
signac-flow Documentation

Release 0.12.0

Carl Simon Adorf, Paul Dodd

Mar 04, 2021

Contents

1	Contents	3
1.1	Installation	3
1.2	Supported Environments	4
1.3	API Reference	12
1.4	Changelog	38
1.5	Support and Development	50
2	Indices and tables	55
	Python Module Index	57
	Index	59

Note: This is documentation for the **signac-flow** package, which is part of the **signac** framework. See [here](#) for a comprehensive introduction to the **signac** *framework*.

1.1 Installation

The recommended installation method for **signac-flow** is via [conda](#) or [pip](#). The software is tested for Python versions 3.6+ and [signac](#) versions 1.0+. Supported Python and NumPy versions are determined according to the [NEP 29](#) deprecation policy.

1.1.1 Install with conda

You can install **signac-flow** via conda (available on the [conda-forge](#) channel), with:

```
$ conda install -c conda-forge signac-flow
```

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

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

1.1.2 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
```

1.1.3 Source Code Installation

Alternatively you can clone the [git repository](https://github.com/glotzerlab/signac-flow) and execute the `setup.py` script to install the package.

```
git clone https://github.com/glotzerlab/signac-flow.git
cd signac-flow
python setup.py install --user
```

1.2 Supported Environments

The **signac-flow** package streamlines the submission of job operations to a supercomputer scheduling system. A different template is used for each scheduler and system, but all templates extend from the *Base Script*.

1.2.1 Base Script

Listing 1: `base_script.sh`

```
{# The following variables are available to all scripts. #}
{% if parallel %}
{% set np_global = operations|map(attribute='directives.np')|sum %}
{% else %}
{% set np_global = operations|map(attribute='directives.np')|max %}
{% endif %}
{% block header %}
{% block preamble %}
{% endblock %}
{% endblock %}

{% block project_header %}
set -e
set -u

cd {{ project.config.project_dir }}
{% endblock %}
{% block body %}
{% set cmd_suffix = cmd_suffix|default('') ~ (' &' if parallel else '') %}
{% for operation in operations %}

# {{ "%s"|format(operation) }}
{% block pre_operation scoped %}
{% endblock %}
{{ operation.cmd }}{{ cmd_suffix }}
{% if operation.eligible_operations|length > 0 %}
# Eligible to run:
{% for run_op in operation.eligible_operations %}
{% The split/join handles multi-line cmd operations. #}
# {{ "\n# ".join(run_op.cmd.strip().split("\n")) }}
{% endfor %}
{% endif %}
{% if operation.operations_with_unmet_preconditions|length > 0 %}
# Operations with unmet preconditions:
{% for run_op in operation.operations_with_unmet_preconditions %}
# {{ "\n# ".join(run_op.cmd.strip().split("\n")) }}
```

(continues on next page)

(continued from previous page)

```

{% endfor %}
{% endif %}
{% if operation.operations_with_met_postconditions|length > 0 %}
# Operations with all postconditions met:
{% for run_op in operation.operations_with_met_postconditions %}
# {{ "\n# ".join(run_op.cmd.strip().split("\n")) }}
{% endfor %}
{% endif %}
{% endblock %}
{% block footer %}
{% if parallel %}
wait
{% endif %}
{% endblock %}

```

The package currently ships with scheduler support for:

1.2.2 Slurm

[Link to official documentation](#)

class `flow.environment.DefaultSlurmEnvironment`

Bases: `flow.environment.NodesEnvironment`, `flow.environment.SlurmEnvironment`

Default environment for clusters with a SLURM scheduler.

Listing 2: slurm.sh

```

{% extends "base_script.sh" %}
{% block header %}
{% block preamble %}
#!/bin/bash
#SBATCH --job-name="{{ id }}"
{% if partition %}
#SBATCH --partition={{ partition }}
{% endif %}
{% if memory %}
#SBATCH --mem={{ memory }}
{% endif %}
{% if walltime %}
#SBATCH -t {{ walltime|format_timedelta }}
{% endif %}
{% if job_output %}
#SBATCH --output={{ job_output }}
#SBATCH --error={{ job_output }}
{% endif %}
{% endblock %}
{% block tasks %}
#SBATCH --ntasks={{ operations|calc_tasks('np', parallel, force) }}
{% endblock %}
{% endblock %}

```

1.2.3 TORQUE

[Link to qsub man page](#)

class `flow.environment.DefaultTorqueEnvironment`

Bases: `flow.environment.NodesEnvironment`, `flow.environment.TorqueEnvironment`

Default environment for clusters with a TORQUE scheduler.

Listing 3: torque.sh

```
{% extends "base_script.sh" %}
{% block header %}
#PBS -N {{ id }}
{% if walltime %}
#PBS -l walltime={{ walltime|format_timedelta }}
{% endif %}
{% if not no_copy_env %}
#PBS -V
{% endif %}
{% if memory %}
#PBS -l pmem={{ memory }}
{% endif %}
{% block tasks %}
{% set threshold = 0 if force else 0.9 %}
{% set cpu_tasks = operations|calc_tasks('np', parallel, force) %}
{% set gpu_tasks = operations|calc_tasks('ngpu', parallel, force) %}
{% set s_gpu = ':gpus=1' if gpu_tasks else '' %}
{% set ppn = ppn|default(operations|calc_tasks('omp_num_threads', parallel, force),
↳true) %}
{% if ppn %}
#PBS -l nodes={{ nn|default(cpu_tasks|calc_num_nodes(ppn, threshold, 'CPU'), true) }}
↳:ppn={{ ppn }}{{ s_gpu }}
{% else %}
#PBS -l procs={{ cpu_tasks }}{{ s_gpu }}
{% endif %}
{% endblock %}
{% endblock %}
```

1.2.4 LSF

[Link to LSF on IBM Knowledge Center](#)

class `flow.environment.DefaultLSFEnvironment`

Bases: `flow.environment.NodesEnvironment`, `flow.environment.LSFEnvironment`

Default environment for clusters with a LSF scheduler.

Listing 4: lsf.sh

```
{% extends "base_script.sh" %}
{% block header %}
#!/bin/bash
#BSUB -J {{ id }}
{% if partition %}
#BSUB -q {{ partition }}
{% endif %}
```

(continues on next page)

(continued from previous page)

```

{% if walltime %}
#BSUB -W {{ walltime|format_timedelta(style='HH:MM') }}
{% endif %}
{% if job_output %}
#BSUB -eo {{ job_output }}
{% endif %}
{% block tasks %}
#BSUB -n {{ operations|calc_tasks('np', parallel, force) }}
{% endblock %}
{% endblock %}

```

Any supercomputing system utilizing these schedulers is supported out of the box. In addition, the package provides specialized submission templates for the following environments:

1.2.5 INCITE Summit

Link to official documentation

class `flow.environments.incite.SummitEnvironment`

Bases: `flow.environment.DefaultLSFEnvironment`

Environment profile for the Summit supercomputer.

Example:

```

@Project.operation
@directives(nranks=3) # 3 MPI ranks per operation
@directives(ngpu=3) # 3 GPUs
@directives(np=3) # 3 CPU cores
@directives(rs_tasks=3) # 3 tasks per resource set
@directives(extra_jsrun_args='--smpiargs="-gpu"') # extra jsrun arguments
def my_operation(job):
    ...

```

<https://www.olcf.ornl.gov/summit/>

Listing 5: `summit.sh`

```

{# Templated in accordance with: https://www.olcf.ornl.gov/for-users/system-user-
↳guides/summit/running-jobs/ #}
{% extends "lsf.sh" %}
{% block tasks %}
{% set threshold = 0 if force else 0.9 %}
{% set nn = operations|map('guess_resource_sets')|calc_num_nodes(parallel) %}
#BSUB -nnodes {{ nn }}
{% endblock %}
{% block header %}
{{ super() -}}
{% set account = account|default(environment|get_account_name, true) %}
{% if account %}
#BSUB -P {{ account }}
{% endif %}
{% endblock %}

```

1.2.6 XSEDE Stampede2

Link to official documentation

class flow.environments.xsede.Stampede2Environment

Bases: `flow.environment.DefaultSlurmEnvironment`

Environment profile for the Stampede2 supercomputer.

<https://www.tacc.utexas.edu/systems/stampede2>

Listing 6: stampede2.sh

```
{# Templated in accordance with: https://portal.tacc.utexas.edu/user-guides/stampede2
↪#}
{% extends "slurm.sh" %}
{% set ns = namespace(use_launcher=True) %}
{% if operations|length() < 2 %}
{% set ns.use_launcher = False %}
{% endif %}
{% for operation in operations %}
{% if operation.directives.nranks or operation.directives.omp_num_threads or_
↪operation.directives.np > 1 %}
{% set ns.use_launcher = False %}
{% endif %}
{% endfor %}

{% block tasks %}
{% set threshold = 0 if force else 0.9 %}
{% if operations|calc_tasks('ngpu', false, true) and not force %}
{% raise "GPUs were requested but are unsupported by Stampede2!" %}
{% endif %}
{% set cpn = 48 if 'skx' in partition else 68 %}
{% if ns.use_launcher %}
{% set cpu_tasks = operations|calc_tasks('np', true, force) %}
#SBATCH --nodes={{ nn|default(cpu_tasks|calc_num_nodes(cpn, threshold, 'CPU'), true) }}
↪
#SBATCH --ntasks={{ nn|default(cpu_tasks|calc_num_nodes(cpn, threshold, 'CPU'), true)
↪* cpn }}
{% else %}
{% set cpu_tasks = operations|calc_tasks('np', parallel, force) %}
#SBATCH --nodes={{ nn|default(cpu_tasks|calc_num_nodes(cpn, threshold, 'CPU'), true) }}
↪
#SBATCH --ntasks={{ (operations|calc_tasks('nranks', parallel, force), 1)|max }}
{% endif %}
{% endblock %}

{% block header %}
{{ super () -}}
{% set account = account|default(environment|get_account_name, true) %}
{% if account %}
#SBATCH -A {{ account }}
{% endif %}
{% endblock %}

{% block body %}
{% if ns.use_launcher %}
{% if parallel %}
{{("Bundled submission without MPI on Stampede2 is using launcher; the --parallel_
↪option is therefore ignored.")|print_warning}}
```

(continues on next page)

(continued from previous page)

```

{% endif %}
{% set launcher_file = 'launcher_' ~ id|replace('/', '_') %}
{% set cmd_suffix = cmd_suffix|default('') %}
cat << EOF > {{ launcher_file }}
{% for operation in (operations|with_np_offset) %}
{{ operation.cmd }}{{ cmd_suffix }}
{% endfor %}
EOF

export LAUNCHER_PLUGIN_DIR=$LAUNCHER_DIR/plugins
export LAUNCHER_RMI=SLURM
export LAUNCHER_JOB_FILE={{ launcher_file }}

$LAUNCHER_DIR/paramrun
rm {{ launcher_file }}
{% else %}
{# Only the pre_operation block is overridden, all other behavior is inherited from
↳base_script.sh #}
{{ super () -}}
{# We need to reset the environment's base offset in between script generation for
↳separate bundles. #}
{# Since Jinja's bytecode optimizes out calls to filters with a constant argument, we
↳are forced to #}
{# rerun this function on the environment's base offset at the end of each run to
↳return the offset to 0. #}
{{ "%d"|format(environment.base_offset)|decrement_offset }}
{% endif %}
{% endblock %}

{# This override needs to happen outside the body block above, otherwise jinja2 doesn
↳t seem to #}
{# respect block scope and the operations variable is left undefined. #}
{% block pre_operation %}
export _FLOW_STAMPEDE_OFFSET={{ "%d"|format(operation.directives['nranks']|return_
↳and_increment) }}
{% endblock %}

```

1.2.7 XSEDE Comet

Link to official documentation

```

class flow.environments.xsede.CometEnvironment
    Bases: flow.environment.DefaultSlurmEnvironment

    Environment profile for the Comet supercomputer.

    https://www.sdsc.edu/services/hpc/hpc\_systems.html#comet

```

Listing 7: comet.sh

```

{# Templated in accordance with: http://www.sdsc.edu/support/user\_guides/comet.html
↳#running #}
{# This template can only be used with P100 GPUs! #}
{% extends "slurm.sh" %}
{% block tasks %}
{% set threshold = 0 if force else 0.9 %}

```

(continues on next page)

(continued from previous page)

```

{% set cpu_tasks = operations|calc_tasks('np', parallel, force) %}
{% set gpu_tasks = operations|calc_tasks('ngpu', parallel, force) %}
{% if gpu_tasks and 'gpu' not in partition and not force %}
{% raise "Requesting GPUs requires a gpu partition!" %}
{% endif %}
{% set nn_cpu = cpu_tasks|calc_num_nodes(24) %}
{% set nn_gpu = gpu_tasks|calc_num_nodes(4) if 'gpu' in partition else 0 %}
{% set nn = nn|default((nn_cpu, nn_gpu)|max, true) %}
{% if partition == 'gpu' %}
#SBATCH --nodes={{ nn|check_utilization(gpu_tasks, 4, threshold, 'GPU') }}
#SBATCH --gres=gpu:p100:{{ (gpu_tasks, 4)|min }}
{% elif partition == 'gpu-shared' %}
#SBATCH --nodes={{ nn|default(1, true)|check_utilization(gpu_tasks, 1, threshold, 'GPU
↪') }}
#SBATCH --ntasks-per-node={{ (gpu_tasks * 7, cpu_tasks)|max }}
#SBATCH --gres=gpu:p100:{{ gpu_tasks }}
{% elif 'shared' in partition %}{# standard shared partition #}
#SBATCH --nodes={{ nn|default(1, true) }}
#SBATCH --ntasks-per-node={{ cpu_tasks }}
{% else %}{# standard compute partition #}
#SBATCH --nodes={{ nn|check_utilization(cpu_tasks, 24, threshold, 'CPU') }}
#SBATCH --ntasks-per-node={{ (24, cpu_tasks)|min }}
{% endif %}
{% endblock tasks %}
{% block header %}
{{ super () -}}
{% set account = account|default(environment|get_account_name, true) %}
{% if account %}
#SBATCH -A {{ account }}
{% endif %}
{% endblock %}

```

1.2.8 XSEDE Bridges-2

Link to official documentation

class flow.environments.xsede.Bridges2Environment

Bases: *flow.environment.DefaultSlurmEnvironment*

Environment profile for the Bridges-2 supercomputer.

<https://www.psc.edu/resources/bridges-2/user-guide>

Listing 8: bridges2.sh

```

{# Templated in accordance with: https://www.psc.edu/resources/bridges-2/user-guide #}
{% extends "slurm.sh" %}
{% block tasks %}
{% set threshold = 0 if force else 0.9 %}
{% set cpu_tasks = operations|calc_tasks('np', parallel, force) %}
{% set gpu_tasks = operations|calc_tasks('ngpu', parallel, force) %}
{% if gpu_tasks %}
{% if not ('GPU' in partition or force) %}
{% raise "GPU operations require a GPU partition!" %}
{% endif %}
{# GPU nodes have 8 NVIDIA V100-32GB SXM2 #}

```

(continues on next page)

(continued from previous page)

```

{% set nn_gpu = gpu_tasks|calc_num_nodes(8) %}
{% set nn = nn_gpu %}
{% else %}
{% if 'GPU' in partition and not force %}
{% raise "Requesting GPU partition, but no GPUs requested!" %}
{% endif %}
{% set nn = nn|default(cpu_tasks|calc_num_nodes(128), true) %}
{% endif %}
{% if 'GPU' in partition %}
{% set gpus_per_node = (gpu_tasks / nn)|round(0, 'ceil')|int %}
{% set cpus_per_node = (cpu_tasks / nn)|round(0, 'ceil')|int %}
{% if cpus_per_node > gpus_per_node * 5 and not force %}
{% raise "Cannot request more than 5 CPUs per GPU." %}
{% endif %}
{% endif %}
{% if partition == 'GPU' %}
#SBATCH -N {{ nn|check_utilization(gpu_tasks, 8, threshold, 'GPU') }}
#SBATCH --tasks-per-node={{ cpus_per_node }}
#SBATCH --gpus={{ gpu_tasks }}
{% elif partition == 'GPU-shared' %}
#SBATCH -N {{ nn|check_utilization(gpu_tasks, 1, threshold, 'GPU') }}
#SBATCH --tasks-per-node={{ cpus_per_node }}
#SBATCH --gpus={{ gpu_tasks }}
{% elif partition == 'EM' %}
#SBATCH -N {{ nn|check_utilization(cpu_tasks, 96, threshold, 'CPU') }}
#SBATCH --tasks-per-node={{ (96, cpu_tasks)|min }}
{% elif partition == 'RM-shared' %}
{% if nn|default(1, true) > 1 or cpu_tasks > 64 %}
{% raise "Cannot request RM-shared with more than 64 tasks or multiple nodes." %}
{% endif %}
#SBATCH -N {{ nn|default(1, true) }}
#SBATCH --tasks={{ cpu_tasks }}
{% else %}
{# This should cover RM, RM-512, and possibly RM-small (not documented) #}
#SBATCH -N {{ nn|check_utilization(cpu_tasks, 128, threshold, 'CPU') }}
#SBATCH --tasks-per-node={{ (128, cpu_tasks)|min }}
{% endif %}
{% endblock tasks %}
{% block header %}
{{ super() -}}
{% set account = account|default(environment|get_account_name, true) %}
{% if account %}
#SBATCH -A {{ account }}
{% endif %}
{% endblock header %}

```

1.2.9 University of Michigan Great Lakes

Link to official documentation

class `flow.environments.umich.GreatLakesEnvironment`

Bases: `flow.environment.DefaultSlurmEnvironment`

Environment profile for the Great Lakes supercomputer.

<https://arc-ts.umich.edu/greatlakes/>

Listing 9: umich-greatlakes.sh

```
{% extends "slurm.sh" %}
{% set partition = partition|default('standard', true) %}
{% block tasks %}
{% set threshold = 0 if force else 0.9 %}
{% set cpu_tasks = operations|calc_tasks('np', parallel, force) %}
{% set gpu_tasks = operations|calc_tasks('ngpu', parallel, force) %}
{% if gpu_tasks and 'gpu' not in partition and not force %}
{% raise "Requesting GPUs requires a gpu partition!" %}
{% endif %}
{% set nn_cpu = cpu_tasks|calc_num_nodes(36) if 'gpu' not in partition else cpu_
↪tasks|calc_num_nodes(40) %}
{% set nn_gpu = gpu_tasks|calc_num_nodes(2) if 'gpu' in partition else 0 %}
{% set nn = nn|default((nn_cpu, nn_gpu)|max, true) %}
{% if partition == 'gpu' %}
#SBATCH --nodes={{ nn|default(1, true) }}
#SBATCH --ntasks-per-node={{ (gpu_tasks, cpu_tasks)|max }}
#SBATCH --gpus={{ gpu_tasks }}
{% else %}{# standard compute partition #}
#SBATCH --nodes={{ nn }}
#SBATCH --ntasks-per-node={{ (36, cpu_tasks)|min }}
{% endif %}
{% endblock tasks %}
{% block header %}
{{ super () -}}
{% set account = account|default(environment|get_account_name, true) %}
{% if account %}
#SBATCH --account={{ account }}
{% endif %}
{% endblock %}
```

1.2.10 University of Minnesota Mangi

Link to official documentation

class `flow.environments.umn.MangiEnvironment`

Bases: `flow.environment.DefaultSlurmEnvironment`

Environment profile for the Mangi supercomputer.

Mangi is managed by the Minnesota Supercomputing Institute (University of Minnesota).

Cluster information: <https://www.msi.umn.edu/mangi>

Queue information: <https://www.msi.umn.edu/queues>

We develop environment templates and add scheduler support on an as-needed basis. Please [contact us](#) if you have trouble running **signac-flow** on your local cluster or need assistance with creating a template or supporting a new scheduler.

1.3 API Reference

This is the API for the **signac-flow** application.

1.3.1 Command Line Interface

Some core **signac-flow** functions are—in addition to the Python interface—accessible directly via the `$ flow` command.

For more information, please see `$ flow --help`.

```
usage: flow [-h] [--debug] [--version] {init} ...

flow provides the basic components to set up workflows for projects as part of
the signac framework.

positional arguments:
  {init}
  init                Initialize a signac-flow project.

optional arguments:
  -h, --help          show this help message and exit
  --debug             Show traceback on error for debugging.
  --version           Display the version number and exit.
```

1.3.2 The FlowProject

Attributes

<code>FlowProject.ALIASES</code>	Default aliases used within the status output.
<code>FlowProject.add_operation(name, cmd[, pre, post])</code>	Add an operation to the workflow.
<code>FlowProject.completed_operations(job)</code>	Determine which operations have been completed for job.
<code>FlowProject.export_job_statuses(collection, ...)</code>	Export the job statuses to a <code>signac.Collection</code> .
<code>FlowProject.get_job_status(job[, ...])</code>	Return status information about a job.
<code>FlowProject.label([label_name_or_func])</code>	Designate a function as a label function for this class.
<code>FlowProject.labels(job)</code>	Yield all labels for the given job.
<code>FlowProject.main([parser])</code>	Call this function to use the main command line interface.
<code>FlowProject.make_group(name[, options])</code>	Make a <code>FlowGroup</code> named <code>name</code> and return a decorator to make groups.
<code>FlowProject.next_operations</code>	
<code>FlowProject.operation(func[, name])</code>	Add an operation function to the class workflow definition.
<code>FlowProject.operations</code>	Get the dictionary of operations that have been added to the workflow.
<code>FlowProject.post(condition[, tag])</code>	Define and evaluate postconditions for operations.
<code>FlowProject.post.copy_from(*other_funcs)</code>	Copy postconditions from other operation(s).
<code>FlowProject.post.false(key)</code>	Evaluate if a document key is False for the job(s).
<code>FlowProject.post.isfile(filename)</code>	Determine if the specified file exists for the job(s).
<code>FlowProject.post.never(func)</code>	Return False.
<code>FlowProject.post.not_(condition)</code>	Return <code>not condition(*jobs)</code> for the provided condition function.
<code>FlowProject.post.true(key)</code>	Evaluate if a document key is True for the job(s).

Continued on next page

Table 1 – continued from previous page

<code>FlowProject.pre(condition[, tag])</code>	Define and evaluate preconditions for operations.
<code>FlowProject.pre.after(*other_funcs)</code>	Precondition to run an operation after other operations.
<code>FlowProject.pre.copy_from(*other_funcs)</code>	Copy preconditions from other operation(s).
<code>FlowProject.pre.false(key)</code>	Evaluate if a document key is False for the job(s).
<code>FlowProject.pre.isfile(filename)</code>	Determine if the specified file exists for the job(s).
<code>FlowProject.pre.never(func)</code>	Return False.
<code>FlowProject.pre.not_(condition)</code>	Return not condition(*jobs) for the provided condition function.
<code>FlowProject.pre.true(key)</code>	Evaluate if a document key is True for the job(s).
<code>FlowProject.run([jobs, names, pretend, np, ...])</code>	Execute all eligible operations for the given selection.
<code>FlowProject.run_operations</code>	
<code>FlowProject.scheduler_jobs(scheduler)</code>	Fetch jobs from the scheduler.
<code>FlowProject.script</code>	
<code>FlowProject.submit([bundle_size, jobs, ...])</code>	Submit function for the project's main submit interface.
<code>FlowProject.submit_operations</code>	

class `flow.FlowProject` (*config=None, environment=None, entrypoint=None*)

Bases: `signac.contrib.project.Project`

A signac project class specialized for workflow management.

This class is used to define, execute, and submit workflows based on operations and conditions.

Users typically interact with this class through its command line interface.

This is a typical example of how to use this class:

```
@FlowProject.operation
def hello(job):
    print('hello', job)

FlowProject().main()
```

Parameters

- **config** (`signac.contrib.project._ProjectConfig`) – A signac configuration, defaults to the configuration loaded from the current directory.
- **environment** (`flow.environment.ComputeEnvironment`) – An environment to use for scheduler submission. If `None`, the environment is automatically identified. The default is `None`.
- **entrypoint** (`dict`) – A dictionary with two possible keys: 'executable' and 'path'. The path represents the location of the script file (the script file must call `FlowProject.main()`). The executable represents the location of the Python interpreter used for the execution of `BaseFlowOperation` that are Python functions.

ALIASES = {'active': 'A', 'error': 'E', 'held': 'H', 'inactive': 'I', 'queued': 'Q'}

Default aliases used within the status output.

PRINT_STATUS_ALL_VARYING_PARAMETERS = `True`

This constant can be used to signal that the `print_status()` method is supposed to automatically show all varying parameters.

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

Add an operation to the workflow.

This method will add an instance of *FlowOperation* to the operations of this project.

See also:

A Python function may be defined as an operation function directly using the *operation()* decorator.

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 construct 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}')
```

Preconditions (*pre*) and postconditions (*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 preconditions are met and when at least one of the postconditions is not met. Preconditions are always met when the list of preconditions is empty. Postconditions are never met when the list of postconditions is empty.

Please note, eligibility in this contexts refers only to the workflow pipeline 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, which may be freely chosen.
- **cmd** (*str or callable*) – The command to execute operation; should be a function of job.
- **pre** (*sequence of callables*) – List of preconditions. (Default value = *None*)
- **post** (*sequence of callables*) – List of postconditions. (Default value = *None*)
- ****kwargs** – Keyword arguments passed as directives.

completed_operations (*job*)

Determine which operations have been completed for job.

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

Yields *str* – The names of the operations that are complete.

detect_operation_graph ()

Determine the directed acyclic graph given by operation conditions.

In general, executing a given operation registered with a *FlowProject* just involves checking the operation’s preconditions and postconditions to determine eligibility. More generally, however, the preconditions and postconditions define a directed acyclic graph that governs the execution of all operations. Visualizing this graph can be useful for finding logic errors in the specified conditions, and having this graph computed also enables additional execution modes. For example, using this graph it is possible to determine exactly what operations need to be executed in order to make the operation eligible so that the task of executing all necessary operations can be automated.

The graph is determined by iterating over all pairs of operations and checking for equality of preconditions and postconditions. The algorithm builds an adjacency matrix based on whether the preconditions for one operation match the postconditions for another. The comparison of operations is conservative; by default, conditions must be composed of identical code to be identified as equal (technically, they must have equivalent bytecode, i.e. `cond1.__code__.co_code == cond2.__code__.co_code`). Users can specify that conditions should be treated as equal by providing tags to the operations.

Given a `FlowProject` subclass defined in a module `project.py`, the output graph could be visualized using Matplotlib and NetworkX with the following code:

```
import numpy as np
import networkx as nx
from matplotlib import pyplot as plt

from project import Project

project = Project()
ops = project.operations.keys()
adj = np.asarray(project.detect_operation_graph())

plt.figure()
g = nx.DiGraph(adj)
pos = nx.spring_layout(g)
nx.draw(g, pos)
nx.draw_networkx_labels(
    g, pos,
    labels={key: name for (key, name) in
            zip(range(len(ops)), [o for o in ops])})

plt.show()
```

Returns The adjacency matrix of operation dependencies. A zero indicates no dependency, and a 1 indicates dependency. This can be converted to a graph using NetworkX.

Return type list of lists of int

Raises `RuntimeError` – If a condition does not have a tag. This can occur when using `functools.partial`, and a manually specified condition tag has not been set.

export_job_statuses (*collection, statuses*)

Export the job statuses to a `signac.Collection`.

Deprecated since version 0.12: This will be removed in 0.14.

get_job_status (*job, ignore_errors=False, cached_status=None*)

Return status information about a job.

Parameters

- **job** (`Job`) – The signac job.
- **ignore_errors** (`bool`) – Whether to ignore exceptions raised during status check. (Default value = `False`)
- **cached_status** (`dict`) – Dictionary of cached status information. The keys are uniquely generated ids for each group and job. The values are instances of `JobStatus`. (Default value = `None`)

Returns A dictionary containing job status for all jobs.

Return type `dict`

groups

Get the dictionary of groups that have been added to the workflow.

classmethod label (*label_name_or_func=None*)

Designate a function as a label function for this class.

For example, we can define a label function like this:

```
@FlowProject.label
def foo_label(job):
    if job.document.get('foo', False):
        return 'foo-label-text'
```

The `foo-label-text` label will now show up in the status view for each job, where the `foo` key evaluates true.

If the label functions returns any type other than `str`, the label name will be the name of the function if and only if the return value evaluates to `True`, for example:

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

Finally, specify a label name by providing it as the first argument to the `label()` decorator.

Parameters `label_name_or_func` (*str or callable*) – A label name or callable. (Default value = None)

Returns A decorator for the label function.

Return type callable

labels (*job*)

Yield all labels for the given job.

See also: `label()`.

Parameters `job` (`signac.contrib.job.Job`) – Job handle.

Yields *str* – Label value.

main (*parser=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:

```
# my_project.py
from flow import FlowProject

class MyProject(FlowProject):
    pass

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

The project can then be executed on the command line:

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

Parameters `parser` (`argparse.ArgumentParser`) – The argument parser used to implement the command line interface. If None, a new parser is constructed. (Default value = None)

classmethod `make_group` (*name*, *options=""*)

Make a *FlowGroup* named *name* and return a decorator to make groups.

A *FlowGroup* is used to group operations together for running and submitting `_JobOperations`.

Examples

The code below creates a group and adds an operation to that group.

```
example_group = FlowProject.make_group('example')

@example_group
@FlowProject.operation
def foo(job):
    return "hello world"
```

Parameters

- **name** (*str*) – The name of the *FlowGroup*.
- **options** (*str*) – A string to append to submissions. Can be any valid `FlowOperation.run()` option. (Default value = "")

Returns The created group.

Return type *FlowGroupEntry*

classmethod `operation` (*func*, *name=None*)

Add an operation function to the class workflow definition.

This function is designed to be used as a decorator, for example:

```
@FlowProject.operation
def hello(job):
    print('Hello', job)
```

See also: `add_operation()`.

Parameters

- **func** (*callable*) – The function to add to the workflow.
- **name** (*str*) –
The operation name. Uses the name of the function if None. (Default value = None)

Returns The operation function.

Return type *callable*

operations

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

print_status (*jobs=None*, *overview=True*, *overview_max_lines=None*, *detailed=False*, *parameters=None*, *param_max_width=None*, *expand=False*, *all_ops=False*, *only_incomplete=False*, *dump_json=False*, *unroll=True*, *compact=False*, *pretty=False*, *file=None*, *err=None*, *ignore_errors=False*, *template=None*, *profile=False*, *eligible_jobs_max_lines=None*, *output_format='terminal'*)

Print the status of the project.

Parameters

- **jobs** (iterable of `Job`) – Only print status for the given jobs, or all if the argument is None. (Default value = None)
- **overview** (*bool*) – Display an overview of the project status. (Default value = True)
- **overview_max_lines** (*int*) – Limit the number of overview lines. (Default value = None)
- **detailed** (*bool*) – Print a detailed status of each job. (Default value = False)
- **parameters** (*list of str*) – Print the value of the specified parameters. (Default value = None)
- **param_max_width** (*int*) – Limit the number of characters of parameter columns. (Default value = None)
- **expand** (*bool*) – Present labels and operations in two separate tables. (Default value = False)
- **all_ops** (*bool*) – Include operations that are not eligible to run. (Default value = False)
- **only_incomplete** (*bool*) – Only show jobs that have eligible operations. (Default value = False)
- **dump_json** (*bool*) – Output the data as JSON instead of printing the formatted output. (Default value = False)
- **unroll** (*bool*) – Separate columns for jobs and the corresponding operations. (Default value = True)
- **compact** (*bool*) – Print a compact version of the output. (Default value = False)
- **pretty** (*bool*) – Prettify the output. (Default value = False)
- **file** (*str*) – Redirect all output to this file, defaults to `sys.stdout`.
- **err** (*str*) – Redirect all error output to this file, defaults to `sys.stderr`.
- **ignore_errors** (*bool*) – Print status even if querying the scheduler fails. (Default value = False)
- **template** (*str*) – User provided Jinja2 template file. (Default value = None)
- **profile** (*bool*) – Show profile result. (Default value = False)
- **eligible_jobs_max_lines** (*int*) – Limit the number of operations and its eligible job count printed in the overview. (Default value = None)
- **output_format** (*str*) – Status output format, supports: ‘terminal’ (default), ‘mark-down’ or ‘html’.

run (*jobs=None, names=None, pretend=False, np=None, timeout=None, num=None, num_passes=1, progress=False, order=None, ignore_conditions=<IgnoreConditions.NONE: 0>*)
Execute all eligible operations for the given selection.

This function will run in an infinite loop until all eligible operations are executed, unless it reaches the maximum number of passes per operation or the maximum number of executions.

By default there is no limit on the total number of executions, but a specific operation will only be executed once per job. This is to avoid accidental infinite loops when no or faulty postconditions are provided.

Parameters

- **jobs** (iterable of `Job`) – Only execute operations for the given jobs, or all if the argument is None. (Default value = None)

- **names** (iterable of *str*) – Only execute operations that match the provided set of names (interpreted as regular expressions), or all if the argument is None. (Default value = None)
- **pretend** (*bool*) – Do not actually execute the operations, but show the commands that would have been executed. (Default value = False)
- **np** (*int*) – Parallelize to the specified number of processors. Use -1 to parallelize to all available processing units. (Default value = None)
- **timeout** (*int*) – An optional timeout for each operation in seconds after which execution will be cancelled. Use -1 to indicate no timeout (the default).
- **num** (*int*) – The total number of operations that are executed will not exceed this argument if provided. (Default value = None)
- **num_passes** (*int or None*) – The total number of executions of one specific job-operation pair will not exceed this argument. The default is 1, there is no limit if this argument is None.
- **progress** (*bool*) – Show a progress bar during execution. (Default value = False)
- **order** (*str, callable, or None*) – Specify the order of operations. Possible values are:
 - 'none' or None (no specific order)
 - 'by-job' (operations are grouped by job)
 - 'cyclic' (order operations cyclic by job)
 - 'random' (shuffle the execution order randomly)
 - callable (a callable returning a comparison key for an operation used to sort operations)

The default value is 'none', which is equivalent to 'by-job' in the current implementation.

Note: Users are advised to not rely on a specific execution order as a substitute for defining the workflow in terms of preconditions and postconditions. However, a specific execution order may be more performant in cases where operations need to access and potentially lock shared resources.

- **ignore_conditions** (*IgnoreConditions*) – Specify if preconditions and/or postconditions are to be ignored when determining eligibility. The default is *IgnoreConditions.NONE*.

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 *ClusterJob* – All cluster jobs fetched from the scheduler.

submit (*bundle_size=1, jobs=None, names=None, num=None, parallel=False, force=False, walltime=None, ignore_conditions=<IgnoreConditions.NONE: 0>, ignore_conditions_on_execution=<IgnoreConditions.NONE: 0>, **kwargs*)

Submit function for the project's main submit interface.

Parameters

- **bundle_size** (*int*) – Specify the number of operations to be bundled into one submission, defaults to 1.
- **jobs** (iterable of `Job`) – Only submit operations for the given jobs, or all if the argument is `None`. (Default value = `None`)
- **names** (iterable of `str`) – Only submit operations that match the provided set of names (interpreted as regular expressions), or all if the argument is `None`. (Default value = `None`)
- **num** (*int*) – Limit the total number of submitted operations, defaults to no limit.
- **parallel** (*bool*) – Execute all bundled operations in parallel. (Default value = `False`)
- **force** (*bool*) – Ignore all warnings or checks during submission, just submit. (Default value = `False`)
- **walltime** (`datetime.timedelta`) – Specify the walltime in hours or as instance of `datetime.timedelta`. (Default value = `None`)
- **ignore_conditions** (`IgnoreConditions`) – Specify if preconditions and/or postconditions are to be ignored when determining eligibility. The default is `IgnoreConditions.NONE`.
- **ignore_conditions_on_execution** (`IgnoreConditions`) – Specify if preconditions and/or postconditions are to be ignored when determining eligibility after submitting. The default is `IgnoreConditions.NONE`.
- ****kwargs** – Additional keyword arguments forwarded to `submit()`.

`FlowProject.post` (*tag=None*)

Define and evaluate postconditions for operations.

A postcondition is a function accepting one or more jobs as positional arguments (`*jobs`) that must evaluate to `True` for this operation to be considered complete. For example:

```
@Project.operation
@Project.post(lambda job: job.doc.get('bye'))
def bye(job):
    print('bye', job)
    job.doc.bye = True
```

The `bye` operation would be considered complete and therefore no longer eligible for execution once the ‘bye’ key in the job document evaluates to `True`.

An optional tag may be associated with the condition. These tags are used by `detect_operation_graph()` when comparing conditions for equality. The tag defaults to the bytecode of the function.

classmethod `post.copy_from` (**other_funcs*)

Copy postconditions from other operation(s).

True if and only if all postconditions of other operation function(s) are met.

classmethod `post.false` (*key*)

Evaluate if a document key is `False` for the job(s).

Returns `True` if the specified key is present in the job document(s) and evaluates to `False`.

classmethod `post.isfile` (*filename*)

Determine if the specified file exists for the job(s).

classmethod `post.never` (*func*)
Return False.

classmethod `post.not_` (*condition*)
Return `not condition(*jobs)` for the provided condition function.

classmethod `post.true` (*key*)
Evaluate if a document key is True for the job(s).
Returns True if the specified key is present in the job document(s) and evaluates to True.

`FlowProject.pre` (*tag=None*)
Define and evaluate preconditions for operations.

A precondition is a function accepting one or more jobs as positional arguments (**jobs*) that must evaluate to True for this operation to be eligible for execution. For example:

```
@Project.operation
@Project.pre(lambda job: not job.doc.get('hello'))
def hello(job):
    print('hello', job)
    job.doc.hello = True
```

The *hello* operation would only execute if the 'hello' key in the job document does not evaluate to True.

An optional tag may be associated with the condition. These tags are used by `detect_operation_graph()` when comparing conditions for equality. The tag defaults to the bytecode of the function.

classmethod `pre.after` (**other_funcs*)
Precondition to run an operation after other operations.
True if and only if all postconditions of other operation function(s) are met.

classmethod `pre.copy_from` (**other_funcs*)
Copy preconditions from other operation(s).
True if and only if all preconditions of other operation function(s) are met.

classmethod `pre.false` (*key*)
Evaluate if a document key is False for the job(s).
Returns True if the specified key is present in the job document(s) and evaluates to False.

classmethod `pre.isfile` (*filename*)
Determine if the specified file exists for the job(s).

classmethod `pre.never` (*func*)
Return False.

classmethod `pre.not_` (*condition*)
Return `not condition(*jobs)` for the provided condition function.

classmethod `pre.true` (*key*)
Evaluate if a document key is True for the job(s).
Returns True if the specified key is present in the job document(s) and evaluates to True.

class `flow.IgnoreConditions`
Bases: `enum.IntFlag`
Flags that determine which conditions are used to determine job eligibility.
ALL = 3
Ignore all conditions.

NONE = 0
Check all conditions.

POST = 2
Ignore postconditions.

PRE = 1
Ignore preconditions.

1.3.3 Operations and Status

class `flow.project.BaseFlowOperation` (*pre=None, post=None*)
Bases: `object`

A *BaseFlowOperation* represents a data space operation acting on any job.

Every *BaseFlowOperation* is associated with a specific command.

Preconditions (*pre*) and postconditions (*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 preconditions are met and when at least one of the postconditions is not met. Preconditions are always met when the list of preconditions is empty. Postconditions are never met when the list of postconditions is empty.

Note: This class should not be instantiated directly.

Parameters

- **pre** (*sequence of callables*) – List of preconditions.
- **post** (*sequence of callables*) – List of postconditions.

class `flow.project.FlowOperation` (*op_func, pre=None, post=None*)
Bases: `flow.project.BaseFlowOperation`

An operation that executes a Python function.

All operations without the `@cmd` directive use this class. The callable *op_func* should be a function of one or more instances of `Job`.

Note: This class should not be instantiated directly.

Parameters

- **op_func** (*callable*) – A callable function of `*jobs`.
- **pre** (*sequence of callables*) – List of preconditions.
- **post** (*sequence of callables*) – List of postconditions.

__call__ (**jobs*)

Call the operation on the provided jobs.

Parameters `*jobs` (One or more instances of `Job`.) – The jobs passed to the operation.

Returns The result of the operation function.

Return type `object`

class `flow.project.FlowCmdOperation` (*cmd*, *pre=None*, *post=None*)

Bases: `flow.project.BaseFlowOperation`

An operation that executes a shell command.

When an operation has the `@cmd` directive specified, it is instantiated as a `FlowCmdOperation`. The operation should be a function of one or more positional arguments that are instances of `Job`. The command (*cmd*) may either be a callable that expects one or more instances of `Job` as positional arguments and returns a string containing valid shell commands, or the string of commands itself. In either case, the resulting string may contain any attributes of the job (or jobs) placed in curly braces, which will then be substituted by Python string formatting.

Note: This class should not be instantiated directly.

Parameters

- **cmd** (*str* or *callable*) – The command to execute the operation. Callable values will be provided one or more positional arguments (**jobs*) that are instances of `Job`. String values will be formatted with `cmd.format(jobs=jobs)` where *jobs* is a tuple of `Job`, or `cmd.format(jobs=jobs, job=job)` if only one job is provided.
- **pre** (*sequence of callables*) – List of preconditions.
- **post** (*sequence of callables*) – List of postconditions.

`__call__` (**jobs*)

Return the command formatted with the supplied job(s).

1.3.4 Labels

class `flow.label` (*name=None*)

Bases: `object`

Decorate a `FlowProject` class function as a label function.

For example:

```
class MyProject(FlowProject):  
  
    @label()  
    def foo(self, job):  
        return True
```

`__call__` (*func*)

Add the function as a label.

This call operator allows the class to be used as a decorator.

Parameters **func** (*callable*) – The function to decorate.

Returns The decorated function.

Return type `callable`

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

Bases: `flow.labels.label`

A label decorator for staticmethods.

This decorator implies “staticmethod”!

`__call__` (*func*)

Add the function as a label.

This call operator allows the class to be used as a decorator.

Parameters `func` (*callable*) – The function to decorate.

Returns The decorated function.

Return type callable

class `flow.classlabel` (*name=None*)

Bases: `flow.labels.label`

A label decorator for classmethods.

This decorator implies “classmethod”!

`__call__` (*func*)

Add the function as a label.

This call operator allows the class to be used as a decorator.

Parameters `func` (*callable*) – The function to decorate.

Returns The decorated function.

Return type callable

1.3.5 @flow.cmd

`flow.cmd` (*func*)

Indicate that `func` returns a shell command with this decorator.

If this function is an operation function defined by `FlowProject`, it will be interpreted to return a shell command, instead of executing the function itself.

For example:

```
@FlowProject.operation
@flow.cmd
def hello(job):
    return "echo {job.id}"
```

Note: The final shell command generated for `run()` or `submit()` still respects directives and will prepend e.g. MPI or OpenMP prefixes to the shell command provided here.

1.3.6 @flow.with_job

`flow.with_job` (*func*)

Use `arg` as a context manager for `func(arg)` with this decorator.

If this function is an operation function defined by `FlowProject`, it will be the same as using `with job:`.

For example:

```
@FlowProject.operation
@flow.with_job
def hello(job):
    print("hello {}".format(job))
```

Is equivalent to:

```
@FlowProject.operation
def hello(job):
    with job:
        print("hello {}".format(job))
```

This also works with the `@cmd` decorator:

```
@FlowProject.operation
@with_job
@cmd
def hello(job):
    return "echo 'hello {}'.format(job)
```

Is equivalent to:

```
@FlowProject.operation
@cmd
def hello_cmd(job):
    return 'trap "cd `pwd`" EXIT && cd {} && echo "hello {job}"'.format(job.ws)
```

1.3.7 @flow.directives

class `flow.directives` (***kwargs*)

Bases: `object`

Decorator for operation functions to provide additional execution directives.

Directives can for example be used to provide information about required resources such as the number of processes required for execution of parallelized operations. For more information, read about [Submission Directives](#).

In addition, you can use the `@directives(fork=True)` directive to enforce that a particular operation is always executed within a subprocess and not within the Python interpreter's process even if there are no other reasons that would prevent that.

Note: Setting `fork=False` will not prevent forking if there are other reasons for forking, such as a timeout.

`__call__` (*func*)

Add directives to the function.

This call operator allows the class to be used as a decorator.

Parameters `func` (*callable*) – The function to decorate.

Returns The decorated function.

Return type callable

classmethod `copy_from` (*func*)

Copy directives from another operation.

1.3.8 flow.run()

`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 interpreted 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: You can control the degree of parallelization with the `--np` argument.

For more information, see:

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

Deprecated since version 0.12: This will be removed in 0.14.

1.3.9 flow.init()

`flow.init` (*alias=None, template=None, root=None*)

Initialize a templated `FlowProject` module.

Parameters

- **alias** (*str*) – Python identifier used as a file name for the template output. Uses "project" if None. (Default value = None)
- **template** (*str*) – Name of the template to use. Built-in templates are:
 - "minimal"
 - "example"
 - "testing"
 Uses "minimal" if None. (Default value = None)
- **root** (*str*) – Directory where the output file is placed. Uses the current working directory if None. (Default value = None)

Returns List of file names created.

Return type `list`

1.3.10 flow.get_environment()

`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 returns the first environment where the `is_present()` method returns True.

Parameters

- **test** (*bool*) – Whether to return the TestEnvironment. (Default value = False)
- **import_configured** (*bool*) – Whether to import environments specified in the flow configuration. (Default value = True)

Returns The detected environment class.

Return type `ComputeEnvironment`

1.3.11 The FlowGroup

class `flow.project.FlowGroup` (*name, operations=None, operation_directives=None, options=""*)

Bases: `object`

A `FlowGroup` represents a subset of a workflow for a project.

A `FlowGroup` is associated with one or more instances of `BaseFlowOperation`.

Examples

In the example below, the directives will be `{'nranks': 4}` for `op1` and `{'nranks': 2, 'executable': 'python3'}` for `op2`.

```
group = FlowProject.make_group(name='example_group')

@group.with_directives(nranks=4)
@FlowProject.operation
@directives(nranks=2, executable="python3")
def op1(job):
    pass

@group
@FlowProject.operation
@directives(nranks=2, executable="python3")
def op2(job):
    pass
```

Parameters

- **name** (*str*) – The name of the group to be used when calling from the command line.
- **operations** (*dict*) – A dictionary of operations where the keys are operation names and each value is a `BaseFlowOperation`.
- **operation_directives** (*dict*) – A dictionary of additional parameters that provide instructions on how to execute a particular operation, e.g., specifically required resources. Operation names are keys and the dictionaries of directives are values. If an operation does not have directives specified, then the directives of the singleton group containing

that operation are used. To prevent this, set the directives to an empty dictionary for that operation.

- **options** (*str*) – A string of options to append to the output of the object’s call method. This allows options like `--num_passes` to be given to a group.

add_operation (*name*, *operation*, *directives=None*)

Add an operation to the *FlowGroup*.

Parameters

- **name** (*str*) – The name of the operation.
- **operation** (*BaseFlowOperation*) – The workflow operation to add to the *FlowGroup*.
- **directives** (*dict*) – The operation specific directives. (Default value = None)

isdisjoint (*group*)

Return whether two groups are disjoint.

Groups are disjoint if they do not share any common operations.

Parameters **group** (*FlowGroup*) – The other *FlowGroup* to compare to.

Returns Returns True if *group* and *self* share no operations, otherwise returns False.

Return type bool

class `flow.project.FlowGroupEntry` (*name*, *options=""*)

Bases: `object`

A `FlowGroupEntry` registers operations for inclusion into a *FlowGroup*.

Application developers should not directly instantiate this class, but use `make_group()` instead.

Operation functions can be marked for inclusion into a *FlowGroup* by decorating the functions with a corresponding *FlowGroupEntry*. If the operation requires specific directives, `with_directives()` accepts keyword arguments that are mapped to directives and returns a decorator that can be applied to the operation to mark it for inclusion in the group and indicate that it should be executed using the specified directives. This overrides the default directives specified by `flow.directives()`.

Parameters

- **name** (*str*) – The name of the *FlowGroup* to be created.
- **options** (*str*) – The `FlowProject.run()` options to pass when submitting the group. These will be included in all submissions. Submissions use run commands to execute.

__call__ (*func*)

Add the function into the group’s operations.

This call operator allows the class to be used as a decorator.

Parameters **func** (*callable*) – The function to decorate.

Returns The decorated function.

Return type callable

with_directives (*directives*)

Return a decorator that sets group specific directives to the operation.

Parameters **directives** (*dict*) – Directives to use for resource requests and running the operation through the group.

Returns A decorator which registers the function into the group with specified directives.

Return type function

1.3.12 Compute Environments

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 adjustment of scheduler submission scripts.

`flow.environment.setup(py_modules, **attrs)`
Set up user-defined environment modules.

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

`flow.environment.template_filter(func)`
Decorate a function as a `ComputeEnvironment` template filter.

This decorator is applied to methods defined in a subclass of `ComputeEnvironment` that are used in that environment's templates. The decorated function becomes a class method that is available as a `jinja2` filter in templates rendered by a `FlowProject` with that `ComputeEnvironment`.

Parameters `func` (*callable*) – Function to decorate.

Returns Decorated function.

Return type callable

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

Define computational environments.

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

The default method for the detection of a specific environment 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'`.

classmethod `add_args(parser)`
Add arguments related to this compute environment to an argument parser.

Parameters `parser` (`argparse.ArgumentParser`) – The argument parser where arguments will be added.

classmethod `get_config_value(key, default=<flow.util.config._GetConfigValueNoneType object>)`
Request a value from the user's configuration.

This method should be used whenever values need to be provided that are specific to a user's environment, e.g. 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** (*str*) – A default value in case the key cannot be found within the user's configuration.

Returns The value or default value.

Return type `object`

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

classmethod `get_prefix` (*operation*, *parallel=False*, *mpi_prefix=None*, *cmd_prefix=None*)
Template filter generating a command prefix from directives.

Parameters

- **operation** (`flow.project._JobOperation`) – The operation to be prefixed.
- **parallel** (*bool*) – If True, operations are assumed to be executed in parallel, which means that the number of total tasks is the sum of all tasks instead of the maximum number of tasks. Default is set to False.
- **mpi_prefix** (*str*) – User defined mpi_prefix string. Default is set to None. This will be deprecated and removed in the future.
- **cmd_prefix** (*str*) – User defined cmd_prefix string. Default is set to None. This will be deprecated and removed in the future.

Returns The prefix to be added to the operation's command.

Return type `str`

classmethod `get_scheduler` ()
Return an 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 or delegate the environment check to the associated scheduler type.

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.

Parameters

- **script** (*str*) – The script to submit.
- **flags** (*list*) – A list of additional flags to provide to the scheduler. (Default value = None)
- ***args** – Positional arguments forwarded to the scheduler's submit method.
- ****kwargs** – Keyword arguments forwarded to the scheduler's submit method.

Returns Status of job, if submitted.

Return type JobStatus.submitted or None

class `flow.environment.StandardEnvironment`

Bases: `flow.environment.ComputeEnvironment`

Default environment which is always present.

classmethod `is_present()`

Determine whether this specific compute environment is present.

The StandardEnvironment is always present, so this returns True.

class `flow.environment.NodesEnvironment`

Bases: `flow.environment.ComputeEnvironment`

A compute environment consisting of multiple compute nodes.

Each compute node is assumed to have a specific number of compute units, e.g., CPUs.

class `flow.environment.TestEnvironment`

Bases: `flow.environment.ComputeEnvironment`

Environment used for testing.

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 a real scheduler.

scheduler_type

alias of `flow.scheduling.fake_scheduler.FakeScheduler`

class `flow.environment.SimpleSchedulerEnvironment`

Bases: `flow.environment.ComputeEnvironment`

An environment for the simple-scheduler scheduler.

scheduler_type

alias of `flow.scheduling.simple_scheduler.SimpleScheduler`

class `flow.environment.TorqueEnvironment`

Bases: `flow.environment.ComputeEnvironment`

An environment with TORQUE scheduler.

scheduler_type

alias of `flow.scheduling.torque.TorqueScheduler`

class `flow.environment.SlurmEnvironment`

Bases: `flow.environment.ComputeEnvironment`

An environment with SLURM scheduler.

scheduler_type

alias of `flow.scheduling.slurm.SlurmScheduler`

class `flow.environment.LSFEnvironment`

Bases: `flow.environment.ComputeEnvironment`

An environment with LSF scheduler.

scheduler_type

alias of `flow.scheduling.lsf.LSFScheduler`

`flow.environment.registered_environments` (*import_configured=True*)

Return a list of registered environments.

Parameters `import_configured` (*bool*) – Whether to import environments specified in the flow configuration. (Default value = True)

Returns List of registered environments.

Return type *list*

1.3.13 Schedulers

Defines the API for the scheduling system.

class `flow.scheduling.base.JobStatus`

Bases: `enum.IntEnum`

Classifies the job's execution status.

active = 7

The cluster job is actively running.

error = 8

The cluster job is in an error or failed state.

held = 5

The cluster job is held.

inactive = 3

The cluster job is inactive.

This includes states like completed, cancelled, or timed out.

placeholder = 127

A placeholder state that is used for status rendering when no operations are eligible.

queued = 6

The cluster job is queued.

registered = 2

The cluster job is registered with the scheduler, but no other status is known.

submitted = 4

The cluster job has been submitted.

Note that this state is never returned by a scheduler, but is an assumed state immediately after a cluster job is submitted.

unknown = 1

Unknown cluster job status.

user = 128

All user-defined states must be ≥ 128 in value.

class `flow.scheduling.base.ClusterJob` (*job_id, status=None*)

Bases: `object`

Class representing a cluster job.

name ()

Return the name of the cluster job.

status ()

Return the status of the cluster job.

class `flow.scheduling.base.Scheduler`

Bases: `abc.ABC`

Abstract base class for schedulers.

classmethod `is_present ()`

Return True if the scheduler is detected.

jobs ()

Yield all cluster jobs.

Yields `ClusterJob` – Cluster job.

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

Submit a job script to the scheduler for execution.

class `flow.scheduling.fake_scheduler.FakeScheduler`

Bases: `flow.scheduling.base.Scheduler`

Implementation of the abstract `Scheduler` class for a fake scheduler.

This scheduler does not actually schedule (or execute) any jobs, but it can be used to test the submission workflow.

classmethod `is_present ()`

Return False.

The `FakeScheduler` is never present unless manually specified.

jobs ()

Return None (no jobs are scheduled by the FakeScheduler).

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

Print the script to screen.

Parameters

- **script** (`str`) – Script to print.
- ****kwargs** – Keyword arguments (ignored).

class `flow.scheduling.lsf.LSFScheduler` (`user=None`)

Bases: `flow.scheduling.base.Scheduler`

Implementation of the abstract Scheduler class for LSF schedulers.

This class can submit cluster jobs to a LSF scheduler and query their current status.

Parameters

- **user** (`str`) – Limit the status information to cluster jobs submitted by user.
- ****kwargs** – Forwarded to the parent constructor.

classmethod `is_present ()`

Return True if an LSF scheduler is detected.

jobs ()

Yield cluster jobs by querying the scheduler.

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

Submit a job script for execution to the scheduler.

Parameters

- **script** (`str`) – The job script submitted for execution.

- **after** (*str*) – Execute the submitted script after a job with this id has completed. (Default value = None)
- **hold** (*bool*) – Whether to hold the job upon submission. (Default value = False)
- **pretend** (*bool*) – If True, do not actually submit the script, but only simulate the submission. Can be used to test whether the submission would be successful. Please note: A successful “pretend” submission is not guaranteed to succeed. (Default value = False)
- **flags** (*list*) – Additional arguments to pass through to the scheduler submission command. (Default value = None)
- ****kwargs** – Additional keyword arguments (ignored).

Returns True if the submission command succeeds (or in pretend mode).

Return type bool

Raises *SubmitError* – If the submission command fails.

class `flow.scheduling.simple_scheduler.SimpleScheduler`

Bases: `flow.scheduling.base.Scheduler`

Implementation of the abstract Scheduler class for the bundled `simple-scheduler`.

The package `signac-flow` includes a script in `bin/simple-scheduler` that is a simple model of a cluster job scheduler. The `simple-scheduler` script is designed primarily for testing and demonstration.

This class can submit cluster jobs to the built-in simple scheduler and query their current status.

classmethod `is_present()`

Return True if a SimpleScheduler is detected.

jobs ()

Yield cluster jobs by querying the scheduler.

submit (*script*, *, *pretend=False*, ***kwargs*)

Submit a job script for execution to the scheduler.

Parameters

- **script** (*str*) – The job script submitted for execution.
- **pretend** (*bool*) – If True, do not actually submit the script, but only simulate the submission. Can be used to test whether the submission would be successful. A successful “pretend” submission is not guaranteed to succeed. (Default value = False)
- ****kwargs** – Keyword arguments (ignored).

Returns True if the submission command succeeds (or in pretend mode).

Return type bool

Raises *SubmitError* – If the submission command fails.

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

Bases: `flow.scheduling.base.Scheduler`

Implementation of the abstract Scheduler class for SLURM schedulers.

This class can submit cluster jobs to a SLURM scheduler and query their current status.

Parameters

- **user** (*str*) – Limit the status information to cluster jobs submitted by user.
- ****kwargs** – Forwarded to the parent constructor.

classmethod `is_present()`

Return True if a SLURM scheduler is detected.

jobs()

Yield cluster jobs by querying the scheduler.

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

Submit a job script for execution to the scheduler.

Parameters

- **script** (*str*) – The job script submitted for execution.
- **after** (*str*) – Execute the submitted script after a job with this id has completed. (Default value = None)
- **hold** (*bool*) – Whether to hold the job upon submission. (Default value = False)
- **pretend** (*bool*) – If True, do not actually submit the script, but only simulate the submission. Can be used to test whether the submission would be successful. Please note: A successful “pretend” submission is not guaranteed to succeed. (Default value = False)
- **flags** (*list*) – Additional arguments to pass through to the scheduler submission command. (Default value = None)
- ****kwargs** – Additional keyword arguments (ignored).

Returns True if the submission command succeeds (or in pretend mode).

Return type `bool`

Raises `SubmitError` – If the submission command fails.

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

Bases: `flow.scheduling.base.Scheduler`

Implementation of the abstract Scheduler class for TORQUE schedulers.

This class can submit cluster jobs to a TORQUE scheduler and query their current status.

Parameters

- **user** (*str*) – Limit the status information to cluster jobs submitted by user.
- ****kwargs** – Forwarded to the parent constructor.

classmethod `is_present()`

Return True if a TORQUE scheduler is detected.

jobs()

Yield cluster jobs by querying the scheduler.

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

Submit a job script for execution to the scheduler.

Parameters

- **script** (*str*) – The job script submitted for execution.
- **after** (*str*) – Execute the submitted script after a job with this id has completed. (Default value = None)
- **hold** (*bool*) – Whether to hold the job upon submission. (Default value = False)
- **pretend** (*bool*) – If True, do not actually submit the script, but only simulate the submission. Can be used to test whether the submission would be successful. Please note: A successful “pretend” submission is not guaranteed to succeed. (Default value = False)

- **flags** (*list*) – Additional arguments to pass through to the scheduler submission command. (Default value = None)
- ****kwargs** – Additional keyword arguments (ignored).

Returns True if the submission command succeeds (or in pretend mode).

Return type `bool`

Raises `SubmitError` – If the submission command fails.

1.3.14 Error Classes

Definitions of Exception classes used in this package.

exception `flow.errors.ConfigKeyError`

Bases: `KeyError`

Indicates that a config key was not found.

exception `flow.errors.DirectivesError`

Bases: `ValueError`

Indicates that a directive was incorrectly set.

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.

class `flow.errors.TemplateError` (*environment*)

Bases: `jinja2.ext.Extension`

Indicates an error in a jinja2 template.

err (*msg, caller*)

Raise a template error.

parse (*parser*)

Call `err()` when a template raises an Exception.

exception `flow.errors.UserConditionError`

Bases: `RuntimeError`

Indicates an error during evaluation of a condition.

exception `flow.errors.UserOperationError`

Bases: `RuntimeError`

Indicates an error during execution of a `BaseFlowOperation`.

Feature Deprecation

Deprecated features are supported for *at least* one more minor release. A detailed deprecation policy is outlined [here](#).

1.4 Changelog

The **signac-flow** package follows [semantic versioning](#). The numbers in brackets denote the related GitHub issue and/or pull request.

1.4.1 Version 0.13

[0.13.0] – 2021-xx-xx

Added

- Add official support for Bridges-2 cluster (#441).

Fixed

- Support for multi-line `@flow.cmd` operations (#451, #453).
- FlowProject status shows labels and correct number of jobs for projects with zero operations (#454, #460).

Removed

- Removed public API of deprecated class `JobOperation` (#445).
- Removed public API of deprecated methods `eligible` and `complete` of `BaseFlowOperation` and `FlowGroup` (#445).
- Removed configuration option `use_buffered_mode` (#445).
- Removed public API of `script`, `next_operations` and `submit_operations` of `FlowProject` (#445).
- Removed support for decommissioned Bridges cluster (#441).

1.4.2 Version 0.12

[0.12.0] – 2021-01-30

Added

- Code is formatted with `black` and `isort` pre-commit hooks (#365).
- Add official support for Python version 3.9 (#365).
- Documentation has been added for all public classes and methods (#387, #389).
- Added internal support for aggregates of jobs (#334, #348, #351, #364, #383, #390, #415, #422, #430).
- Added code coverage to continuous integration (#405).

Changed

- Command line interface always uses `--job-id` instead of `--jobid` (#363, #386).
- `CPUEnvironment` and `GPUEnvironment` classes are deprecated (#381).
- Docstrings are now written in `numpydoc` style (#392).
- Default environment for the University of Minnesota Mangi cluster changed from SLURM to Torque (#393).
- Run commands are evaluated lazily (#70, #396).
- Deprecated method `export_job_statuses` (#402).
- Improved internal caching of scheduler status (#410).
- Refactored status fetching code (#368, #417).
- Optimization: Directives are no longer deep-copied (#420, #421).
- The `use_buffered_mode` config option is deprecated. Buffering is always internally enabled (#425).
- Evaluate directives when called instead of when defined (#398, #402).
- Various internal refactorings and optimizations (#371, #373, #374, #375, #376, #377, #378, #379, #380, #400, #410, #416, #423, #426).
- Scheduler is now an abstract base class (#426).
- `flow.scheduling.fakescheduler` has been renamed to `flow.scheduling.fake_scheduler` (#426).
- Arguments to `submit` have been changed for all scheduler classes (#426).
- Python 3.6 is only tested with oldest dependencies (#436).
- Drop support for `tqdm` versions older than *4.48.1* (#436, #440).
- Drop support for `Jinja2` versions older than *2.10.0* (#436).
- Deprecated `operations.py` method `flow.run` (#221, #463, #467).

Fixed

- Ensure that directives are always evaluated before running or submitting (#408, #409).
- Cache the fully qualified domain name during environment detection to fix a performance issue on macOS (#339, #394).
- Ensure that `next` CLI command displays eligible jobs for the exact operation name provided (#443).
- Display warning when a non-existing operation/group is passed to `run`, `submit`, or `next` commands (#291, #442).

Removed

- Removed the deprecated method `flow.util.misc.write_human_readable_statepoints` (#397).
- Removed the deprecated argument `--no-parallelize` (#424).
- Removed the deprecated `env` argument from submission methods (#424).
- `flow.render_status.Renderer` class has been removed. `FlowProject.print_status` no longer returns the renderer (#426).

- Removed deprecated `status.py` module (#426).
- Removed the `--test` argument from `FlowProject.submit` (#439).

1.4.3 Version 0.11

[0.11.0] – 2020-10-09

Added

- Added classes `_Directives` and `_Directive` that serve as a smart mapping for directives specified by the environment or user (#265, #283).
- Added support for [pre-commit hooks](#) (#333).
- Add environment profile for University of Minnesota, Minnesota Supercomputing Institute, Mangi supercomputer (#353).

Changed

- Make `FlowCondition` class private (#307, #315).
- Deprecate `JobOperation` class, make `SubmissionJobOperation` a private class and deprecate the following methods of `FlowProject`: `script`, `run_operations`, `submit_operations`, `next_operations`. (#313)
- Deprecate the following methods: `FlowGroup.eligible`, `FlowGroup.complete`, `BaseFlowOperation.eligible`, `BaseFlowOperation.complete` (#337).

Fixed

- Serial execution on Summit correctly counts total node requirements (#342).
- Fixed performance regression in job submission in large workspaces (#354).

Removed

- Drop support for Python 3.5 (#305). The signac project will follow the [NEP 29 deprecation policy](#) going forward.
- Remove the deprecated methods `always`, `make_bundles`, and `JobOperation.get_id` (#312).

1.4.4 Version 0.10

[0.10.1] – 2020-08-20

Fixed

- Fix issue with the submission of bundled operations on cluster environments that do not allow slashes (`/`) in cluster scheduler job names (#343).

[0.10.0] – 2020-06-27

Added

- Add `FlowGroup` (one or more operations can be grouped within an execution environment) (#114).
- Add official support for University of Michigan Great Lakes cluster (#185).
- Add official support for Bridges AI cluster (#222).
- Add `IgnoreConditions` option for `submit()`, `run()` and `script()` (#38, #209).
- Add `pytest` support for testing framework (#227, #232).
- Add `markdown` and `html` format support for `print_status()` (#113, #163).
- Add `memory` flag option for default Slurm scheduler (#256).
- Add optional environment variable to specify submission script separator (#262).
- Add `status_parallelization` configuration to specify the parallelization used for fetching status (#264, #271).

Changed

- Raises `ValueError` when an operation function is passed to `FlowProject.pre()` and `FlowProject.post()`, or a non-operation function passed to `FlowProject.pre.after()` (#248, #249).
- The option to provide the `env` argument to `submit` and `submit_operations` has been deprecated (#245).
- The command line option `--cmd` for `script` has been deprecated and will trigger a `DeprecationWarning` upon use until removed (#243, #218).
- Raises `ValueError` when `--job-name` is passed by the user because that interferes with status checking (#164, #241).
- Submitting with `--memory` no longer assumes a unit of gigabytes on Bridges and Comet clusters (#257).
- Buffering is enabled by default, improving the performance of status checks (#273).
- Deprecate the use of `no_parallelize` argument while printing status (#264, #271).
- Submission via the command-line interface now calls the `FlowProject.submit` function instead of bypassing it for `FlowProject.submit_operations` (#238, #286).
- Updated Great Lakes GPU request syntax (#299).

Fixed

- Ensure that label names are used when displaying status (#263).
- Fix node counting for large resource sets on Summit (#294).

Removed

- Removed `ENVIRONMENT` global variable in the `flow.environment` module (#245).
- Removed vendored `tqdm` module and replaced it with a requirement (#247).

1.4.5 Version 0.9

[0.9.0] – 2020-01-09

Added

- Add official support for Python version 3.8 (#190, #210).
- Add descriptive error message when tag is not set and cannot be autogenerated for conditions (#195).
- Add “fork” directive to enforce the execution of operations within a subprocess (#159).
- Operation graph detection based on function comparison (#178).
- Exceptions raised during operations always show tracebacks of user code (#169, #171).

Changed

- Raise a warning when a condition’s tag is not set and raise an error if this occurs during graph detection (#195).
- Raise errors if a forked process or `@cmd` operation returns a non-zero exit code. (#170, #172).

Removed

- Drop support for Python version 2.7 (#157, #158, #201).
- The “always” condition has been deprecated and will trigger a `DeprecationWarning` upon use until removed (#179).
- Removed deprecated `UnknownEnvironment` in favor of `StandardEnvironment` (#204).
- Removed support for decommissioned INCITE Titan and Eos computers (#204).
- Removed support for the legacy Python-based submission script generation (#200).
- Removed legacy compatibility layers for Python 2, `signac < 1.0`, and soft dependencies (#205).
- Removed deprecated support for implied operation names with the `run` command (#205).

1.4.6 Version 0.8

[0.8.0] – 2019-09-01

Added

- Add feature for integrated profiling of status updates (`status --profile`) to aid with the optimization of a `FlowProject` implementation (#107, #110).
- The status view is generated with Jinja2 templates and thus more easily customizable (#67, #111).
- Automatically show an overview of the number of eligible jobs for each operation in status view (#134).
- Allow the provision of multiple operation-functions to the `pre.after` and `*.copy_from` conditions (#120).
- Add option to specify the operation execution order (#121).
- Add a testing module to easily initialize a test project (#130).

- Enable option to always show the full traceback with `show_traceback = on` within the `[flow]` section of the signac configuration (#61, #144).
- Add full launcher support for job submission on XSEDE Stampede2 for large parallel single processor jobs (#85, #91).

Fixed

- Both the `n ranks` and `omp_num_threads` directives properly support callables (#118).
- Show submission error messages in combination with a TORQUE scheduler (#103, #104).
- Fix issue that caused the “Fetching operation status” progress bar to be inaccurate (#108).
- Fix erroneous line in the torque submission template (#126).
- Ensure default parameter range detection in status printing succeeds for nested state points (#154).
- Fix issue with the resource set calculation on INCITE Summit (#101).

Changed

- Packaged environments are now available by default. Set `import_packaged_environments = off` within the `[flow]` section of the signac configuration to revert to previous behavior.
- The following methods of the `FlowProject` class have been deprecated and will trigger a `DeprecationWarning` upon use until their removal:
 - `classify` (use `labels()` instead)
 - `next_operation` (use `next_operations()` instead)
 - `export_job_stati` (replaced by `export_job_statuses`)
 - `eligible_for_submission` (removed without replacement)
 - `update_aliases` (removed without replacement)
- The support for Python version 2.7 is deprecated.

Removed

- The support for Python version 3.4 has been dropped.
- Support for signac version 0.9 has been dropped.

1.4.7 Version 0.7

[0.7.1] – 2019-03-25

Added

- Add function to automatically print all varying state point parameters in the detailed status view triggered by providing option `-p/--parameters` without arguments (#19, #87).
- Add clear environment notification when submitting job scripts (#43, #88).

Fixed

- Fix issue where the scheduler status of job-operations would not be properly updated for *ineligible* operations (#96).

Fixed (compute environments)

- Fix issue with the TORQUE scheduler that occurred when there was no job scheduled at all on the system (for any user) (#92, #93).

Changed

- The performance of status updates has been significantly improved (up to a factor of 1,000 for large data spaces) by applying a more efficient caching strategy (#94).
- Add clear environment notification when submitting job scripts.

[0.7.0] – 2019-03-14

Added

- Add legend explaining the scheduler-related symbols to the detailed status view (#68).
- Allow the specification of the number of tasks per resource set and additional jsrun arguments for Summit scripts.

Fixed (general)

- Fixes issue where callable cmd-directives were not evaluated (#47).
- Fixes issue where the source file of wrapped functions was not determined correctly (#55).
- Fix a Python 2.7 incompatibility and another unrelated issue with the TORQUE scheduler driver (#54, #81).
- Fixes issue where providing the wrong argument type to Project.submit() would go undetected and lead to unexpected behavior (#58).
- Fixes issue where using the buffered mode would lead to confusing error messages when condition-functions would raise an AttributeError exception.
- Fixes issue with erroneous unused-directive-keys-warning.

Fixed (compute environments)

- Fixes issues with the Summit environment resource set calculation for parallel operations under specific conditions (#63).
- Fix the node size specified in the template for the ORNL Eos system (#77).
- Fixes issue with a missing `--gres` directive when using the GPU-shared partition on the XSEDE Bridges system (#59).
- Fixed University of Michigan Flux hostname pattern to ignore the Flux Hadoop cluster (#82).

- Remove the Ascent environment (host decommissioned).

Note: The official support for Python 3.4 will be dropped beginning with version 0.8.0.

1.4.8 Version 0.6

Major changes

1. The generation of execution and submission scripts is now based on the `jinja2` templating system.
2. The new decorator API for the definition of a `FlowProject` class largely reduces the amount of boiler plate code needed to implement FlowProjects. It also removes the necessity to have at least two modules, *e.g.*, one `project.py` and one `operations.py` module.
3. Serial execution is now the default for all project sub commands, that includes `run`, `script`, and `submit`. Parallel execution must be explicitly enabled with the `--parallel` option.
4. The `run` command executes **all eligible** operations, that means you don't have to run the command multiple times to "cycle" through all pending operations. Accidental infinite loops are automatically avoided.
5. Execution scripts generated with the `script` option are always bundled. The previous behavior, where the `script` command would print multiple scripts to screen unless the `--bundle` option was provided did not make much sense.

See the full [changelog](#) below for detailed information on all changes.

How to migrate existing projects

If your project runs with flow 0.5 without any DeprecationWarnings (that means no messages when running Python with the `-W` flag), then you don't *have* to do anything. Version 0.6 is mostly backwards compatible to 0.5, with the exception of custom script templating.

Since 0.6 uses `jinja2` to generate execution and submission scripts, the previous method of generating custom scripts by overloading the `FlowProject.write_script*()` methods is now deprecated. That means that if you over-loaded any of these functions, the new templating system is disabled, and **flow** will fallback to the old templating system and you won't be able to use `jinja2` templates.

If you decide to migrate to the new API, those are the steps you most likely have to take:

1. Replace all `write_script*()` methods and replace them with a custom template script, *e.g.*, `templates/script.sh` within your project root directory.
2. Optionally, use the new decorator API instead of `FlowProject.add_operation` to add operations to your `FlowProject`.
3. Optionally, use the new decorator API to define label functions for your `FlowProject`.
4. Execute your project with the Python `-w` option to make `DeprecationWarnings` visible and address all issues.

We recommend to go through the tutorial on [signac-docs](#) to learn how to best take advantage of **flow** 0.6.

[0.6.4] – 2018-12-28

- Add the `@with_job` decorator that allows the definition of operations to take place within the `job` context. Works with `@cmd`.
- Add the `not_` condition prefix to negate condition functions.
- Add the `false` condition prefix as analogue to the `true` condition prefix.

- Add support for the Summit supercomputer (U.S. DOE, Oak Ridge National Laboratory) and Ascent testing cluster.
- Add support for the IBM LSF scheduler.
- Add warning about explicitly set, but unused directives during submission.
- Add official support for Python version 3.7.
- Fix issue where the status sub-command ignored the `--show-traceback` option.
- Fix SLURM scheduler driver to show full error message in case that submission with `squeue` failed.
- Better specification of (optional) dependencies in `setup.py` and `requirements.txt`.
- Overall revision of all cluster submission templates; improved structure and abstraction of logic.
- The serialization of operations was improved to optimize execution speed for local runs.
- The evaluation of preconditions and postconditions was optimized for optimally lazy evaluation: cheaper conditions should be placed above more expensive conditions for maximal performance.
- When gathering operations, signac will automatically use the buffered mode when config value `'flow.use_buffered_mode'` is set to True (requires signac \geq 0.9.3).
- Improved documentation for developers and contributors.

[0.6.3] – 2018-08-22

- Fix issue related to dynamic data spaces, that means data spaces where jobs are either added, removed, or changed during the execution of the workflow. Specifically, flow will now execute operations also for jobs that were added during execution.
- Fix issue where command line options would be ignored when provided before the sub-command.
- Fix issue where the table symbols in the `--stack --pretty` view were swapped.

[0.6.2] – 2018-08-11

- Increase performance of condition evaluation (switch from eager to lazy evaluation). Speeds up detailed status update and `run/script/submit` sub commands.
- Fix issue with the detailed status update failing on older Python versions (#29).
- Fix issue with the XSEDE Bridges template in combination with GPU operations.

[0.6.1] – 2018-07-01

- Add the `-v/--verbosity` and `--show-traceback` option to the project interface, which allows for more fine-grained control over the message verbosity. The `--debug` option is now equivalent to `-vv --show-traceback`.
- The message verbosity of the project class was overall reduced.
- Global options including (`--debug` and `--verbose`) can be used at any place within the project command and must no longer be placed before the sub command, e.g., the following commands are equivalent:
`$ python project.py run --debug` and `$ python project.py --debug run.`
- Implement the `-p/--parallel` option for the project run command.
- Use cloudpickle when encountering pickling issues during parallel execution (when installed).

- Implement the status `--ignore-errors` option.
- Handle changes to the project data space during running execution, *e.g.*, removed jobs.
- Print the `operation.cmd` attribute to screen in `run --pretend mode`, not `repr(operation)`.
- Show progressbar while gathering pending operations.

[0.6.0] – 2018-05-24

Major updates and new features

- Use jinja2 as templating engine for the generation of execution and submission scripts.
- Add decorator API for the definition of FlowProject operations and label functions.
- Revise the status view to render on a per job-operation basis, not on a per job basis.
- The `<project> run` function executes all pending operations, not just the next pending ones.
- The `<project> script` function no longer supports explicit bundling, all operations are bundled by default.
- The default execution mode for script and submission script bundling is serial, not parallel.
- Add the `operations` directive parameter, which provides a more generalized interface to specify required resources or other execution related metadata.
- Add support for XSEDE Stampede2.
- Add simple-scheduler, for local testing of scheduled workflows.
- Allow the override of the detected environment with the `SIGNAC_FLOW_ENVIRONMENT` variable.
- The `$flow init` command initializes the signac project if not project is found.

API changes (breaking)

- The `FlowProject.run()` method arguments were changed [1]; the old API is better supported by the new `FlowProject.run_operations()` function.
- The `FlowProject.submit()` and `.submit_operations()` method arguments were changed [1].
- The `JobOperation` constructor arguments were changed; the old API is still supported.

API changes (non-breaking)

- Unify the job and operation selection API for the `run/script/submit` commands.
- Add `FlowProject.operation()` decorator function.
- Add `FlowProject.label()` decorator function.
- The `FlowProject.write_human_readable_statepoints()` method is deprecated.
- All `FlowProject` methods relating to the old templating system are deprecated, that includes all `write_script*()` methods.
- Add `flow.cmd` decorator function.
- Add `flow.directives` decorator function.

[1] A reasonable attempt to support legacy API use is made, but may fail under some circumstances.

1.4.9 Version 0.5

[0.5.6] – 2018-02-22

- Fix issue, where operations with spaces in their name would not be accepted by the SLURM scheduler.
- Add environment profile for XSEDE Bridges.
- Update the environment profile for XSEDE comet to use the shared queue by default and provide options to specify the memory allocation.
- Improve performance of the project update status function.

[0.5.5] – 2017-10-05

- Fix issue with the SLURM scheduler, where the queue status could not be parsed.

[0.5.4] – 2017-08-01

- Fix issue with `<project> run`, where operation commands consist of multiple shell commands.
- Fix issue where the `<project> status` output showed negative values for the number of lines omitted (issue #12).
- Raise error when trying to provide a timeout for `<project> run` in serial execution in combination with Python 2.7; this mode of execution is not supported for Python versions 2.7.x.
- Enforce that the `<project> status --overview-max-lines (-m)` argument is positive.

[0.5.3] – 2017-07-18

- Fix issue where the return value of `FlowProject.next_operation()` is ignored in combination with the `<project> submit / run / script` interface.

[0.5.2] – 2017-07-12

- Fix bug in detailed status output in combination with unhashable types.
- Do not fork when executing only a single operation with `flow.run()`.
- Run all next operations for each job with `flow.run()` instead of only one of the next operations.
- Gather all next operations when submitting, instead of only one of the next operations for each job.

[0.5.1] – 2017-06-08

- Exclude private functions, that means functions with a name that start with an underscore, from the operations listing when using `flow.run()`.
- Forward all extra submit arguments into the `write_script()` methods.
- Fix an issue with `$flow init/flow.init()` in combination with Python 2.7.

[0.5.0] – 2017-05-24

Major updates and new features

- The documentation has been completely revised; most components are now covered by a reference documentation; the reference documentation serves also as basic tutorial.
- The signac-flow package now officially supports Python version 2.7 in addition to versions 3.4+; the support for version 3.3 has been dropped.
- Add comand line interface for instances of *FlowProject*, to be accessed via the *FlowProject.main()* function. This makes it easier to interact with specific workflow implementations on the command line, for example to view the project's status, execute operations or submit them to a scheduler.
- The *\$ flow init* command initializes a new very lean workflow module that replaces the need to use project templates. Setting up a workflow with signac-flow is now much easier; template projects are no longer needed. The *\$ flow init* command can be invoked with an optional *-t/-template* argument to initialize project modules with example code.
- Add the *flow.run()* function to turn python modules that implement functions to be used as data space operations into executables. Executing the *flow.run()* function opens a command line interface that can be used to execute operations defined within the same module directly from the command line in serial and parallel.
- The definition of operations on the project level is now possible via the *FlowProject.operations* dictionary; operations can either be added directly or via the *FlowProject.add_operation()* function.
- Environment with torque or slurm scheduler are now immediately supported via a default environment profile.
- The submission process is generally streamlined and it is easier to forward arguments to the underlying scheduler; this is is supposed to enable the user to directly submit scripts and operations without the need to setup a custom environment profile.
- Some environment profiles for large cluster environments are bundled with the package; it is no longer needed to install external packages to be able to use profiles on some HPC resources.

API changes (breaking)

- The use of *JobScript.write_cmd()* with an *np* argument is pending deprecation, the adjustment of commands to the local environment is moved to an earlier stage (for instance, during project instance construction).
- The official project template is still functional via a legacy API layer, however it is recommended that users update projects to use with this version; the update process is described in the README document.
- Most of the environment specific command line arguments are now directly provided by the environment profile via profile specific *add_parser_args()* functions; that means that existing environment might be require some tweaking to work with this version.

1.4.10 Version 0.4

[0.4.2] – 2017-02-28

- Fix issue in the submit legacy mode, the *write_header()* method was previously ignored.

[0.4.1] – 2017-02-24

- Fix ppn issue when submitting in legacy mode.
- Enable optional parallelization during submit.

[0.4.0] – 2017-02-23

Major revision to the job-operation submission function.

- The `write_user()` function has been replaced by `submit_user()` with slightly adjusted API.
- The header and environment module have been merged into a single environment module.
- All submit logic has been removed from the scheduler drivers.
- Any submit logic implemented as part of the environment module has been reduced to the bare minimum.
- The submission flow has been refactored to be based on `JobOperations`.
- An attempt is made to detect the use of the deprecated API which will trigger the use of a legacy code path and the emission of warnings.
- Improved testing capabilities for unknown environments.
- The determination of present environments is deterministic and based on reversed definition order.
- Add the label decorators, allowing a more concise definition of methods which are to be used for classification.
- Add the `FlowGraph`, which allows the user to implement workflows in form of a graph.
- Implement unit tests for all core functionalities.

1.5 Support and Development

1.5.1 Getting help and reporting issues

To get help using the **signac-flow** package, either send an email to signac-support@umich.edu or join the **signac** gitter chatroom.

The **signac-flow** package is hosted on [GitHub](#) and licensed under the open-source BSD 3-Clause license. Please use the repository's [issue tracker](#) to report bugs or request new features.

1.5.2 Code contributions

This project is open-source. Users are highly encouraged to contribute directly by implementing new features and fixing issues. Development for packages as part of the **signac** framework should follow the general development guidelines outlined [here](#).

A brief summary of contributing guidelines are outlined in the [CONTRIBUTING.md](#) file as part of the repository. All contributors must agree to the [Contributor Agreement](#) before their pull request can be merged.

Set up a development environment

Start by [forking](#) the project.

We highly recommend to setup a dedicated development environment, for example with [venv](#):

```
~ $ python -m venv ~/envs/signac-flow-dev
~ $ source ~/envs/signac-flow-dev/bin/activate
(signac-flow-dev) ~ $ pip install -r requirements-dev.txt -r requirements-precommit.
↳txt
```

or alternatively with [conda](#):

```
~ $ conda create -n signac-flow-dev python --file requirements-dev.txt --file_
↳requirements-precommit.txt
~ $ conda activate signac-flow-dev
```

Then clone your fork and install the package from source with:

```
(signac-flow-dev) ~ $ cd path/to/my/fork/of/signac-flow
(signac-flow-dev) signac-flow $ pip install -e .
```

The `-e` option stands for *editable*, which means that the package is directly loaded from the source code repository. That means any changes made to the source code are immediately reflected upon reloading the Python interpreter.

The [pre-commit](#) tool is used to enforce code style guidelines. To install the tool and configure pre-commit hooks, execute:

```
(signac-flow-dev) signac $ pip install pre-commit
(signac-flow-dev) signac $ pre-commit install
```

With the pre-commit hook, your code will be checked for syntax and style before you make a commit. The continuous integration pipeline for the package will perform these checks as well, so running these tests before committing / pushing will prevent the pipeline from failing due to style-related issues.

The development workflow

Prior to working on a patch, it is advisable to create an [issue](#) that describes the problem or proposed feature. This means that the code maintainers and other users get a chance to provide some input on the scope and possible limitations of the proposed changes, as well as advise on the actual implementation.

All code changes should be developed within a dedicated git branch and must all be related to each other. Unrelated changes, such as minor fixes to unrelated bugs encountered during implementation, spelling errors, and similar typographical mistakes must be developed within a separate branch.

Branches should be named after the following pattern: `<prefix>/issue-<#>-optional-short-description`. Choose from one of the following prefixes depending on the type of change:

- `fix/`: Any changes that fix the code and documentation.
- `feature/`: Any changes that introduce a new feature.
- `release/`: Reserved for release branches.

If your change does not seem to fall into any of the above mentioned categories, use `misc/`.

Once you are content with your changes, push the new branch to your forked repository and create a pull request into the main repository. Feel free to push a branch before completion to get input from the maintainers and other users, but make sure to add a comment that clarifies that the branch is not ready for merge yet.

Testing

Prior to fixing an issue, implement unit tests that *fail* for the described problem. New features must be tested with unit and integration tests. To run tests, execute:

```
(signac-flow-dev) signac-flow $ python -m pytest tests/
```

Building documentation

Building documentation requires the `sphinx` package which you will need to install into your development environment.

```
(signac-flow-dev) signac-flow $ pip install Sphinx sphinx_rtd_theme
```

Then you can build the documentation from within the `doc/` directory as part of the source code repository:

```
(signac-flow-dev) signac-flow $ cd doc/
(signac-flow-dev) doc $ make html
```

Note: Documentation as part of the package should be largely limited to the API. More elaborate documentation on how to integrate **signac-flow** into a computational workflow should be documented as part of the [framework documentation](#), which is maintained [here](#).

Updating the changelog

To update the changelog, add a one-line description to the `changelog.txt` file within the `next` section. For example:

```
next
----

- Fix issue with launching rockets to the moon.

[0.6.3] -- 2018-08-22
-----

- Fix issue related to dynamic data spaces, ...
```

Just add the `next` section in case it doesn't exist yet.

Contributing Environments to the Package

Users are also **highly encouraged** to contribute environment profiles that they developed for their local environments. While there are a few steps, they are almost all entirely automated, with the exception of actually reviewing the scripts your environment generates.

Before you begin the process, make sure you have the following packages installed (in addition to **signac-flow**):

1. `python-docx`
2. `GitPython`

Once you've written the environment class and the template as described above, contributing the environments to the package involves the following:

1. Create a new branch of **signac-flow** based on the *master* branch.
2. Add your environment class to the *flow/environments/* directory, and add the corresponding template to the *flow/templates/* directory.
3. Run the *tests/test_templates.py* test script. It should fail on your environment, indicating that no reference scripts exist yet.
4. Update the *environments* dictionary in the *init* function of *tests/generate_template_reference_data.py*. The dictionary indicates the submission argument combinations that need to be tested for your environment.
5. Run the *tests/generate_template_reference_data.py* script, which will create the appropriate reference data in the *tests/template_reference_data.tar.gz* tarball based on your modifications. The *test_templates.py* script should now succeed.
6. Run the *tests/extract_templates.py* script, which will extract the tarball into a **signac** project folder.
7. Run the *tests/generate_template_review_document.py* script, which will generate docx files in the *tests/compiled_scripts/* directory, one for each environment.
8. You should see one named after your new environment class. **Review the generated scripts thoroughly.** This step is critical, as it ensures that the environment is correctly generating scripts for various types of submission.
9. Once you've fixed any issues with your environment and template, push your changes and create a pull request. You're done!

Deprecation Policy

While the signac-flow API is not considered stable yet (a *1.0* release has not been made), we apply the following deprecation policy:

Some features may be deprecated in future releases in which case the deprecation is announced as part of the documentation, the change log, and their use will trigger warnings. A deprecated feature is removed in the next minor version, unless it is considered part of the core API in which case a reasonable attempt at maintaining backwards compatibility is made in the next minor version, but is then completely removed in any following minor or major release.

A feature is considered to be part of the core API if it is likely to be used by the majority of existing projects.

A feature which is deprecated in version *0.x*, will trigger warnings for all releases with release number *0.x.**, and will be removed in version *0.y.0*. A feature, which is deprecated in version *0.x* and which is considered core API will trigger warnings for versions *0.x.** and *0.y.**, limited backwards compatibility will be maintained throughout versions *0.y.**, and the feature will be removed in version *0.z.0*.

For example: A feature deprecated in version 0.6, will be removed in version 0.7, unless it is considered core API, in which case, some backwards compatibility is maintained in version 0.7, and it is removed in version 0.8.

CHAPTER 2

Indices and tables

- `genindex`
- `modindex`
- `search`

f

`flow.environment`, 30
`flow.errors`, 37
`flow.scheduling`, 33

Symbols

__call__() (*flow.classlabel* method), 25
 __call__() (*flow.directives* method), 26
 __call__() (*flow.label* method), 24
 __call__() (*flow.project.FlowCmdOperation* method), 24
 __call__() (*flow.project.FlowGroupEntry* method), 29
 __call__() (*flow.project.FlowOperation* method), 23
 __call__() (*flow.staticlabel* method), 25

A

active (*flow.scheduling.base.JobStatus* attribute), 33
 add_args() (*flow.environment.ComputeEnvironment* class method), 30
 add_operation() (*flow.FlowProject* method), 14
 add_operation() (*flow.project.FlowGroup* method), 29
 after() (*flow.FlowProject.pre* class method), 22
 ALIASES (*flow.FlowProject* attribute), 14
 ALL (*flow.IgnoreConditions* attribute), 22

B

BaseFlowOperation (class in *flow.project*), 23
 Bridges2Environment (class in *flow.environments.xsede*), 10

C

classlabel (class in *flow*), 25
 ClusterJob (class in *flow.scheduling.base*), 33
 cmd() (in module *flow*), 25
 CometEnvironment (class in *flow.environments.xsede*), 9
 completed_operations() (*flow.FlowProject* method), 15
 ComputeEnvironment (class in *flow.environment*), 30
 ConfigKeyError, 37
 copy_from() (*flow.directives* class method), 26

copy_from() (*flow.FlowProject.post* class method), 21
 copy_from() (*flow.FlowProject.pre* class method), 22

D

DefaultLSFEnvironment (class in *flow.environment*), 6
 DefaultSlurmEnvironment (class in *flow.environment*), 5
 DefaultTorqueEnvironment (class in *flow.environment*), 6
 detect_operation_graph() (*flow.FlowProject* method), 15
 directives (class in *flow*), 26
 DirectivesError, 37

E

err() (*flow.errors.TemplateError* method), 37
 error (*flow.scheduling.base.JobStatus* attribute), 33
 export_job_statuses() (*flow.FlowProject* method), 16

F

FakeScheduler (class in *flow.scheduling.fake_scheduler*), 34
 false() (*flow.FlowProject.post* class method), 21
 false() (*flow.FlowProject.pre* class method), 22
 flow.environment (module), 30
 flow.errors (module), 37
 flow.scheduling (module), 33
 FlowCmdOperation (class in *flow.project*), 24
 FlowGroup (class in *flow.project*), 28
 FlowGroupEntry (class in *flow.project*), 29
 FlowOperation (class in *flow.project*), 23
 FlowProject (class in *flow*), 14

G

get_config_value() (*flow.environment.ComputeEnvironment* class method), 30

get_environment() (in module flow), 28

get_job_status() (flow.FlowProject method), 16

get_prefix() (flow.environment.ComputeEnvironment class method), 31

get_scheduler() (flow.environment.ComputeEnvironment class method), 31

GreatLakesEnvironment (class in flow.environments.umich), 11

groups (flow.FlowProject attribute), 16

H

held (flow.scheduling.base.JobStatus attribute), 33

I

IgnoreConditions (class in flow), 22

inactive (flow.scheduling.base.JobStatus attribute), 33

init() (in module flow), 27

is_present() (flow.environment.ComputeEnvironment class method), 31

is_present() (flow.environment.StandardEnvironment class method), 32

is_present() (flow.scheduling.base.Scheduler class method), 34

is_present() (flow.scheduling.fake_scheduler.FakeScheduler class method), 34

is_present() (flow.scheduling.lsf.LSFScheduler class method), 34

is_present() (flow.scheduling.simple_scheduler.SimpleScheduler class method), 35

is_present() (flow.scheduling.slurm.SlurmScheduler class method), 35

is_present() (flow.scheduling.torque.TorqueScheduler class method), 36

isdisjoint() (flow.project.FlowGroup method), 29

isfile() (flow.FlowProject.post class method), 21

isfile() (flow.FlowProject.pre class method), 22

J

jobs() (flow.scheduling.base.Scheduler method), 34

jobs() (flow.scheduling.fake_scheduler.FakeScheduler method), 34

jobs() (flow.scheduling.lsf.LSFScheduler method), 34

jobs() (flow.scheduling.simple_scheduler.SimpleScheduler method), 35

jobs() (flow.scheduling.slurm.SlurmScheduler method), 36

jobs() (flow.scheduling.torque.TorqueScheduler method), 36

JobStatus (class in flow.scheduling.base), 33

L

label (class in flow), 24

label() (flow.FlowProject class method), 17

labels() (flow.FlowProject method), 17

LSFEnvironment (class in flow.environment), 32

LSFScheduler (class in flow.scheduling.lsf), 34

M

main() (flow.FlowProject method), 17

make_group() (flow.FlowProject class method), 18

MangiEnvironment (class in flow.environments.umn), 12

N

name() (flow.scheduling.base.ClusterJob method), 33

never() (flow.FlowProject.post class method), 21

never() (flow.FlowProject.pre class method), 22

NodesEnvironment (class in flow.environment), 32

NONE (flow.IgnoreConditions attribute), 23

NoSchedulerError, 37

not_() (flow.FlowProject.post class method), 22

not_() (flow.FlowProject.pre class method), 22

O

operation() (flow.FlowProject class method), 18

operations (flow.FlowProject attribute), 18

P

parse() (flow.errors.TemplateError method), 37

placeholder (flow.scheduling.base.JobStatus attribute), 33

POST (flow.IgnoreConditions attribute), 23

post() (flow.FlowProject method), 21

PRE (flow.IgnoreConditions attribute), 23

pre() (flow.FlowProject method), 22

print_status() (flow.FlowProject method), 18

PRINT_STATUS_ALL_VARYING_PARAMETERS (flow.FlowProject attribute), 14

Q

queued (flow.scheduling.base.JobStatus attribute), 33

R

registered (flow.scheduling.base.JobStatus attribute), 33

registered_environments() (in module flow.environment), 32

run() (flow.FlowProject method), 19

run() (in module flow), 27

S

Scheduler (class in flow.scheduling.base), 33

scheduler_jobs() (flow.FlowProject method), 20

scheduler_type (flow.environment.LSFEnvironment attribute), 32

scheduler_type (*flow.environment.SimpleSchedulerEnvironment* attribute), 32
scheduler_type (*flow.environment.SlurmEnvironment* attribute), 32
scheduler_type (*flow.environment.TestEnvironment* attribute), 32
scheduler_type (*flow.environment.TorqueEnvironment* attribute), 32
setup() (in module *flow.environment*), 30
SimpleScheduler (class in *flow.scheduling.simple_scheduler*), 35
SimpleSchedulerEnvironment (class in *flow.environment*), 32
SlurmEnvironment (class in *flow.environment*), 32
SlurmScheduler (class in *flow.scheduling.slurm*), 35
Stampede2Environment (class in *flow.environments.xsede*), 8
StandardEnvironment (class in *flow.environment*), 32
staticlabel (class in *flow*), 24
status() (*flow.scheduling.base.ClusterJob* method), 33
submit() (*flow.environment.ComputeEnvironment* class method), 31
submit() (*flow.FlowProject* method), 20
submit() (*flow.scheduling.base.Scheduler* method), 34
submit() (*flow.scheduling.fake_scheduler.FakeScheduler* method), 34
submit() (*flow.scheduling.lsf.LSFScheduler* method), 34
submit() (*flow.scheduling.simple_scheduler.SimpleScheduler* method), 35
submit() (*flow.scheduling.slurm.SlurmScheduler* method), 36
submit() (*flow.scheduling.torque.TorqueScheduler* method), 36
SubmitError, 37
submitted (*flow.scheduling.base.JobStatus* attribute), 33
SummitEnvironment (class in *flow.environments.incite*), 7

T

template_filter() (in module *flow.environment*), 30
TemplateError (class in *flow.errors*), 37
TestEnvironment (class in *flow.environment*), 32
TorqueEnvironment (class in *flow.environment*), 32
TorqueScheduler (class in *flow.scheduling.torque*), 36
true() (*flow.FlowProject.post* class method), 22
true() (*flow.FlowProject.pre* class method), 22