*, *start_collapsed: bool = False*, *flag: str = ''*)

Bases: *ResultsDisplayObject*

An object for the GUI to display html

It is best to not instantiate this class directly. Instead create it using *create_results_display_object*

This can be used for general HTML display in Doppio. Either provide a directory with index.html or specify a html file or provide a html string as input.

html_dir

Path to the html directory (optional)

Type
str

html_file

Path to a standalone html file or in the given html_dir (optional)

Type
str

html_str

Input html as string (optional)

Type
str

start_collapsed

Should the object start out collapsed when displayed in the GUI

Type
bool

class pipeliner.results_display_objects.**ResultsDisplayImage**(*, *title: str*, *image_path: str*, *image_desc: str*, *associated_data: list*, *start_collapsed: bool = False*, *flag: str = ''*)

Bases: *ResultsDisplayObject*

A class for the GUI to display a single image

It is best to not instantiate this class directly. Instead create it using *create_results_display_object*

title

The title for the image

Type
str

image_path

The path to the image

Type
str

image_desc

A description of the image

Type
str

associated_data

Data files associated with the image

Type
list

start_collapsed

Should the object start out collapsed when displayed in the GUI

Type
bool

class pipeliner.results_display_objects.**ResultsDisplayJson**(*, *file_path: str*, *title: str = ''*, *start_collapsed: bool = False*, *flag: str = ''*)

Bases: *ResultsDisplayObject*

An object for the GUI to display JSON files

It is best to not instantiate this class directly. Instead create it using *create_results_display_object*

file_path

Path to the file

Type
str

start_collapsed

Should the object start out collapsed when displayed in the GUI

Type
bool

class pipeliner.results_display_objects.**ResultsDisplayMapModel**(*title: str*, *associated_data: list*, *maps: list | None = None*, *models: list | None = None*, *maps_data: str = ''*, *models_data: str = ''*, *maps_opacity: list | None = None*, *maps_colours: list | None = None*, *models_colours: list | None = None*, *start_collapsed: bool = True*, *flag: str = ''*)

Bases: *ResultsDisplayObject*

An object for overlaying multiple maps and/or models

It is best to not instantiate this class directly. Instead create it using *create_results_display_object*

title

The title that appears at the top of the accordian in the GUI

Type
str

`associated_data`
A list of associated data files

Type
list

`maps`
List of map paths, mrc format

Type
list

`models`
List of model paths, pdb or mmcif format

Type
list

`maps_opacity`
Opacity for each map from 0-1 if not specified set at 0.5 for all maps

Type
list

`models_data`
Any extra info about the models

Type
str

`maps_data`
Any extra info about the maps

Type
str

`maps_colours`
Hex values for colouring the maps specific colours, in the form "#XXXXXX" where X is a hex digit (0-9 or a-f). If None, the standard colours will be used

Type
list

`models_colours`
Hex values for colouring the models specific colours, in the form "#XXXXXX" where X is a hex digit (0-9 or a-f). If None, the standard colours will be used

Type
list

Raises

- `ValueError` – If no maps or models were specified
- `ValueError` – If the map is not .mrc format
- `ValueError` – If models are not in pdb of mmcif format
- `ValueError` – If the number of maps and map opacities don't match

class pipeliner.results_display_objects.**ResultsDisplayMontage**(*, *xvalues: list*, *yvalues: list*, *img: str*, *title: str*, *associated_data: list*, *labels: list | None = None*, *start_collapsed: bool = False*, *flag: str = ''*)

Bases: *ResultsDisplayObject*

An object to send to the GUI to make an image montage

This one is an image montage with info about the specific images It is best to not instantiate this class directly. Instead create it using *create_results_display_object*

title

The title of the object/graph

Type
str

xvalues

(list): The x coordinates by image

yvalues

(list): The y coordinates by image

labels

Data labels for the images

Type
list

associated_data

A list of files that contributed the data used in the image/graph

Type
list

img

Path to an image to display

Type
str

start_collapsed

Should the object start out collapsed when displayed in the GUI

Type
bool

class pipeliner.results_display_objects.**ResultsDisplayObject**(*title: str*, *start_collapsed: bool = False*, *flag=''*)

Bases: object

Abstract super-class for results display objects

title

The title

Type
str

start_collapsed

Should the object start out collapsed when displayed in the GUI

Type
bool

dobj_type

Used to identify what kind of ResultsDisplayObject it is

Type
str

flag

A message that is displayed if the results display object is showing somthing scientifically dubious.

Type
str

write_displayobj_file(*outdir*) → None

Write a json file from a ResultsDisplayObject object

Parameters
outdir (*str*) – The directory to write the output in

Raises
NotImplementedError – If a write attempt is made from the superclass

class pipeliner.results_display_objects.**ResultsDisplayPdfFile**(*, *file_path: str*, *title: str = ''*, *start_collapsed: bool = False*, *flag: str = ''*)

Bases: *ResultsDisplayObject*

An object for the GUI to display pdf files

It is best to not instantiate this class directly. Instead create it using *create_results_display_object*

file_path

Path to the file

Type
str

start_collapsed

Should the object start out collapsed when displayed in the GUI

Type
bool

class pipeliner.results_display_objects.**ResultsDisplayPending**(*, *title: str = 'Results pending...'*, *message: str = 'The result not available yet'*, *reason: str = 'unknown'*, *start_collapsed: bool = False*, *flag: str = ''*)

Bases: *ResultsDisplayObject*

A placeholder class for when a job is not able to produce results yet

class pipeliner.results_display_objects.**ResultsDisplayPlotlyHistogram**(*, *data: List[float] | DataFrame | ndarray | dict | None = None*, *title: str*, *x: str | list | None = None*, *y: str | list | None = None*, *color: str | int | list | None = None*, *nbins: int | None = None*, *range_x: list | None = None*, *range_y: list | None = None*, *category_orders: dict | None = None*, *labels: dict | None = None*, *bin_counts: List[float] | None = None*, *bin_centres: List[float] | None = None*, *associated_data: list*, *start_collapsed: bool = False*, *flag: str = ''*, ***kwargs*)

Bases: *ResultsDisplayObject*

A class that generates plotly.graph_objects.Figure object to display a histogram Uses plotly express histogram https://plotly.com/python-api-reference/generated/plotly.express.histogram.html Examples here: https://plotly.com/python/histograms/

data

The data to bin. Following types are allowed list - list of values to be binned array and dict - converted to a pandas dataframe internally pandas dataframe - ensure column names are added if 'x' indicates a column name More details https://plotly.com/python/px-arguments/

title

The title of the plot

Type
str

plotlyfig

plotly.graph_objects.Figure object generated from input data

Type
plotly.graph_objects.Figure

associated_data

A list of the associated data files

Type
list

start_collapsed

Should the object start out collapsed when displayed in the GUI

Type
bool

class pipeliner.results_display_objects.**ResultsDisplayPlotlyObj**(*, *data: list | DataFrame | ndarray | dict*, *plot_type: list | str*, *title: str*, *associated_data: list*, *multi_series: bool = False*, *subplot: bool = False*, *make_subplot_args: dict | None = None*, *subplot_order: str | List[tuple] | None = None*, *subplot_size: Sequence[int] | None = None*, *subplot_args: List[dict] | None = None*, *series_args: List[dict] | None = None*, *layout_args: dict | None = None*, *trace_args: dict | None = None*, *xaxes_args: List[dict] | dict | None = None*, *yaxes_args: List[dict] | dict | None = None*, *start_collapsed: bool = False*, *flag: str = ''*, ***kwargs*)

Bases: *ResultsDisplayObject*

This uses the plotly express class to create plotly.graph_objects.Figure object https://plotly.com/python/plotly-express/ Use this class to generate plotly Figure objects for custom plots including facet-plots: https://plotly.com/python/facet-plots/ subplots: https://plotly.com/python/subplots/ multi_series: e.g. https://plotly.com/python/creating-and-updating-figures/#adding-traces

data

The data to plot. For a single plot, following types are allowed: list - list of values to be binned array and dict - converted to a pandas dataframe internally pandas dataframe - ensure column names are added if 'x' indicates a column name More details https://plotly.com/python/px-arguments/ For subplots and/or multi_series: list - list with dictionary of arguments for each plot/series

plot_type

Required, type of plot. For a single plot, it is the plotly express function to call https://plotly.com/python-api-reference/plotly.express.html For subplots and/or multi_series, plotly.graph_objects function to call https://plotly.com/python-api-reference/plotly.graph_objects.html

Type
str

title

The title of the plot

Type
str

plotlyfig

plotly.graph_objects.Figure object generated from input data

Type
plotly.graph_objects.Figure

associated_data

A list of the associated data files

Type
list

start_collapsed

Should the object start out collapsed when displayed in the GUI

Type
bool

check_multiseries_arguments(*data*, *plot_type*, *series_args*) → None

check_plottype_list(*plot_type*, *data*) → None

check_singleplot_arguments(*plot_type*) → None

check_subplot_arguments(*data*, *subplot_size*, *subplot_order*, *plot_type*, *subplot_args*, *xaxes_args*, *yaxes_args*) → None

generate_multiseries_plots(*plot_type*, *plot_args*) → Figure

generate_subplots(*subplot_size*, *plot_type*, *subplot_order*, *plot_args*, *make_subplot_args*) → Figure

set_multiplot_data(*data*) → None

set_singleplot_data(*data*) → None

class pipeliner.results_display_objects.**ResultsDisplayPlotlyScatter**(*, *data: List[List[float]] | DataFrame | ndarray | dict | None = None*, *title: str*, *x: str | List[float] | None = None*, *y: str | List[float] | None = None*, *color: str | int | Sequence[str] | None = None*, *size: str | int | Sequence[str] | None = None*, *symbol: str | int | Sequence[str] | None = None*, *hover_name: str | int | Sequence[str] | None = None*, *range_color: list | None = None*, *range_x: list | None = None*, *range_y: list | None = None*, *category_orders: dict | None = None*, *labels: dict | None = None*, *associated_data: list*, *start_collapsed: bool = False*, *flag: str = ''*, ***kwargs*)

Bases: *ResultsDisplayObject*

A class that generates plotly.graph_objects.Figure object to display a scatter plot Uses plotly express scatter https://plotly.com/python-api-reference/generated/plotly.express.scatter.html Examples here: https://plotly.com/python/line-and-scatter/

data

The data to bin. Following types are allowed list - list of values to be binned array and dict - converted to a pandas dataframe internally pandas dataframe - ensure column names are added if 'x' indicates a column name More details https://plotly.com/python/px-arguments/

title

The title of the plot

Type
str

plotlyfig

plotly.graph_objects.Figure object generated from input data

Type
plotly.graph_objects.Figure

associated_data

A list of the associated data files

Type
list

start_collapsed

Should the object start out collapsed when displayed in the GUI

Type
bool

class pipeliner.results_display_objects.**ResultsDisplayRvapi**(*, *title: str*, *rvapi_dir: str*, *start_collapsed: bool = False*, *flag: str = ''*)

Bases: *ResultsDisplayObject*

An object for the GUI to display rvapi objects

It is best to not instantiate this class directly. Instead create it using *create_results_display_object*

This can be used for general HTML display in Doppio. Create a directory with index.html and it will be shown in the results display tab

rvapi_dir

Path to the rvapi directory

Type
str

start_collapsed

Should the object start out collapsed when displayed in the GUI

Type
bool

class pipeliner.results_display_objects.**ResultsDisplayTable**(*, *title: str*, *headers: list*, *table_data: list*, *associated_data: list*, *header_tooltips: list | None = None*, *start_collapsed: bool = False*, *flag: str = ''*)

Bases: *ResultsDisplayObject*

An object for the GUI to display a table

It is best to not instantiate this class directly. Instead create it using *create_results_display_object*

title

The title of the table

Type
str

headers

The column headers for the table

Type
list

table_data

A list of lists, on per row

Type
list

associated_data

A list of the associated data files

Type
list

header_tooltips

Tooltips for each column. Column header by default

Type
list

start_collapsed

Should the object start out collapsed when displayed in the GUI

Type
bool

class pipeliner.results_display_objects.**ResultsDisplayText**(*, *title: str*, *display_data: str*, *associated_data: list*, *start_collapsed: bool = False*, *flag: str = ''*)

Bases: *ResultsDisplayObject*

A class to display general text in the GUI results tab

It is best to not instantiate this class directly. Instead create it using *create_results_display_object*

title

the title of the section

Type
str

display_data

The text to display

Type
str

associated_data

Data files associated with this result

Type
list

start_collapsed

Should the object start out collapsed when displayed in the GUI

Type
bool

class pipeliner.results_display_objects.**ResultsDisplayTextFile**(*, *file_path: str*, *title: str = ''*, *start_collapsed: bool = False*, *flag: str = ''*)

Bases: *ResultsDisplayObject*

An object for the GUI to display ascii tecxt files

It is best to not instantiate this class directly. Instead create it using *create_results_display_object*

This can be used for default display of files that have ascii encoded text but the formats are too variable to make a more complex ResultsDisplayFile

file_path

Path to the file

Type
str

start_collapsed

Should the object start out collapsed when displayed in the GUI

Type
bool

pipeliner.results_display_objects.**get_next_resultsfile_name**(*dir: str*, *search_str: str*) → str

Get the name of the next results file

taking into account existing files of this type in the output dir to prevent overwriting existing ones

Parameters

- **dir** (*str*) – The output directory
- **search_str** (*str*) – The full name for the file with * in place of the number

Returns
The name of the file

Return type
str

7.1.4 Deposition Objects

DepositionObjects are returned by a `PipelinerJob`'s `prepare_deposition_data` function and are used to prepare automated depositions to the PDB, EMDB, and EMPIAR.

EMPIAR DepositionObjects

class pipeliner.deposition_tools.empiar_deposition_objects.**EmpiarCorrectedMics**(*name: str = 'Corrected micrographs'*, *directory: str | NoneType = None*, *category: str | NoneType = None*, *header_format: str | NoneType = None*, *data_format: str | NoneType = None*, *num_images_or_tilt_series: int | NoneType = None*, *frames_per_image: int | NoneType = None*, *voxel_type: str | NoneType = None*, *pixel_width: float | NoneType = None*, *pixel_height: float | NoneType = None*, *details: str | NoneType = None*, *image_width: int | NoneType = None*, *image_height: int | NoneType = None*, *micrographs_file_pattern: str | NoneType = None*)

Bases: object

category: str | None = None

data_format: str | None = None

details: str | None = None

directory: str | None = None

frames_per_image: int | None = None

header_format: str | None = None

image_height: int | None = None

image_width: int | None = None

micrographs_file_pattern: str | None = None

name: str = 'Corrected micrographs'

num_images_or_tilt_series: int | None = None

pixel_height: float | None = None

pixel_width: float | None = None

voxel_type: str | None = None

class pipeliner.deposition_tools.empiar_deposition_objects.**EmpiarData**(*name: str = ''*, *directory: str | NoneType = None*, *category: str | NoneType = None*, *header_format: str | NoneType = None*, *data_format: str | NoneType = None*, *num_images_or_tilt_series: int | NoneType = None*, *frames_per_image: int | NoneType = None*, *voxel_type: str | NoneType = None*, *pixel_width: float | NoneType = None*, *pixel_height: float | NoneType = None*, *details: str | NoneType = None*, *image_width: int | NoneType = None*, *image_height: int | NoneType = None*, *micrographs_file_pattern: str | NoneType = None*, *picked_particles_file_pattern: str | NoneType = None*)

Bases: object

category: str | None = None

```
    data_format: str | None = None
    details: str | None = None
    directory: str | None = None
    frames_per_image: int | None = None
    header_format: str | None = None
    image_height: int | None = None
    image_width: int | None = None
    micrographs_file_pattern: str | None = None
    name: str = ''
    num_images_or_tilt_series: int | None = None
    picked_particles_file_pattern: str | None = None
    pixel_height: float | None = None
    pixel_width: float | None = None
    voxel_type: str | None = None
```

class pipeliner.deposition_tools.empiar_deposition_objects.**EmpiarMovieSet**(*name: str = 'Multiframe micrograph movies'*, *directory: str | NoneType = None*, *category: str | NoneType = None*, *header_format: str | NoneType = None*, *data_format: str | NoneType = None*, *num_images_or_tilt_series: int | NoneType = None*, *frames_per_image: int | NoneType = None*, *voxel_type: str | NoneType = None*, *pixel_width: float | NoneType = None*, *pixel_height: float | NoneType = None*, *details: str | NoneType = None*, *image_width: int | NoneType = None*, *image_height: int | NoneType = None*, *micrographs_file_pattern: str | NoneType = None*)

Bases: object

category: str | None = None

data_format: str | None = None

details: str | None = None

directory: str | None = None

frames_per_image: int | None = None

header_format: str | None = None

image_height: int | None = None

image_width: int | None = None

micrographs_file_pattern: str | None = None

name: str = 'Multiframe micrograph movies'

num_images_or_tilt_series: int | None = None

pixel_height: float | None = None

pixel_width: float | None = None

voxel_type: str | None = None

class pipeliner.deposition_tools.empiar_deposition_objects.**EmpiarParticles**(*name: str = 'Particle images', directory: str | NoneType = None, category: str | NoneType = None, header_format: str | NoneType = None, data_format: str | NoneType = None, num_images_or_tilt_series: int | NoneType = None, frames_per_image: int | NoneType = None, voxel_type: str | NoneType = None, pixel_width: float | NoneType = None, pixel_height: float | NoneType = None, details: str | NoneType = None, image_width: int | NoneType = None, image_height: int | NoneType = None, micrographs_file_pattern: str | NoneType = None, picked_particles_file_pattern: str | NoneType = None*)

Bases: object

category: str | None = None

data_format: str | None = None

details: str | None = None

directory: str | None = None

frames_per_image: int | None = None

header_format: str | None = None

```
    image_height: int | None = None
    image_width: int | None = None
    micrographs_file_pattern: str | None = None
    name: str = 'Particle images'
    num_images_or_tilt_series: int | None = None
    picked_particles_file_pattern: str | None = None
    pixel_height: float | None = None
    pixel_width: float | None = None
    voxel_type: str | None = None
```

class pipeliner.deposition_tools.empiar_deposition_objects.**EmpiarRefinedParticles**(*name: str = 'Per-particle motion corrected particle images'*, *directory: str | None-Type = None*, *category: str | None-Type = None*, *header_format: str | None-Type = None*, *data_format: str | None-Type = None*, *num_images_or_tilt_series: int | None-Type = None*, *frames_per_image: int | None-Type = None*, *voxel_type: str | None-Type = None*, *pixel_width: float | None-Type = None*, *pixel_height: float | None-Type = None*, *details: str | None-Type = None*, *image_width: int | None-*

Bases: object

category: str | None = None

data_format: str | None = None

details: str | None = None

directory: str | None = None

frames_per_image: int | None = None

header_format: str | None = None

image_height: int | None = None

image_width: int | None = None

micrographs_file_pattern: str | None = None

name: str = 'Per-particle motion corrected particle images'

num_images_or_tilt_series: int | None = None

picked_particles_file_pattern: str | None = None

pixel_height: float | None = None

pixel_width: float | None = None

voxel_type: str | None = None

class pipeliner.deposition_tools.empiar_deposition_objects.**Micrograph**(*file: str*, *ext: str*, *n_frames: int*, *dimx: int*, *dimy: int*, *dtype: str*, *headtype: str*, *apix: float*, *voltage: float*, *spherical_abberation: float*)

Bases: object

apix: float

dimx: int

dimy: int

dtype: str

ext: str

file: str

headtype: str

n_frames: int

spherical_abberation: float

voltage: *float*

pipeliner.deposition_tools.empiar_deposition_objects.**empiar_check**(*emp_dep_obj: Any*, *attribute: str*, *number: int*)

Check that an attribute in empiar format is valid

Checks values in the form ("T<n>, "") for things like EMPIARs, header_format value

Parameters

- **emp_dep_obj** (*object*) – The EMPIAR DepositionObject
- **attribute** (*str*) – Name of the attribute to check
- **number** (*int*) – Max value for the 'T' value, (OT OtherType) will always be added as well

Returns
The error or None if there is no error

Return type
str

pipeliner.deposition_tools.empiar_deposition_objects.**empiar_type_is_valid**(*empobj: Any*)

pipeliner.deposition_tools.empiar_deposition_objects.**get_imgfile_info**(*imgfile: str*, *blockname: str*, *img_block_col: str*) → Tuple[Dict[str, Tuple[float, float, float]], List[str]]

Get information from the starfile containing image info

Parameters

- **imgfile** (*str*) – The node object for the file
- **blockname** (*str*) – Name of the images block in the starfile
- **img_block_col** (*str*) – The name of the column for the images in the image data block of the starfile

Returns
(Dict with info about the opitcs groups {og_number: (apix, voltage, sphere. ab)}, List of full paths (relative to the working dir) for all the images in the file, except in the case of movies then the path is relative to import dir)

Return type
tuple

pipeliner.deposition_tools.empiar_deposition_objects.**prepare_empiar_mics**(*in_file: str*) → List[*EmpiarCorrectedMics*]

pipeliner.deposition_tools.empiar_deposition_objects.**prepare_empiar_mics_parts_data**(*mpfile: str*, *is_parts: bool*, *is_cor_parts: bool*) → List[*EmpiarData*]

Prepare the micrographs or particles portion of an EMPIAR deposition

Parameters

- **mpfile** (*str*) – The name of the file containing the micrographs or particles
- **is_parts** (*bool*) – Is the image set particles? will affect the info in the details
- **is_cor_parts** (*bool*) – Is the image set corrected (polished particles)?

Returns
A list of deposition objects

Return type
list

pipeliner.deposition_tools.empiar_deposition_objects.**prepare_empiar_parts**(*in_file: str*, *is_polished: bool = False*) → Sequence[*EmpiarParticles* | *EmpiarRefinedParticles*]

pipeliner.deposition_tools.empiar_deposition_objects.**prepare_empiar_raw_mics**(*movfile: str*) → List[*EmpiarMovieSet*]

Prepare the raw micrographs portion of an EMPIAR deposition

Parameters
movfile (*str*) – Movies star file to operate on

Returns

A
DepositionObject used to create a deposition

Return type
List[*EmpiarMovieSet*]

PDB/EMDB DepositionObjects

PDB/EMDB Deposition object correspond to the schema here: http://ftp.ebi.ac.uk/pub/databases/emdb/doc/XML-schemas/emdb-schemas/v3/current_v3/doc/Untitled.html

class pipeliner.deposition_tools.onedep_deposition_objects.**Em2dProjectionSelection**(*id: int | List[str | None] | str | None = None, entry_id: str = 'ENTRY_ID', details: str | None = None, method: str | None = None, num_particles: int | None = None, software_name: str | None = None, citation_id: int | List[str | None] | str | None = None*)

Bases: *OneDepData*

citation_id: int | List[str | None] | str | None = None

details: str | None = None

entry_id: str = 'ENTRY_ID'

method: str | None = None

num_particles: int | None = None

```
    software_name:  str | None = None
```

class pipeliner.deposition_tools.onedep_deposition_objects.**Em3dReconstruction**(*id: int | List[str | NoneType] | str | NoneType = None*, *entry_id: str = 'ENTRY_ID'*, *num_particles: int | NoneType = None*, *symmetry_type: str | NoneType = None*, *image_processing_id: int | List[str | NoneType] | str | NoneType = None*, *actual_pixel_size: float | NoneType = None*, *algorithm: str | NoneType = None*, *citation_id: int | List[str | NoneType] | str | NoneType = None*, *ctf_correction_method: str | NoneType = None*, *details: str | NoneType = None*, *euler_angles_details: str | NoneType = None*, *fsc_type: str | NoneType = None*, *magnification_calibration: str | NoneType = None*, *method: str | NoneType = None*, *nominal_pixel_size: float | NoneType = None*, *num_class_averages: int | NoneType = None*, *reconstruction_type: str | NoneType = None*, *resolution:*

Bases: *OneDepData*

actual_pixel_size: float | None = None

algorithm: str | None = None

citation_id: int | List[str | None] | str | None = None

ctf_correction_method: str | None = None

details: str | None = None

entry_id: str = 'ENTRY_ID'

euler_angles_details: str | None = None

fsc_type: str | None = None

image_processing_id: int | List[str | None] | str | None = None

magnification_calibration: str | None = None

method: str | None = None

nominal_pixel_size: float | None = None

num_class_averages: int | None = None

num_particles: int | None = None

refinement_type: str | None = None

resolution: float | None = None

resolution_method: str | None = None

software: str | None = None

symmetry_type: str | None = None

class pipeliner.deposition_tools.onedep_deposition_objects.**EmCtfCorrection**(*id: int | List[str | NoneType] | str | NoneType = None*, *image_processing_id: int | List[str | NoneType] | str | NoneType = None*, *type: Literal['PHASE FLIPPING AND AMPLITUDE CORRECTION'] = 'PHASE FLIPPING AND AMPLITUDE CORRECTION'*, *details: str | NoneType = None*)

Bases: *OneDepData*

details: str | None = None

image_processing_id: int | List[str | None] | str | None = None

type: Literal['PHASE FLIPPING AND AMPLITUDE CORRECTION'] = 'PHASE FLIPPING AND AMPLITUDE CORRECTION'

class pipeliner.deposition_tools.onedep_deposition_objects.**EmImageProcessing**(*id: int | List[str | NoneType] | str | NoneType = None*, *image_recording_id: int | List[str | NoneType] | str | NoneType = None*, *details: str | NoneType = None*)

Bases: *OneDepData*

details: str | None = None

image_recording_id: int | List[str | None] | str | None = None

class pipeliner.deposition_tools.onedep_deposition_objects.**EmImageRecording**(*id: int | List[str | NoneType] | str | NoneType = None*, *imaging_id: int | List[str | NoneType] | str | NoneType = None*, *avg_electron_dose_per_image: float | NoneType = None*, *average_exposure_time: float | NoneType = None*, *details: str | NoneType = None*, *detector_mode: str | NoneType = None*, *film_or_detector_model: str | NoneType = None*, *num_diffraction_images: int | NoneType = None*, *num_grids_imaged: int | NoneType = None*, *num_real_images: int | NoneType = None*)

Bases: *OneDepData*

average_exposure_time: float | None = None

avg_electron_dose_per_image: float | None = None

details: str | None = None

detector_mode: str | None = None

film_or_detector_model: str | None = None

imaging_id: int | List[str | None] | str | None = None

num_diffraction_images: int | None = None

num_grids_imaged: int | None = None

num_real_images: int | None = None

class pipeliner.deposition_tools.onedep_deposition_objects.**EmImaging**(*id: int | List[str | NoneType] | str | NoneType = None*, *entry_id: str = 'ENTRY_ID'*, *accelerating_voltage: int | NoneType = None*, *illumination_mode: str | NoneType = None*, *electron_source: str | NoneType = None*, *microscope_model: str | NoneType = None*, *imaging_mode: str | NoneType = None*, *sample_support_id: int | List[str | NoneType] | str | NoneType = None*, *specimen_holder_type: str | NoneType = None*, *specimen_holder_model: str | NoneType = None*, *details: str | NoneType = None*, *date: str | NoneType = None*, *mode: str | NoneType = None*, *nominal_cs: float | NoneType = None*, *nominal_defocus_min: float | NoneType = None*, *nominal_defocus_max: float | NoneType = None*, *tilt_angle_min: float | NoneType = None*, *tilt_angle_max: float | NoneType = None*, *nominal_magnification: float | NoneType = None*, *calibrated_magnification: float | NoneType = None*, *energy_filter: str | NoneType = None*, *energy_window: str | NoneType = None*, *temperature: float | NoneType = None*, *detector_distance: float | NoneType = None*, *recording_temperature_minimum: float | NoneType = None*, *recording_temperature_maximum: float | NoneType = None*)

Bases: *OneDepData*

```
    accelerating_voltage: int | None = None
    calibrated_magnification: float | None = None
    date: str | None = None
    details: str | None = None
    detector_distance: float | None = None
    electron_source: str | None = None
    energy_filter: str | None = None
    energy_window: str | None = None
    entry_id: str = 'ENTRY_ID'
    illumination_mode: str | None = None
    imaging_mode: str | None = None
    microscope_model: str | None = None
    mode: str | None = None
    nominal_cs: float | None = None
    nominal_defocus_max: float | None = None
    nominal_defocus_min: float | None = None
    nominal_magnification: float | None = None
    recording_temperature_maximum: float | None = None
    recording_temperature_minimum: float | None = None
    sample_support_id: int | List[str | None] | str | None = None
    specimen_holder_model: str | None = None
    specimen_holder_type: str | None = None
    temperature: float | None = None
    tilt_angle_max: float | None = None
    tilt_angle_min: float | None = None
```

class pipeliner.deposition_tools.onedep_deposition_objects.**EmSampleSupport**(*id: int | List[str | NoneType] | str | NoneType = None*, *entry_id: str = 'ENTRY_ID'*, *film_material: str | NoneType = None*, *grid_type: str | NoneType = None*, *grid_material: str | NoneType = None*, *grid_mesh_size: str | NoneType = None*, *details: str | NoneType = None*)

Bases: *OneDepData*

details: str | None = None

entry_id: str = 'ENTRY_ID'

film_material: str | None = None

grid_material: str | None = None

grid_mesh_size: str | None = None

grid_type: str | None = None

class pipeliner.deposition_tools.onedep_deposition_objects.**EmSoftware**(*id: int | List[str | NoneType] | str | NoneType = None, category: Literal['CLASSIFICATION', 'CTF CORRECTION', 'DIFFRACTION INDEXING', 'FINAL EULER ASSIGNMENT', 'IMAGE ACQUISITION', 'INITIAL EULER ASSIGNMENT', 'LAYERLINE INDEXING', 'MASKING', 'MODEL FITTING', 'MODEL REFINEMENT', 'OTHER', 'PARTICLE SELECTION', 'RECONSTRUCTION'] | NoneType = None, details: str | NoneType = None, name: str | NoneType = None, version: str | NoneType = None, image_processing_id: int | List[str | NoneType] | str | NoneType = None, fitting_id: int | List[str | NoneType] | str | NoneType = None, imaging_id: int | List[str | NoneType] | str | NoneType = None*)

Bases: *OneDepData*

category: Literal['CLASSIFICATION', 'CTF CORRECTION', 'DIFFRACTION INDEXING', 'FINAL EULER ASSIGNMENT', 'IMAGE ACQUISITION', 'INITIAL EULER ASSIGNMENT', 'LAYERLINE INDEXING', 'MASKING', 'MODEL FITTING', 'MODEL REFINEMENT', 'OTHER', 'PARTICLE SELECTION', 'RECONSTRUCTION'] | None = None

details: str | None = None

fitting_id: int | List[str | None] | str | None = None

image_processing_id: int | List[str | None] | str | None = None

imaging_id: int | List[str | None] | str | None = None

name: str | None = None

version: str | None = None

class pipeliner.deposition_tools.onedep_deposition_objects.**EmSpecimen**(*id: int | List[str | NoneType] | str | NoneType = None*, *experiment_id: int = 1*, *concentration: str | NoneType = None*, *details: str | NoneType = None*, *embedding_applied: bool = False*, *shadowing_applied: bool = False*, *staining_applied: bool = False*, *vitrification_applied: bool = True*)

Bases: *OneDepData*

concentration: str | None = None

details: str | None = None

embedding_applied: bool = False

experiment_id: int = 1

shadowing_applied: bool = False

staining_applied: bool = False

vitrification_applied: bool = True

class pipeliner.deposition_tools.onedep_deposition_objects.**OneDepData**(*id: int | List[str | NoneType] | str | NoneType = None*)

Bases: object

id: int | List[str | None] | str | None = None

pipeliner.deposition_tools.onedep_deposition_objects.**ddc**

alias of *Em3dReconstruction*

7.1.5 Deposition tools

These functions combine the information from the DepositionObjects returned by a `PipelinerJob` into a format for automated deposition.

class pipeliner.deposition_tools.onedep_deposition.**DepositionData**(*field_name: str*, *data_items: object*, *merge_strat: str*, *reverse: bool = False*)

Bases: object

An object that holds data to be depositied via the onedep system

field_name

The name of the field in the ped/emdb these should be drawn from https://mmcif.wwpdb.org/dictionaries/mmcif_pdbx_v50.dic/Groups/index.html

Type
str

data_items

A dataclass from *pipeliner.deposition_tools.onedep_deposition_objects* containing the data

Type
object

dc_type

The subclass of the dataclass used for data_items

Type
DepositionData

cif_type

("loop" or "pairs") How the data will be written into the final cif file.

Type
str

merge_strategy

("multiple", "combine", or "overwrite") How multiple display objects will be combined, Multiple: means each DepositionData object will be given its own entry in a loop in the cif file. Overwrite: Only the first/last (depending on *self.reverse*) will be used. Combine: the objects will be combined, with each field being overwritten by subsequent data in a given field if it is not None.

Type
str

reverse

What order should the objects be considered in when combined False: The latest items take precedence, True: The earliest items take precedence

Type
bool

add_to_cif(*block: Block*, *did: str*)

Add a DepositionData object to a deposition cif file

Parameters

- **block** (*gemmi.cif.Block*) – The block of the cif.Document that is being written
- **did** (*str*) – The id of the deposition that is being written

Raises
ValueError – If the cif type is not in ("loop", "pairs")

static cifformat(*val: int | str | float | bool | None*, *depoid: str*)

Format data for writing to cif files in gemmi

All data must be strings Anything with spaces gets single quotated None is replaced with "?"

Parameters

- **val** (*Any*) – The data value

- **depoid** (*str*) – The id of the deposition, which will be substituted for if the val is the placeholder "ENTRY_ID"

Returns
The updated value

Return type
str

`update_job_references`(*jobs_dict: Dict[str, list]*)

Update the job references in a DepositionData object

If an object needs to refer to a DepositionData object from another job it uses the placeholder "JOBREF:<jobname>". Replaces the placeholder with the uuid of the appropriate DepositionData object

`write_json`(*output_dir: str*)

Write a json file from DepositionData object

Writes a file called deposition_<field_name>.json :param output_dir: The dir to write the file in. :type output_dir: str

`class pipeliner.deposition_tools.onedep_deposition.OneDepDeposition`(*terminal_job*)

Bases: `object`

Object for making a onedep deposition

Broken down in submethods for testing

`raw_depobjs`

A list of lists each contain deposition dataclasses of the same type

Type
list

`merged_depobjs`

The raw_repobjs list, which each subgroup merged according to the merge strategy of that type of deposition object

Type
list

`final_depobjs`

The merged depobjs with their UIDs and job references updated to their actual values

Type
list

`int_ids`

The integer ids that correspond to the UIDs assigned to each of the raw deposition objects

Type
dict

`make_int_ids`()

Make a dictionary {UID: integer ID}

`merge_depobjs`()

Merge deposition objects according to their merge strategy

`prepare_deposition`()

Do all the steprs to get ready for a deposition

update_jobrefs()

Update job references from 'JOBREF: jobname' to the UID for that job'

update_uids_to_int_ids()

Update all UIDS in deposition objects to the integer ids

write_deposition_cif_file(*depo_id: str | None*)

Write a cif file for deposition through the onedep system

Parameters
depo_id – The id of the deposition, assigned by EMDB/PDB

Raises
ValueError – If the deposition has not been prepared to write yet

pipeliner.deposition_tools.onedep_deposition.**gather_onedep_jobs_and_depobjects**(*terminal_job: str*) → Tuple[List[List[*DepositionData*]], Dict[str, List[*DepositionData*]]]

Prepare a onedep deposition

Parameters
terminal_job (*str*) – The job to use

Returns

The gathered DepositionData objects. Each sublist
contains objects of one type.

Return type
List[List[*DepositionData*]]

Raises
ValueError – If the terminal job is not found.

pipeliner.deposition_tools.onedep_deposition.**make_deposition_data_object**(*data_obj*, *uid: int | List[str | None] | str | None = ''*) → *DepositionData*

Makes a new DepositionData object of the desired type

Parameters

- **data_obj** (*object*) – The data class for the object with the data in it
- **uid** (*str*) – A uuid4 for the object, if not specified one will be created, if None, no UID is needed

Returns
The created DepositionData object

Return type
DepositionData

Raises
ValueError – If the data dict is not appropriate for the selected dataclass

pipeliner.deposition_tools.onedep_deposition.**merge_depdata_items**(*new_di:* DepositionData, *old_di: List[*DepositionData*]*) → List[*DepositionData*]

Merge together DepositionData objects, using their merge strategy

Parameters

- **new_di** (DepositionData) – The object to be merged in
- **old_di** (*List[DepositionData]*) – Object(s) to be merged into. For DepositionData objects with the 'multiple' merge strategy this will be multiple objects for all others it will be a single object in the list

Returns

The merged object(s) a single item for "combine" and
"overwrite" merge strategies and multiple items for "multiple"

Return type
List[*DepositionData*]

Raises
ValueError – If the objects have different merge strategies

7.2 Nodes

class pipeliner.nodes.**AtomCoordsGroupMetadataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**AtomCoordsNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

default_results_display(*output_dir*) → *ResultsDisplayObject*

Generate a ResultsDisplayObject for the node

This method will be used for every node in Process if there is no 'create_results_display' function defined

class pipeliner.nodes.**DensityMapGroupMetadataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**DensityMapMetadataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**DensityMapNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

default_results_display(*output_dir*) → *ResultsDisplayObject*

Generate a ResultsDisplayObject for the node

This method will be used for every node in Process if there is no 'create_results_display' function defined

class pipeliner.nodes.**EulerAnglesNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**EvaluationMetricNode**(*name: str*, *kwds: List[str] | None = None*, *toplevel_description: str = 'A file containing evaluation metrics'*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**GenNodeFormatConverter**(*allowed_ext: Dict[Tuple[str, ...], str]*, *check_funct: Dict[str, Callable]*)

Bases: object

Convert node types for files that have different but equivalent file extensions

allowed_ext

A dict of extensions that are allowed for that nodetype IE: {("mrc", "mrcs", "map"): "mrc"}

Type
dict

check_funct

A function that checks a file is of the expected type, must return a bool

Type
staticmethod

gen_nodetype(*filename: str*, *override_format: str = ''*, *validate_file: bool = True*) → str

Assign a file to a general node type

Parameters

- **filename** (*str*) – File to create the node for
- **override_format** (*str*) – The node's format
- **validate_file** (*bool*) – DO validation on the file type - turn it off to speed up rewriting an entire pipeline

class pipeliner.nodes.**Image2DGroupMetadataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

default_results_display(*output_dir*) → *ResultsDisplayObject*

Generate a ResultsDisplayObject for the node

This method will be used for every node in Process if there is no 'create_results_display' function defined

class pipeliner.nodes.**Image2DMetadataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**Image2DNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

default_results_display(*output_dir*) → *ResultsDisplayObject*

Generate a ResultsDisplayObject for the node

This method will be used for every node in Process if there is no 'create_results_display' function defined

class pipeliner.nodes.**Image2DStackNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

default_results_display(*output_dir*) → *ResultsDisplayObject*

Generate a ResultsDisplayObject for the node

This method will be used for every node in Process if there is no 'create_results_display' function defined

class pipeliner.nodes.**Image3DGroupMetadataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**Image3DMetadataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**Image3DNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

default_results_display(*output_dir*)

Generate a ResultsDisplayObject for the node

This method will be used for every node in Process if there is no 'create_results_display' function defined

class pipeliner.nodes.**LigandDescriptionNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**LogFileNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

default_results_display(*output_dir*) → *ResultsDisplayObject*

Generate a ResultsDisplayObject for the node

This method will be used for every node in Process if there is no 'create_results_display' function defined

class pipeliner.nodes.**Mask2DNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

default_results_display(*output_dir*) → *ResultsDisplayObject*

Generate a ResultsDisplayObject for the node

This method will be used for every node in Process if there is no 'create_results_display' function defined

class pipeliner.nodes.**Mask3DNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

default_results_display(*output_dir*) → *ResultsDisplayObject*

Generate a ResultsDisplayObject for the node

This method will be used for every node in Process if there is no 'create_results_display' function defined

class pipeliner.nodes.**MicrographCoordsGroupNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

default_results_display(*output_dir*) → *ResultsDisplayObject*

Generate a ResultsDisplayObject for the node

This method will be used for every node in Process if there is no 'create_results_display' function defined

class pipeliner.nodes.**MicrographCoordsNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**MicrographGroupMetadataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

default_results_display(*output_dir*) → *ResultsDisplayObject*

Generate a ResultsDisplayObject for the node

This method will be used for every node in Process if there is no 'create_results_display' function defined

class pipeliner.nodes.**MicrographMetadataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**MicrographMovieGroupMetadataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

default_results_display(*output_dir*) → *ResultsDisplayObject*

Generate a ResultsDisplayObject for the node

This method will be used for every node in Process if there is no 'create_results_display' function defined

class pipeliner.nodes.**MicrographMovieMetadataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**MicrographMovieNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

default_results_display(*output_dir*) → *ResultsDisplayObject*

Generate a ResultsDisplayObject for the node

This method will be used for every node in Process if there is no 'create_results_display' function defined

class pipeliner.nodes.**MicrographNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

default_results_display(*output_dir*) → *ResultsDisplayObject*

Generate a ResultsDisplayObject for the node

This method will be used for every node in Process if there is no 'create_results_display' function defined

class pipeliner.nodes.**MicroscopeDataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**MlModelNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

default_results_display(*output_dir*) → *ResultsDisplayObject*

Generate a ResultsDisplayObject for the node

This method will be used for every node in Process if there is no 'create_results_display' function defined

class pipeliner.nodes.**Node**(*name: str*, *toplevel_type: str*, *kwds: List[str] | None = None*, *toplevel_description: str = 'A general node type for files without a more specific Node class defined'*, *format_converter:* GenNodeFormatConverter *| None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: object

Nodes store info about input and output files of jobs that have been run

name

The name of the file the node represents

Type
str

toplevel_type

The toplevel node type

Type
str

toplevel_description

(str): A description of the toplevel type

format

The node's generalised file type IE ('mrc' for mrc, 'mrcs' or 'map')

Type
str

kwds

Keywords associated with the node

Type
list[str]

output_from_process

The Process object for process that created the file

Type
Process

input_for_processes_list

A list of *Process* objects for processes that use the node as an input

Type
list

format_converter

An object that defines the expected extensions for the node and how to validate them

Type
Optional[*GenNodeFormatConverter*]

type

The full node type

Type
str

default_results_display(*output_dir*) → *ResultsDisplayObject*

Generate a ResultsDisplayObject for the node

This method will be used for every node in Process if there is no 'create_results_display' function defined

generalise_format() → None

Set the node's format

generalise node format for files that have different but equivalent file extensions EX 'mrc' for '.map', or '.mrc' files

defaults to the file extension

get_result_display_object() → *ResultsDisplayObject*

Get the default results display object for the node.

This method loads the results display object from a saved file on disk if there is one. If not, a new results display object is generated and saved.

Returns
The default results display object for the node

Return type
ResultsDisplayObject

load_results_display_file() → *ResultsDisplayObject* | None

Load the node's results display object from a file on disk.

This method must be fast because it is used by the GUI to load node results. Therefore, if the display object fails to load properly, no attempt is made to recalculate it and a `ResultsDisplayPending` object is returned instead.

If there is no results display file yet, `None` is returned.

Returns
A *ResultsDisplayObject*, or `None`.

not_compatible_with_default_output() → *ResultsDisplayObject*

Return a ResultsDisplayText object for jobs with no default display object

write_default_result_file() → None

Write the default display file for this node

class pipeliner.nodes.**OptimiserDataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**ParamsDataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**ParticleGroupMetadataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

default_results_display(*output_dir*) → *ResultsDisplayObject*

Generate a ResultsDisplayObject for the node

This method will be used for every node in Process if there is no 'create_results_display' function defined

class pipeliner.nodes.**ProcessDataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**RestraintsNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**RigidBodiesNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**SequenceAlignmentNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**SequenceGroupNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**SequenceNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**StructureFactorsNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**TiltSeriesGroupMetadataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**TiltSeriesMetadataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**TiltSeriesMovieGroupMetadataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**TiltSeriesMovieMetadataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**TiltSeriesMovieNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**TiltSeriesNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**TomoManifoldDataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**TomoOptimisationSetNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**TomoTrajectoryDataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**TomogramGroupMetadataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**TomogramMetadataNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

class pipeliner.nodes.**TomogramNode**(*name: str*, *kwds: List[str] | None = None*, *do_validation: bool = True*, *override_format: str = ''*)

Bases: *Node*

pipeliner.nodes.**add_to_node_display_file**(*node_name: str*, *display_file: str*) → None

Add an entry to the node display file

Parameters

- **node_name** (*str*) – The name of the node file
- **display_file** (*str*) – Path to the display file

pipeliner.nodes.**check_file_is_cif**(*file: str*) → bool

Validate that a file is a cif or star file

pipeliner.nodes.**check_file_is_json**(*file: str*) → bool

pipeliner.nodes.**check_file_is_mrc**(*file: str*) → bool

Validate that a file is a mrc file

pipeliner.nodes.**check_file_is_text**(*file: str*) → bool

pipeliner.nodes.**check_file_is_tif**(*file: str*) → bool

pipeliner.nodes.**get_node_results_display_file_name**(*node_name: str*) → str | None

Get the results display file name for a node from the node display file.

Returns: the name of the node results display file, or None if there is no entry
for this node name in the node display file.

pipeliner.nodes.**remove_from_node_display_file**(*node_name: str*) → None

Remove an entry from the node display file

Parameters
node_name (*str*) – The file to remove

7.3 Job Options

7.3.1 JobOptions

JobOptions are used to store parameters for jobs. They contain all the info necessary for on-the-fly GUI generation.

class pipeliner.job_options.**BooleanJobOption**(*, *label: str*, *default_value: bool*, *help_text: str = ''*, *in_continue: bool = False*, *only_in_continue: bool = False*, *deactivate_if:* JobOptionCondition | *None = None*, *jobop_group: str = 'Main'*)

Bases: *JobOption*

Define a job option as a boolean

Parameters

- **label** (*str*) – A verbose label for the parameter. This is what appears in a run.job file
- **default_value** (*bool*) – The default value
- **help_text** (*str*) – Text that will be displayed in the GUI if help is clicked
- **in_continue** (*bool*) – If this parameter can be modified in a job that is continued

get_boolean() → bool

Get a boolean value

Returns
The value

Return type
bool

joboption_type = 'BOOLEAN'

validate() → List[*JobOptionValidationResult*]

Basic validation of the input parameters

Returns

The validation
error of one exists

Return type
JobOptionValidationResult

class pipeliner.job_options.**DirPathJobOption**(*, *label: str*, *default_value: str = ''*, *help_text: str = ''*, *must_be_in_project: bool = False*, *must_exist: bool = True*, *in_continue: bool = False*, *only_in_continue: bool = False*, *is_required: bool = False*, *required_if:* JobOptionCondition | *None = None*, *deactivate_if:* JobOptionCondition | *None = None*, *validation_regex: str | None = None*, *regex_error_message: str | None = None*, *jobop_group: str = 'Main'*)

Bases: *StringJobOption*

A JobOption for directory paths

Parameters

- **label** (*str*) – A verbose label for the parameter.
- **default_value** (*None*) –
- **help_text** (*str*) – Text that will be displayed in the GUI if help is clicked.
- **in_continue** (*bool*) – If this parameter can be modified in a job that is continued
- **must_be_in_project** (*bool*) – Does the directory have to be in the project
- **must_exist** (*bool*) – Does this directory have to exist when the job option is validated? Usually true, unless it will be created by a previous command in the job

check_dir() → List[*JobOptionValidationResult*]

Check that the specified dir exists.

Returns

An error if the dir does not exist, or a warning if the dir does not exist and `must_exist` is `False`

Return type

List[*JobOptionValidationResult*]

joboption_type = 'DIRPATH'

validate() → List[*JobOptionValidationResult*]

Basic validation of the input parameters

class pipeliner.job_options.**ExternalFileJobOption**(*, *label: str*, *pattern: str = '*', *default_value: str = ''*, *help_text: str = ''*, *in_continue: bool = False*, *only_in_continue: bool = False*, *is_required: bool = False*, *required_if:* JobOptionCondition | *None = None*, *deactivate_if:* JobOptionCondition | *None = None*, *validation_regex: str | None = None*, *regex_error_message: str | None = None*, *suggestion: str = ''*, *jobop_group: str = 'Main'*)

Bases: *JobOption*

Define a job option as a file name

GUI will open a file browser for the user to find an input

Parameters

- **label** (*str*) – A verbose label for the parameter. This is what appears in a run.job file
- **default_value** (*str*) – will usually be set from the string and pattern.

- **pattern** (*str*) – Info about the search string for file types. It should be in the format <description> (<example>) e.g.: *"MRC files (*.mrc)"*
- **help_text** (*str*) – Text that will be displayed in the GUI if help is clicked
- **in_continue** (*bool*) – If this parameter can be modified in a job that is continued
- **suggestion** (*str*) – A hint that will appear in the data entry field if no no default value is set, it will not be used. IE: 'Enter file path here' If you want the suggestion used as the default then just put it in default_value

check_file() → List[*JobOptionValidationResult*]

Check that specified input files exist, also checks in path

Returns
An error message if the file does not exist

Return type
str

get_basename_mapping() → Dict[str, str]

Get a mapping of names from input files to unique basenames.

The purpose of this method is to get unique basenames (i.e. the file name alone without any directory part) for all input files, by adding a prefix if necessary to make sure files that would otherwise have identical basenames are all assigned a unique basename to use for output files.

Returns

A dict with the full original input file names as keys and the unique basenames as values, for example:

```
{
  "Import/job001/myfile.txt": "job001_myfile.txt",
  "Import/job002/myfile.txt": "job002_myfile.txt",
}
```

joboption_type = 'FILENAME'

validate() → List[*JobOptionValidationResult*]

Basic validation of the input parameters

Returns

The
validation error of one exists

Return type
List[*JobOptionValidationResult*]

class pipeliner.job_options.**FloatJobOption**(*, *label: str*, *default_value: str | float | None*, *suggested_min: float | None = None*, *suggested_max: float | None = None*, *step_value: float = 1.0*, *help_text: str = ''*, *in_continue: bool = False*, *only_in_continue: bool = False*, *is_required: bool = False*, *required_if:* JobOptionCondition | *None = None*, *deactivate_if:* JobOptionCondition | *None = None*, *hard_min: float | None = None*, *hard_max: float | None = None*, *jobop_group: str = 'Main'*)

Bases: *JobOption*

Define a job option as a slider for inputting numbers as floats

The min value and max value are for display only, they do not limit the actual values that can be input

Parameters

- **label** (*str*) – A verbose label for the parameter. This is what appears in a run.job file
- **default_value** (*float*) – The default value for the parameter
- **suggested_min** (*float*) – The suggested minimum value for the slider
- **suggested_max** (*float*) – The suggested maximum value for the slider
- **step_value** (*float*) – The step value for the slider
- **help_text** (*str*) – Text that will be displayed in the GUI if help is clicked
- **in_continue** (*bool*) – If this parameter can be modified in a job that is continued
- **hard_min** (*float*) – An error will be raised if the value is < this value
- **hard_max** (*float*) – An error will be raised if the value is > this value

get_number() → float

Get the value of a JobOption as a number

Returns
The value

Return type
float

Raises

- **ValueError** – If the value cannot be converted to a float
- **ValueError** – If the value is required and missing

joboption_type = 'FLOAT'

validate() → List[*JobOptionValidationResult*]

Basic validation check on the job option

Returns

The validation
error of one exists

Return type
JobOptionValidationResult

class pipeliner.job_options.**InputNodeJobOption**(*, *label: str*, *node_type: str*, *default_value: str = ''*, *pattern: str = '*', *help_text: str = ''*, *in_continue: bool = False*, *only_in_continue: bool = False*, *is_required: bool = False*, *required_if:* JobOptionCondition | *None = None*, *deactivate_if:* JobOptionCondition | *None = None*, *validation_regex: str* | *None = None*, *regex_error_message: str* | *None = None*, *suggestion: str = ''*, *jobop_group: str = 'Main'*, *node_kwds: List[str]* | *None = None*)

Bases: *JobOption*

Define a job option as a file name, create an input node for it

GUI will open a file browser for the user to find an input

Parameters

- **label** (*str*) – A verbose label for the parameter. This is what appears in a run.job file
- **node_type** (*str*) – Top level node type for this node
- **default_value** (*str*) – will be set by the specific class method
- **pattern** (*str*) – Info about the search string for file types. It should be in the format <description> (<example>) IE: *"MRC files (*.mrc)"*
- **help_text** (*str*) – Text that will be displayed in the GUI if help is clicked
- **in_continue** (*bool*) – If this parameter can be modified in a job that is continued
- **suggestion** (*str*) – A hint that will appear in the data entry field if no no default value is set, it will not be used. IE: 'Enter file path here' If you want the suggestion used as the default then just put it in default_value

check_file() → List[*JobOptionValidationResult*]

Check that specified input files exist

Returns

An error message if the file does not exist

Return type

str

get_basename_mapping() → Dict[str, str]

Get a mapping of names from input files to unique basenames.

The purpose of this method is to get unique basenames (i.e. the file name alone without any directory part) for all input files, by adding a prefix if necessary to make sure files that would otherwise have identical basenames are all assigned a unique basename to use for output files.

Returns

A dict with the full original input file names as keys and the unique basenames as values, for example:

```
{
  "Import/job001/myfile.txt": "job001_myfile.txt",
  "Import/job002/myfile.txt": "job002_myfile.txt",
}
```

get_input_nodes() → List[*Node*]

Get the input node(s) related to this job option.

Most job option types do not create input nodes and so they should rely on this default implementation that returns an empty list.

Options that do create nodes should return a list of the corresponding Node objects.

joboption_type = 'INPUTNODE'

validate() → List[*JobOptionValidationResult*]

Basic validation of the input parameters

Returns

The validation error of one exists

Return type

List[*JobOptionValidationResult*]

class pipeliner.job_options.**IntJobOption**(*, *label: str*, *default_value: int | None*, *suggested_min: int | None = None*, *suggested_max: int | None = None*, *step_value: int = 1*, *help_text: str = ''*, *in_continue: bool = False*, *only_in_continue: bool = False*, *is_required: bool = False*, *required_if:* JobOptionCondition *| None = None*, *deactivate_if:* JobOptionCondition *| None = None*, *hard_min: int | None = None*, *hard_max: int | None = None*, *jobop_group: str = 'Main'*)

Bases: *JobOption*

Define a job option as a slider for inputting numbers as integers

The slider min value and max value are for display only, they do not limit the actual values that can be input

Parameters

- **label** (*str*) – A verbose label for the parameter. This is what appears in a run.job file
- **default_value** (*int*) – The default value for the parameter
- **suggested_min** (*int*) – The suggested minimum value for the slider
- **suggested_max** (*int*) – The suggested maximum value for the slider
- **step_value** (*int*) – The step value for the slider
- **help_text** (*str*) – Text that will be displayed in the GUI if help is clicked
- **in_continue** (*bool*) – If this parameter can be modified in a job that is continued
- **hard_min** (*int*) – An error will be raised if the value is < this value
- **hard_max** (*int*) – An error will be raised if the value is > this value

Raises

ValueError – If the suggested_min, suggested_max, default, hard_min , hard_max, or step values are not integers

get_number() → int

Get the value of a JobOption as a number

Returns

The value

Return type

int

Raises

- **ValueError** – If the value cannot be converted to a `float`
- **ValueError** – If the value is required and missing

joboption_type = 'INT'

validate() → List[*JobOptionValidationResult*]

Basic validation check on the job option

Returns

The validation
error of one exists

Return type

JobOptionValidationResult

class pipeliner.job_options.**JobOption**(*, *label: str*, *default_value: Any = ''*, *help_text: str = ''*, *in_continue: bool = False*, *is_required: bool = False*, *required_if:* JobOptionCondition | *None = None*, *deactivate_if:* JobOptionCondition | *None = None*, *only_in_continue: bool = False*, *jobop_group: str = 'Main'*)

Bases: object

A JobOption stores a parameter, its value, and info about the GUI for that param

This is a general class with several more specialised class subclasses.

check_dir() → List[*JobOptionValidationResult*]

Check that the value is an existing directory

Returns
A JobOptionValidationResult for any errors

Return type
list

check_file() → List[*JobOptionValidationResult*]

Check that specified input files exist

Returns
A JobOptionValidationResult for any errors

Return type
list

check_required_not_empty() → None

Check if the job option is required but has no value

get_basename_mapping() → Dict[str, str]

Get a mapping of names from input files to unique basenames.

The purpose of this method is to get unique basenames (i.e. the file name alone without any directory part) for all input files, by adding a prefix if necessary to make sure files that would otherwise have identical basenames are all assigned a unique basename to use for output files.

Returns

A dict with the full original input file names as keys and the unique basenames as values, for example:

```
{
  "Import/job001/myfile.txt": "job001_myfile.txt",
  "Import/job002/myfile.txt": "job002_myfile.txt",
}
```

get_boolean() → bool

get_input_nodes() → List[*Node*]

Get the input node(s) related to this job option.

Most job option types do not create input nodes and so they should rely on this default implementation that returns an empty list.

Options that do create nodes should return a list of the corresponding Node objects.

get_list() → List[str]

get_number() → int | float

get_string() → str

Get the value of a JobOption as a string

Returns
The value

Return type
str

Raises
ValueError – If the value is required and missing

joboption_type = 'ERROR: no job option type set'

set_string(*set_to: str*) → None

Set the value of a JobOption to a string

Parameters
set_to (*str*) – The string to set the value to

validate() → List[*JobOptionValidationResult*]

Basic validation of input parameters

will be set by individual JobOption subtypes

class pipeliner.job_options.**JobOptionCondition**(*conditions: List[Tuple[str, str, str | bool | int | float]]*, *operation: Literal['ANY', 'ALL'] = 'ANY'*)

Bases: object

Definines conditions for JobOption based on other JobOptions

Used to determine if a JobOption should be required or deactivated

check_condition_is_met(*jobops: Dict[str, JobOption]*) → Tuple[bool, List[Tuple[str, str, str | bool | int | float]], List[Tuple[str, str, str | bool | int | float]]]

Has the condition been met?

Parameters
jobops (*dict*) – The JobOptions dict for the job

Returns

(bool: Has the condition been met?, List[conditions that passed],
List[conditions that failed]

Return type
Tuple

class pipeliner.job_options.**JobOptionValidationResult**(*result_type: Literal['warning', 'error', 'info']*, *raised_by: List[JobOption]*, *message: str*)

Bases: object

A class for handling validation of joboptions

type

One of 'warning', 'error' or 'info'

Type
str

raised_by

List of JobOption objects that raised the warning/error. Generally this will only be one JobOption, unless the warning/error was raised from comparing two or more JobOptions

Type
list

message

Description of the error/warning

Type
str

class pipeliner.job_options.**MultiExternalFileJobOption**(*, *label: str*, *pattern: str = ''*, *default_value: str = ''*, *help_text: str = ''*, *in_continue: bool = False*, *only_in_continue: bool = False*, *is_required: bool = False*, *required_if:* JobOptionCondition | *None = None*, *deactivate_if:* JobOptionCondition | *None = None*, *validation_regex: str* | *None = None*, *regex_error_message: str* | *None = None*, *suggestion: str = ''*, *jobop_group: str = 'Main'*, *allow_upload: bool = False*)

Bases: *JobOption*

Define a job option as multiple file names

GUI will open a file browser for the user to find an input for each one. The value for this should be a string of values separated by ::: which will be converted into a list by: *[x.strip() for x in the_str.split(":::")]*

Parameters

- **label** (*str*) – A verbose label for the parameter. This is what appears in a run.job file
- **default_value** (*str*) – will usually be set from the string and pattern
- **pattern** (*str*) – Info about the search string for file types. It should be in the format <description> (<example>) e.g.: *"MRC files (*.mrc)"*
- **help_text** (*str*) – Text that will be displayed in the GUI if help is clicked
- **in_continue** (*bool*) – If this parameter can be modified in a job that is continued
- **suggestion** (*str*) – A hint that will appear in the data entry field if no no default value is set, it will not be used. IE: 'Enter file path here' If you want the suggestion used as the default then just put it in default_value
- **allow_upload** (*bool*) – Should a file upload be allowed for this input? If yes a copy of the file will end up in the project. This should be turned off for external programs and files that are expected to be very large like ML models.

check_file() → List[*JobOptionValidationResult*]

Check that specified input files exist

For each input file that does not exist, a JobOptionValidationResult of type "error" is returned.

Returns
class:~pipeliner.job_options.JobOptionValidationResult]: if any of the files do not exist

Return type
List[

get_basename_mapping() → Dict[str, str]

Get a mapping of names from input files to unique basenames.

The purpose of this method is to get unique basenames (i.e. the file name alone without any directory part) for all input files, by adding a prefix if necessary to make sure files that would otherwise have identical basenames are all assigned a unique basename to use for output files.

Returns

A dict with the full original input file names as keys and the unique basenames as values, for example:

```
{
  "Import/job001/myfile.txt": "job001_myfile.txt",
  "Import/job002/myfile.txt": "job002_myfile.txt",
}
```

get_list() → List[str]

Return the items in self.value as a list of strings

Returns

The items in the JobOptions comma separated string
as a list, with any white space at the beginning and/or end removed

Return type
List[str]

joboption_type = 'MULTIFILENAME'

validate() → List[*JobOptionValidationResult*]

Basic validation of the input parameters

Returns
The validation errors of any exist

Return type
List[*JobOptionValidationResult*]

class pipeliner.job_options.**MultiInputNodeJobOption**(*, *label: str*, *node_type: str*, *default_value: str = ''*, *pattern: str = ''*, *help_text: str = ''*, *in_continue: bool = False*, *only_in_continue: bool = False*, *is_required: bool = False*, *required_if:* JobOptionCondition | *None = None*, *deactivate_if:* JobOptionCondition | *None = None*, *validation_regex: str* | *None = None*, *regex_error_message: str* | *None = None*, *suggestion: str = ''*, *jobop_group: str = 'Main'*, *node_kwds: List[str]* | *None = None*)

Bases: *JobOption*

Define a job option as a file name, create an input node for it

GUI will open a file browser for the user to find an input

Parameters

- **label** (*str*) – A verbose label for the parameter. This is what appears in a run.job file
- **node_type** (*str*) – Toplevel node type for this node.

- **node_kwds** – Optional[List[Optional[List[str]]]]: keywords for each toplevel type in node_types. The index of the toplevel type is used to match the kwds to toplevel type.
- **default_value** (*str*) – will be set by the specific class method
- **pattern** (*str*) – Info about the search string for file types. It should be in the format <description> (<example>), e.g. *"MRC files (*.mrc)"*
- **help_text** (*str*) – Text that will be displayed in the GUI if help is clicked
- **in_continue** (*bool*) – If this parameter can be modified in a job that is continued
- **suggestion** (*str*) – A hint that will appear in the data entry field if no no default value is set, it will not be used. IE: 'Enter file path here' If you want the suggestion used as the default then just put it in default_value

check_file() → List[*JobOptionValidationResult*]

Check that specified input files exist

For each input file that does not exist, a JobOptionValidationResult of type "error" is returned.

Returns
class:~pipeliner.job_options.JobOptionValidationResult]: if any of the files do not exist

Return type
List[

get_basename_mapping() → Dict[str, str]

Get a mapping of names from input files to unique basenames.

The purpose of this method is to get unique basenames (i.e. the file name alone without any directory part) for all input files, by adding a prefix if necessary to make sure files that would otherwise have identical basenames are all assigned a unique basename to use for output files.

Returns

A dict with the full original input file names as keys and the unique basenames as values, for example:

```
{
  "Import/job001/myfile.txt": "job001_myfile.txt",
  "Import/job002/myfile.txt": "job002_myfile.txt",
}
```

get_input_nodes() → List[*Node*]

Get the input node(s) related to this job option.

Most job option types do not create input nodes and so they should rely on this default implementation that returns an empty list.

Options that do create nodes should return a list of the corresponding Node objects.

get_list()

joboption_type = 'MULTIINPUTNODE'

validate() → List[*JobOptionValidationResult*]

Basic validation of the input parameters

Returns
The validation results (if any)

Return type
List[*JobOptionValidationResult*]

class pipeliner.job_options.**MultiStringJobOption**(*, *label: str*, *default_value: str = ''*, *help_text: str = ''*, *in_continue: bool = False*, *only_in_continue: bool = False*, *is_required: bool = False*, *required_if:* JobOptionCondition | *None = None*, *deactivate_if:* JobOptionCondition | *None = None*, *validation_regex: str | None = None*, *regex_error_message: str | None = None*, *jobop_group: str = 'Main'*)

Bases: *JobOption*

Create a job option object for a multiple strings. For things like keyword lists

The value for this should be a string containing values separated by ::: which will be converted into a list by: *[x.strip() for x in the_str.split(":::")]*

The only reason there can't just be a self.get_list() function in a regular StringJobOption is the GUI must handle them differently.

For regex validation all values in the list of inputs must match the regex

Parameters

- **label** (*str*) – A verbose label for the parameter.
- **default_value** (*None*) –
- **help_text** (*str*) – Text that will be displayed in the GUI if help is clicked.
- **in_continue** (*bool*) – If this parameter can be modified in a job that is continued

get_list() → List[str]

Return the items in self.value as a list of strings

Returns

The items in the JobOptions separated string
as a list, with any white space at the beginning and/or end removed

Return type
List[str]

joboption_type = 'MULTISTRING'

validate() → List[*JobOptionValidationResult*]

Basic validation of the input parameters

class pipeliner.job_options.**MultipleChoiceJobOption**(*, *label: str*, *choices: List[str]*, *default_value_index: int*, *help_text: str = ''*, *in_continue: bool = False*, *only_in_continue: bool = False*, *is_required: bool = True*, *required_if:* JobOptionCondition | *None = None*, *deactivate_if:* JobOptionCondition | *None = None*, *jobop_group: str = 'Main'*)

Bases: *JobOption*

Define a job option as a pull down menu

Radio is a misnomer, the GUI will display a pull down menu with the options

Parameters

- **label** (*str*) – A verbose label for the parameter. This is what appears in a run.job file
- **choices** (*list*) – A list str options for the menu
- **default_value_index** (*int*) – Index of the initial option for the radio; also used as the default value
- **help_text** (*str*) – Text that will be displayed in the GUI if help is clicked
- **in_continue** (*bool*) – If this parameter can be modified in a job that is continued

Raises
ValueError – If the index of the initial option is not in the radio menu

joboption_type = 'MULTIPLECHOICE'

validate() → List[*JobOptionValidationResult*]

Basic validation of the input parameters

Returns
The validation error of one exists

Return type
List[*JobOptionValidationResult*]

class pipeliner.job_options.**SearchStringJobOption**(*, *label: str*, *default_value: str = ''*, *help_text: str = ''*, *must_be_in_project: bool = False*, *in_continue: bool = False*, *only_in_continue: bool = False*, *is_required: bool = False*, *required_if:* JobOptionCondition | *None = None*, *deactivate_if:* JobOptionCondition | *None = None*, *validation_regex: str | None = None*, *regex_error_message: str | None = None*, *jobop_group: str = 'Main'*)

Bases: *StringJobOption*

A JobOption for search strings

Parameters

- **label** (*str*) – A verbose label for the parameter.
- **default_value** (*None*) –
- **help_text** (*str*) – Text that will be displayed in the GUI if help is clicked.
- **in_continue** (*bool*) – If this parameter can be modified in a job that is continued

check_file() → List[*JobOptionValidationResult*]

Check that at least one input file exists

Returns
Error if no files found

Return type
List[*JobOptionValidationResult*]

get_list()

joboption_type = 'SEARCHSTRING'

validate() → List[*JobOptionValidationResult*]

Basic validation of the input parameters

class pipeliner.job_options.**StringJobOption**(*, *label: str*, *default_value: str = ''*, *help_text: str = ''*, *in_continue: bool = False*, *only_in_continue: bool = False*, *is_required: bool = False*, *required_if:* JobOptionCondition | *None = None*, *deactivate_if:* JobOptionCondition | *None = None*, *validation_regex: str* | *None = None*, *regex_error_message: str* | *None = None*, *jobop_group: str = 'Main'*)

Bases: *JobOption*

Create a job option object for a string parameter.

Parameters

- **label** (*str*) – A verbose label for the parameter.
- **default_value** (*None*) –
- **help_text** (*str*) – Text that will be displayed in the GUI if help is clicked.
- **in_continue** (*bool*) – If this parameter can be modified in a job that is continued

joboption_type = 'STRING'

validate() → List[*JobOptionValidationResult*]

Basic validation of the input parameters

pipeliner.job_options.**files_exts**(*name: str = 'File'*, *exts: List[str]* | *None = None*, *exact: bool = False*) → str

Produce a description of files and their extensions

In a format compatible with PyQt5 QFileDialog

Parameters

- **name** (*str*) – The type of file IE: 'Micrograph movies'
- **exts** (*list*) – The acceptable extensions or file search strings, e.g. '.txt' to find `*.txt` or '_half1.star' for `*_half1.star`
- **exact** (*bool*) – Match the file name(s) exactly, do not prepend a *

Returns

Formatted for PyQt5 QFileDialog: 'name (`*ext1 *ext2 *ext3`)'

Return type

str

7.4 Processes

class pipeliner.process.**Process**(*name: str*, *p_type: str*, *status: str*, *alias: str* | *None = None*)

Bases: `object`

A Process represents a job that has been run by the Pipeliner

name

The name of the process. It should be in the format *"<jobtype>/jobxxx/"*. The trailing slash is required

Type

str

alias

An alternate name for the process to make it easier to identify

Type
str

outdir

The directory the process was written into

Type
str

p_type

The process' type

Type
str

status

The processes' status 'running, scheduled, successful, failed, or aborted'

Type
str

input_nodes

Node objects for files the process use as inputs

Type
list

output_nodes

Node objects for files the process produces

Type
list

remove_nonexistent_nodes() → List[str]

Checks the files for the output nodes of the process actually exist

Removes them from the output node list if the files are not found

Returns
The names of the removed nodes

Return type
List[str]

update_jobinfo_file(*action: str*, *comment: str | None = None*, *command_list: list | None = None*) → None

Update the file in the jobdir that stores info about the job

Parameters

- **current_proc** (*Process*) – Process object for the job being operated on
- **action** (*str*) – what action was performed on the job, e.g. Run, Scheduled, Cleaned up
- **comment** (*str*) – Comment to append to the job's comments list
- **command_list** (*list*) – Commands that were run. Generally *None* if action was any other than Run or Scheduled

7.5 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

A

B

C

D

E

F

G

H

K

L

M

N

O

P

R

S

T

U

V

W

X

Y