signac-flow Documentation

Release 0.5.0

Carl Simon Adorf, Paul Dodd

May 24, 2017

Contents

The signac-flow tool provides the basic components to setup simple to complex workflows for signac projects. That includes the definition of data pipelines, execution of data space operations and the submission of operations to high-performance super computers.

The implementation is in pure Python, requires signac and is tested for Python versions 2.7 and 3.4+.

The screencast below demonstrates the general concept of setting up a workflow with **signac-flow**. For a detailed introduction, please checkout the *Reference* documentation!

CHAPTER 1

Installation

The recommendend installation method for **signac-flow** is via conda or pip. The software is tested for Python versions 3.4+ and only depends on the signac package.

Install with conda

To install **signac-flow** via conda, you first need to add the glotzer channel with:

```
$ conda config --add channels glotzer
```

Once the **glotzer** channel has been enabled, **signac-flow** can be installed with:

```
$ conda install signac-flow
```

All additional dependencies will be installed automatically. To upgrade the package, execute:

```
$ conda update signac-flow
```

Install with pip

To install the package with the package manager pip, execute

```
$ pip install signac-flow --user
```

Note: It is highly recommended to install the package into the user space and not as superuser!

To upgrade the package, simply execute the same command with the `--upgrade` option.

```
$ pip install signac-flow --user --upgrade
```

Source Code Installation

Alternatively you can clone the git repository and execute the `setup.py` script to install the package.

```
git clone https://bitbucket.org/glotzer/signac-flow.git
cd signac-flow
python setup.py install --user
```

CHAPTER 2

Reference

A complete reference to all major components of the **signac-flow** package.

Contents:

Basics

This chapter introduces the two **fundamental concepts** for the implementation of workflows with the **signac-flow** package: *Data Space Operations* and *Conditions*.

Data Space Operations

Concept

It is highly recommended to divide individual modifications of your project's data space into distinct functions. In this context, a *data space operation* is defined as a unary function with an instance of `Job` as its only argument.

This is an example for a simple *operation*, which creates a file called `hello.txt` within the job's workspace directory:

```
def hello(job):
    print('hello', job)
    with job:
        with open('hello.txt', 'w') as f:
            file.write('world!\n')
```

Let's initialize a few jobs, by executing the following script:

```
# init.py
import signac

project = signac.init_project('MyProject')
```

```
for i in range(10):
    project.open_job({'a': i}).init()
```

We could execute this *operation* for the complete data space, for example in the following manner:

```
>>> import signac
>>> from operations import hello
>>> project = signac.get_project()
>>> for job in project:
...     hello(job)
...
hello 0d32543f785d3459f27b8746f2053824
hello 14fb5d016557165019abaac200785048
hello 2af7905ebe91ada597a8d4bb91a1c0fc
>>>
```

The *flow.run()*-interface

However, we can simplify this. The flow package provides a command line interface for modules that contain *operations*. We can access this interface by calling the `run()` function:

The *run*-interface

The `run()` function parses the module from where it was called and interprets all **top-level unary functions** as operations.

```
# operations.py

def hello(job):
    print('hello', job)
    with job:
        with open('hello.txt', 'w') as f:
            file.write('world!\n')

if __name__ == '__main__':
    import flow
    flow.run()
```

Since the `hello()` function is a top-level function with only one argument, it is interpreted as a dataspace-operation. That means we can execute it directly from the command line:

```
$ python operations.py hello
hello 0d32543f785d3459f27b8746f2053824
hello 14fb5d016557165019abaac200785048
hello 2af7905ebe91ada597a8d4bb91a1c0fc
```

This is a brief demonstration on how to implement the `operations.py` module:

Parallelized Execution

The `run()` function automatically executes all operations in parallel on as a many processors as there are available. We can test that by adding a "cost-function" to our example *operation*:

```
from time import sleep

def hello(job):
    sleep(1)
    # ...
```

Executing this with `$ python operations.py hello` we can now see how many operations are executed in parallel:

Conditions

In the context of signac-flow, a workflow is defined by the **ordered** execution of *operations*. The execution order is determined by specific *conditions*.

That means in order to implement a workflow, we need to determine two things:

1. What is the **current state** of the data space?
2. What needs to happen **next**?

We answer the first question by evaluating unary condition functions for each job. Based on those *conditions*, we can then determine what should happen next.

Following the example from above, we define a `greeted` condition that determines whether the `hello()` operation was executed, e.g. the `hello.txt` file exists:

```
def greeted(job):
    return job.isfile('hello.txt')
```

Our workflow would then be completely determined like this:

```
for job in project:
    if not greeted(job):
        hello(job)
```

This is fine for simple workflows, however in the next chapter, we will see how to automate things further.

The FlowProject

This chapter describes how to setup a complete workflow via the implementation of a *FlowProject*.

Setup and Interface

To implement a more automated workflow, we can subclass a *FlowProject*:

```
# project.py
from flow import FlowProject

class MyProject(FlowProject):
    pass

if __name__ == '__main__':
    MyProject().main()
```

Tip: You can generate boiler-plate templates like the one above with the `$ flow init` function. There are multiple different templates available via the `-t/--template` option.

Executing this script on the command line will give us access to this project's specific command line interface:

```
$ python project.py
usage: project.py [-h] {status,next,run,script,submit} ...
```

Note: You can have **multiple** implementations of *FlowProject* that all operate on the same **signac** project! This may be useful, for example, if you want to implement two very distinct workflows that operate on the same data space. Simply put those in different modules, *e.g.*, `project_a.py` and `project_b.py`.

Classification

The *FlowProject* uses a *classify()* method to generate *labels* for a job. A label is a short text string, that essentially represents a condition. Following last chapter's example, we could implement a `greeted` label like this:

```
# project.py
from flow import FlowProject
from flow import staticlabel

class MyProject(FlowProject):

    @staticlabel()
    def greeted(job):
        return job.isfile('hello.txt')
# ...
```

Using the `staticlabel` decorator turns the `greeted()` function into a function, which will be evaluated for our classification. We can check that by executing the `hello` operation for a few job and then looking at the project's status:

```
$ python operations.py hello 0d32 2e6
hello 0d32543f785d3459f27b8746f2053824
hello 2e6ba580a9975cf0c01cb3c3f373a412
$ python project.py status --detailed
Status project 'MyProject':
Total # of jobs: 10

label    progress
-------  ----------
greeted  |########--------------------------------| 20.00%

Detailed view:
job_id                            S      next_op  labels
--------------------------------  ---  ---------  --------
0d32543f785d3459f27b8746f2053824  U               greeted
14fb5d016557165019abaac200785048  U
2af7905ebe91ada597a8d4bb91a1c0fc  U
2e6ba580a9975cf0c01cb3c3f373a412  U               greeted
42b7b4f2921788ea14dac5566e6f06d0  U
751c7156cca734e22d1c70e5d3c5a27f  U
81ee11f5f9eb97a84b6fc934d4335d3d  U
```

```
9bfd29df07674bc4aa960cf661b5acd2  U
9f8a8e5ba8c70c774d410a9107e2a32b  U
b1d43cd340a6b095b41ad645446b6800  U

Abbreviations used:
S: status
U: unknown
```

Determine the next-operation

Next, we should tell the project, that the `hello()` operation is to be executed, whenever the `greeted` condition is **not met**. We achieve this by adding the operation to the project:

```
class MyProject(FlowProject):

  def __init__(self, *args, **kwargs):
      super(MyProject, self).__init__(*args, **kwargs)

      self.add_operation(
        name='hello',
        cmd='python operations.py hello {job._id}',
        post=[MyProject.greeted])
```

Let's go through the individual arguments of the *`add_operation()`* method:

The `name` argument is arbitrary, but must be unique for all operations part of the project's workflow. It simply helps us to identify the operation without needing to look at the full command.

The `cmd` argument actually determines how to execute the particular operation, ideally it should be a function of job. We can construct the `cmd` either by using formatting fields, as shown above. We can use any attribute of our job instance, that includes state points (e.g. `job.sp.a`) or the workspace directory (`job.ws`). The command is later evaluated like this: `cmd.format(job=job)`.

Alternatively, we can define a function that returns a command or script, e.g.:

```
# ...
    self.add_operation(
        name='hello',
        cmd=lambda job: "python operations.py hello {}".format(job),
        post=[MyProject.greeted])
```

Finally, the `post` argument is a list of unary condition functions.

Definition:

A specific operation is **eligible for execution**, whenever all pre-conditions (`pre`) are met and at least one of the post-conditions (`post`) is not met.

In this case, the `hello` operation will only be executed, when `greeted()` returns `False`; we can check that again by looking at the status:

```
$ python project.py status --detailed
Status project 'MyProject':
Total # of jobs: 10

label    progress
```

```
-------  ------------------------------------------------
greeted  |########------------------------------| 20.00%

Detailed view:
job_id                            S    next_op    labels
--------------------------------  ---  ---------  --------
0d32543f785d3459f27b8746f2053824  U               greeted
14fb5d016557165019abaac200785048  U !  hello
2af7905ebe91ada597a8d4bb91a1c0fc  U !  hello
2e6ba580a9975cf0c01cb3c3f373a412  U               greeted
42b7b4f2921788ea14dac5566e6f06d0  U !  hello
751c7156cca734e22d1c70e5d3c5a27f  U !  hello
81ee11f5f9eb97a84b6fc934d4335d3d  U !  hello
9bfd29df07674bc4aa960cf661b5acd2  U !  hello
9f8a8e5ba8c70c774d410a9107e2a32b  U !  hello
b1d43cd340a6b095b41ad645446b6800  U !  hello

Abbreviations used:
!: requires_attention
S: status
U: unknown
```

Running project operations

Similar to the *run()* interface earlier, we can execute all pending operations with the `python project.py run` command:

```
$ python project.py run
hello 42b7b4f2921788ea14dac5566e6f06d0
hello 2af7905ebe91ada597a8d4bb91a1c0fc
hello 14fb5d016557165019abaac200785048
hello 751c7156cca734e22d1c70e5d3c5a27f
hello 9bfd29df07674bc4aa960cf661b5acd2
hello 81ee11f5f9eb97a84b6fc934d4335d3d
hello 9f8a8e5ba8c70c774d410a9107e2a32b
hello b1d43cd340a6b095b41ad645446b6800
```

Again, the execution is automatically parallelized.

Let's remove a few random `hello.txt` files to regain pending operations:

```
$ rm workspace/2af7905ebe91ada597a8d4bb91a1c0fc/hello.txt
$ rm workspace/9bfd29df07674bc4aa960cf661b5acd2/hello.txt
```

Generating Execution Scripts:

Using the `script` command, we can generate an **operation** execution script based on the pending operations, which might look like this:

```
$ python project.py script
---- BEGIN SCRIPT ----

set -u
set -e
cd /Users/johndoe/my_project
```

```
# Statepoint:
#
# {{
#   "a": 4
# }}
python operations.py hello 2af7905ebe91ada597a8d4bb91a1c0fc &

wait
---- END SCRIPT ----


---- BEGIN SCRIPT ----

set -u
set -e
cd /Users/johndoe/my_project

# Statepoint:
#
# {{
#   "a": 0
# }}
python operations.py hello 9bfd29df07674bc4aa960cf661b5acd2 &

wait
---- END SCRIPT ----
```

These scripts can be used for the execution of operations directly, or they could be submitted to a cluster environment for remote execution. For more information about how to submit operations for execution to a cluster environment, see the *Cluster Submission* chapter.

Full Demonstration

The screencast below is a complete demonstration of all steps:

Checkout the *next chapter* for a guide on how to submit operations to a cluster environment.

Cluster Submission

While it is always possible to manually submit scripts like the one shown in the *previous chapter* to a cluster, using the *flow interface* will allows us to **keep track of submitted operations** for example to prevent the resubmission of active operations.

In addition, **signac-flow** may utilize *environment profiles* to adjust the submission process based on your local environment. That is because different cluster environments will offer different resources and require slightly different options for submission. While the basic options will be as similar as possible, the *submit interface* will be slightly adapted to the local environment. You can check out the available options with the `python project.py submit --help` command.

The *submit* interface

In general, we submit operations through the primary interface of the *FlowProject*. If we have a `project.py` module (*as shown earlier*), which looks something like this:

```
# project.py
from flow import FlowProject

class Project(FlowProject):

    def __init__(*args, **kwargs):
        super(Project, self).__init__(*args, **kwargs)

if __name__ == '__main__':
    Project().main()
```

Then we can submit operations from the command line with the following command:

```
$ python project.py submit
```

Note: From here on we will abbreviate the `$ python project.py submit` command with `$ <project> submit`. That is because the module may be named differently.

In many cases you will need to provide additional arguments to the scheduler, such as your *account name*, the required *walltime*, and other information about requested resources. Some of these options can be specified through the native interface, that means flow knows about these options and you can see them when executing `submit --help`.

However, you can **always** forward any arguments directly to the scheduler command as positional arguments. For example, if we wanted to specify an account name with a *torque* scheduler, we could use the following command:

```
$ <project> submit -- -l A:my_account_name
```

Everything after the two dashes `--` will not be interpreted by the *submit* interface, but directly forwarded to the scheduler *as is*.

Note: Unless you have one of the *supported schedulers* installed, you will not be able to submit any operations on your computer, however you will be able to run some test commands in order to debug the process as best as you can. On the other hand, if you are in one of the natively supported high-performance super computing environments (e.g. XSEDE), you may take advantage of configurations profiles specifically tailored to those environments.

Submitting Operations

The submission process consists of the following steps:

1. *Gathering* of all operations *eligible* for submission.
2. Generating of scripts to execute those operations.
3. Submission of those scripts to the scheduler.

The first step is largely determined by your project *workflow*. You can see which operation might be submitted by looking at the output of `$ <project> status --detailed`. You may further reduce the operations to be submitted by selecting specifc jobs (`-j`), specific operations (`-o`), or generally reduce the total number of operations to be submitted (`-n`). For example the following command would submit up to 5 `hello` operations:

```
$ <project> submit -o hello -n 5
```

By default, all operations are *eligible for submission*, however you can overload the *FlowProject.eligible_for_submission()* method to customize this behavior.

The scripts for submission are generated by the *FlowProject.write_script()* method. This method itself calls the following methods:

```
write_script_header(script)
write_script_operations(script, operations, ...)
write_script_footer(script)
```

This means by default, each script will contain one header and footer at the beginning and end of the script and the commands for each operation will be written in between. In order to customize the generation of scripts, it is recommended to overload any of these three functions, or to overload the *write_script()* method itself.

Tip: Use the *script command* to debug the generation of execution scripts.

Parallelization

When submitting operations to the cluster, **signac-flow** assumes that each operations requires one processor and will generate a script requesting the resources accordingly.

When you execute *parallelized* operations you need to specify that with your *operation*. For example, assuming that we want to execute a program called `foo`, which will automatically parallelize onto 24 cores. Then we would need to specify the operation like this:

```
class MyProject(FlowProject):

    def __init__(self, *args, **kwargs):
        super(MyProject, self).__init__(*args, **kwargs)
            self.add_operation(
              name='foo',                           # name of the operation
              cmd='cd {job.ws}; foo input.txt',     # the execution command
              np=24,                                # foo requires 24 cores
            )
```

If you are using MPI for parallelization, you may need to prefix your command accordingly:

```
cmd='cd {job.ws}; mpirun -np 24 foo input.txt'
```

Different environment use different MPI-commands, you can use your environment-specific MPI-command like that:

```
from flow import get_environment

# ..
    env = get_environment()

    self.add_operation(
      name='foo',
      cmd='cd {job.ws};' +  env.mpi_cmd('foo input.txt', np=24),
      np=24,
    )
```

Tip: Both the `cmd`-argument and the `np`-argument may be *callables*, that means you can specify both the command itself, but also the number of processors **as a function of job**!

Here is an example using lambda-expressions:

```
self.add_operation(
    name='foo',
    cmd=lambda job: env.mpi_cmd("foo input.txt", np=job.sp.a),
    np=lambda job: job.sp.a)
```

Operation Bundling

By default all operations will be submitted as separate cluster jobs. This is usually the best model for clusters that scale well with the *size* of your operations. However, you may choose to *bundle* multiple operations into one submission using the `--bundle` option, *e.g.*, if you need to run multiple processes in parallel to fully utilize one node.

For example, the following command will bundle *up to 5* operations into a single cluster job:

```
$ <project> submit --bundle 5
```

These 5 operations will be executed *in parallel*, that means the resources for this cluster jobs will be the sum of the resources required for each operation. Without any argument the `--bundle` option will bundle **all** operations into a single cluster job.

Finally, if you have many small operations, you could bundle them into a single cluster job submission with the `--serial` option. In this mode, all bundled operations will be executed in serial and the resources required will be determined by the operation which requires the most resources.

Managing Environments

The **signac-flow** package attempts to detect your local environment and based on that adjusts the options provided by the `submit` interface. You can check which environment you are using, by looking at the output of `submit --help`.

For more information, see the *next chapter*.

Manage Environments

The **signac-flow** package uses environment profiles to adjust the submission process to local environments. That is because different environments provide different resources and options for the submission of operations to those resources. The basic options will always be the same, however there might be some subtle differences depending on where you want to submit your operations.

Tip: If you are running on a high-performance super computer, add the following line to your `project.py` module to import packaged profiles: `import flow.environments`

How to Use Environments

Environments are defined by subclassing from the *ComputeEnvironment* class. The *ComputeEnvironment* class is a *meta-class* that is automatically globally registered whenever you define one.

This enables us to use environments simply by defining them or importing them from a different module. The *get_environment()* function will go through all defined *ComputeEnvironment* classes and return the one, where the *is_present()* class method returns `True`.

Default Environments

The package comes with a few *default environments* which are **always available**. That includes the *DefaultTorqueEnvironment* and the *DefaultSlurmEnvironment*. This means that if you are within an environment with a *torque* or *slurm scheduler* you should be immediately able to submit to the cluster.

There is also a *TestEnvironment*, which you can use by calling the *get_environment()* function with `test=True` or by using the `--test` argument on the command line.

Packaged Environments

In addition, **signac-flow** comes with some additional *packaged environments*. These environments are defined within the *flow.environments* module. These environments are not automatically available, instead you need to *explictly import* the *flow.environments* module.

For a full list of all packaged environments, please see *Packaged Environments*.

Defining New Environments

In order to implement a new environment, create a new class that inherits from *ComputeEnvironment*. You will need to define a detection algorithm for your environment, be default we use a regular expression that matches the return value of `socket.gethostname()`.

Those are ususally the steps we need to take:

1. Subclass from *ComputeEnvironment*.
2. Determine a host name pattern that would match the output of `socket.gethostname()`.
3. Optionally specify the `cores_per_node` for environments with compute nodes.
4. Optionally overload the `mpi_cmd()` classmethod.
5. Overload the `script()` method to add specific options to the header of the submission script.

This is an example for a typical environment class definition:

```
class MyUniversityCluster(flow.TorqueEnvironment):

    hostname_pattern = 'mycluster.*.university.edu'
    cores_per_node = 32

    @classmethod
    def mpi_cmd(cls, cmd, np):
        return 'mpirun -np {np} {cmd}'.format(n=np, cmd=cmd)

    @classmethod
    def script(cls, _id, **kwargs):
```

```
        js = super(MyUniversityCluster, cls).script(_id=_id, **kwargs)
        js.writeline("$PBS -A {}".format(cls.get_config_value('account'))
        return js
```

The *get_config_value()* method allows us to get information from **signac**'s configuration which would be different for different users. Unless you provide a default value as the second argument, the user will be prompted to add the requested value to their configuration when using this specific profile for the first time.

Contributing Environments to the Package

Users are **highly encouraged** to contribute environment profiles that they developed for their local environments. In order to contribute an environment, either simply email them to the package maintainers (see the README for contact information), or add your environment directly to the `flow.environments/__init__.py` module and create a pull request!

Packaged Environments

The environments module contains additional *opt-in* environment profiles.

Add the following line to your project modules, to use these profiles:

```
import flow.environments
```

Environments for incite supercomputers.

class `flow.environments.incite.`**`TitanEnvironment`**
Environment profile for the titan super computer.

https://www.olcf.ornl.gov/titan/

class `flow.environments.incite.`**`EosEnvironment`**
Environment profile for the eos super computer.

https://www.olcf.ornl.gov/computing-resources/eos/

Environments for XSEDE supercomputers.

class `flow.environments.xsede.`**`CometEnvironment`**
Environment profile for the Comet supercomputer.

http://www.sdsc.edu/services/hpc/hpc_systems.html#comet

Environments for the University of Michigan HPC environment.

class `flow.environments.umich.`**`FluxEnvironment`**
Environment profile for the flux supercomputing environment.

http://arc-ts.umich.edu/systems-and-services/flux/

CHAPTER 3

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.

`add_operation`(*name*, *cmd*, *pre=None*, *post=None*, *np=None*, ***kwargs*)

Add an operation to the workflow.

This method will add an instance of `FlowOperation` to the operations-dict of this project.

Any FlowOperation is associated with a specific command, which should be a function of `Job`. The command (cmd) can be stated as function, either by using str-substitution based on a job's attributes, or by providing a unary callable, which expects an instance of job as its first and only positional argument.

For example, if we wanted to define a command for a program called 'hello', which expects a job id as its first argument, we could contruct the following two equivalent operations:

```
op = FlowOperation('hello', cmd='hello {job._id}')
op = FlowOperation('hello', cmd=lambda 'hello {}'.format(job._id))
```

Here are some more useful examples for str-substitutions:

Substitute job state point parameters: op = FlowOperation('hello', cmd='cd {job.ws}; hello {job.sp.a}')

Pre-requirements (pre) and post-conditions (post) can be used to trigger an operation only when certain conditions are met. Conditions are unary callables, which expect an instance of job as their first and only positional argument and return either True or False.

An operation is considered "eligible" for execution when all pre-requirements are met and when at least one of the post-conditions is not met. Requirements are always met when the list of requirements is empty and post-conditions are never met when the list of post-conditions is empty.

Please note, eligibility in this contexts refers only to the workflow pipline and not to other contributing factors, such as whether the job-operation is currently running or queued.

Parameters

- **name** (*str*) – A unique identifier for this operation, may be freely choosen.
- **cmd** (*str or callable*) – The command to execute operation; should be a function of job.
- **pre** (*sequence of callables*) – required conditions
- **post** – post-conditions to determine completion
- **np** (*int*) – Specify the number of processors this operation requires, defaults to 1.

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

classmethod add_script_args(*parser*)
Add arguments to parser for the *script()* method.

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

classify(*job*)
Generator function which yields labels for job.

By default, this method yields from the project's labels() method.

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

Yields The labels to classify job.

Yield type str

completed_operations(*job*)
Determine which operations have been completed for job.

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

Returns The name of the operations that are complete.

Return 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.

By default, an operation is eligible for submission when it is not considered active, that means already queued or running.

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*.

---

main(*parser=None*, *pool=None*)
Call this function to use the main command line interface.

In most cases one would want to call this function as part of the class definition, e.g.:

```
 my_project.py
from flow import FlowProject

class MyProject(FlowProject):
    pass

if __name__ == '__main__':
    MyProject().main()
```

You can then execute this script on the command line:

```
$ python my_project.py --help
```

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 An instance of JobOperation to execute next or *None*, if no operation is eligible.

Return type *JobOperation* or *NoneType*

next_operations(*job*)
Determine the next operations for job.

You can, but don't have to use this function to simplify the submission process. The default method returns yields all operation that a job is eligible for, as defined by the *add_operation()* method.

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

Yield All instances of JobOperation a job is eligible for.

operations
The dictionary of operations that have been added to the workflow.

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*, *ignore_errors=False*)
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.

run(*operations=None*, *pretend=False*, *np=None*, *timeout=None*, *progress=False*)
Execute the next operations as specified by the project's workflow.

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*, *bundle_size=1*, *serial=False*, *force=False*, *nn=None*, *ppn=None*, *walltime=None*, ***kwargs*)
Submit function for the project's main submit interface.

This method gather and optionally bundle all operations which are eligible for execution, prepare a submission script using the write_script() method, and finally attempting to submit these to the scheduler.

The primary advantage of using this method over a manual submission process, is that submit() will keep track of operation submit status (queued/running/completed/etc.) and will automatically prevent the submission of the same operation multiple times if it is considered active (e.g. queued or running).

submit_operations(*env*, *_id*, *operations*, *nn=None*, *ppn=None*, *serial=False*, *flags=None*, *force=False*, ***kwargs*)
Submit a sequence of operations to the scheduler.

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*, *ignore_errors=False*)
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.

write_script(*script*, *operations*, *background=False*)
Write a script for the execution of operations.

By default, this function will generate a script with the following components:

```
write_script_header(script)
write_script_operations(script, operations, background=background)
write_script_footer(script)
```

Consider overloading any of the methods above, before overloading this method.

Parameters

- **script** – The script to write the commands to.
- **operations** (*sequence of JobOperation*) – The operations to be written to the script.
- **background** (*bool*) – Whether operations should be executed in the background; useful to parallelize execution

write_script_footer(*script*, ***kwargs*)
Write the script footer for the execution script.

write_script_header(*script*, ***kwargs*)
Write the script header for the execution script.

write_script_operations(*script*, *operations*, *background=False*)
Write the commands for the execution of operations as part of a script.

class flow.**JobOperation**(*name*, *job*, *cmd*, *np=None*, *mpi=False*)
Bases: object

Define operations to apply to a job.

An 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.

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*, *import_configured=True*)
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.`**`run`**(*parser=None*)
Access to the "run" interface of an operations module.

Executing this function within a module will start a command line interface, that can be used to execute operations defined within the same module. All **top-level unary functions** will be intepreted as executable operation functions.

For example, if we have a module as such:

```
# operations.py

def hello(job):
    print('hello', job)

if __name__ == '__main__':
    import flow
    flow.run()
```

Then we can execute the `hello` operation for all jobs from the command like like this:

```
$ python operations.py hello
```

Note: The execution of operations is automatically parallelized. You can control the degree of parallelization with the `--np` argument.

For more information, see:

```
$ python operations.py --help
```

flow.scheduler module

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.`**`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_config_value`(*key*, *default=None*)
Request a value from the user's configuration.

This method should be used whenever values need to be provided that are specific to a users's environment. A good example are account names.

When a key is not configured and no default value is provided, a *`SubmitError`* will be raised and the user will be prompted to add the missing key to their configuration.

Please note, that the key will be automatically expanded to be specific to this environment definition. For example, a key should be 'account', not 'MyEnvironment.account'.

Parameters

- **`key`** (*`str`*) – The environment specific configuration key.
- **`default`** – A default value in case the key cannot be found within the user's configuration.

Returns The value or default value.

Raises ***`SubmitError`*** – If the key is not in the user's configuration and no default value is provided.

classmethod `get_scheduler`()
Return a environment specific scheduler driver.

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

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

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

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*, *flags=None*, **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.NodesEnvironment, *flow.environment.SlurmEnvironment*

A default environment for environments with slurm scheduler.

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

A default environment for environments with TORQUE scheduler.

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.

write_cmd(*cmd*, *bg=False*, *np=None*)
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.

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

An environment with slurm scheduler.

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.

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

An environment with TORQUE scheduler.

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

This is a default environment, which is always present.

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

flow.environment.**get_environment**(*test=False*, *import_configured=True*)
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.environment.**setup**(*py_modules*, ***attrs*)
Setup function for environment modules.

Use this function in place of setuptools.setup to not only install a environments module, but also register it with the global signac configuration. Once registered, is automatically imported when the get_environment() function is called.

flow.environments module

The environments module contains additional *opt-in* environment profiles.

Add the following line to your project modules, to use these profiles:

```
import flow.environments
```

flow.manage module

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.

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.errors module

exception flow.errors.**NoSchedulerError**
Bases: AttributeError

Indicates that there is no scheduler type defined for an environment class.

exception flow.errors.**SubmitError**
Bases: RuntimeError

Indicates an error during cluster job submission.

flow.fakescheduler module

flow.torque module

Routines for the MOAB environment.

flow.slurm module

Routines for the SLURM environment.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index