Pydra: A simple dataflow engine with scalable semantics

Release 0.19+0.gac06f8c.dirty

The Nipype Developers team

Jul 26, 2022

CONTENTS:

Pydra is a new lightweight dataflow engine written in Python. Pydra is developed as an open-source project in the neuroimaging community, but it is designed as a general-purpose dataflow engine to support any scientific domain.

Scientific workflows often require sophisticated analyses that encompass a large collection of algorithms. The algorithms, that were originally not necessarily designed to work together, and were written by different authors. Some may be written in Python, while others might require calling external programs. It is a common practice to create semi-manual workflows that require the scientists to handle the files and interact with partial results from algorithms and external tools. This approach is conceptually simple and easy to implement, but the resulting workflow is often time consuming, error-prone and difficult to share with others. Consistency, reproducibility and scalability demand scientific workflows to be organized into fully automated pipelines. This was the motivation behind Pydra - a new dataflow engine written in Python.

The Pydra package is a part of the second generation of the Nipype ecosystem — an open-source framework that provides a uniform interface to existing neuroimaging software and facilitates interaction between different software components. The Nipype project was born in the neuroimaging community, and has been helping scientists build workflows for a decade, providing a uniform interface to such neuroimaging packages as FSL, ANTs, AFNI, FreeSurfer and SPM. This flexibility has made it an ideal basis for popular preprocessing tools, such as fMRIPrep and C-PAC. The second generation of Nipype ecosystem is meant to provide additional flexibility and is being developed with reproducibility, ease of use, and scalability in mind. Pydra itself is a standalone project and is designed as a general-purpose dataflow engine to support any scientific domain.

The goal of Pydra is to provide a lightweight dataflow engine for computational graph construction, manipulation, and distributed execution, as well as ensuring reproducibility of scientific pipelines. In Pydra, a dataflow is represented as a directed acyclic graph, where each node represents a Python function, execution of an external tool, or another reusable dataflow. The combination of several key features makes Pydra a customizable and powerful dataflow engine:

- Composable dataflows: Any node of a dataflow graph can be another dataflow, allowing for nested dataflows of arbitrary depths and encouraging creating reusable dataflows.
- Flexible semantics for creating nested loops over input sets: Any Task or dataflow can be run over input parameter sets and the outputs can be recombined (similar concept to Map-Reduce model, but Pydra extends this to graphs with nested dataflows).
- A content-addressable global cache: Hash values are computed for each graph and each Task. This supports reusing of previously computed and stored dataflows and Tasks.
- Support for Python functions and external (shell) commands: Pydra can decorate and use existing functions in Python libraries alongside external command line tools, allowing easy integration of existing code and software.
- Native container execution support: Any dataflow or Task can be executed in an associated container (via Docker or Singularity) enabling greater consistency for reproducibility.
- Auditing and provenance tracking: Pydra provides a simple JSON-LD-based message passing mechanism to capture the dataflow execution activities as a provenance graph. These messages track inputs and outputs of each task in a dataflow, and the resources consumed by the task.

CHAPTER

ONE

USER GUIDE

1.1 Dataflows Components: Task and Workflow

A *Task* is the basic runnable component of *Pydra* and is described by the class `TaskBase`. A *Task* has named inputs and outputs, thus allowing construction of dataflows. It can be hashed and executes in a specific working directory. Any *Pydra*'s *Task* can be used as a function in a script, thus allowing dual use in *Pydra*'s *Workflows* and in standalone scripts. There are several classes that inherit from `TaskBase` and each has a different application:

1.1.1 Function Tasks

- `FunctionTask` is a *Task* that executes Python functions. Most Python functions declared in an existing library, package, or interactively in a terminal can be converted to a `FunctionTask` by using *Pydra*'s decorator - `mark.task`.

```
import numpy as np
from pydra import mark
fft = mark.annotate({'a': np.ndarray,
                     'return': float})(np.fft.fft)
fft_task = mark.task(fft)()
result = fft_task(a=np.random.rand(512))
```

fft_task is now a *Pydra Task* and result will contain a *Pydra*'s `Result` object. In addition, the user can use Python's function annotation or another *Pydra* decorator — `mark.annotate` in order to specify the output. In the following example, we decorate an arbitrary Python function to create named outputs:

```
@mark.task
@mark.annotate(
    {"return": {"mean": float, "std": float}}
)
def mean_dev(my_data):
    import statistics as st
    return st.mean(my_data), st.stdev(my_data)

result = mean_dev(my_data=[...])()
```

When the *Task* is executed *result.output* will contain two attributes: *mean* and *std*. Named attributes facilitate passing different outputs to different downstream nodes in a dataflow.

1.1.2 Shell Command Tasks

- `ShellCommandTask` is a *Task* used to run shell commands and executables. It can be used with a simple command without any arguments, or with specific set of arguments and flags, e.g.:

```
ShellCommandTask(executable="pwd")

ShellCommandTask(executable="ls", args="my_dir")
```

The *Task* can accommodate more complex shell commands by allowing the user to customize inputs and outputs of the commands. One can generate an input specification to specify names of inputs, positions in the command, types of the inputs, and other metadata. As a specific example, FSL's BET command (Brain Extraction Tool) can be called on the command line as:

```
bet input_file output_file -m
```

Each of the command argument can be treated as a named input to the `ShellCommandTask`, and can be included in the input specification. As shown next, even an output is specified by constructing the *out_file* field form a template:

```
bet_input_spec = SpecInfo(
    name="Input",
    fields=[
    ( "in_file", File,
      { "help_string": "input file ...",
        "position": 1,
        "mandatory": True } ),
    ( "out_file", str,
      { "help_string": "name of output ...",
        "position": 2,
        "output_file_template":
                        "{in_file}_br" } ),
    ( "mask", bool,
      { "help_string": "create binary mask",
        "argstr": "-m", } ) ],
    bases=(ShellSpec,) )

ShellCommandTask(executable="bet",
                 input_spec=bet_input_spec)
```

More details are in the *Input Specification*.

1.1.3 Container Tasks

- `ContainerTask` class is a child class of `ShellCommandTask` and serves as a parent class for `DockerTask` and `SingularityTask`. Both *Container Tasks* run shell commands or executables within containers with specific user defined environments using Docker and Singularity software respectively. This might be extremely useful for users and projects that require environment encapsulation and sharing. Using container technologies helps improve scientific workflows reproducibility, one of the key concept behind *Pydra*.

  These *Container Tasks* can be defined by using `DockerTask` and `SingularityTask` classes directly, or can be created automatically from `ShellCommandTask`, when an optional argument `container_info` is used when creating a *Shell Task*. The following two types of syntax are equivalent:

```
DockerTask(executable="pwd", image="busybox")

ShellCommandTask(executable="ls",
     container_info=("docker", "busybox"))
```

1.1.4 Workflows

- `Workflow` - is a subclass of *Task* that provides support for creating *Pydra* dataflows. As a subclass, a *Workflow* acts like a *Task* and has inputs, outputs, is hashable, and is treated as a single unit. Unlike *Tasks*, workflows embed a directed acyclic graph. Each node of the graph contains a *Task* of any type, including another *Workflow*, and can be added to the *Workflow* simply by calling the `add` method. The connections between *Tasks* are defined by using so called *Lazy Inputs* or *Lazy Outputs*. These are special attributes that allow assignment of values when a *Workflow* is executed rather than at the point of assignment. The following example creates a *Workflow* from two *Pydra Tasks*.

```
# creating workflow with two input fields
wf = Workflow(input_spec=["x", "y"])
# adding a task and connecting task's input
# to the workflow input
wf.add(mult(name="mlt",
               x=wf.lzin.x, y=wf.lzin.y))
# adding another task and connecting
# task's input to the "mult" task's output
wf.add(add2(name="add", x=wf.mlt.lzout.out))
# setting workflow output
wf.set_output([("out", wf.add.lzout.out)])
```

1.1.5 Task's State

All Tasks, including Workflows, can have an optional attribute representing an instance of the State class. This attribute controls the execution of a Task over different input parameter sets. This class is at the heart of Pydra's powerful Map-Reduce over arbitrary inputs of nested dataflows feature. The State class formalizes how users can specify arbitrary combinations. Its functionality is used to create and track different combinations of input parameters, and optionally allow limited or complete recombinations. In order to specify how the inputs should be split into parameter sets, and optionally combined after the Task execution, the user can set splitter and combiner attributes of the State class.

```
task_with_state =
      add2(x=[1, 5]).split("x").combine("x")
```

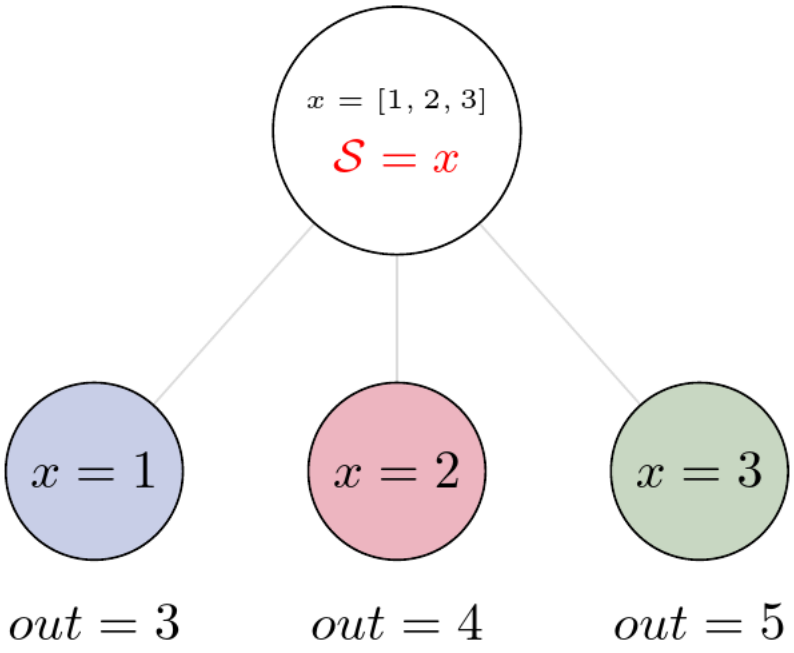
In this example, the `State` class is responsible for creating a list of two separate inputs, *[{x: 1}, {x:5}]*, each run of the *Task* should get one element from the list. The results are grouped back when returning the result from the *Task*. While this example illustrates mapping and grouping of results over a single parameter, *Pydra* extends this to arbitrary combinations of input fields and downstream grouping over nested dataflows. Details of how splitters and combiners power *Pydra*'s scalable dataflows are described in the next section.

1.2 State and Nested Loops over Input

One of the main goals of creating Pydra was to support flexible evaluation of a Task or a Workflow over combinations of input parameters. This is the key feature that distinguishes it from most other dataflow engines. This is similar to the concept of the Map-Reduce, but extends it to work over arbitrary nested graphs. In complex dataflows, this would typically involve significant overhead for data management and use of multiple nested loops. In Pydra, this is controlled by setting specific State related attributes through Task methods. In order to set input splitting (or mapping), Pydra requires setting up a splitter. This is done using Task's split method. The simplest example would be a Task that has one field x in the input, and therefore there is only one way of splitting its input. Assuming that the user provides a list as a value of x, Pydra splits the list, so each copy of the Task will get one element of the list. This can be represented as follow:

$$S = x : x = [x_1, x_2, ..., x_n] \longmapsto x = x_1, x = x_2, ..., x = x_n \ ,$$

where S represents the splitter, and x is the input field. This is also represented in the diagram, where $x = [1, 2, 3]$ as an example, and the coloured nodes represent stateless copies of the original Task after splitting the input, (these are the runnables that are executed).



1.2.1 Types of Splitter

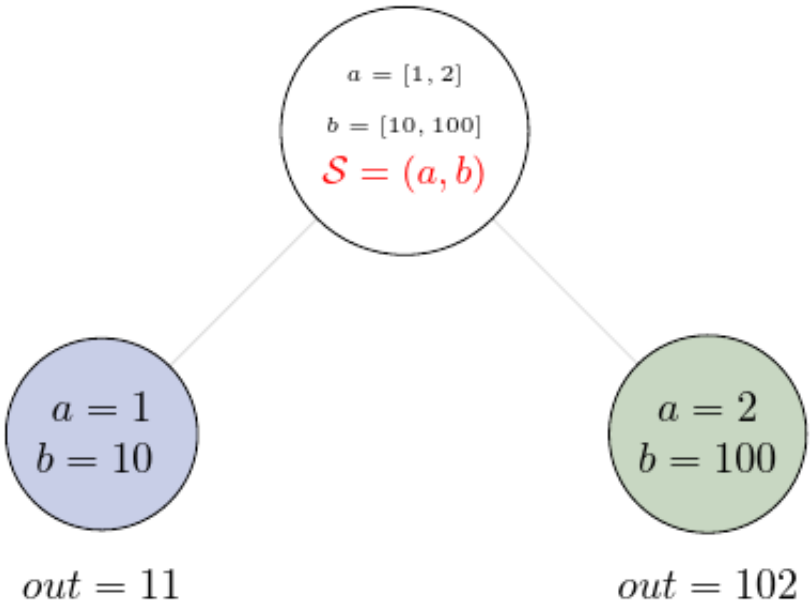
Whenever a *Task* has more complicated inputs, i.e. multiple fields, there are two ways of creating the mapping, each one is used for different application. These *splitters* are called *scalar splitter* and *outer splitter*. They use a special, but Python-based syntax as described next.

1.2.2 Scalar Splitter

A *scalar splitter* performs element-wise mapping and requires that the lists of values for two or more fields to have the same length. The *scalar splitter* uses Python tuples and its operation is therefore represented by a parenthesis, `()`:

$$S = (x, y) : x = [x_1, x_2, .., x_n],\ y = [y_1, y_2, .., y_n] \mapsto (x, y) = (x_1, y_1), ..., (x, y) = (x_n, y_n),$$

where *S* represents the *splitter*, *x* and *y* are the input fields. This is also represented as a diagram:



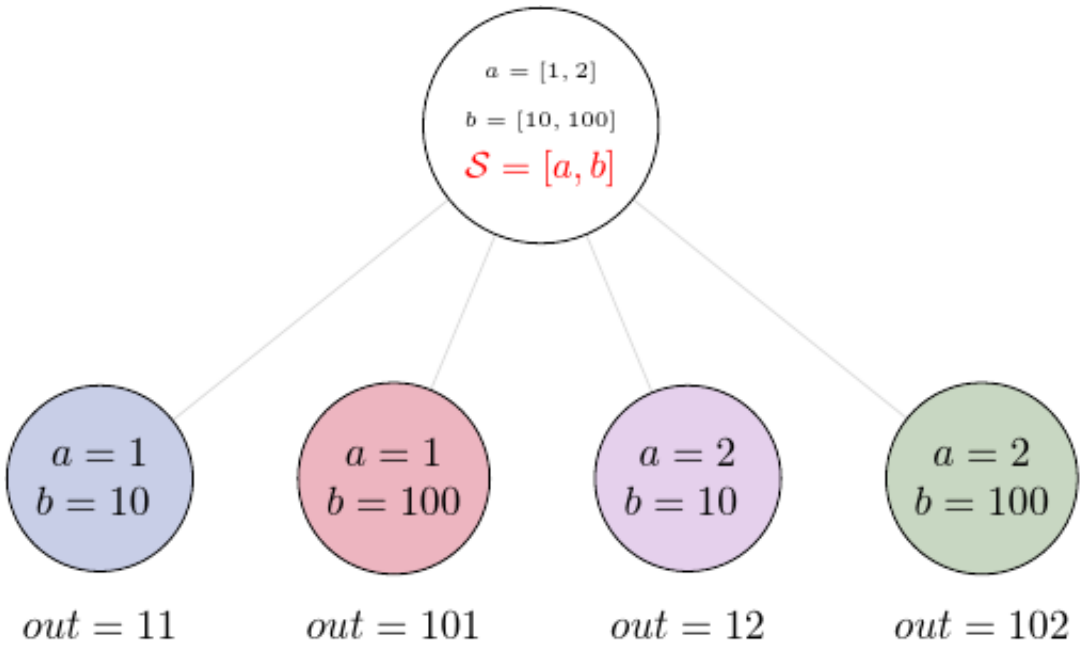
1.2.3 Outer Splitter

The second option of mapping the input, when there are multiple fields, is provided by the *outer splitter*. The *outer splitter* creates all combination of the input values and does not require the lists to have the same lengths. The *outer splitter* uses Python's list syntax and is represented by square brackets, `[]`:

$$S = [x, y] : x = [x_1, x_2, ..., x_n],\ \ y = [y_1, y_2, ..., y_m], \mapsto (x, y) = (x_1, y_1), (x, y) = (x_1, y_2)..., (x, y) = (x_n, y_m).$$

The *outer splitter* for a node with two input fields is schematically represented in the diagram:

Different types of splitters can be combined over inputs such as *[inp1, (inp2, inp3)]*. In this example an *outer splitter* provides all combinations of values of *inp1* with pairwise combinations of values of *inp2* and *inp3*. This can be extended to arbitrary complexity. In additional, the output can be merge at the end if needed. This will be explained in the next section.



1.3 Grouping Task's Output

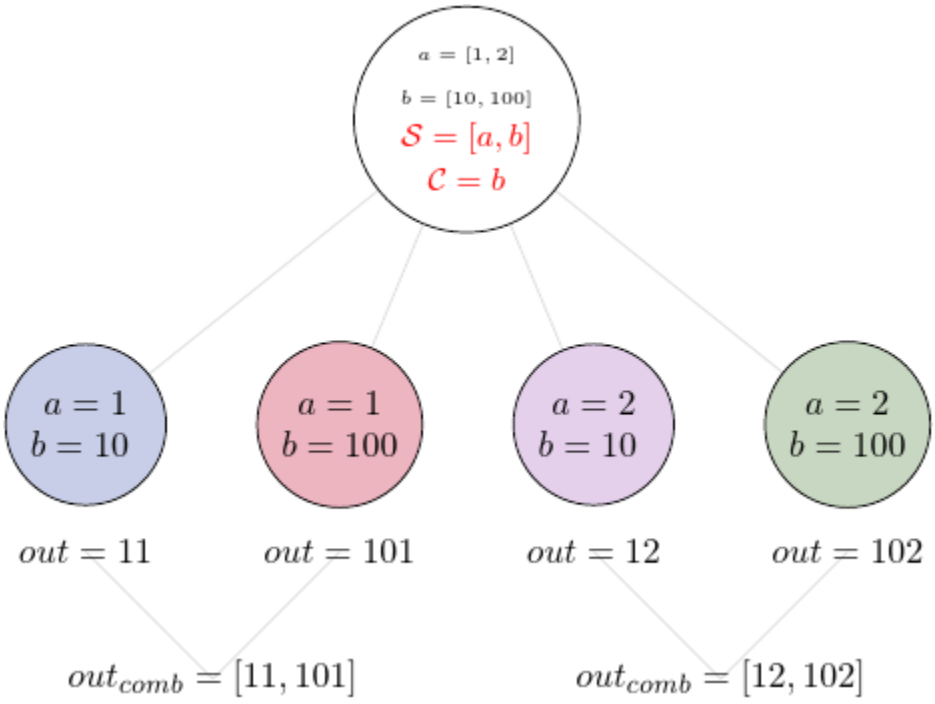
In addition to the splitting the input, *Pydra* supports grouping or combining the output resulting from the splits. In order to achieve this for a *Task*, a user can specify a *combiner*. This can be set by calling `combine` method. Note, the *combiner* only makes sense when a *splitter* is set first. When *combiner*=*x*, all values are combined together within one list, and each element of the list represents an output of the *Task* for the specific value of the input x. Splitting and combining for this example can be written as follows:

$$S = x :$$
$$x = [x_1, x_2, ..., x_n] \mapsto x = x_1, x = x_2, ..., x = x_n,$$
$$C = x :$$
$$out(x_1), ..., out(x_n) \mapsto out_{comb} = [out(x_1), ...out(x_n)],$$

where *S* represents the *splitter*, *C* represents the *combiner*, x is the input field, $out(x_i)$ represents the output of the *Task* for x_i, and out_{comb} is the final output after applying the *combiner*.

In the situation where input has multiple fields and an *outer splitter* is used, there are various ways of combining the output. Taking as an example the task from the previous section, user might want to combine all the outputs for one specific value of x_i and all the values of y. In this situation, the combined output would be a two dimensional list, each inner list for each value of x. This can be written as follow:

$$C = y :$$
$$out(x_1, y1), out(x_1, y2), ...out(x_n, y_m) \longmapsto$$
$$[[out(x_1, y_1), ..., out(x_1, y_m)],$$
$$...,$$
$$[out(x_n, y_1), ..., out(x_n, y_m)]].$$



However, for the same task the user might want to combine all values of x for specific values of y. One may also need to combine all the values together. This can be achieved by providing a list of fields, $[x, y]$ to the combiner. When a full combiner is set, i.e. all the fields from the splitter are also in the combiner, the output is a one dimensional list:

$$C = [x, y] : out(x_1, y1), ...out(x_n, y_m) \longmapsto [out(x_1, y_1), ..., out(x_n, y_m)].$$

These are the basic examples of the *Pydra*'s *splitter-combiner* concept. It is important to note, that *Pydra* allows for mixing *splitters* and *combiners* on various levels of a dataflow. They can be set on a single *Task* or a *Workflow*. They can be passed from one *Task* to following *Tasks* within the *Workflow*.

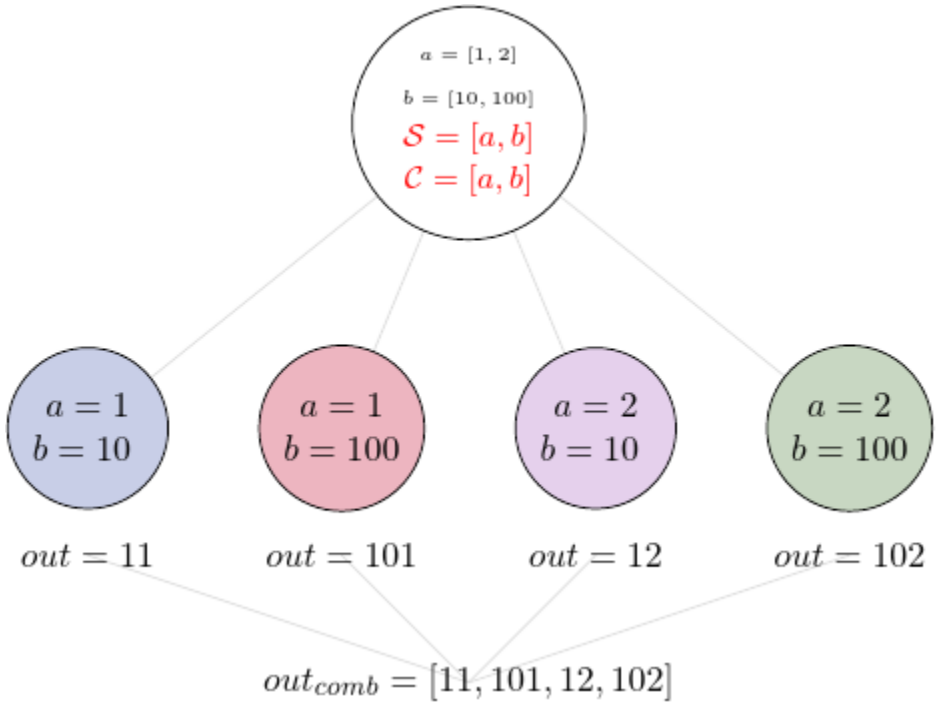
1.4 Input Specification

As it was mentioned in *Shell Command Tasks*, the user can customize the input and output for the *ShellCommandTask*. In this section, more examples of the input specification will be provided.

Let's start from the previous example:

```
bet_input_spec = SpecInfo(
    name="Input",
    fields=[
    ( "in_file", File,
      { "help_string": "input file ...",
        "position": 1,
        "mandatory": True } ),
    ( "out_file", str,
      { "help_string": "name of output ...",
        "position": 2,
```

(continues on next page)



(continued from previous page)

```
        "output_file_template":
                            "{in_file}_br" } ),
    ( "mask", bool,
      { "help_string": "create binary mask",
        "argstr": "-m", } ) ],
    bases=(ShellSpec,) )

ShellCommandTask(executable="bet",
                 input_spec=bet_input_spec)
```

In order to create an input specification, a new *SpecInfo* object has to be created. The field *name* specifies the type of the spec and it should be always "Input" for the input specification. The field *bases* specifies the "base specification" you want to use (can think about it as a *parent class*) and it will usually contains *ShellSpec* only, unless you want to build on top of your other specification (this will not be cover in this section). The part that should be always customised is the *fields* part. Each element of the *fields* is a separate input field that is added to the specification. In this example, three-elements tuples - with name, type and dictionary with additional information - are used. But this is only one of the supported syntax, more options will be described below.

1.4.1 Adding a New Field to the Spec

Pydra uses *attr* classes to represent the input specification, and the full syntax for each field is:

```
field1 = ("field1_name", attr.ib(type=<'field1_type'>, metadata=<'dictionary with␣
↪metadata'>)
```

However, we allow for shorter syntax, that does not include *attr.ib*:

- providing only name and the type

```
field1 = ("field1_name", <'field1_type'>)
```

- providing name, type and metadata (as in the example above)

```
field1 = ("field1_name", <'field1_type'>, <'dictionary with metadata'>))
```

- providing name, type and default value

```
field1 = ("field1_name", <'field1_type'>, <'default value'>)
```

- providing name, type, default value and metadata

```
field1 = ("field1_name", <'field1_type'>, <'default value', <'dictionary with metadata'>
↪))
```

Each of the shorter versions will be converted to the *(name, attr.ib(...)*.

1.4.2 Types

Type can be provided as a simple python type (e.g. *str*, *int*, *float*, etc.) or can be more complex by using *typing.List*, *typing.Dict* and *typing.Union*.

There are also special types provided by Pydra:

- *File* and *Directory* - should be used in *input_spec* if the field is an existing file or directory. Pydra checks if the file or directory exists, and returns an error if it doesn't exist.
- *MultiInputObj* - a special type that takes a any value and if the value is not a list it converts value to a 1-element list (it could be used together with *MultiOutputObj* in the *output_spec* to reverse the conversion of the output values).

1.4.3 Metadata

In the example we used multiple keys in the metadata dictionary including *help_string*, *position*, etc. In this section all allowed key will be described:

***help_string* (*str*, mandatory):**
A short description of the input field.

***mandatory* (*bool*, default: *False*):**
If *True* user has to provide a value for the field.

***sep* (*str*):**
A separator if a list is provided as a value.

***argstr* (*str*):**
A flag or string that is used in the command before the value, e.g. *-v* or *-v {inp_field}*, but it could be and empty string, *""*. If ... are used, e.g. *-v...*, the flag is used before every element if a list is provided as a value. If no *argstr* is used the field is not part of the command.

***position* (*int*):**
Position of the field in the command, could be nonnegative or negative integer. If nothing is provided the field will be inserted between all fields with nonnegative positions and fields with negative positions.

***allowed_values* (*list*):**
List of allowed values for the field.

***requires* (*list*):**
List of field names that are required together with the field.

***xor* (*list*):**
List of field names that are mutually exclusive with the field.

***copyfile* (*bool*, default: *False*):**
If *True*, a hard link is created for the input file in the output directory. If hard link not possible, the file is copied to the output directory.

***container_path* (*bool*, default: *False*, only for *ContainerTask*):**
If *True* a path will be consider as a path inside the container (and not as a local path).

***output_file_template* (*str*):**
If provided, the field is treated also as an output field and it is added to the output spec. The template can use other fields, e.g. *{file1}*. Used in order to create an output specification.

***output_field_name* (*str*, used together with *output_file_template*)**
If provided the field is added to the output spec with changed name. Used in order to create an output specification.

***keep_extension* (*bool*, default: *True*):**
A flag that specifies if the file extension should be removed from the field value. Used in order to create an output specification.

***readonly* (*bool*, default: *False*):**
If *True* the input field can't be provided by the user but it aggregates other input fields (for example the fields with *argstr: -o {fldA} {fldB}*).

***formatter* (*function*):**
If provided the *argstr* of the field is created using the function. This function can for example be used to combine several inputs into one command argument. The function can take *field* (this input field will be passed to the function), *inputs* (entire *inputs* will be passed) or any input field name (a specific input field will be sent).

1.4.4 Validators

Pydra allows for using simple validator for types and *allowev_values*. The validators are disabled by default, but can be enabled by calling *pydra.set_input_validator(flag=True)*.

1.5 Output Specification

As it was mentioned in *Shell Command Tasks*, the user can customize the input and output for the *ShellCommandTask*. In this section, the output specification will be covered.

Instead of using field with *output_file_template* in the customized *input_spec* to specify an output field, a customized *output_spec* can be used, e.g.:

```
output_spec = SpecInfo(
    name="Output",
    fields=[
        (
            "out1",
            attr.ib(
                type=File,
                metadata={
                    "output_file_template": "{inp1}",
```

(continues on next page)

(continued from previous page)

```
                    "help_string": "output file",
                    "requires": ["inp1", "inp2"]
                },
            ),
        )
    ],
    bases=(ShellOutSpec,),
)

ShellCommandTask(executable=executable,
                 output_spec=output_spec)
```

Similarly as for *input_spec*, in order to create an output specification, a new *SpecInfo* object has to be created. The field *name* specifies the type of the spec and it should be always "Output" for the output specification. The field *bases* specifies the "base specification" you want to use (can think about it as a *parent class*) and it will usually contains *ShellOutSpec* only, unless you want to build on top of your other specification (this will not be cover in this section). The part that should be always customised is the *fields* part. Each element of the *fields* is a separate output field that is added to the specification. In this example, a three-elements tuple - with name, type and dictionary with additional information - is used. See *Input Specification* for other recognized syntax for specification's fields and possible types.

1.5.1 Metadata

The metadata dictionary for *output_spec* can include:

***help_string* (*str*, mandatory):**
A short description of the input field. The same as in *input_spec*.

***mandatory* (*bool*, default: *False*):**
If *True* the output file has to exist, otherwise an error will be raised.

***output_file_template* (*str*):**
If provided the output file name (or list of file names) is created using the template. The template can use other fields, e.g. *{file1}*. The same as in *input_spec*.

***output_field_name* (*str*, used together with *output_file_template*)**
If provided the field is added to the output spec with changed name. The same as in *input_spec*.

***keep_extension* (*bool*, default: *True*):**
A flag that specifies if the file extension should be removed from the field value. The same as in *input_spec*.

***requires* (*list*):**
List of field names that are required to create a specific output. The fields do not have to be a part of the *output_file_template* and if any field from the list is not provided in the input, a *NOTHING* is returned for the specific output. This has a different meaning than the *requires* form the *input_spec*.

***callable* (*function*):**
If provided the output file name (or list of file names) is created using the function. The function can take *field* (the specific output field will be passed to the function), *output_dir* (task *output_dir* will be used), *stdout*, *stderr* (*stdout* and *stderr* of the task will be sent) *inputs* (entire *inputs* will be passed) or any input field name (a specific input field will be sent).

CHAPTER

TWO

RELEASE NOTES

2.1 0.8.0

- refactoring template formatting for `input_spec`
- fixing issues with input fields with extension (and using them in templates)
- adding simple validators to input spec (using `attr.validator`)
- adding `create_dotfile` for workflows, that creates graphs as dotfiles (can convert to other formats if dot available)
- adding a simple user guide with `input_spec` description
- expanding docstrings for `State`, `audit` and `messanger`
- updating syntax to newer python

2.2 0.7.0

- refactoring the error handling by padra: improving raised errors, removing nodes from the workflow graph that can't be run
- refactoring of the `input_spec`: adapting better to the nipype interfaces
- switching from `pkg_resources.declare_namespace` to the stdlib `pkgutil.extend_path`
- moving `readme` to rst format

2.3 0.6.2

- Use pkgutil to declare `pydra.tasks` as a namespace package, ensuring better support for editable mode.

2.4 0.6.1

- Add `pydra.tasks` namespace package to enable separate packages of `Task`s to be installed into `pydra.tasks`.
- Raise error when task or workflow name conflicts with names of attributes, methods, or other tasks already added to workflow
- Mention `requirements.txt` in README

2.5 0.6

- removing the tutorial to a separate repo
- adding windows tests to codecov
- accepting `None` as a valid output from a `FunctionTask`, also for function that returns multiple values
- fixing slurm error files
- adding `wf._connection` to `checksum`
- allowing for updates of `wf._connections`
- editing output, so it works with `numpy.arrays`
- removing `to_job` and pickling task instead (workers read the tasks and set the proper input, so the multiple copies of the input are not kept in the memory)
- adding standalone function `load_and_run` that can load and run a task from a pickle file
- removing `create_pyscript` and simplifying the slurm worker
- improving error reports in errors flies
- fixing `make_class` so the `Output` is properly formatted

2.6 0.5

- fixing `hash_dir` function
- adding `get_available_cpus` to get the number of CPUs available to the current process or available on the system
- adding simple implementation for `BoshTask` that uses boutiques descriptor
- adding azure to CI
- fixing code for windows
- etelementry updates
- adding more verbose output for task `result` - returns values or indices for input fields
- adding an experimental implementation of Dask Worker (limited testing with ci)

2.7 0.4

- reorganization of the `State` class, fixing small issues with the class
- fixing some paths issues on windows os
- adding osx and window sto the travis runs (right now allowing for failures for windows)
- adding `PydraStateError` for exception in the `State` class
- small fixes to the hashing functions, adding more tests
- adding `hash_dir` to calculate hash for `Directory` type

2.8 0.3.1

- passing `wf.cache_locations` to the task
- using `rerun` from submitter to all task
- adding `test_rerun` and `propagate_rerun` for workflows
- fixing task with a full combiner
- adding `cont_dim` to specify dimensionality of the input variables (how much the input is nested)

2.9 0.3

- adding sphinx documentation
- moving from `dataclasses` to `attrs`
- adding `container` flag to the `ShellCommandTask`
- fixing `cmdline`, `command_args` and `container_args` for tasks with states
- adding `CONTRIBUTING.md`
- fixing hash calculations for inputs with a list of files
- using `attr.NOTHING` for input that is not set

2.10 0.2.2

- supporting tuple as a single element of an input

2.11 0.2.1

- fixing: nodes with states and input fields (from splitter) that are empty were failing

2.12 0.2

- **big changes in `ShellTask`, `DockerTask` and `SingularityTask`**
  - customized input specification and output specification for `Task`s
  - adding singularity checks to Travis CI
  - binding all input files to the container
- **changes in `Workflow`**
  - passing all outputs to the next node: `lzout.all_`
  - fixing inner splitter
- allowing for `splitter` and `combiner` updates
- adding `etelementry` support

2.13 0.1

- Core dataflow creation and management API
- **Distributed workers:**
  - concurrent futures
  - SLURM
- Notebooks for Pydra concepts

2.14 0.0.1

Initial Pydra Dataflow Engine release.

CHAPTER

THREE

LIBRARY API (APPLICATION PROGRAMMER INTERFACE)

The Pydra workflow engine.

Pydra is a rewrite of the Nipype engine with mapping and joining as first-class operations. It forms the core of the Nipype 2.0 ecosystem.

pydra.**check_latest_version**()

pydra.**set_input_validator**(*flag=False*)

3.1 Subpackages

3.1.1 pydra.engine package

The core of the workflow engine.

class pydra.engine.**AuditFlag**(*value*)

Bases: Flag

Auditing flags.

ALL = 3

Track provenance and resource utilization.

NONE = 0

Do not track provenance or monitor resources.

PROV = 1

Track provenance only.

RESOURCE = 2

Monitor resource utilization only.

class pydra.engine.**DockerTask**(*container_info=None*, **args*, ***kwargs*)

Bases: *ContainerTask*

Extend shell command task for containerized execution with the Docker Engine.

property container_args

Get container-specific CLI arguments, returns a list if the task has a state

init = False

class pydra.engine.**ShellCommandTask**(*container_info=None*, **args*, ***kwargs*)

Bases: *TaskBase*

Wrap a shell command as a task element.

property cmdline

Get the actual command line that will be submitted Returns a list if the task has a state.

property command_args

Get command line arguments

input_spec = None

output_spec = None

class pydra.engine.**Submitter**(*plugin='cf'*, ***kwargs*)

Bases: object

Send a task to the execution backend.

close()

Close submitter.

Do not close previously running loop.

async expand_runnable(*runnable*, *wait=False*, *rerun=False*)

This coroutine handles state expansion.

Removes any states from *runnable*. If *wait* is set to False (default), aggregates all worker execution coroutines and returns them. If *wait* is True, waits for all coroutines to complete / error and returns None.

Parameters

- **runnable** (*pydra Task*) – Task instance (*Task*, *Workflow*)
- **wait** (*bool (False)*) – Await all futures before completing

Returns
futures – Coroutines for *TaskBase* execution.

Return type
set or None

async expand_workflow(*wf*, *rerun=False*)

Expand and execute a stateless *Workflow*. This method is only reached by *Workflow._run_task*.

Parameters
wf (*Workflow*) – Workflow Task object

Returns
wf – The computed workflow

Return type
pydra.engine.core.Workflow

async submit_from_call(*runnable*, *rerun*)

This coroutine should only be called once per Submitter call, and serves as the bridge between sync/async lands.

There are 4 potential paths based on the type of runnable: 0) Workflow has a different plugin than a submitter 1) Workflow without State 2) Task without State 3) (Workflow or Task) with State

Once Python 3.10 is the minimum, this should probably be refactored into using structural pattern matching.

class pydra.engine.**Workflow**(*name*, *audit_flags: AuditFlag = AuditFlag.NONE*, *cache_dir=None*, *cache_locations=None*, *input_spec: Optional[Union[List[str], SpecInfo, BaseSpec]] = None*, *cont_dim=None*, *messenger_args=None*, *messengers=None*, *output_spec: Optional[Union[SpecInfo, BaseSpec]] = None*, *rerun=False*, *propagate_rerun=True*, ***kwargs*)

Bases: *TaskBase*

A composite task with structure of computational graph.

add(*task*)

Add a task to the workflow.

Parameters
task (TaskBase) – The task to be added.

property checksum

Calculates the unique checksum of the task. Used to create specific directory name for task that are run; and to create nodes checksums needed for graph checksums (before the tasks have inputs etc.)

create_connections(*task*, *detailed=False*)

Add and connect a particular task to existing nodes in the workflow.

Parameters

- **task** (TaskBase) – The task to be added.
- **detailed** (bool) – If True, *add_edges_description* is run for self.graph to add a detailed descriptions of the connections (input/output fields names)

create_dotfile(*type='simple'*, *export=None*, *name=None*)

creating a graph - dotfile and optionally exporting to other formats

property graph_sorted

Get a sorted graph representation of the workflow.

property nodes

Get the list of node names.

set_output(*connections*)

Write outputs.

Parameters
connections – TODO

Submodules

pydra.engine.audit module

Module to keep track of provenance information.

class pydra.engine.audit.**Audit**(*audit_flags*, *messengers*, *messenger_args*, *develop=None*)

Bases: object

Handle provenance tracking and resource utilization.

audit_check(*flag*)

Determine whether auditing is enabled for a particular flag.

Parameters
flag (:obj: *bool*) – The flag that is checked.

Returns
Boolean AND for self.oudit_flags and flag

Return type
bool

audit_message(*message*, *flags=None*)

Send auditing message.

Parameters

- **message** (dict) – A message in Pydra is a JSON-LD message object.
- **flags** (bool, optional) – If True and self.audit_flag, the message is sent.

finalize_audit(*result*)

End auditing.

monitor()

Start resource monitoring.

start_audit(*odir*)

Start recording provenance.

Monitored information is not sent until directory is created, in case message directory is inside task output directory.

Parameters
odir (os.pathlike) – Message output directory.

pydra.engine.boutiques module

class pydra.engine.boutiques.**BoshTask**(*container_info=None*, **args*, ***kwargs*)

Bases: *ShellCommandTask*

Shell Command Task based on the Boutiques descriptor

pydra.engine.core module

Basic processing graph elements.

class pydra.engine.core.**TaskBase**(*name: str*, *audit_flags:* AuditFlag *= AuditFlag.NONE*, *cache_dir=None*, *cache_locations=None*, *inputs: Optional[Union[str,* File*, Dict]] = None*, *cont_dim=None*, *messenger_args=None*, *messengers=None*, *rerun=False*)

Bases: object

A base structure for the nodes in the processing graph.

Tasks are a generic compute step from which both elemntary tasks and *Workflow* instances inherit.

audit_flags: *AuditFlag* **= 0**

AuditFlag.

Type
What to audit – available flags

property cache_dir

Get the location of the cache directory.

property cache_locations

Get the list of cache sources.

property can_resume

Whether the task accepts checkpoint-restart.

property checksum

Calculates the unique checksum of the task. Used to create specific directory name for task that are run; and to create nodes checksums needed for graph checksums (before the tasks have inputs etc.)

checksum_states(*state_index=None*)

Calculate a checksum for the specific state or all of the states of the task. Replaces lists in the inputs fields with a specific values for states. Used to recreate names of the task directories,

Parameters
state_index – TODO

combine(*combiner*, *overwrite=False*)

Combine inputs parameterized by one or more previous tasks.

Parameters

- **combiner** – TODO
- **overwrite** (bool) – TODO

property cont_dim

property done

Check whether the tasks has been finalized and all outputs are stored.

property errored

Check if the task has raised an error

property generated_output_names

Get the names of the outputs generated by the task. If the spec doesn't have generated_output_names method, it uses output_names. The results depends on the input provided to the task

get_input_el(*ind*)

Collect all inputs required to run the node (for specific state element).

help(*returnhelp=False*)

Print class help.

property output_dir

Get the filesystem path where outputs will be written.

property output_names

Get the names of the outputs from the task's output_spec (not everything has to be generated, see generated_output_names).

pickle_task()

Pickling the tasks with full inputs

result(*state_index=None*, *return_inputs=False*)

Retrieve the outcomes of this particular task.

Parameters

- **state_index** (:obj: *int*) – index of the element for task with splitter and multiple states
- **return_inputs** (:obj: *bool*, `str`) – if True or "val" result is returned together with values of the input fields, if "ind" result is returned together with indices of the input fields

Returns

Return type
result

set_state(*splitter*, *combiner=None*)

Set a particular state on this task.

Parameters

- **splitter** – TODO
- **combiner** – TODO

split(*splitter*, *overwrite=False*, *cont_dim=None*, ***kwargs*)

Run this task parametrically over lists of split inputs.

Parameters

- **splitter** – TODO
- **overwrite** (`bool`) – TODO
- **cont_dim** (`dict`) – Container dimensions for specific inputs, used in the splitter. If input name is not in cont_dim, it is assumed that the input values has a container dimension of 1, so only the most outer dim will be used for splitting.

property uid

the unique id number for the task It will be used to create unique names for slurm scripts etc. without a need to run checksum

property version

Get version of this task structure.

class pydra.engine.core.**Workflow**(*name*, *audit_flags: AuditFlag = AuditFlag.NONE*, *cache_dir=None*, *cache_locations=None*, *input_spec: Optional[Union[List[str], SpecInfo, BaseSpec]] = None*, *cont_dim=None*, *messenger_args=None*, *messengers=None*, *output_spec: Optional[Union[SpecInfo, BaseSpec]] = None*, *rerun=False*, *propagate_rerun=True*, ***kwargs*)

Bases: *TaskBase*

A composite task with structure of computational graph.

add(*task*)

Add a task to the workflow.

Parameters
task (*TaskBase*) – The task to be added.

property checksum

Calculates the unique checksum of the task. Used to create specific directory name for task that are run; and to create nodes checksums needed for graph checksums (before the tasks have inputs etc.)

create_connections(*task*, *detailed=False*)

Add and connect a particular task to existing nodes in the workflow.

Parameters

- **task** (*TaskBase*) – The task to be added.
- **detailed** (bool) – If True, *add_edges_description* is run for self.graph to add a detailed descriptions of the connections (input/output fields names)

create_dotfile(*type='simple'*, *export=None*, *name=None*)

creating a graph - dotfile and optionally exporting to other formats

property graph_sorted

Get a sorted graph representation of the workflow.

property nodes

Get the list of node names.

set_output(*connections*)

Write outputs.

Parameters
connections – TODO

pydra.engine.core.**is_lazy**(*obj*)

Check whether an object has any field that is a Lazy Field

pydra.engine.core.**is_task**(*obj*)

Check whether an object looks like a task.

pydra.engine.core.**is_workflow**(*obj*)

Check whether an object is a *Workflow* instance.

pydra.engine.graph module

Data structure to support *Workflow* tasks.

class pydra.engine.graph.**DiGraph**(*name=None*, *nodes=None*, *edges=None*)

Bases: object

A simple Directed Graph object.

add_edges(*new_edges*)

Add new edges and sort the new graph.

add_edges_description(*new_edge_details*)

adding detailed description of the connections, filling _nodes_details

add_nodes(*new_nodes*)

Insert new nodes and sort the new graph.

calculate_max_paths()

Calculate maximum paths.

Maximum paths are calculated between any node without "history" (no predecessors) and all of the connections.

`copy`()

Duplicate this graph.

Create a copy that contains new lists and dictionaries, but runnable objects are the same.

`create_dotfile_detailed`(*outdir*, *name='graph_det'*)

creates a detailed dotfile (detailed connections - input/output fields, but no nested structure)

`create_dotfile_nested`(*outdir*, *name='graph'*)

dotfile that includes the nested structures for workflows

`create_dotfile_simple`(*outdir*, *name='graph'*)

creates a simple dotfile (no nested structure)

`property edges`

Get a list of the links between nodes.

`property edges_names`

Get edges as pairs of the nodes they connect.

`export_graph`(*dotfile*, *ext='png'*)

exporting dotfile to other format, equires the dot command

`property nodes`

Get a list of the nodes currently contained in the graph.

`property nodes_details`

dictionary with details of the nodes for each task, there are inputs/outputs and connections (with input/output fields names)

`property nodes_names_map`

Get a map of node names to nodes.

`remove_nodes`(*nodes*, *check_ready=True*)

Mark nodes for removal from the graph, re-sorting if needed.

Important: This method does not remove connections, see `remove_node_connections()`. Nodes are added to the `_node_wip` list, marking them for removal when all referring connections are removed.

Parameters

- **nodes** (`list`) – List of nodes to be marked for removal.
- **check_ready** (:obj: *bool*) – checking if the node is ready to be removed

`remove_nodes_connections`(*nodes*)

Remove connections between nodes.

Also prunes the nodes from `_node_wip`.

Parameters

nodes (`list`) – List of nodes which connections are to be removed.

`remove_previous_connections`(*nodes*)

Remove connections that the node has with predecessors.

Also prunes the nodes from `_node_wip`.

Parameters
nodes (list) – List of nodes which connections are to be removed.

`remove_successors_nodes`(*node*)

Removing all the nodes that follow the node

`property sorted_nodes`

Return sorted nodes (runs sorting if needed).

`property sorted_nodes_names`

Return a list of sorted nodes names.

`sorting`(*presorted=None*)

Sort this graph.

Sorting starts either from self.nodes or the previously sorted list.

Parameters
presorted (list) – A list of previously sorted nodes.

pydra.engine.helpers module

Administrative support for the engine framework.

`class` pydra.engine.helpers.`PydraFileLock`(*lockfile*)

Bases: object

Wrapper for filelock's SoftFileLock that makes it work with asyncio.

pydra.engine.helpers.`argstr_formatting`(*argstr*, *inputs*, *value_updates=None*)

formatting argstr that have form {field_name}, using values from inputs and updating with value_update if provided

pydra.engine.helpers.`copyfile_workflow`(*wf_path*, *result*)

if file in the wf results, the file will be copied to the workflow directory

pydra.engine.helpers.`create_checksum`(*name*, *inputs*)

Generate a checksum name for a given combination of task name and inputs.

Parameters

- **name** (str) – Task name.
- **inputs** (str) – String of inputs.

pydra.engine.helpers.`custom_validator`(*instance*, *attribute*, *value*)

simple custom validation take into account ty.Union, ty.List, ty.Dict (but only one level depth) adding an additional validator, if allowe_values provided

pydra.engine.helpers.`ensure_list`(*obj*, *tuple2list=False*)

Return a list whatever the input object is.

Examples

```
>>> ensure_list(list("abc"))
['a', 'b', 'c']
>>> ensure_list("abc")
['abc']
>>> ensure_list(tuple("abc"))
[('a', 'b', 'c')]
>>> ensure_list(tuple("abc"), tuple2list=True)
['a', 'b', 'c']
>>> ensure_list(None)
[]
>>> ensure_list(5.0)
[5.0]
```

pydra.engine.helpers.**execute**(*cmd*, *strip=False*)

Run the event loop with coroutine.

Uses *read_and_display_async()* unless a loop is already running, in which case *read_and_display()* is used.

Parameters

- **cmd** (list or tuple) – The command line to be executed.
- **strip** (bool) – TODO

pydra.engine.helpers.**gather_runtime_info**(*fname*)

Extract runtime information from a file.

Parameters
fname (os.pathlike) – The file containing runtime information

Returns
runtime – A runtime object containing the collected information.

Return type
Runtime

pydra.engine.helpers.**get_available_cpus**()

Return the number of CPUs available to the current process or, if that is not available, the total number of CPUs on the system.

Returns
n_proc – The number of available CPUs.

Return type
int

pydra.engine.helpers.**get_open_loop**()

Get current event loop.

If the loop is closed, a new loop is created and set as the current event loop.

Returns
loop – The current event loop

Return type
asyncio.EventLoop

pydra.engine.helpers.**hash_function**(*obj*)

Generate hash of object.

pydra.engine.helpers.**hash_value**(*value*, *tp=None*, *metadata=None*, *precalculated=None*)

calculating hash or returning values recursively

pydra.engine.helpers.**load_and_run**(*task_pkl*, *ind=None*, *rerun=False*, *submitter=None*, *plugin=None*, ***kwargs*)

loading a task from a pickle file, settings proper input and running the task

async pydra.engine.helpers.**load_and_run_async**(*task_pkl*, *ind=None*, *submitter=None*, *rerun=False*, ***kwargs*)

loading a task from a pickle file, settings proper input and running the workflow

pydra.engine.helpers.**load_result**(*checksum*, *cache_locations*)

Restore a result from the cache.

Parameters

- **checksum** (`str`) – Unique identifier of the task to be loaded.
- **cache_locations** (`list` of `os.pathlike`) – List of cache directories, in order of priority, where the checksum will be looked for.

pydra.engine.helpers.**load_task**(*task_pkl*, *ind=None*)

loading a task from a pickle file, settings proper input for the specific ind

pydra.engine.helpers.**make_klass**(*spec*)

Create a data class given a spec.

Parameters

spec – TODO

pydra.engine.helpers.**output_from_inputfields**(*output_spec*, *input_spec*)

Collect values from output from input fields. If names_only is False, the output_spec is updated, if names_only is True only the names are returned

Parameters

- **output_spec** – TODO
- **input_spec** – TODO

pydra.engine.helpers.**position_sort**(*args*)

Sort objects by position, following Python indexing conventions.

Ordering is positive positions, lowest to highest, followed by unspecified positions (`None`) and negative positions, lowest to highest.

```
>>> position_sort([(None, "d"), (-3, "e"), (2, "b"), (-2, "f"), (5, "c"), (1, "a")])
['a', 'b', 'c', 'd', 'e', 'f']
```

Parameters

args (*list of (int/None, object) tuples*)

Returns

Return type

list of objects

pydra.engine.helpers.**print_help**(*obj*)

Visit a task object and print its input/output interface.

pydra.engine.helpers.**read_and_display**(**cmd*, *strip=False*, *hide_display=False*)

Capture a process' standard output.

async pydra.engine.helpers.**read_and_display_async**(**cmd*, *hide_display=False*, *strip=False*)

Capture standard input and output of a process, displaying them as they arrive.

Works line-by-line.

async pydra.engine.helpers.**read_stream_and_display**(*stream*, *display*)

Read from stream line by line until EOF, display, and capture the lines.

See also:

This discussion on StackOverflow.

pydra.engine.helpers.**record_error**(*error_path*, *error*)

Write an error file.

pydra.engine.helpers.**save**(*task_path: Path*, *result=None*, *task=None*, *name_prefix=None*)

Save a *TaskBase* object and/or results.

Parameters

- **task_path** (Path) – Write directory
- **result** (Result) – Result to pickle and write
- **task** (*TaskBase*) – Task to pickle and write

pydra.engine.helpers.**task_hash**(*task*)

Calculate the checksum of a task.

input hash, output hash, environment hash

Parameters

task (*TaskBase*) – The input task.

pydra.engine.helpers_file module

Functions ported from Nipype 1, after removing parts that were related to py2.

pydra.engine.helpers_file.**copyfile**(*originalfile*, *newfile*, *copy=False*, *create_new=False*, *use_hardlink=True*, *copy_related_files=True*)

Copy or link files.

If use_hardlink is True, and the file can be hard-linked, then a link is created, instead of copying the file.

If a hard link is not created and copy is False, then a symbolic link is created.

Copy options for existing files

- symlink
  - to regular file originalfile (keep if symlinking)
  - to same dest as symlink originalfile (keep if symlinking)
  - to other file (unlink)

- regular file
  - hard link to originalfile (keep)
  - copy of file (same hash) (keep)
  - different file (diff hash) (unlink)

Copy options for new files

- `use_hardlink` & `can_hardlink` => hardlink
- `~hardlink` & `~copy` & `can_symlink` => symlink
- `~hardlink` & `~symlink` => copy

Parameters

- **originalfile** (`str`) – full path to original file
- **newfile** (`str`) – full path to new file
- **copy** (*Bool*) – specifies whether to copy or symlink files (default=False) but only for POSIX systems
- **use_hardlink** (*Bool*) – specifies whether to hard-link files, when able (Default=False), taking precedence over copy
- **copy_related_files** (*Bool*) – specifies whether to also operate on related files, as defined in `related_filetype_sets`

Returns

Return type
None

`pydra.engine.helpers_file.`**`copyfile_input`**(*inputs*, *output_dir*)

Implement the base class method.

`pydra.engine.helpers_file.`**`copyfiles`**(*filelist*, *dest*, *copy=False*, *create_new=False*)

Copy or symlink files in `filelist` to `dest` directory.

Parameters

- **filelist** (*list*) – List of files to copy.
- **dest** (*path/files*) – full path to destination. If it is a list of length greater than 1, then it assumes that these are the names of the new files.
- **copy** (*Bool*) – specifies whether to copy or symlink files (default=False) but only for posix systems

Returns

Return type
None

`pydra.engine.helpers_file.`**`ensure_list`**(*filename*)

Return a list given either a string or a list.

pydra.engine.helpers_file.**fname_presuffix**(*fname*, *prefix=''*, *suffix=''*, *newpath=None*, *use_ext=True*)

Manipulate path and name of input filename.

Parameters

- **fname** (str) – A filename (may or may not include path)
- **prefix** (str) – Characters to prepend to the filename
- **suffix** (str) – Characters to append to the filename
- **newpath** (str) – Path to replace the path of the input fname
- **use_ext** (bool) – If True (default), appends the extension of the original file to the output name.

Returns

path – Absolute path of the modified filename

Return type

str

Examples

```
>>> import pytest, sys
>>> if sys.platform.startswith('win'): pytest.skip()
>>> from pydra.engine.helpers_file import fname_presuffix
>>> fname = 'foo.nii.gz'
>>> fname_presuffix(fname,'pre','post','/tmp')
'/tmp/prefoopost.nii.gz'
```

pydra.engine.helpers_file.**get_related_files**(*filename*, *include_this_file=True*)

Return a list of related files.

As defined in *related_filetype_sets*, for a filename (e.g., Nifti-Pair, Analyze (SPM), and AFNI files).

Parameters

- **filename** (str) – File name to find related filetypes of.
- **include_this_file** (*bool*) – If true, output includes the input filename.

pydra.engine.helpers_file.**hash_dir**(*dirpath*, *crypto=<built-in function openssl_sha256>*, *ignore_hidden_files=False*, *ignore_hidden_dirs=False*, *raise_notfound=True*, *precalculated=None*)

Compute hash of directory contents.

This function computes the hash of every file in directory *dirpath* and then computes the hash of that list of hashes to return a single hash value. The directory is traversed recursively.

Parameters

- **dirpath** (str) – Path to directory.
- **crypto** (:obj: *function*) – cryptographic hash functions
- **ignore_hidden_files** (bool) – If *True*, ignore filenames that begin with ..
- **ignore_hidden_dirs** (bool) – If *True*, ignore files in directories that begin with ..
- **raise_notfound** (bool) – If *True* and *dirpath* does not exist, raise *FileNotFound* exception. If *False* and *dirpath* does not exist, return *None*.

Returns
hash – Hash of the directory contents.

Return type
str

pydra.engine.helpers_file.**hash_file**(*afile*, *chunk_len=8192*, *crypto=<built-in function openssl_sha256>*, *raise_notfound=True*, *precalculated=None*)

Compute hash of a file using 'crypto' module.

pydra.engine.helpers_file.**is_container**(*item*)

Check if item is a container (list, tuple, dict, set).

Parameters
item (object) – Input object to check.

Returns
output – True if container False otherwise.

Return type
bool

pydra.engine.helpers_file.**is_existing_file**(*value*)

checking if an object is an existing file

pydra.engine.helpers_file.**is_local_file**(*f*)

pydra.engine.helpers_file.**on_cifs**(*fname*)

Check whether a file path is on a CIFS filesystem mounted in a POSIX host.

POSIX hosts are assumed to have the mount command.

On Windows, Docker mounts host directories into containers through CIFS shares, which has support for Minshall+French symlinks, or text files that the CIFS driver exposes to the OS as symlinks. We have found that under concurrent access to the filesystem, this feature can result in failures to create or read recently-created symlinks, leading to inconsistent behavior and FileNotFoundError errors.

This check is written to support disabling symlinks on CIFS shares.

pydra.engine.helpers_file.**related_filetype_sets = [('.hdr', '.img', '.mat'), ('.nii', '.mat'), ('.BRIK', '.HEAD')]**

List of neuroimaging file types that are to be interpreted together.

pydra.engine.helpers_file.**split_filename**(*fname*)

Split a filename into parts: path, base filename and extension.

Parameters
fname (str) – file or path name

Returns

- **pth** (str) – base path from fname
- **fname** (str) – filename from fname, without extension
- **ext** (str) – file extension from fname

Examples

```
>>> pth, fname, ext = split_filename('/home/data/subject.nii.gz')
>>> pth
'/home/data'
```

```
>>> fname
'subject'
```

```
>>> ext
'.nii.gz'
```

pydra.engine.helpers_file.**template_update**(*inputs*, *output_dir*, *state_ind=None*, *map_copyfiles=None*)

Update all templates that are present in the input spec.

Should be run when all inputs used in the templates are already set.

pydra.engine.helpers_file.**template_update_single**(*field*, *inputs*, *inputs_dict_st=None*, *output_dir=None*, *spec_type='input'*)

Update a single template from the input_spec or output_spec based on the value from inputs_dict (checking the types of the fields, that have "output_file_template)"

pydra.engine.helpers_state module

Additional functions used mostly by the State class.

exception pydra.engine.helpers_state.**PydraStateError**(*value*)

Bases: Exception

Custom error for Pydra State

pydra.engine.helpers_state.**add_name_combiner**(*combiner*, *name*)

adding a node's name to each field from the combiner

pydra.engine.helpers_state.**add_name_splitter**(*splitter*, *name*)

adding a node's name to each field from the splitter

pydra.engine.helpers_state.**combine_final_groups**(*combiner*, *groups*, *groups_stack*, *keys*)

Combine the final groups.

pydra.engine.helpers_state.**converter_groups_to_input**(*group_for_inputs*)

Return fields for each axis and number of all groups.

Requires having axes for all the input fields.

Parameters

group_for_inputs – specified axes (groups) for each input

pydra.engine.helpers_state.**flatten**(*vals*, *cur_depth=0*, *max_depth=None*)

Flatten a list of values.

pydra.engine.helpers_state.**input_shape**(*inp*, *cont_dim=1*)

Get input shape, depends on the container dimension, if not specify it is assumed to be 1

pydra.engine.helpers_state.**inputs_types_to_dict**(*name*, *inputs*)

Convert type.Inputs to dictionary.

pydra.engine.helpers_state.**iter_splits**(*iterable*, *keys*)

Generate splits.

pydra.engine.helpers_state.**map_splits**(*split_iter*, *inputs*, *cont_dim=None*)

generate a dictionary of inputs prescribed by the splitter.

pydra.engine.helpers_state.**remove_inp_from_splitter_rpn**(*splitter_rpn*, *inputs_to_remove*)

Remove inputs due to combining.

Mutates a splitter.

Parameters

- **splitter_rpn** – The splitter in reverse polish notation
- **inputs_to_remove** – input names that should be removed from the splitter

pydra.engine.helpers_state.**rpn2splitter**(*splitter_rpn*)

Convert from splitter_rpn to splitter.

Recurrent algorithm to perform the conversion. Every time combines pairs of input in one input, ends when the length is one.

Parameters
splitter_rpn – splitter in reverse polish notation

Returns
splitter in the standard/original form

Return type
splitter

pydra.engine.helpers_state.**splits_groups**(*splitter_rpn*, *combiner=None*, *inner_inputs=None*)

splits inputs to groups (axes) and creates stacks for these groups This is used to specify which input can be combined.

pydra.engine.helpers_state.**splitter2rpn**(*splitter*, *other_states=None*, *state_fields=True*)

Translate user-provided splitter into *reverse polish notation*.

The reverse polish notation is imposed by `State`.

Parameters

- **splitter** – splitter (standard form)
- **other_states** – other states that are connected to the state
- **state_fields** (`bool`) – if False the splitter from the previous states are unwrapped

pydra.engine.specs module

Task I/O specifications.

class pydra.engine.specs.**BaseSpec**

Bases: `object`

The base dataclass specs for all inputs and outputs.

check_fields_input_spec()

Check fields from input spec based on the medatada.

e.g., if xor, requires are fulfilled, if value provided when mandatory.

check_metadata()

Check contained metadata.

collect_additional_outputs(*inputs*, *output_dir*, *outputs*)

Get additional outputs.

copyfile_input(*output_dir*)

Copy the file pointed by a *File* input.

property hash

Compute a basic hash for any given set of fields.

retrieve_values(*wf*, *state_index=None*)

Get values contained by this spec.

template_update()

Update template.

class pydra.engine.specs.**ContainerSpec**(*, *executable: Union[str, List[str]]*, *args: Optional[Union[str, List[str]]] = None*, *image: Union[File, str]*, *container: Optional[Union[File, str]]*, *container_xargs: Optional[List[str]] = None*)

Bases: *ShellSpec*

Refine the generic command-line specification to container execution.

container: Optional[Union[*File*, str]]

The container.

container_xargs: Optional[List[str]]

image: Union[*File*, str]

The image to be containerized.

class pydra.engine.specs.**Directory**

Bases: object

An os.pathlike object, designating a folder.

class pydra.engine.specs.**DockerSpec**(*, *executable: Union[str, List[str]]*, *args: Optional[Union[str, List[str]]] = None*, *image: Union[File, str]*, *container_xargs: Optional[List[str]] = None*, *container: str = 'docker'*)

Bases: *ContainerSpec*

Particularize container specifications to the Docker engine.

container: str

The container.

class pydra.engine.specs.**File**

Bases: object

An os.pathlike object, designating a file.

class pydra.engine.specs.**FunctionSpec**

Bases: *BaseSpec*

Specification for a process invoked from a shell.

check_metadata()

Check the metadata for fields in input_spec and fields.

Also sets the default values when available and needed.

class pydra.engine.specs.**LazyField**(*node*, *attr_type*)

Bases: object

Lazy fields implement promises.

get_value(*wf*, *state_index=None*)

Return the value of a lazy field.

class pydra.engine.specs.**MultiInputFile**

Bases: *MultiInputObj*

A ty.List[File] object, converter changes a single file path to a list

class pydra.engine.specs.**MultiInputObj**

Bases: object

A ty.List[ty.Any] object, converter changes a single values to a list

classmethod converter(*value*)

class pydra.engine.specs.**MultiOutputFile**

Bases: *MultiOutputObj*

A ty.List[File] object, converter changes an 1-el list to the single value

class pydra.engine.specs.**MultiOutputObj**

Bases: object

A ty.List[ty.Any] object, converter changes an 1-el list to the single value

classmethod converter(*value*)

class pydra.engine.specs.**Result**(**, output: Optional[Any] = None, runtime: Optional[*Runtime*] = None, errored: bool = False*)

Bases: object

Metadata regarding the outputs of processing.

errored: bool

get_output_field(*field_name*)

Used in get_values in Workflow

Parameters

field_name (*str*) – Name of field in LazyField object

output: Optional[Any]

runtime: Optional[*Runtime*]

class pydra.engine.specs.**Runtime**(**, rss_peak_gb: Optional[float] = None, vms_peak_gb: Optional[float] = None, cpu_peak_percent: Optional[float] = None*)

Bases: object

Represent run time metadata.

cpu_peak_percent: Optional[float]

Peak in cpu consumption.

rss_peak_gb: Optional[float]

Peak in consumption of physical RAM.

vms_peak_gb: Optional[float]

Peak in consumption of virtual memory.

class pydra.engine.specs.**RuntimeSpec**(*, *outdir: Optional[str] = None*, *container: Optional[str] = 'shell'*, *network: bool = False*)

Bases: object

Specification for a task.

From CWL:

```
InlineJavascriptRequirement
SchemaDefRequirement
DockerRequirement
SoftwareRequirement
InitialWorkDirRequirement
EnvVarRequirement
ShellCommandRequirement
ResourceRequirement

InlineScriptRequirement
```

container: Optional[str]

network: bool

outdir: Optional[str]

class pydra.engine.specs.**ShellOutSpec**(*, *return_code: int*, *stdout: Union[File, str]*, *stderr: Union[File, str]*)

Bases: object

Output specification of a generic shell process.

collect_additional_outputs(*inputs*, *output_dir*, *outputs*)

Collect additional outputs from shelltask output_spec.

generated_output_names(*inputs*, *output_dir*)

Returns a list of all outputs that will be generated by the task. Takes into account the task input and the requires list for the output fields. TODO: should be in all Output specs?

return_code: int

The process' exit code.

stderr: Union[*File*, str]

The process' standard input.

stdout: Union[*File*, str]

The process' standard output.

class pydra.engine.specs.**ShellSpec**(*, *executable: Union[str, List[str]]*, *args: Optional[Union[str, List[str]]] = None*)

Bases: *BaseSpec*

Specification for a process invoked from a shell.

args: Optional[Union[str, List[str]]]

check_metadata()

Check the metadata for fields in input_spec and fields.

Also sets the default values when available and needed.

executable: Union[str, List[str]]

retrieve_values(*wf*, *state_index=None*)

Parse output results.

class pydra.engine.specs.**SingularitySpec**(*, *executable: Union[str, List[str]]*, *args: Optional[Union[str, List[str]]] = None*, *image: Union[*File*, str]*, *container_xargs: Optional[List[str]] = None*, *container: str = 'singularity'*)

Bases: *ContainerSpec*

Particularize container specifications to Singularity.

container: str

The container.

class pydra.engine.specs.**SpecInfo**(*, *name: str*, *fields: List[Tuple] = NOTHING*, *bases: Tuple[Type] = NOTHING*)

Bases: object

Base data structure for metadata of specifications.

bases: Tuple[Type]

Keeps track of specification inheritance. Should be a tuple containing at least one BaseSpec

fields: List[Tuple]

List of names of fields (can be inputs or outputs).

name: str

A name for the specification.

class pydra.engine.specs.**TaskHook**(*, *pre_run_task: ~typing.Callable = <function donothing>*, *post_run_task: ~typing.Callable = <function donothing>*, *pre_run: ~typing.Callable = <function donothing>*, *post_run: ~typing.Callable = <function donothing>*)

Bases: object

Callable task hooks.

post_run: Callable

post_run_task: Callable

pre_run: Callable

pre_run_task: Callable

reset()

pydra.engine.specs.**attr_fields**(*spec*, *exclude_names=()*)

pydra.engine.specs.**attr_fields_dict**(*spec*, *exclude_names=()*)

pydra.engine.specs.**donothing**(**args*, ***kwargs*)

pydra.engine.specs.**path_to_string**(*value*)

Convert paths to strings.

pydra.engine.state module

Keeping track of mapping and reduce operations over tasks.

class pydra.engine.state.**State**(*name*, *splitter=None*, *combiner=None*, *other_states=None*)

Bases: object

A class that specifies a State of all tasks.

- It's only used when a task have a splitter.
- It contains all information about splitter, combiner, final splitter, and input values for specific task states (specified by the splitter and the input).
- It also contains information about the final groups and the final splitter if combiner is available.

name

name of the state that is the same as a name of the task

Type

str

splitter

can be a str (name of a single input), tuple for scalar splitter, or list for outer splitter

Type

str, tuple, list

splitter_rpn_compact

splitter in RPN (reverse Polish notation), using a compact notation for splitter from previous states, e.g. _NA

Type

list

splitter_rpn

splitter represented in RPN, unwrapping splitters from previous states

Type

list

combiner

list of fields that should be combined (order is not important)

Type

list

splitter_final

final splitter that includes the combining process

other_states

used to create connections with previous states:

```
{
  name of a previous state:
    (previous state, input from current state needed the connection)
}
```

Type
dict

inner_inputs

used to create connections with previous states {"{self.name}.input name for current inp": previous state}

Type
dict

states_ind

dictionary for every state that contains indices for all state inputs (i.e. inputs that are part of the splitter)

Type
list of dict

states_val

dictionary for every state that contains values for all state inputs (i.e. inputs that are part of the splitter)

Type
list of dict

inputs_ind

dictionary for every state that contains indices for all task inputs (i.e. inputs that are relevant for current task, can be outputs from previous nodes)

Type
list of dict

group_for_inputs

specifying groups (axes) for each input field (depends on the splitter)

Type
dict

group_for_inputs_final

specifying final groups (axes) for each input field (depends on the splitter and combiner)

Type
dict

groups_stack_final

specify stack of groups/axes (used to determine which field could be combined)

Type
list

final_combined_ind_mapping

mapping between final indices after combining and partial indices of the results

Type
dict

property combiner

the combiner associated to the state.

combiner_validation()

validating if the combiner is correct (after all states are connected)

property current_combiner

the current part of the combiner, i.e. the part that is related to the current task's state only (doesn't include fields propagated from the previous tasks)

property current_combiner_all

the current part of the combiner including all the fields that should be combined (i.e. not only the fields that are explicitly set, but also the fields that re in the same group/axis and had to be combined together, e.g., if splitter is (a, b) a and b has to be combined together)

property current_splitter

the current part of the splitter, i.e. the part that is related to the current task's state only (doesn't include fields propagated from the previous tasks)

property current_splitter_rpn

the current part of the splitter using RPN

property inner_inputs

specifies connections between fields from the current state with the specific state from the previous states, uses dictionary `{input name for current state: the previous state}`

property other_states

specifies the connections with previous states, uses dictionary: {name of a previous state: (previous state, input field from current state)}

prepare_inputs()

Preparing inputs indices, merges input from previous states.

Includes indices for fields from inner splitters (removes elements connected to the inner splitters fields).

prepare_states(*inputs*, *cont_dim=None*)

Prepare a full list of state indices and state values.

State Indices
number of elements depends on the splitter

State Values
specific elements from inputs that can be used running interfaces

Parameters

- **inputs** (dict) – inputs of the task
- **cont_dim** (dict or *None*) – container's dimensions for a specific input's fields

prepare_states_combined_ind(*elements_to_remove_comb*)

Prepare the final list of dictionaries with indices after combiner.

Parameters
elements_to_remove_comb (list) – elements of the splitter that should be removed due to the combining

prepare_states_ind()

Calculate a list of dictionaries with state indices.

Uses hlpst.splits.

prepare_states_val()

Evaluate states values having states indices.

property prev_state_combiner

the prev-state part of the combiner, i.e. the part that comes from the previous tasks' states

property prev_state_combiner_all

the prev-state part of the combiner including all the fields that should be combined (i.e. not only the fields that are explicitly set, but also the fields that re in the same group/axis and had to be combined together, e.g., if splitter is (a, b) a and b has to be combined together)

property prev_state_splitter

the prev-state part of the splitter, i.e. the part that comes from the previous tasks' states

property prev_state_splitter_rpn

the prev-state art of the splitter using RPN

property prev_state_splitter_rpn_compact

the prev-state part of the splitter using RPN in a compact form, (without unwrapping the states from previous nodes), e.g. [_NA, _NB, *]

set_input_groups(*state_fields=True*)

Evaluates groups, especially the final groups that address the combiner.

Parameters
state_fields (bool) – if False the splitter from the previous states are unwrapped

splits(*splitter_rpn*)

Splits input variable as specified by splitter

Parameters
splitter_rpn (*list*) – splitter in RPN notation

Returns

- **splitter** (*list*) – each element contains indices for input variables
- **keys** (*list*) – names of input variables

property splitter

Get the splitter of the state.

property splitter_final

the final splitter, after removing the combined fields

property splitter_rpn

splitter in RPN

property splitter_rpn_compact

splitter in RPN with a compact representation of the prev-state part (i.e. without unwrapping the part that comes from the previous states), e.g., [_NA, _NB, *]

property splitter_rpn_final

splitter_validation()

validating if the splitter is correct (after all states are connected)

update_connections(*new_other_states=None*, *new_combiner=None*)

updating connections, can use a new other_states and combiner

Parameters

- **new_other_states** (dict, optional) – dictionary with new other_states, will be set before updating connections
- **new_combiner** (str, or list, optional) – new combiner

pydra.engine.submitter module

Handle execution backends.

class pydra.engine.submitter.**Submitter**(*plugin='cf'*, ***kwargs*)

Bases: object

Send a task to the execution backend.

close()

Close submitter.

Do not close previously running loop.

async expand_runnable(*runnable*, *wait=False*, *rerun=False*)

This coroutine handles state expansion.

Removes any states from *runnable*. If *wait* is set to False (default), aggregates all worker execution coroutines and returns them. If *wait* is True, waits for all coroutines to complete / error and returns None.

Parameters

- **runnable** (*pydra Task*) – Task instance (*Task*, *Workflow*)
- **wait** (*bool (False)*) – Await all futures before completing

Returns
futures – Coroutines for *TaskBase* execution.

Return type
set or None

async expand_workflow(*wf*, *rerun=False*)

Expand and execute a stateless *Workflow*. This method is only reached by *Workflow._run_task*.

Parameters
wf (*Workflow*) – Workflow Task object

Returns
wf – The computed workflow

Return type
pydra.engine.core.Workflow

async submit_from_call(*runnable*, *rerun*)

This coroutine should only be called once per Submitter call, and serves as the bridge between sync/async lands.

There are 4 potential paths based on the type of runnable: 0) Workflow has a different plugin than a submitter 1) Workflow without State 2) Task without State 3) (Workflow or Task) with State

Once Python 3.10 is the minimum, this should probably be refactored into using structural pattern matching.

pydra.engine.submitter.**get_runnable_tasks**(*graph*)

Parse a graph and return all runnable tasks.

pydra.engine.submitter.**is_runnable**(*graph*, *obj*)

Check if a task within a graph is runnable.

async pydra.engine.submitter.**prepare_runnable_with_state**(*runnable*)

pydra.engine.task module

Implement processing nodes.

Notes:

- Environment specs
  1. neurodocker json
  2. singularity file+hash
  3. docker hash
  4. conda env
  5. niceman config
  6. environment variables
- Monitors/Audit
  1. internal monitor
  2. external monitor
  3. callbacks
- Resuming
  1. internal tracking
  2. external tracking (DMTCP)
- Provenance
  1. Local fragments
  2. Remote server
- Isolation
  1. Working directory
  2. File (copy to local on write)
  3. read only file system
- Original implementation

---

class pydra.engine.task.**ContainerTask**(*container_info=None*, **args*, ***kwargs*)

Bases: *ShellCommandTask*

Extend shell command task for containerized execution.

bind_paths()

Get bound mount points

Returns
mount points – mapping from local path to tuple of container path + mode

Return type
dict

binds(*opt*)

Specify mounts to bind from local filesystems to container and working directory.

Uses py:meth:*bind_paths*

container_check(*container_type*)

Get container-specific CLI arguments.

class pydra.engine.task.**DockerTask**(*container_info=None*, **args*, ***kwargs*)

Bases: *ContainerTask*

Extend shell command task for containerized execution with the Docker Engine.

property container_args

Get container-specific CLI arguments, returns a list if the task has a state

init = False

class pydra.engine.task.**FunctionTask**(*func: Callable*, *audit_flags:* AuditFlag = *AuditFlag.NONE*, *cache_dir=None*, *cache_locations=None*, *input_spec: Optional[Union[*SpecInfo*,* BaseSpec*]] = None*, *cont_dim=None*, *messenger_args=None*, *messengers=None*, *name=None*, *output_spec: Optional[Union[*SpecInfo*,* BaseSpec*]] = None*, *rerun=False*, ***kwargs*)

Bases: *TaskBase*

Wrap a Python callable as a task element.

class pydra.engine.task.**ShellCommandTask**(*container_info=None*, **args*, ***kwargs*)

Bases: *TaskBase*

Wrap a shell command as a task element.

property cmdline

Get the actual command line that will be submitted Returns a list if the task has a state.

property command_args

Get command line arguments

input_spec = None

output_spec = None

class pydra.engine.task.**SingularityTask**(*container_info=None*, **args*, ***kwargs*)

Bases: *ContainerTask*

Extend shell command task for containerized execution with Singularity.

property container_args

Get container-specific CLI arguments.

init = False

pydra.engine.task.**split_cmd**(*cmd: str*)

Splits a shell command line into separate arguments respecting quotes

Parameters
cmd (*str*) – Command line string or part thereof

Returns
the command line string split into process args

Return type
str

pydra.engine.workers module

Execution workers.

class pydra.engine.workers.**ConcurrentFuturesWorker**(*n_procs=None*)

Bases: *Worker*

A worker to execute in parallel using Python's concurrent futures.

close()

Finalize the internal pool of tasks.

async exec_as_coro(*runnable*, *rerun=False*)

Run a task (coroutine wrapper).

run_el(*runnable*, *rerun=False*, ***kwargs*)

Run a task.

class pydra.engine.workers.**DaskWorker**(***kwargs*)

Bases: *Worker*

A worker to execute in parallel using Dask.distributed. This is an experimental implementation with limited testing.

close()

Finalize the internal pool of tasks.

async exec_dask(*runnable*, *rerun=False*)

Run a task (coroutine wrapper).

run_el(*runnable*, *rerun=False*, ***kwargs*)

Run a task.

class pydra.engine.workers.**DistributedWorker**(*loop=None*, *max_jobs=None*)

Bases: *Worker*

Base Worker for distributed execution.

async fetch_finished(*futures*)

Awaits asyncio's `asyncio.Task` until one is finished.

Limits number of submissions based on py:attr:*DistributedWorker.max_jobs*.

Parameters
futures (*set of asyncio awaitables*) – Task execution coroutines or asyncio `asyncio.Task`

Returns
pending – Pending asyncio `asyncio.Task`.

Return type
set

max_jobs

Maximum number of concurrently running jobs.

class pydra.engine.workers.**SGEWorker**(*loop=None*, *max_jobs=None*, *poll_delay=1*, *qsub_args=None*, *write_output_files=True*, *max_job_array_length=50*, *indirect_submit_host=None*, *max_threads=None*, *poll_for_result_file=True*, *default_threads_per_task=1*, *polls_before_checking_evicted=60*, *collect_jobs_delay=30*, *default_qsub_args=''*, *max_mem_free=None*)

Bases: *DistributedWorker*

A worker to execute tasks on SLURM systems.

async check_for_results_files(*jobid*, *threads_requested*)

async get_output_by_task_pkl(*task_pkl*)

async get_tasks_to_run(*task_qsub_args*, *mem_free*)

run_el(*runnable*, *rerun=False*)

Worker submission API.

async submit_array_job(*sargs*, *tasks_to_run*, *error_file*)

class pydra.engine.workers.**SerialWorker**(***kwargs*)

Bases: *Worker*

A worker to execute linearly.

close()

Return whether the task is finished.

async exec_serial(*runnable*, *rerun=False*)

async fetch_finished(*futures*)

Awaits asyncio's `asyncio.Task` until one is finished.

Parameters
futures (*set of asyncio awaitables*) – Task execution coroutines or asyncio `asyncio.Task`

Returns
pending – Pending asyncio `asyncio.Task`.

Return type
set

run_el(*interface*, *rerun=False*, ***kwargs*)

Run a task.

class pydra.engine.workers.**SlurmWorker**(*loop=None*, *max_jobs=None*, *poll_delay=1*, *sbatch_args=None*)

Bases: *DistributedWorker*

A worker to execute tasks on SLURM systems.

run_el(*runnable*, *rerun=False*)

Worker submission API.

class pydra.engine.workers.**Worker**(*loop=None*)

Bases: object

A base class for execution of tasks.

close()

Close this worker.

async fetch_finished(*futures*)

Awaits asyncio's asyncio.Task until one is finished.

Parameters
futures (*set of asyncio awaitables*) – Task execution coroutines or asyncio asyncio.Task

Returns
pending – Pending asyncio asyncio.Task.

Return type
set

run_el(*interface*, ***kwargs*)

Return coroutine for task execution.

3.1.2 pydra.mark package

Submodules

pydra.mark.functions module

Decorators to apply to functions used in Pydra workflows

pydra.mark.functions.**annotate**(*annotation*)

Update the annotation of a function.

Example

```
>>> import pydra
>>> @pydra.mark.annotate({'a': int, 'return': float})
... def square(a):
...     return a ** 2.0
```

pydra.mark.functions.**task**(*func*)

Promote a function to a *FunctionTask*.

Example

```
>>> import pydra
>>> @pydra.mark.task
... def square(a: int) -> float:
...     return a ** 2.0
```

3.1.3 pydra.tasks package

Pydra tasks

The `pydra.tasks` namespace is reserved for collections of Tasks, to be managed and packaged separately. To create a task package, please fork the pydra-tasks-template.

3.1.4 pydra.utils package

Submodules

pydra.utils.messenger module

Messaging of states.

class pydra.utils.messenger.**AuditFlag**(*value*)

Bases: `Flag`

Auditing flags.

ALL = 3

Track provenance and resource utilization.

NONE = 0

Do not track provenance or monitor resources.

PROV = 1

Track provenance only.

RESOURCE = 2

Monitor resource utilization only.

class pydra.utils.messenger.**FileMessenger**

Bases: *Messenger*

A messenger that redirects to a file.

send(*message*, *append=True*, ***kwargs*)

Append message to file.

Parameters

- **message** (`dict`) – The message to be printed.
- **append** (`bool`) – Do not truncate file when opening (i.e. append to it).

Returns

Returns the unique identifier used in the file's name.

Return type
str

class pydra.utils.messenger.**Messenger**

Bases: object

Base messenger class.

abstract send(*message*, ***kwargs*)

Send a message.

class pydra.utils.messenger.**PrintMessenger**

Bases: *Messenger*

A messenger that redirects to standard output.

send(*message*, ***kwargs*)

Send the message to standard output.

Parameters
message (dict) – The message to be printed.

class pydra.utils.messenger.**RemoteRESTMessenger**

Bases: *Messenger*

A messenger that redirects to remote REST endpoint.

send(*message*, ***kwargs*)

Append message to file.

Parameters
message (dict) – The message to be printed.

Returns
The status code from the *request.post*

Return type
int

class pydra.utils.messenger.**RuntimeHooks**(*value*)

Bases: IntEnum

Allowed points to hook into the process.

resource_monitor_post_stop = 4

resource_monitor_pre_start = 3

task_execute_post_exit = 6

task_execute_pre_entry = 5

task_run_entry = 1

task_run_exit = 2

pydra.utils.messenger.**collect_messages**(*collected_path*, *message_path*, *ld_op='compact'*)

Compile all messages into a single provenance graph.

Parameters

- **collected_path** (os.pathlike) – A place to write all of the collected messages. (?TODO)

- **message_path** (os.pathlike) – A path with the message file (?TODO)
- **ld_op** (str, optional) – Option used by pld.jsonld

pydra.utils.messenger.**gen_uuid**()

Generate a unique identifier.

pydra.utils.messenger.**make_message**(*obj*, *context=None*)

Build a message using the specific context

Parameters

- **obj** (dict) – A dictionary containing the non-context information of a message record.
- **context** (dict, optional) – Dictionary with the link to the context file or containing a JSON-LD context.

Returns

The message with the context.

Return type

dict

pydra.utils.messenger.**now**()

Get a formatted timestamp.

pydra.utils.messenger.**send_message**(*message*, *messengers=None*, ***kwargs*)

Send NIDM messages for logging provenance and auditing.

pydra.utils.profiler module

Utilities to keep track of performance and resource utilization.

class pydra.utils.profiler.**ResourceMonitor**(*pid*, *interval=5*, *logdir=None*, *fname=None*)

Bases: Thread

A thread to monitor a specific PID with a certain frequence to a file.

property fname

Get/set the internal filename.

run()

Core monitoring function, called by start().

stop()

Stop monitoring.

pydra.utils.profiler.**get_max_resources_used**(*pid*, *mem_mb*, *num_threads*, *pyfunc=False*)

Get the RAM and threads utilized by a given process.

Parameters

- **pid** (*integer*) – the process ID of process to profile
- **mem_mb** (*float*) – the high memory watermark so far during process execution (in MB)
- **num_threads** (*int*) – the high thread watermark so far during process execution

Returns

- **mem_mb** (*float*) – the new high memory watermark of process (MB)

- **num_threads** (*float*) – the new high thread watermark of process

pydra.utils.profiler.**get_system_total_memory_gb**()

Get the total RAM of the running system, in GB.

pydra.utils.profiler.**log_nodes_cb**(*node*, *status*)

Record node run statistics to a log file as json dictionaries.

Parameters

- **node** (*nipype.pipeline.engine.Node*) – the node being logged
- **status** (*string*) – acceptable values are 'start', 'end'; otherwise it is considered and error

Returns

this function does not return any values, it logs the node status info to the callback logger

Return type

None

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

D

E

M

N

O

P

R

S

T

U

V

W