signac-flow Documentation

Release 0.4.1

Carl Simon Adorf, Paul Dodd

February 24, 2017

Contents

Workflow management based on the signac framework.

The signac-flow package provides the basic infrastructure to easily configure and implement a workflow to operate on a signac data space.

Tip: The signac-project-template is a complete working implementation of a *`FlowProject`*, which serves both as example, but can also be used as a template to start new projects.

CHAPTER 1

Example Project

Basics

The signac-flow package is designed to operate with a scheduler. We submit operations to the scheduler by generating a script, which contains the commands to execute these.

Important: The following example is designed to teach how to use a FlowProject. To implement an actual project, it is usually better to fork the signac-project-template instead of starting from scratch.

For the sake of example, let's make the following assumptions:

1. We operate on a signac data space spanned by parameters `a` and `b`.
2. We initialize the data space in an `init.py` script by creating an `init.txt` file for each job.
3. After initialization we want to execute a *job-operation* on a cluster using the `foo` program which requires the name of the input file and a parameter `a` as arguments.
4. The output of `foo` will be stored in a file called `out.txt`.
5. The `foo` program may be parallelized with MPI on up to `b` ranks.
6. Finally, we want to label our jobs, so that we can assess the project's overall state.

Initialization

To get started we implement an initialization routine:

```
# init.py
import signac

project = signac.init_project('MyProject')

for a in range(10):
    for b in range(10):
        with project.open_job(dict(a=a, b=b)) as job:
            with open('init.txt', 'w') as file:
                # Some initialization routine
```

Workflow logic

Then we will implement the workflow logic in a module called `project.py` by specializing a *FlowProject*:

```python
# project.py
from math import ceil

from flow import FlowProject
from flow import JobOperation


class MyProject(FlowProject):

    # The classification of our job workspace is simple:
    def classify(self, job):
        if job.isfile('init.txt'):
            yield 'initialized'
        if job.isfile('out.txt'):
            yield 'processed'

    # There is only one operation in this project:
    def next_operation(self, job):
        labels = set(self.classify(job))
        # We can't run any operation if the job has not been properly initialized.
        if 'initialized' not in labels:
            return

        # If 'processed' is not among the labels, we return the command that
        # will execute the operation.
        if 'processed' not in labels:
            return JobOperation('process', job, 'foo -a {} init.txt > out.txt'.format(job.sp.a))

    # The FlowProject will automatically gather and bundle operations that are eligible
    # for execution. The user needs to specify how exactly these bundled operations
    # are executed on the cluster.
    def submit_user(self, env, _id, operations, **kwargs):
        # First we define a helper function to determine the # of processors
        # required for a specific operation

        def np(op):
            return op.job.sp.b if op.name == 'process' else 1

        # We use this function to determine the total number of required processors.
        np_total = sum(map(op, operations))

        # Then we calculate the # of required nodes based on the
        # processors per node(ppn) provided during submission.
        nn = ceil(np_total / ppn)

        # We start writing the job submission script by creating a JobScript instance
        # from the environment (env). The JobScript class is a thin io.StringIO wrapper,
        # essentially a in-memory text file.
        #
        # The JobScript instance contains a environment specific header and the exact
        # arguments required depend on the specific environment, but most environments
        # require a name (_id), the number of nodes (nn) and processors per node (ppn).
        sscript = env.script(_id, nn=nn, ppn=ppn)
        # Then we write the command for each operation to the script, assuming that we
        # want to execute the operation from within the job's workspace.
        #
        # The `write_cmd()` method will write the command to the script, wrapping it with
        # the environment specific mpi execution command if np is larger than one and making
        # sure that the command is executed in the background if bg is True.
        for op in operations:
            sscript.writeline('cd {}'.format(op.job.workspace()))
            sscript.write_cmd(op.cmd, np=np(op), bg=True)

        # Finally, we want to wait until all processes have finished before exiting the script.
        sscript.writeline('wait')
```

Workflow Execution

We can initialize the project's data space by executing the `init.py` script:

```
$ python init.py
```

To print the status of the project to screen, execute:

```
>>> project = MyProject()
>>> project.print_status(detailed=True, params=('a',))
Status project 'test-project':
Total # of jobs: 10
label         progress
------------  --------------------------------------------------
initialized   |########################################| 100.00%
processed     |##########################--------------| 66.67%
Detailed view:
job_id                            S  next_job  a  labels
--------------------------------  -  --------  -  -------------------------
108ef78ec381244447a108f931fe80db  U            1  initialized, processed
be01a9fd6b3044cf12c4a83ee9612f84  U            2  initialized, processed
32764c28ef130baefebeba76a158ac4e  U  process   3  initialized
# ...
>>>
```

We can submit the operations to the cluster by executing the `project.submit()` function. In many cases it is useful to create a separate `submit.py` script, which allows to submit operations directly from the command line, for example like this:

```
# submit.py
import argparse

from flow import get_environment

from project import MyProject

project = MyProject.get_project()
env = get_environment()

parser = argparser.ArgumentParser()
MyProject.add_submit_arguments(parser)
args = parser.parse_args()

project.submit(env, **vars(args))
```

This means we can execute the submission like this:

```
$ python submit.py
```

To get help concerning possible command line options, use the `--help` argument:

```
$ python submit.py --help
```

CHAPTER 2

flow API

Module contents

Workflow management based on the signac framework.

The signac-flow package provides the basic infrastructure to easily configure and implement a workflow to operate on a signac data space.

class `flow.`**`FlowProject`**(*config=None*)

Bases: `signac.contrib.project.Project`

A signac project class assisting in workflow management.

Parameters **`config`** (*`A signac config object.`*) – A signac configuaration, defaults to the configuration loaded from the environment.

`ALIASES` **= {'queued': 'Q', 'registered': 'R', 'unknown': 'U', 'requires_attention': '!', 'active': 'A', 'inactive': 'I', 'status**

`NAMES` **= {'next_operation': 'next_op'}**

classmethod **`add_print_status_args`**(*parser*)

Add arguments to parser for the *`print_status()`* method.

classmethod **`add_submit_args`**(*parser*)

Add arguments to parser for the *`submit()`* method.

`classify`(*job*)

Generator function which yields labels for job.

Parameters **`job`** (`Job`) – The signac job handle.

Yields The labels to classify job.

Yield type str

`eligible`(*job_operation*, ***kwargs*)

Determine if job is eligible for operation.

Warning: This function is deprecated, please use *`eligible_for_submission()`* instead.

`eligible_for_submission`(*job_operation*)

Determine if a job-operation is eligible for submission.

`export_job_stati`(*collection*, *stati*)

Export the job stati to a database collection.

format_row(*status*, *statepoint=None*, *max_width=None*)
Format each row in the detailed status output.

get_job_status(*job*)
Return the detailed status of a job.

labels(*job*)
Auto-generate labels from label-functions.

This generator function will automatically yield labels, from project methods decorated with the `@label` decorator.

For example, we can define a function like this:

```
class MyProject(FlowProject):

    @label()
    def is_foo(self, job):
        return job.document.get('foo', False)
```

The `labels()` generator method will now yield a `is_foo` label whenever the job document has a field `foo` which evaluates to True.

By default, the label name is equal to the function's name, but you can specify a custom label as the first argument to the label decorator, e.g.: `@label('foo_label')`.

Tip: In this particular case it may make sense to define the `is_foo()` method as a *staticmethod*, since it does not actually depend on the project instance. We can do this by using the `@staticlabel()` decorator, equivalently the `@classlabel()` for *class methods*.

map_scheduler_jobs(*scheduler_jobs*)
Map all scheduler jobs by job id.

This function fetches all scheduled jobs from the scheduler and generates a nested dictionary, where the first key is the job id, the second key the operation name and the last value are the cooresponding scheduler jobs.

For example, to print the status of all scheduler jobs, associated with a specific job operation, execute:

```
sjobs = project.scheduler_jobs(scheduler)
sjobs_map = project.map_scheduler_jobs(sjobs)
for sjob in sjobs_map[job.get_id()][operation]:
    print(sjob._id(), sjob.status())
```

Parameters **scheduler_jobs** – An iterable of scheduler job instances.

Returns A nested dictionary (job_id, op_name, scheduler jobs)

next_operation(*job*)
Determine the next operation for this job.

Parameters **job** (Job) – The signac job handle.

Returns A JobOpereation instance to execute next.

Return type *JobOperation*

print_status(*scheduler=None*, *job_filter=None*, *overview=True*, *overview_max_lines=None*, *detailed=False*, *parameters=None*, *skip_active=False*, *param_max_width=None*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *err=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>*, *pool=None*)

Print the status of the project.

Parameters

- **scheduler** (`Scheduler`) – The scheduler instance used to fetch the job stati.
- **job_filter** – A JSON encoded filter, that all jobs to be submitted need to match.
- **overview** (`bool`) – Aggregate an overview of the project' status.
- **overview_max_lines** (`int`) – Limit the number of overview lines.
- **detailed** (`bool`) – Print a detailed status of each job.
- **parameters** (`list of str`) – Print the value of the specified parameters.
- **skip_active** (`bool`) – Only print jobs that are currently inactive.
- **param_max_width** – Limit the number of characters of parameter columns, see also: `update_aliases()`.
- **file** – Redirect all output to this file, defaults to sys.stdout
- **err** – Redirect all error output to this file, defaults to sys.stderr
- **pool** – A multiprocessing or threading pool. Providing a pool parallelizes this method.

scheduler_jobs(*scheduler*)

Fetch jobs from the scheduler.

This function will fetch all scheduler jobs from the scheduler and also expand bundled jobs automatically.

However, this function will not automatically filter scheduler jobs which are not associated with this project.

Parameters **scheduler** (`Scheduler`) – The scheduler instance.

Yields All scheduler jobs fetched from the scheduler instance.

submit(*env*, *job_ids=None*, *operation_name=None*, *walltime=None*, *num=None*, *force=False*, *bundle_size=1*, *cmd=None*, *requires=None*, *pool=None*, ***kwargs*)

Submit job-operations to the scheduler.

This method will submit an operation for each job to the environment's scheduler, unless the job is considered active, e.g., because an operation associated with the same job has alreay been submitted.

The actual execution of operations is controlled in the `submit_user()` method which must be implemented by the user.

Parameters

- **env** (`ComputeEnvironment`) – The env instance.
- **job_ids** – A list of job_id's, whose next operation shall be executed. Defaults to all jobs found in the workspace.
- **operation_name** – If not None, only execute operations with this name.
- **walltime** (`float`) – The maximum wallclock time in hours.
- **num** (`int`) – If not None, limit number of submitted operations to *num*.
- **force** (`bool`) – Ignore warnings and checks during submission, just submit.

- **bundle_size** (*int*) – Bundle up to 'bundle_size' number of operations during submission.
- **cmd** (*str*) – Construct and submit an operation "on-the-fly" instead of submitting the "next operation".
- **requires** (*Iterable of str*) – A job's set of classification labels must fully intersect with the labels provided as part of this argument to be considered for submission.
- **kwargs** – Other keyword arguments which are forwarded to down-stream methods.

submit_user(*env*, *_id*, *operations*, *walltime=None*, *force=False*, ***kwargs*)

Implement this method to submit operations in combination with submit().

The `submit()` method provides an interface for the submission of operations to the environment's scheduler. Operations will be optionally bundled into one submission and.

The submit_user() method enables the user to create and submit a job submission script that controls the execution of all operations for this particular project.

Parameters

- **env** (`ComputeEnvironment`) – The environment to submit to.
- **_id** – A unique identifier, automatically calculated for this submission.
- **operations** – A list of operations that should be executed as part of this submission.
- **walltime** (`datetime.timedelta`) – The submission should be limited to the provided walltime.

Tpype _id str

Force Warnings and other checks should be ignored if this argument is True.

classmethod update_aliases(*aliases*)

Update the ALIASES table for this class.

update_stati(*scheduler*, *jobs=None*, *file=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>*, *pool=None*)

Update the status of all jobs with the given scheduler.

Parameters

- **scheduler** (*Scheduler*) – The scheduler instance used to feth the job stati.
- **jobs** – A sequence of `Job` instances.
- **file** – The file to write output to, defaults to *sys.stderr*.

write_human_readable_statepoint(*script*, *job*)

Write statepoint of job in human-readable format to script.

class `flow.`**JobOperation**(*name*, *job*, *cmd*)

Bases: `object`

Define operations to apply to a job.

A operation function in the context of signac is a function, with only one job argument. This in principle ensures that operations are deterministic in the sense that both input and output only depend on the job's metadata and data.

This class is designed to define commands to be executed on the command line that constitute an operation.

Note: The command arguments should only depend on the job metadata to ensure deterministic operations.

Parameters

- **name** (*str*) – The name of this JobOperation instance. The name is arbitrary, but helps to concisely identify the operation in various contexts.
- **job** (signac.Job.) – The job instance associated with this operation.
- **cmd** (*str*) – The command that constitutes the operation.

get_id()
Return a name, which identifies this job-operation.

get_status()
Retrieve the operation's last known status.

set_status(*value*)
Store the operation's status.

class flow.**label**(*name=None*)
Bases: object

Decorate a function to be a label function.

The label() method as part of FlowProject iterates over all methods decorated with this label and yields the method's name or the provided name.

For example:

```
class MyProject(FlowProject):

    @label()
    def foo(self, job):
        return True

    @label()
    def bar(self, job):
        return 'a' in job.statepoint()

>>> for label in MyProject().labels(job):
...     print(label)
```

The code segment above will always print the label 'foo', but the label 'bar' only if 'a' is part of a job's state point.

This enables the user to quickly write classification functions and use them for labeling, for example in the classify() method.

class flow.**classlabel**(*name=None*)
Bases: *flow.project.label*

A label decorator for classmethods.

This decorator implies "classmethod"!

class flow.**staticlabel**(*name=None*)
Bases: *flow.project.label*

A label decorator for staticmethods.

This decorator implies "staticmethod"!

flow.**get_environment**(*test=False*)
Attempt to detect the present environment.

This function iterates through all defined ComputeEnvironment classes in reversed order of definition and and returns the first EnvironmentClass where the is_present() method returns True.

Parameters **`test`** – Return the TestEnvironment

Returns The detected environment class.

flow.project module

Workflow definition with the FlowProject.

The FlowProject is a signac Project, that allows the user to define a workflow based on job classification and job operations.

A job may be classified based on its metadata and data in the form of str labels. These str-labels are yielded in the classify() method.

Based on the classification a "next operation" may be identified, that should be executed next to further the workflow. While the user is free to choose any method for the determination of the "next operation", one option is to use a FlowGraph.

class `flow.project.`**`FlowProject`**(*config=None*)

Bases: `signac.contrib.project.Project`

A signac project class assisting in workflow management.

Parameters **`config`** (*`A signac config object.`*) – A signac configuararion, defaults to the configuration loaded from the environment.

`ALIASES` **= {'queued': 'Q', 'registered': 'R', 'unknown': 'U', 'requires_attention': '!', 'active': 'A', 'inactive': 'I', 'status**

`NAMES` **= {'next_operation': 'next_op'}**

classmethod **`add_print_status_args`**(*parser*)

Add arguments to parser for the *`print_status()`* method.

classmethod **`add_submit_args`**(*parser*)

Add arguments to parser for the *`submit()`* method.

`classify`(*job*)

Generator function which yields labels for job.

Parameters **`job`** (`Job`) – The signac job handle.

Yields The labels to classify job.

Yield type str

`eligible`(*job_operation*, ***kwargs*)

Determine if job is eligible for operation.

Warning: This function is deprecated, please use *`eligible_for_submission()`* instead.

`eligible_for_submission`(*job_operation*)

Determine if a job-operation is eligible for submission.

`export_job_stati`(*collection*, *stati*)

Export the job stati to a database collection.

`format_row`(*status*, *statepoint=None*, *max_width=None*)

Format each row in the detailed status output.

get_job_status (*job*)
Return the detailed status of a job.

labels (*job*)
Auto-generate labels from label-functions.

This generator function will automatically yield labels, from project methods decorated with the `@label` decorator.

For example, we can define a function like this:

```
class MyProject(FlowProject):

    @label()
    def is_foo(self, job):
        return job.document.get('foo', False)
```

The `labels()` generator method will now yield a `is_foo` label whenever the job document has a field `foo` which evaluates to True.

By default, the label name is equal to the function's name, but you can specify a custom label as the first argument to the label decorator, e.g.: `@label('foo_label')`.

Tip: In this particular case it may make sense to define the `is_foo()` method as a *staticmethod*, since it does not actually depend on the project instance. We can do this by using the `@staticlabel()` decorator, equivalently the `@classlabel()` for *class methods*.

map_scheduler_jobs (*scheduler_jobs*)
Map all scheduler jobs by job id.

This function fetches all scheduled jobs from the scheduler and generates a nested dictionary, where the first key is the job id, the second key the operation name and the last value are the cooresponding scheduler jobs.

For example, to print the status of all scheduler jobs, associated with a specific job operation, execute:

```
sjobs = project.scheduler_jobs(scheduler)
sjobs_map = project.map_scheduler_jobs(sjobs)
for sjob in sjobs_map[job.get_id()][operation]:
    print(sjob._id(), sjob.status())
```

Parameters **scheduler_jobs** – An iterable of scheduler job instances.

Returns A nested dictionary (job_id, op_name, scheduler jobs)

next_operation (*job*)
Determine the next operation for this job.

Parameters **job** (Job) – The signac job handle.

Returns A JobOpereation instance to execute next.

Return type *JobOperation*

print_status (*scheduler=None, job_filter=None, overview=True, overview_max_lines=None, detailed=False, parameters=None, skip_active=False, param_max_width=None, file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>, err=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>, pool=None*)
Print the status of the project.

Parameters

- **scheduler** (*Scheduler*) – The scheduler instance used to fetch the job stati.
- **job_filter** – A JSON encoded filter, that all jobs to be submitted need to match.
- **overview** (*bool*) – Aggregate an overview of the project' status.
- **overview_max_lines** (*int*) – Limit the number of overview lines.
- **detailed** (*bool*) – Print a detailed status of each job.
- **parameters** (*list of str*) – Print the value of the specified parameters.
- **skip_active** (*bool*) – Only print jobs that are currently inactive.
- **param_max_width** – Limit the number of characters of parameter columns, see also: `update_aliases()`.
- **file** – Redirect all output to this file, defaults to sys.stdout
- **err** – Redirect all error output to this file, defaults to sys.stderr
- **pool** – A multiprocessing or threading pool. Providing a pool parallelizes this method.

scheduler_jobs(*scheduler*)
Fetch jobs from the scheduler.

This function will fetch all scheduler jobs from the scheduler and also expand bundled jobs automatically.

However, this function will not automatically filter scheduler jobs which are not associated with this project.

Parameters **scheduler** (*Scheduler*) – The scheduler instance.

Yields All scheduler jobs fetched from the scheduler instance.

submit(*env*, *job_ids=None*, *operation_name=None*, *walltime=None*, *num=None*, *force=False*, *bundle_size=1*, *cmd=None*, *requires=None*, *pool=None*, ***kwargs*)
Submit job-operations to the scheduler.

This method will submit an operation for each job to the environment's scheduler, unless the job is considered active, e.g., because an operation associated with the same job has alreay been submitted.

The actual execution of operations is controlled in the `submit_user()` method which must be implemented by the user.

Parameters

- **env** (`ComputeEnvironment`) – The env instance.
- **job_ids** – A list of job_id's, whose next operation shall be executed. Defaults to all jobs found in the workspace.
- **operation_name** – If not None, only execute operations with this name.
- **walltime** (*float*) – The maximum wallclock time in hours.
- **num** (*int*) – If not None, limit number of submitted operations to *num*.
- **force** (*bool*) – Ignore warnings and checks during submission, just submit.
- **bundle_size** (*int*) – Bundle up to 'bundle_size' number of operations during submission.
- **cmd** (*str*) – Construct and submit an operation "on-the-fly" instead of submitting the "next operation".

- **requires** (*Iterable of str*) – A job's set of classification labels must fully intersect with the labels provided as part of this argument to be considered for submission.
- **kwargs** – Other keyword arguments which are forwarded to down-stream methods.

submit_user(*env*, *_id*, *operations*, *walltime=None*, *force=False*, ***kwargs*)
Implement this method to submit operations in combination with submit().

The *submit()* method provides an interface for the submission of operations to the environment's scheduler. Operations will be optionally bundled into one submission and.

The submit_user() method enables the user to create and submit a job submission script that controls the execution of all operations for this particular project.

Parameters

- **env** (ComputeEnvironment) – The environment to submit to.
- **_id** – A unique identifier, automatically calculated for this submission.
- **operations** – A list of operations that should be executed as part of this submission.
- **walltime** (datetime.timedelta) – The submission should be limited to the provided walltime.

Tpype _id str

Force Warnings and other checks should be ignored if this argument is True.

classmethod update_aliases(*aliases*)
Update the ALIASES table for this class.

update_stati(*scheduler*, *jobs=None*, *file=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>*, *pool=None*)
Update the status of all jobs with the given scheduler.

Parameters

- **scheduler** (*Scheduler*) – The scheduler instance used to feth the job stati.
- **jobs** – A sequence of Job instances.
- **file** – The file to write output to, defaults to *sys.stderr*.

write_human_readable_statepoint(*script*, *job*)
Write statepoint of job in human-readable format to script.

class flow.project.**JobOperation**(*name*, *job*, *cmd*)
Bases: object

Define operations to apply to a job.

A operation function in the context of signac is a function, with only one job argument. This in principle ensures that operations are deterministic in the sense that both input and output only depend on the job's metadata and data.

This class is designed to define commands to be executed on the command line that constitute an operation.

Note: The command arguments should only depend on the job metadata to ensure deterministic operations.

Parameters

- **name** (*str*) – The name of this JobOperation instance. The name is arbitrary, but helps to concisely identify the operation in various contexts.

- **job** (signac.Job.) – The job instance associated with this operation.
- **cmd** (*str*) – The command that constitutes the operation.

get_id()
Return a name, which identifies this job-operation.

get_status()
Retrieve the operation's last known status.

set_status(*value*)
Store the operation's status.

flow.project.**abbreviate**(*x*, *a*)
Abbreviate x with a and add to the abbreviation table.

class flow.project.**classlabel**(*name=None*)
Bases: *flow.project.label*

A label decorator for classmethods.

This decorator implies "classmethod"!

flow.project.**draw_progressbar**(*value*, *total*, *width=40*)
Helper function for the visualization of progress.

flow.project.**is_active**(*status*)
True if a specific status is considered 'active'.

A active status usually means that no further operation should be executed at the same time to prevent race conditions and other related issues.

class flow.project.**label**(*name=None*)
Bases: object

Decorate a function to be a label function.

The label() method as part of FlowProject iterates over all methods decorated with this label and yields the method's name or the provided name.

For example:

```
class MyProject(FlowProject):

    @label()
    def foo(self, job):
        return True

    @label()
    def bar(self, job):
        return 'a' in job.statepoint()

>>> for label in MyProject().labels(job):
...     print(label)
```

The code segment above will always print the label 'foo', but the label 'bar' only if 'a' is part of a job's state point.

This enables the user to quickly write classification functions and use them for labeling, for example in the classify() method.

flow.project.**make_bundles**(*operations*, *size=None*)
Utility function for the generation of bundles.

This function splits a iterable of operations into equally sized bundles and a possibly smaller final bundle.

`flow.project.`**`shorten`**`(`*x*, *max_length=None*`)`

Shorten x to max_length and add to abbreviation table.

class `flow.project.`**`staticlabel`**`(`*name=None*`)`

Bases: *`flow.project.label`*

A label decorator for staticmethods.

This decorator implies "staticmethod"!

flow.graph module

class `flow.graph.`**`FlowCondition`**`(`*callback*`)`

Bases: `flow.graph._FlowNode`

A node within the FlowGraph corresponding to a condition function.

class `flow.graph.`**`FlowGraph`**

Bases: `object`

Define a workflow based on conditions and operations as a graph.

The `FlowGraph` class is designed to simplify the definition of more complex workflows, by adding operations to a graph, linking them by pre- and post-conditions. The assumption is that an operation is *eligible* for operation when the pre-condition is met, and at least one of the post-conditions is not met.

For example, assuming that we a have a *foo* operation, that requires a *ready* condition for execution and *should* result in a *started* and in a *done* condition, we can express this in graph form like this:

```
g = FlowGraph()
g.add_operation('foo', prereq=ready, postconds=[started, done])
```

The condition functions (`ready`, `started`, and `done`) need to be implemented as functions with a single argument, for example a signac job.

We can then determine *all* eligible operations within the graph, by calling the *`eligible_operations()`* method.

In the example above, the operation was defined as a `str`, however in principle we can use any object as operation, as long as they are uniquely comparable, e.g., an instance of *`flow.JobOperation`*, or a callable.

If we use callables, we can execute all eligible operations for all jobs in a project like this:

```
for i in range(max_num_iterations):  # make sure to limit this loop!
    for job in project:
        for op in g.eligible_operations(job):
            op(job)
```

`add_operation``(`*operation*, *prereq=None*, *postconds=None*`)`

Add an operation to the graph.

This method adds the operation, the optional pre-requirement condition, and optional post-conditions to the graph.

The method will be considered *eligible* for execution when the pre-condition is met and at least one of the post-conditions is **not** met. The assumption is that the executing the operation will eventually lead to all post-conditions to be met.

The operation may be any object that can be uniquely compared via `hash()`, e.g. a `str` or an instance of *`flow.JobOperation`* and *callables*.

All conditions must be callables, with exactly one argument.

Note: The operation and condition arguments can also be provided as instances of *`FlowOperation`* and *`FlowCondition`* since they will be internally converted to these classes anyways.

Parameters

- **`operation`** – The operation to add to the graph.
- **`prereq`** – The pre-condition callback function.
- **`postconds`** – The post-condition callback functions.

`conditions`()

Yields all conditions which are part of the graph.

`eligible`(*operation*, *job*)

Determine whether operation is eligible for execution.

The operation is considered to be *eligible* for execution if the pre-condition is met and at least one of the post-conditions is not met.

Parameters

- **`operation`** – The operation to check.
- **`job`** – The argument passed to the condition functions.

`eligible_operations`(*job*)

Determine all eligible operations within the graph.

This method yields all operations that are determined to be *eligible* for execution given their respective pre- and post-conditions.

Parameters **`job`** – The argument passed to the condition functions.

`link_conditions`(*a*, *b*)

Link a condition a with a condition b.

Linking conditions may be necessary to define a path within the graph, that is otherwise difficult to express.

The arguments may passedd as condition callback functions or instances of *`FlowCondition`*.

Parameters

- **`a`** – The condition to link with `b`.
- **`b`** – The condition to link with `a`.

`operation_chain`(*job*, *target*, *start=None*)

Generate a chain of operations to link a start and target condition.

This function will yield operations that need to be executed in order to reach a target condition given an optional start condition.

Parameters

- **`job`** – The argument passed to the condition functions.
- **`target`** – The target condition that is to be reached.

- **start** – The start condition, by default None.

operations()
Yields all operations which are part of the graph.

class flow.graph.**FlowOperation**(*callback*)
Bases: flow.graph._FlowNode

A node within the FlowGraph corresponding to an operation.

flow.scheduler module

class flow.scheduler.**FakeScheduler**(*header=None*, *cores_per_node=None*, **args*, ***kwargs*)
Bases: *flow.manage.Scheduler*

jobs()

submit(*script*, **args*, ***kwargs*)

class flow.scheduler.**TorqueScheduler**(*user=None*, ***kwargs*)
Bases: *flow.manage.Scheduler*

classmethod is_present()

jobs()

submit(*script*, *resume=None*, *after=None*, *pretend=False*, *hold=False*, **args*, ***kwargs*)

submit_cmd = ['qsub']

class flow.scheduler.**SlurmScheduler**(*user=None*, ***kwargs*)
Bases: *flow.manage.Scheduler*

classmethod is_present()

jobs()

submit(*script*, *resume=None*, *after=None*, *hold=False*, *pretend=False*, ***kwargs*)

submit_cmd = ['sbatch']

class flow.scheduler.**MoabScheduler**(**args*, ***kwargs*)
Bases: *flow.torque.TorqueScheduler*

flow.environment module

Detection of compute environments.

This module provides the ComputeEnvironment class, which can be subclassed to automatically detect specific computational environments.

This enables the user to adjust their workflow based on the present environment, e.g. for the adjustemt of scheduler submission scripts.

class flow.environment.**CPUEnvironment**
Bases: *flow.environment.ComputeEnvironment*

class flow.environment.**ComputeEnvironment**
Bases: object

Define computational environments.

The ComputeEnvironment class allows us to automatically determine specific environments in order to programatically adjust workflows in different environments.

The default method for the detection of a specific environemnt is to provide a regular expression matching the environment's hostname. For example, if the hostname is my_server.com, one could identify the environment by setting the hostname_pattern to 'my_server'.

static bg (*cmd*)
Wrap a command (cmd) to be executed in the background.

classmethod get_scheduler ()
Return a environment specific scheduler driver.

The returned scheduler class provides a standardized interface to different scheduler implementations.

hostname_pattern = None

classmethod is_present ()
Determine whether this specific compute environment is present.

The default method for environment detection is trying to match a hostname pattern.

registry = OrderedDict([('UnknownEnvironment', <class 'flow.environment.UnknownEnvironment'>), ('TestEnvironm

scheduler = None

classmethod script (***kwargs*)
Return a JobScript instance.

Derived ComputeEnvironment classes may require additional arguments for the creation of a job submission script.

classmethod submit (*script*, **args*, ***kwargs*)
Submit a job submission script to the environment's scheduler.

Scripts should be submitted to the environment, instead of directly to the scheduler to allow for environment specific post-processing.

class flow.environment.**ComputeEnvironmentType** (*name*, *bases*, *dct*)
Bases: type

Meta class for the definition of ComputeEnvironments.

This meta class automatically registers ComputeEnvironment definitions, which enables the automatic determination of the present environment.

class flow.environment.**DefaultSlurmEnvironment**
Bases: *flow.environment.SlurmEnvironment*

A default environment for environments with slurm scheduler.

classmethod is_present ()

classmethod mpi_cmd (*cmd*, *np*)

classmethod script (*_id*, *nn*, *walltime*, *ppn=None*, ***kwargs*)

class flow.environment.**DefaultTorqueEnvironment**
Bases: *flow.environment.TorqueEnvironment*

A default environment for environments with TORQUE scheduler.

classmethod is_present ()

classmethod mpi_cmd (*cmd*, *np*)

classmethod script (*_id*, *nn*, *walltime*, *ppn=None*, ***kwargs*)

class `flow.environment.`**`GPUEnvironment`**

Bases: *`flow.environment.ComputeEnvironment`*

class `flow.environment.`**`JobScript`**(*env*)

Bases: `_io.StringIO`

"Simple StringIO wrapper for the creation of job submission scripts.

Using this class to write a job submission script allows us to use environment specific expressions, for example for MPI commands.

`eol` **= '\n'**

`write_cmd`(*cmd*, *np=1*, *bg=False*)

Write a command to the jobscript.

This command wrapper function is a convenience function, which adds mpi and other directives whenever necessary.

Parameters

- **`cmd`** (*`str`*) – The command to write to the jobscript.
- **`np`** (*`int`*) – The number of processors required for execution.

`writeline`(*line=''*)

Write one line to the job script.

class `flow.environment.`**`MoabEnvironment`**(**args*, ***kwargs*)

Bases: *`flow.environment.ComputeEnvironment`*

"An environment with TORQUE scheduler.

This class is deprecated and only kept for backwards compatibility.

`scheduler_type`

alias of `TorqueScheduler`

class `flow.environment.`**`SlurmEnvironment`**

Bases: *`flow.environment.ComputeEnvironment`*

An environment with slurm scheduler.

`scheduler_type`

alias of `SlurmScheduler`

class `flow.environment.`**`TestEnvironment`**

Bases: *`flow.environment.ComputeEnvironment`*

This is a test environment.

The test environment will print a mocked submission script and submission commands to screen. This enables testing of the job submission script generation in environments without an real scheduler.

`cores_per_node` **= 1**

classmethod **`mpi_cmd`**(*cmd*, *np*)

`scheduler_type`

alias of `FakeScheduler`

classmethod **`script`**(***kwargs*)

class `flow.environment.`**`TorqueEnvironment`**

Bases: *`flow.environment.ComputeEnvironment`*

An environment with TORQUE scheduler.

scheduler_type
alias of TorqueScheduler

class flow.environment.**UnknownEnvironment**
Bases: *flow.environment.ComputeEnvironment*

This is a default environment, which is always present.

classmethod get_scheduler ()

classmethod is_present ()

scheduler_type = None

flow.environment.**format_timedelta** (*delta*)
Format a time delta for interpretation by schedulers.

flow.environment.**get_environment** (*test=False*)
Attempt to detect the present environment.

This function iterates through all defined ComputeEnvironment classes in reversed order of definition and and returns the first EnvironmentClass where the is_present() method returns True.

Parameters **test** – Return the TestEnvironment

Returns The detected environment class.

flow.manage module

class flow.manage.**ClusterJob** (*jobid*, *status=None*)
Bases: object

name ()

status ()

class flow.manage.**JobStatus**
Bases: enum.IntEnum

Classifies the job's execution status.

The stati are ordered by the significance of the execution status. This enables easy comparison, such as

which prevents a submission of a job, which is already submitted, queued, active or in an error state.

active = <JobStatus.active: 7>

error = <JobStatus.error: 8>

held = <JobStatus.held: 5>

inactive = <JobStatus.inactive: 3>

queued = <JobStatus.queued: 6>

registered = <JobStatus.registered: 2>

submitted = <JobStatus.submitted: 4>

unknown = <JobStatus.unknown: 1>

user = <JobStatus.user: 128>

class `flow.manage.`**`Scheduler`**(*header=None*, *cores_per_node=None*, **args*, ***kwargs*)

Bases: `object`

Generic Scheduler ABC

`jobs`()

yields ClusterJob

`flow.manage.`**`submit`**(*env*, *project*, *state_point*, *script*, *identifier='default'*, *force=False*, *pretend=False*, **args*, ***kwargs*)

Attempt to submit a job to the scheduler of the current environment.

The job status will be determined from the job's status document. If the job's status is greater or equal than JobStatus.submitted, the job will not be submitted, unless the force option is provided.

`flow.manage.`**`update_status`**(*job*, *scheduler_jobs=None*)

Update the job's status dictionary.

flow.fakescheduler module

class `flow.fakescheduler.`**`FakeScheduler`**(*header=None*, *cores_per_node=None*, **args*, ***kwargs*)

Bases: *`flow.manage.Scheduler`*

`jobs`()

`submit`(*script*, **args*, ***kwargs*)

flow.torque module

Routines for the MOAB environment.

class `flow.torque.`**`TorqueJob`**(*node*)

Bases: *`flow.manage.ClusterJob`*

`name`()

`status`()

class `flow.torque.`**`TorqueScheduler`**(*user=None*, ***kwargs*)

Bases: *`flow.manage.Scheduler`*

classmethod `is_present`()

`jobs`()

`submit`(*script*, *resume=None*, *after=None*, *pretend=False*, *hold=False*, **args*, ***kwargs*)

`submit_cmd` = ['qsub']

flow.slurm module

Routines for the MOAB environment.

class `flow.slurm.`**`SlurmJob`**(*jobid*, *status=None*)

Bases: *`flow.manage.ClusterJob`*

class `flow.slurm.`**`SlurmScheduler`**(*user=None*, ***kwargs*)

Bases: *`flow.manage.Scheduler`*

classmethod **`is_present`**()

`jobs`()

`submit`(*script*, *resume=None*, *after=None*, *hold=False*, *pretend=False*, ***kwargs*)

`submit_cmd` = **['sbatch']**

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index