signac-flow Documentation

Release 0.10.0

Carl Simon Adorf, Paul Dodd

Jun 27, 2020

Contents

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*.

CHAPTER 1

Contents

1.1 Installation

The recommended installation method for **signac-flow** is via conda or pip. The software is tested for Python versions 3.5+ and signac versions 1.0+.

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 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 %}
{% 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) }}
{{ operation.cmd }}{{ cmd_suffix }}
{% if operation.eligible_operations|length > 0 %}
# Eligible to run:
{% for run_op in operation.eligible_operations %}
# {{ run_op.cmd }}
{% endfor %}
{% endif %}
{% if operation.operations_with_unmet_preconditions|length > 0 %}
# Operations with unmet preconditions:
{% for run_op in operation.operations_with_unmet_preconditions %}
# {{ run_op.cmd }}
{% endfor %}
{% endif %}
{% if operation.operations_with_met_postconditions|length > 0 %}
# Operations with all postconditions met:
{% for run_op in operation.operations_with_met_postconditions %}
```

(continues on next page)

(continued from previous page)

```
# {{ run_op.cmd }}
{% endfor %}
{% endif %}
{% endfor %}
{% 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**
A default environment for environments with SLURM scheduler.

Listing 2: slurm.sh

```
{% extends "base_script.sh" %}
{% block header %}
#!/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 %}
{% 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**
A default environment for environments with 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**
A default environment for environments with LSF scheduler.

Listing 4: lsf.sh

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

(continues on next page)

(continued from previous page)

```
{% 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}}
{% 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
```

(continues on next page)

(continued from previous page)

```
rm {{ launcher_file }}
{% else %}
{% set cmd_suffix = cmd_suffix|default('') ~ (' &' if parallel else '') %}
{% for operation in operations %}

# {{ "%s"|format(operation) }}
{{ "_FLOW_STAMPEDE_OFFSET_=%d "|format(operation.directives['nranks']|return_and_
↪increment) }}{{ operation.cmd }}{{ cmd_suffix }}
{% if operation.eligible_operations|length > 0 %}
# Eligible to run:
{% for run_op in operation.eligible_operations %}
# {{ run_op.cmd }}
{% endfor %}
{% endif %}
{% if operation.operations_with_unmet_preconditions|length > 0 %}
# Operations with unmet preconditions:
{% for run_op in operation.operations_with_unmet_preconditions %}
# {{ run_op.cmd }}
{% endfor %}
{% endif %}
{% if operation.operations_with_met_postconditions|length > 0 %}
# Operations with all postconditions met:
{% for run_op in operation.operations_with_met_postconditions %}
# {{ run_op.cmd }}
{% endfor %}
{% endif %}
{% endfor %}
{# 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 %}
```

1.2.7 XSEDE Comet

Link to official documentation

class flow.environments.xsede.**CometEnvironment**

Environment profile for the Comet supercomputer.

http://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 %}
{% set cpu_tasks = operations|calc_tasks('np', parallel, force) %}
{% set gpu_tasks = operations|calc_tasks('ngpu', parallel, force) %}
```

(continues on next page)

(continued from previous page)

```
{% 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

Link to official documentation

class flow.environments.xsede.**BridgesEnvironment**

Environment profile for the Bridges super computer.

https://portal.xsede.org/psc-bridges

Listing 8: bridges.sh

```
{# Templated in accordance with: https://www.psc.edu/bridges/user-guide #}
{# This template can only be used with P100 GPUs on GPU, or V100 GPUs on GPU-AI. #}
{% 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 %}
{% if partition == 'GPU-AI' %}
{# GPU-AI nodes with V100s #}
{% set nn_cpu = cpu_tasks|calc_num_nodes(40) %}
{% set nn_gpu = gpu_tasks|calc_num_nodes(8) %}
```

(continues on next page)

(continued from previous page)

```
{% set nn = nn|default((nn_cpu, nn_gpu)|max, true) %}
{% else %}
{# GPU nodes with P100s #}
{% set nn_cpu = cpu_tasks|calc_num_nodes(32) %}
{% set nn_gpu = gpu_tasks|calc_num_nodes(2) %}
{% set nn = nn|default((nn_cpu, nn_gpu)|max, true) %}
{% endif %}
{% 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(32), true) %}
{% endif %}
{% if partition in ('GPU', 'GPU-small') %}
#SBATCH -N {{ nn|check_utilization(gpu_tasks, 2, threshold, 'GPU') }}
#SBATCH --ntasks-per-node=32
#SBATCH --gres=gpu:p100:2
{% elif partition == 'GPU-shared' %}
{% if gpu_tasks > 1 and gpu_tasks % 2 %}
{% raise "Can only request multiples of two GPUs when submitting to the GPU-shared␣
↪partition." %}
{% endif %}
#SBATCH -N {{ nn|default(1, true)|check_utilization(gpu_tasks, 1, threshold, 'GPU') }}
#SBATCH --ntasks-per-node=16
#SBATCH --gres=gpu:p100:{{ 2 if gpu_tasks > 1 else 1 }}
{% elif partition == 'GPU-AI' %}
{% 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 %}
#SBATCH -N {{ nn|default(1, true)|check_utilization(gpu_tasks, 1, threshold, 'GPU') }}
#SBATCH --ntasks-per-node={{ cpus_per_node }}
#SBATCH --gres=gpu:volta16:{{ gpus_per_node }}
{% elif 'shared' in partition %}
#SBATCH -N {{ nn|default(1, true) }}
#SBATCH --ntasks-per-node={{ cpu_tasks }}
{% else %}
#SBATCH -N {{ nn|check_utilization(cpu_tasks, 28, threshold, 'CPU') }}
#SBATCH --ntasks-per-node={{ (28, 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**

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 %}
```

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:
```

(continues on next page)

(continued from previous page)

```
  {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

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

A signac project class specialized for workflow management.

This class provides a command line interface for the definition, execution, and submission of workflows based on condition and operation functions.

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

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

FlowProject().main()
```

Parameters

- **config** (*A signac config object.*) – A signac configuration, defaults to the configuration loaded from the environment.
- **entrypoint** (*dict*) – A dictionary with two possible keys: executable and path. Path represents the filepath location of the script file (the script file must call `main`). Executable represents the location of the Python interpreter used for the executable of *FlowOperation* that are Python functions.

Attributes

FlowProject.ALIASES	These are default aliases used within the status output.
FlowProject.add_operation(name, cmd[, pre, post])	Add an operation to the workflow.
FlowProject.completed_operations(job)	Determine which operations have been completed for job.
FlowProject.export_job_statuses(collection, ...)	Export the job statuses to a database collection.
FlowProject.get_job_status(job[, ...])	Return a dict with detailed information about the status of a job.
FlowProject.label([label_name_or_func])	Designate a function to be a label function of this class.
FlowProject.labels(job)	Yields all labels for the given `job`.
FlowProject.main([parser])	Call this function to use the main command line interface.
FlowProject.make_group(name[, options])	Make a FlowGroup named `name` and return a decorator to make groups.

Continued on next page

Table 1 – continued from previous page

FlowProject.next_operations(*jobs[, ...])	Determine the next eligible operations for jobs.
FlowProject.operation(func[, name])	Add the function *func* as operation function to the class workflow definition.
FlowProject.operations	The dictionary of operations that have been added to the workflow.
FlowProject.post(condition[, tag])	Specify a function of job that must evaluate to True for this operation to be considered complete.
FlowProject.post.always(func)	Returns True.
FlowProject.post.copy_from(*other_funcs)	True if and only if all post conditions of other operation-function(s) are met.
FlowProject.post.false(key)	True if the specified key is present in the job document and evaluates to False.
FlowProject.post.isfile(filename)	True if the specified file exists for this job.
FlowProject.post.never(func)	Returns False.
*FlowProject.post.not_*(condition)	Returns `not condition(job)` for the provided condition function.
FlowProject.post.true(key)	True if the specified key is present in the job document and evaluates to True.
FlowProject.pre(condition[, tag])	Specify a function of job that must be true for this operation to be eligible for execution.
FlowProject.pre.after(*other_funcs)	True if and only if all post conditions of other operation-function(s) are met.
FlowProject.pre.always(func)	Returns True.
FlowProject.pre.copy_from(*other_funcs)	True if and only if all pre conditions of other operation-function(s) are met.
FlowProject.pre.false(key)	True if the specified key is present in the job document and evaluates to False.
FlowProject.pre.isfile(filename)	True if the specified file exists for this job.
FlowProject.pre.never(func)	Returns False.
*FlowProject.pre.not_*(condition)	Returns `not condition(job)` for the provided condition function.
FlowProject.pre.true(key)	True if the specified key is present in the job document and evaluates to True.
FlowProject.run([jobs, names, pretend, np, ...])	Execute all pending operations for the given selection.
FlowProject.run_operations([operations, ...])	Execute the next operations as specified by the project's workflow.
FlowProject.scheduler_jobs(scheduler)	Fetch jobs from the scheduler.
FlowProject.script(operations[, parallel, ...])	Generate a run script to execute given operations.
FlowProject.submit([bundle_size, jobs, ...])	Submit function for the project's main submit interface.
FlowProject.submit_operations(operations[, ...])	Submit a sequence of operations to the scheduler.

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

Bases: signac.contrib.project.Project

A signac project class specialized for workflow management.

This class provides a command line interface for the definition, execution, and submission of workflows based on condition and operation functions.

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

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

FlowProject().main()
```

Parameters

- **config** (*A signac config object.*) – A signac configuration, defaults to the configuration loaded from the environment.
- **entrypoint** (*dict*) – A dictionary with two possible keys: executable and path. Path represents the filepath location of the script file (the script file must call `main`). Executable represents the location of the Python interpreter used for the executable of *FlowOperation* that are Python functions.

ALIASES = {'active': 'A', 'error': 'E', 'held': 'H', 'inactive': 'I', 'queued': '
These are 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-dict 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}')
```

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

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

Please note, eligibility in this contexts refers only to the workflow 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*) – Required conditions.
- **post** – Post-conditions to determine completion.

completed_operations(*job*)

Determine which operations have been completed for job.

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

Returns The name of the operations that are complete.

Return type str

detect_operation_graph()

Determine the directed acyclic graph defined by operation pre- and post-conditions.

In general, executing a given operation registered with a FlowProject just involves checking the operation's pre- and post-conditions to determine eligibility. More generally, however, the pre- and post-conditions 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 pre- and post-conditions. The algorithm builds an adjacency matrix based on whether the pre-conditions for one operation match the post-conditions 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 be bytecode equivalent, 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()
```

Raises a `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.

Raises RuntimeError

export_job_statuses(*collection*, *statuses*)
Export the job statuses to a database collection.

get_job_status(*job*, *ignore_errors=False*, *cached_status=None*)
Return a dict with detailed information about the status of a job.

classmethod label(*label_name_or_func=None*)
Designate a function to be a label function of 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, you can specify a different default 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.

labels(*job*)
Yields all labels for the given `job`.

See also: *label()*

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, e.g.:

```
 my_project.py
from flow import FlowProject

class MyProject(FlowProject):
    pass

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

You can then execute this script on the command line:

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

classmethod make_group(*name*, *options=''*)
Make a FlowGroup named `name` and return a decorator to make groups.

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

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

FlowGroups group operations together for running and submitting JobOperations.

Parameters

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

next_operations(**jobs*, *ignore_conditions=<IgnoreConditions.NONE: 0>*)

Determine the next eligible operations for jobs.

Parameters

- **jobs** – The signac job handles.
- **ignore_conditions** (*IgnoreConditions*) – Specify if pre and/or post conditions check is to be ignored for eligibility check. The default is `IgnoreConditions.NONE`.

Yield All instances of `JobOperation` jobs are eligible for.

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

Add the function *func* as operation function to the class workflow definition.

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

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

See also: *add_operation()*.

operations

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*, *no_parallelize=False*, *template=None*, *profile=False*, *eligible_jobs_max_lines=None*, *output_format='terminal'*)

Print the status of the project.

Parameters

- **jobs** (Sequence of instances `Job`.) – Only execute operations for the given jobs, or all if the argument is omitted.
- **overview** (*bool*) – Aggregate an overview of the project' status.
- **overview_max_lines** (*int*) – Limit the number of overview lines.
- **detailed** (*bool*) – Print a detailed status of each job.
- **parameters** (*list of str*) – Print the value of the specified parameters.
- **param_max_width** (*int*) – Limit the number of characters of parameter columns.

- **expand** (*bool*) – Present labels and operations in two separate tables.
- **all_ops** (*bool*) – Include operations that are not eligible to run.
- **only_incomplete** (*bool*) – Only show jobs that have eligible operations.
- **dump_json** (*bool*) – Output the data as JSON instead of printing the formatted output.
- **unroll** (*bool*) – Separate columns for jobs and the corresponding operations.
- **compact** (*bool*) – Print a compact version of the output.
- **pretty** (*bool*) – Prettify the output.
- **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.
- **template** (*str*) – User provided Jinja2 template file.
- **profile** (*bool*) – Show profile result.
- **eligible_jobs_max_lines** (*int*) – Limit the number of operations and its eligible job count printed in the overview.
- **output_format** (*str*) – Status output format, supports: 'terminal' (default), 'markdown' or 'html'.

Returns A Renderer class object that contains the rendered string.

Return type *Renderer*

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 pending operations for the given selection.

This function will run in an infinite loop until all pending 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 post conditions are provided.

See also: *run_operations()*

Parameters

- **jobs** (Sequence of instances `Job`.) – Only execute operations for the given jobs, or all if the argument is omitted.
- **names** (Sequence of `str`) – Only execute operations that are in the provided set of names, or all, if the argument is omitted.
- **pretend** (*bool*) – Do not actually execute the operations, but show which command would have been used.
- **np** (*int*) – Parallelize to the specified number of processors. Use -1 to parallelize to all available processing units.
- **timeout** (*int*) – An optional timeout for each operation in seconds after which execution will be cancelled. Use -1 to indicate not timeout (the default).
- **num** (*int*) – The total number of operations that are executed will not exceed this argument if provided.
- **num_passes** (*int*) – The total number 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.
- **order** (*str, callable, or NoneType*) –

  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 pre- and post-conditions. 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 pre and/or post conditions check is to be ignored for eligibility check. The default is `IgnoreConditions.NONE`.

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

See also: *run()*

Parameters

- **operations** (Sequence of instances of `JobOperation`) – The operations to execute (optional).
- **pretend** (*bool*) – Do not actually execute the operations, but show which command would have been used.
- **np** (*int*) – The number of processors to use for each operation.
- **timeout** (*int*) – An optional timeout for each operation in seconds after which execution will be cancelled. Use -1 to indicate not timeout (the default).
- **progress** (*bool*) – Show a progress bar during execution.

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

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

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

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

Yields All scheduler jobs fetched from the scheduler instance.

script (*operations*, *parallel=False*, *template='script.sh'*, *show_template_help=False*)
Generate a run script to execute given operations.

Parameters

- **operations** (Sequence of instances of `JobOperation`) – The operations to execute.
- **parallel** (*bool*) – Execute all operations in parallel (default is False).
- **template** (*str*) – The name of the template to use to generate the script.
- **show_template_help** (*bool*) – Show help related to the templating system and then exit.

submit(*bundle_size=1*, *jobs=None*, *names=None*, *num=None*, *parallel=False*, *force=False*, *walltime=None*, *env=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** (Sequence of instances `Job`.) – Only submit operations associated with the provided jobs. Defaults to all jobs.
- **names** (Sequence of `str`) – Only submit operations with any of the given names, defaults to all names.
- **num** (*int*) – Limit the total number of submitted operations, defaults to no limit.
- **parallel** (*bool*) – Execute all bundled operations in parallel.
- **force** (*bool*) – Ignore all warnings or checks during submission, just submit.
- **walltime** – Specify the walltime in hours or as instance of `datetime.timedelta`.
- **ignore_conditions** (*IgnoreConditions*) – Specify if pre and/or post conditions check is to be ignored for eligibility check. The default is `IgnoreConditions.NONE`.
- **ignore_conditions_on_execution** – Specify if pre and/or post conditions check is to be ignored for eligibility check after submitting. The default is `IgnoreConditions.NONE`.

submit_operations(*operations*, *_id=None*, *env=None*, *parallel=False*, *flags=None*, *force=False*, *template='script.sh'*, *pretend=False*, *show_template_help=False*, ***kwargs*)

Submit a sequence of operations to the scheduler.

Parameters

- **operations** (A sequence of instances of `JobOperation`) – The operations to submit.
- **_id** (*str*) – The _id to be used for this submission.
- **parallel** (*bool*) – Execute all bundled operations in parallel.
- **flags** (*list*) – Additional options to be forwarded to the scheduler.
- **force** (*bool*) – Ignore all warnings or checks during submission, just submit.
- **template** (*str*) – The name of the template file to be used to generate the submission script.
- **pretend** (*bool*) – Do not actually submit, but only print the submission script to screen. Useful for testing the submission workflow.

- **show_template_help** (*bool*) – Show information about available template variables and filters and exit.
- ****kwargs** – Additional keyword arguments to be forwarded to the scheduler.

Returns Returns the submission status after successful submission or None.

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

Specify a function of job 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.`**`always`**`(`*func*`)`

Returns True.

Deprecated since version 0.9: This will be removed in 0.11. This condition decorator is obsolete.

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

True if and only if all post conditions of other operation-function(s) are met.

classmethod `post.`**`false`**`(`*key*`)`

True if the specified key is present in the job document and evaluates to False.

classmethod `post.`**`isfile`**`(`*filename*`)`

True if the specified file exists for this job.

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

Returns False.

classmethod `post.`**`not_`**`(`*condition*`)`

Returns `not condition(job)` for the provided condition function.

classmethod `post.`**`true`**`(`*key*`)`

True if the specified key is present in the job document and evaluates to True.

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

Specify a function of job that must be 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*`)`

True if and only if all post conditions of other operation-function(s) are met.

classmethod pre.**always**(*func*)
Returns True.

Deprecated since version 0.9: This will be removed in 0.11. This condition decorator is obsolete.

classmethod pre.**copy_from**(**other_funcs*)
True if and only if all pre conditions of other operation-function(s) are met.

classmethod pre.**false**(*key*)
True if the specified key is present in the job document and evaluates to False.

classmethod pre.**isfile**(*filename*)
True if the specified file exists for this job.

classmethod pre.**never**(*func*)
Returns False.

classmethod pre.**not_**(*condition*)
Returns `not condition(job)` for the provided condition function.

classmethod pre.**true**(*key*)
True if the specified key is present in the job document and evaluates to True.

class flow.**IgnoreConditions**
Flags that determine which conditions are used to determine job eligibility.
The options include:

- IgnoreConditions.NONE: check all conditions
- IgnoreConditions.PRE: ignore pre conditions
- IgnoreConditions.POST: ignore post conditions
- IgnoreConditions.ALL: ignore all conditions

class flow.render_status.**Renderer**
A class for rendering status in different format.

This class provides method and string output for rendering status output in different format, currently supports: terminal, markdown and html

generate_html_output()
Get status string in html format.

Returns Status string in html format.

Return type str

generate_terminal_output()
Get status string in format for terminal.

Returns Status string in format for terminal.

Return type str

render(*template*, *template_environment*, *context*, *detailed*, *expand*, *unroll*, *compact*, *output_format*)
Render the status in different format for print_status.

Parameters

- **template** (*str*) – User provided Jinja2 template file.
- **template_environment** (jinja2.Environment) – Template environment.

- **context** (*dict*) – Context that includes all the information for rendering status output.
- **detailed** (*bool*) – Print a detailed status of each job.
- **expand** (*bool*) – Present labels and operations in two separate tables.
- **unroll** (*bool*) – Separate columns for jobs and the corresponding operations.
- **compact** (*bool*) – Print a compact version of the output.
- **output_format** (*str*) – Rendering output format, supports: 'terminal' (default), 'markdown' or 'html'.

1.3.3 @flow.cmd

`flow.`**`cmd`**(*func*)

Specifies that `func` returns a shell command.

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.

1.3.4 @flow.with_job

`flow.`**`with_job`**(*func*)

Specifies that `func(arg)` will use `arg` as a context manager.

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)
```

1.3.5 @flow.directives

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

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.

1.3.6 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
```

1.3.7 flow.init()

`flow.`**`init`**(*alias=None*, *template=None*, *root=None*, *out=None*)
Initialize a templated FlowProject module.

1.3.8 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.

Returns The detected environment class.

1.3.9 The FlowGroup

`class` `flow.project.`**`FlowGroup`**(*name*, *operations=None*, *operation_directives=None*, *options=''*)
A FlowGroup represents a subset of a workflow for a project.

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

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 lets 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.

complete(*job*)
True when all BaseFlowOperation post-conditions are met.

Parameters **job** (`signac.Job`) – A `signac.Job` from the signac workspace.

Returns Whether the group is complete (all contained operations are complete).

Return type bool

eligible(*job*, *ignore_conditions=<IgnoreConditions.NONE: 0>*)
Eligible, when at least one BaseFlowOperation is eligible.

Parameters

- **job** (`signac.Job`) – A `signac.Job` from the signac workspace.
- **ignore_conditions** (*IgnoreConditions*) – Specify if pre and/or post conditions check is to be ignored for eligibility check. The default is `IgnoreConditions.NONE`.

Returns Whether the group is eligible.

Return type bool

isdisjoint(*group*)
Returns whether two groups are disjoint (do not share any common operations).

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

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

Return type bool

Feature Deprecation

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

1.4 Changes

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

1.4.1 Version 0.10

[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).

1.4.2 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).

1.4.3 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).

Fixes

- Both the `nranks` 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.

1.4.4 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).

Fixes

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

Fixes (compute environments)

- Fix issue with the TORQUE scheduler that occured 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.

Fixes (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.

Fixes (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.

1.4.5 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 execption 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 overloaded 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 pre- and post-conditions 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 >= 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* commad 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 ist 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() docorator 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.

1.4.6 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 nex 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.

1.4.7 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
```

or alternatively with conda:

```
~ $ conda create -n signac-flow-dev python --file requirements-dev.txt
~ $ 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.

Finally, we recommend to setup a Flake8 git commit hook with:

```
(signac-flow-dev) signac-flow $ flake8 --install-hook git
(signac-flow-dev) signac-flow $ git config --bool flake8.strict true
```

With the *flake8* 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 unittest discover 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

Index

O

P

R

S

T

W