signac Documentation

Carl Simon Adorf, Vyas Ramasubramani, Bradley Dice

Oct 24, 2018

Contents

The **signac** framework supports researchers in managing project-related data with a well-defined indexable storage layout for data and metadata. This streamlines post-processing and analysis and makes data collectively accessible.

This is the overall **framework documentation**. It provides a comprehensive overview on what you can do with packages that are part of the **signac** framework. If you are new to **signac**, the best place to start is to read the *Introduction* and the *Tutorial*.

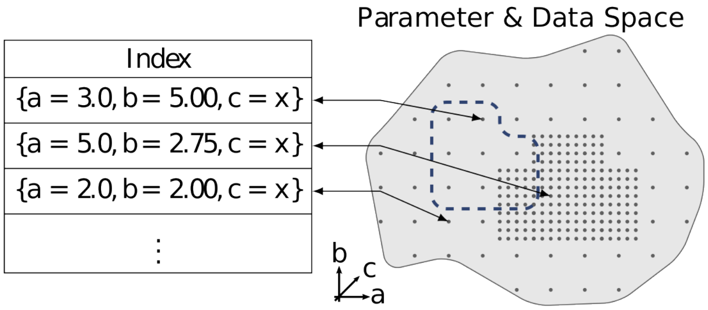
CHAPTER 1

Contents

1.1 Introduction

The **signac** framework is designed to simplify the storage, generation and analysis of multidimensional data sets associated with large-scale, file-based computational studies. Any computational work that requires you to manage files and execute workflows may benefit from an integration with **signac**. Typical examples include hyperparameter optimization for machine learning applications and high-throughput screening of material properties with various simulation methods. The data model assumes that the work can be divided into so called *projects*, where each project is roughly confined by similarly structured data, e.g., a parameter study.

In **signac**, the elements of a project's data space are called *jobs*. Every job is defined by a unique set of well-defined parameters that define the job's context, and it also contains all the data associated with this metadata. This means that all data is uniquely addressable from the associated parameters. With **signac**, we define the processes generating and manipulating a specific data set as a sequence of operations on a job. Using this abstraction, **signac** can define workflows on an arbitrary **signac** data space.


Parameter & Data Space
Index
{a = 3.0, b = 5.00, c = x}
{a = 5.0, b = 2.75, c = x}
{a = 2.0, b = 2.00, c = x}
⋮
b
c
a

1.2 Installation

All packages in the **signac** framework depend on the core **signac** package, which provides the data management functionality used by all other packages (See *Packages (API)* for more information). Most users should install the

signac and the **signac-flow** packages, which are tested for Python version 2.7.x and 3.4+ and do not have any *hard* dependencies, ensuring that no packages outside the **signac** framework are required for basic functionality. Please see the individual package documentation for instructions on how to install additional packages.

1.2.1 Install with conda

The recommended installation method for installing **signac** packages is *via* conda. The **signac** packages are distributed *via* the conda-forge channel. For a standard installation, execute:

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

Tip: Consider adding the conda-forge channel to your default channels with: `$ conda config --add channels conda-forge`.

1.2.2 Install with pip

For a standard installation with pip, execute:

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

Note: If you want to install packages for all users on a machine, you can remove the `--user` option in the install command.

1.2.3 Installation from Source

Alternatively, you can clone any of the package's source code repositories and install them manually. For example, to install the signac core package you can execute the following code:

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

Note: If you want to install packages for all users on a machine, you can remove the `--user` option in the install command.

1.3 Quickstart

To get started, first *install* **signac** and then setup a new project with:

```
~ $ mkdir my_project
~ $ cd my_project/
~/my_project $ signac init MyProject
Initialized project 'MyProject'.
```

Important: If you need to interface with non-Python code, see *How to integrate signac-flow with MATLAB or other software without Python interface*.

Once a project has been created, the next step is to initialize the *data space* with, *e.g.*, a script called `init.py`.:

```
# init.py
import signac

project = signac.get_project()

for foo in range(3):
    project.open_job({'foo': foo}).init()
```

The key is using the Python *project* handle as the interface to initialize jobs (data points) in your data space. You can then implement a simple *data space operation* within a `project.py` script:

```
# project.py
from flow import FlowProject

@FlowProject.operation
def hello_job(job):
    print("Hello from job {}, my foo is '{}'.".format(job, job.sp.foo))


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

Note the use of the `FlowProject.operation` decorator to indicate that the `hello_job` function should be interpreted as an operation acting on the data space.

Operations can be executed for all of your jobs with:

```
~/my_project $ python project.py run
Execute operation 'hello_job(15e548a2d943845b33030e68801bd125)'...
Hello from job 15e548a2d943845b33030e68801bd125, my foo is '1'.
Execute operation 'hello_job(2b985fa90138327bef586f9ad87fc310)'...
Hello from job 2b985fa90138327bef586f9ad87fc310, my foo is '2'.
Execute operation 'hello_job(7f3e901b4266f28348b38721c099d612)'...
Hello from job 7f3e901b4266f28348b38721c099d612, my foo is '0'.
```

See the *Tutorial* for a more detailed introduction to how to use **signac** to manage data and implement workflows.

1.4 Tutorial

License

The code shown in this tutorial is part of the *Examples*. It can be downloaded from the signac-docs repository and is released into the public domain.

This tutorial is designed to step new users through the basics of setting up a **signac** data space, defining and executing a simple workflow, and analyzing the data. For the complete code corresponding to this tutorial, see the *Ideal Gas* example.

1.4.1 Basics

Initializing the data space

In this tutorial, we will perform a simple study of the pressure-volume (p-V) relationship of a noble gas. As a first approximation, we could model the gas as an *ideal gas*, so the *ideal gas law* applies:

$$pV = Nk_BT$$

Therefore, we can assume that the volume V can be directly calculated as a function of system size N, Boltzmann's constant k_B, and temperature T.

To test this relationship, we start by creating an empty project directory where we will place all the code and data associated with this computational study.

```
~ $ mkdir ideal_gas_project
~ $ cd ideal_gas_project/
~/ideal_gas_project $
```

We then proceed by initializing the data space within a Python script called `init.py`:

```python
# init.py
import signac

project = signac.init_project('ideal-gas-project')

for p in range(1, 10):
    sp = {'p': p, 'kT': 1.0, 'N': 1000}
    job = project.open_job(sp)
    job.init()
```

The `signac.init_project()` function initializes the **signac** project in the current working directory by creating a configuration file called `signac.rc`. The location of this file defines the *project root directory*. We can access the project interface from anywhere within and below the root directory by calling the `signac.get_project()` function, or from outside this directory by providing an explicit path, *e.g.*, `signac.get_project('~/ideal_gas_project')`.

Note: The name of the project stored in the configuration file is independent of the directory name it resides in.

We can verify that the initialization worked by examining the *implicit* schema of the project we just created:

```
~/ideal_gas_project $ signac schema
{
 'N': 'int([1000], 1)',
 'kT': 'float([1.0], 1)',
 'p': 'int([1, 2, 3, ..., 9, 10], 10)',
}
```

The output of the `$ signac schema` command gives us a brief overview of all keys that were used as well as their value (range).

Note: The `job.init()` function is idempotent, meaning that it is safe to call it multiple times even after a job has already been initialized. It is good practice make *all* steps that are part of the data space initialization routine idempotent.

Exploring the data space

The core function that **signac** offers is the ability to associate metadata — for example, a specific set of parameters such as temperature, pressure, and system size — with a distinct directory on the file system that contains all data related to said metadata. The `open_job()` method associates the metadata specified as its first argument with a distinct directory called a *job workspace*. These directories are located in the `workspace` sub-directory within the project directory and the directory name is the so called *job id*.

```
~/ideal_gas_project $ ls -1 workspace/
03585df0f87fada67bd0f540c102cce7
22a51374466c4e01ef0e67e65f73c52e
71855b321a04dd9ee27ce6c9cc0436f4
# ...
```

The *job id* is a highly compact, unambiguous representation of the *full metadata*, *i.e.*, a distinct set of key-value pairs will always map to the same job id. However, it can also be somewhat cryptic, especially for users who would like to browse the data directly on the file system. Fortunately, you don't need to worry about this *internal representation* of the data space while you are actively working with the data. Instead, you can create a *linked view* with the `signac view` command:

```
~/ideal_gas_project $ signac view
~/ideal_gas_project $ ls view/
p_1 p_10  p_2 p_3 p_4 p_5 p_6 p_7 p_8 p_9
```

The linked view is **the most compact** representation of the data space in form of a nested directory structure. *Most compact* means in this case, that **signac** detected that the values for *kT* and *N* are constant across all jobs and are therefore safely omitted. It is designed to provide a human-readable representation of the metadata in the form of a nested directory structure. Each directory contains a `job` directory, which is a symbolic link to the actual workspace directory.

Note: Make sure to update the view paths by executing the `$ signac view` command (or equivalently with the `create_linked_view()` method) everytime you add or remove jobs from your data space.

Interacting with the signac project

You interact with the **signac** project on the command line using the `signac` command. You can also interact with the project within Python *via* the `signac.Project` class. You can obtain an instance of that class within the project root directory and all sub-directories with:

```
>>> import signac
>>> project = signac.get_project()
```

Iterating through all jobs within the data space is then as easy as:

```
>>> for job in project:
...     print(job)
...
03585df0f87fada67bd0f540c102cce7
22a51374466c4e01ef0e67e65f73c52e
71855b321a04dd9ee27ce6c9cc0436f4
# ...
```

We can iterate through a select set of jobs with the `find_jobs()` method in combination with a query expression:

```
>>> for job in project.find_jobs({"kT": 1.0, "p.$lt": 3.0}):
...     print(job, job.sp.p)
...
742c883cbee8e417bbb236d40aea9543 1
ee550647e3f707b251eeb094f43d434c 2
>>>
```

In this example we selected all jobs, where the value for kT is equal to 1.0 – which would be all of them – and where the value for p is less than 3.0. The equivalent selection on the command line would be achieved with `$ signac find kT 1.0 p.\$lt 3.0`. See the detailed query documentation for more information on how to find and select specific jobs.

Note: The following expressions are all equivalent: `for job in project:`, `for job in project.find_jobs():`, and `for job in project.find_jobs(None):`.

Operating on the data space

Each job represents a data set associated with specific metadata. The point is to generate data which is a **function** of that metadata. Within the framework's language, such a function is called a *data space operation*.

Coming back to our example, we could implement a very simple operation that calculates the volume V as a function of our metadata like this:

```
def volume(N, kT, p):
    return N * kT / p
```

Let's store the volume within our data space in a file called `volume.txt`, by implementing this function in a Python script called `project.py`:

```
# project.py
import signac


def compute_volume(job):
    volume = job.sp.N * job.sp.kT / job.sp.p
    with open(job.fn('volume.txt'), 'w') as file:
        file.write(str(volume) + '\n')

project = signac.get_project()
for job in project:
    compute_volume(job)
```

Executing this script will calculate and store the volume for each pressure-temperature combination in a file called `volume.txt` within each job's workspace.

Note: The `job.fn('volume.txt')` expression is a short-cut for `os.path.join(job.workspace(), 'volume.txt')`.

1.4.2 Workflows

Implementing a simple workflow

In many cases, it is desirable to avoid the repeat execution of data space operations, especially if they are not idempotent or are significantly more expensive than our simple example. For this, we will incorporate the `compute_volume()` function into a workflow using the `FlowProject` class. We slightly modify our `project.py` script:

```
# project.py
from flow import FlowProject


@FlowProject.operation
def compute_volume(job):
    volume = job.sp.N * job.sp.kT / job.sp.p
    with open(job.fn('volume.txt'), 'w') as file:
        file.write(str(volume) + '\n')


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

The `operation()` decorator identifies the `compute_volume` function as an *operation function* of our project. Furthermore, it is now directly executable from the command line via an interface provided by the `main()` method.

We can then execute all operations defined within the project with:

```
~/ideal_gas_project $ python project.py run
Execute operation 'compute_volume(03585df0f87fada67bd0f540c102cce7)'...
Execute operation 'compute_volume(22a51374466c4e01ef0e67e65f73c52e)'...
Execute operation 'compute_volume(71855b321a04dd9ee27ce6c9cc0436f4)'...
# ...
```

However, if you execute this in your own terminal, you might have noticed a bunch of warning messages printed out at the end, that read similar to:

```
Operation 'compute_volume(03585df0f87fada67bd0f540c102cce7)' exceeds max. # of␣
↪allowed passes (1).
Operation 'compute_volume(22a51374466c4e01ef0e67e65f73c52e)' exceeds max. # of␣
↪allowed passes (1).
# and so on
```

That is because by default, the `run` command will continue to execute all defined operations until they are considered *completed*. An operation is considered completed when all its *post conditions* are met, and it is up to the user to define those post conditions. Since we have not defined any post conditions yet, **signac** would continue to execute the same operation indefinitely.

For this example, a good post condition would be the existence of the `volume.txt` file. To tell the `FlowProject` class when an operation is *completed*, we can modify the above example by adding a function that defines this condition:

```
# project.py
from flow import FlowProject


def volume_computed(job):
```

(continues on next page)

(continued from previous page)

```
    return os.path.isfile("volume.txt")


@FlowProject.operation
@FlowProject.post(volume_computed)
def compute_volume(job):
    volume = job.sp.N * job.sp.kT / job.sp.p
    with open(job.fn('volume.txt'), 'w') as file:
        file.write(str(volume) + '\n')


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

Tip: Simple conditions can be conveniently defined inline as lambda expressions: `@FlowProject.post(lambda job: job.isfile("volume.txt"))`.

We can check that we implemented the condition correctly by executing `$ python project.py run` again. This should now return without any message because all operations have already been completed.

Note: To simply, execute a specific operation from the command line ignoring all logic, use the `exec` command, *e.g.*: `$ python project.py exec compute_volume`. This command (as well as the run command) also accepts jobs as arguments, so you can specify that you only want to run operations for a specific set of jobs.

Extending the workflow

So far we learned how to define and implement *data space operations* and how to define simple post conditions to control the execution of said operations. In the next step, we will learn how to integrate multiple operations into a cohesive workflow.

First, let's verify that the volume has actually been computed for all jobs. For this we transform the `volume_computed()` function into a *label function* by decorating it with the `label()` decorator:

```
# project.py
from flow import FlowProject


@FlowProject.label
def volume_computed(job):
    return job.isfile("volume.txt")

# ...
```

We can then view the project's status with the `status` command:

```
~/ideal_gas_project $ python project.py status
Generate output...

Status project 'ideal-gas-project':
Total # of jobs: 10
```

(continues on next page)

(continued from previous page)

```
label            progress
---------------  --------------------------------------------------
volume_computed  |########################################| 100.00%
```

That means that there is a `volume.txt` file in each and every job workspace directory.

Let's assume that instead of storing the volume in a text file, we wanted to store in it in a JSON file called `data.json`. Since we are pretending that computing the volume is an expensive operation, we will implement a second operation that copies the result stored in the `volume.txt` file into the `data.json` file instead of recomputing it:

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

@FlowProject.operation
@FlowProject.pre(volume_computed)
@FlowProject.post.isfile("data.json")
def store_volume_in_json_file(job):
    with open(job.fn("volume.txt")) as textfile:
        with open(job.fn("data.json"), "w") as jsonfile:
            data = {"volume": float(textfile.read())}
            jsonfile.write(json.dumps(data) + "\n")

# ...
```

Here we reused the `volume_computed` condition function as a **pre-condition** and took advantage of the `post.isfile` short-cut function to define the post-condition for this operation function.

Important: An operation function is **eligible** for execution if all pre-conditions are met, at least one post-condition is not met and the operation is not currently submitted or running.

Next, instead of running this new function for all jobs, let's test it for one job first.

```
~/ideal_gas_project $ python project.py run -n 1
Execute operation 'store_volume_in_json_file(742c883cbee8e417bbb236d40aea9543)'...
```

We can verify the output with:

```
~/ideal_gas_project $ cat workspace/742c883cbee8e417bbb236d40aea9543/data.json
{"volume": 1000.0}
```

Since that seems right, we can then store all other volumes in the respective `data.json` files by executing `$ python project run`.

Tip: We could further simplify our workflow definition by replacing the `pre(volume_computed)` condition with `pre.after(compute_volume)`, which is a short-cut to reuse all of `compute_volume()`'s post-conditions as pre-conditions for the `store_volume_in_json_file()` operation.

The job document

Storing results in JSON format – as shown in the previous section – is good practice because the JSON format is an open, human-readable format, and parsers are readily available in a wide range of languages. Because of this, **signac** stores all metadata in JSON files and in addition comes with a built-in JSON-storage container for each job: the job document.

Let's add another operation to our `project.py` script that stores the volume in the *job document*:

```
# project.py
# ...

@FlowProject.operation
@FlowProject.pre.after(compute_volume)
@FlowProject.post(lambda job: 'volume' in job.document)
def store_volume_in_document(job):
    with open(job.fn("volume.txt")) as textfile:
        job.document.volume = float(textfile.read())
```

Besides needing fewer lines of code, storing data in the *job document* has one more distinct advantage: it is directly searchable. That means that we can find and select jobs based on its content.

Executing the `$ python project.py run` command after adding the above function to the `project.py` script will store all volume in the job documents. We can then inspect all *searchable* data with the `$ signac find` command in combination with the `--show` option:

```
~/ideal_gas_project $ signac find --show
03585df0f87fada67bd0f540c102cce7
{'N': 1000, 'kT': 1.0, 'p': 3}
{'volume': 333.3333333333333}
22a51374466c4e01ef0e67e65f73c52e
{'N': 1000, 'kT': 1.0, 'p': 5}
{'volume': 200.0}
71855b321a04dd9ee27ce6c9cc0436f4
{'N': 1000, 'kT': 1.0, 'p': 4}
{'volume': 250.0}
# ...
```

When executed with `--show`, the `find` command not only prints the *job id*, but also the metadata and the document for each job. In addition to selecting by metadata as shown earlier, we can also find and select jobs by their *job document* content, *e.g.*:

```
~/ideal_gas_project $ signac find --doc-filter volume.\$lte 125 --show
Interpreted filter arguments as '{"volume.$lte": 125}'.
df1794892c1ec0909e5955079754fb0b
{'N': 1000, 'kT': 1.0, 'p': 10}
{'volume': 100.0}
dbe8094b72da6b3dd7c8f17abdcb7608
{'N': 1000, 'kT': 1.0, 'p': 9}
{'volume': 111.11111111111111}
97ac0114bb2269561556b16aef030d43
{'N': 1000, 'kT': 1.0, 'p': 8}
{'volume': 125.0}
```

Note: The job document is a feature of the core **signac** package, and can be used even outside the context of a `FlowProject`.

1.4.3 Job scripts and cluster submission

Generating scripts

So far, we executed all operations directly on the command line with the `run` command. However we can also generate scripts for execution, which is especially relevant if you intend to submit the workflow to a scheduling system typically encountered in high-performance computing (HPC) environments.

Scripts are generated using the jinja2 templating system, but you don't have to worry about that unless you want to change any of the default templates.

We can generate a script for the execution of the *next eligible operations* with the `script` command. We need to reset our workflow before we can test that:

```
~/ideal_gas_project $ rm -r workspace/
~/ideal_gas_project $ python init.py
```

Let's start by generating a script for the execution of up to two *eligible* operations:

```
~/ideal_gas_project $ python project.py script -n 2
set -e
set -u

cd /Users/csadorf/ideal_gas_project

# Operation 'compute_volume' for job '03585df0f87fada67bd0f540c102cce7':
python project.py exec compute_volume 03585df0f87fada67bd0f540c102cce7
# Operation 'compute_volume' for job '22a51374466c4e01ef0e67e65f73c52e':
python project.py exec compute_volume 22a51374466c4e01ef0e67e65f73c52e
```

By default, the generated script will change into the *project root directory* and then execute the command for each next eligible operation for all selected jobs. We then have two ways to run this script. One option would be to pipe it into a file and then execute it:

```
~/ideal_gas_project $ python project.py script > run.sh
~/ideal_gas_project $ /bin/bash run.sh
```

Alternatively, we could pipe it directly into the command processor:

```
~/ideal_gas_project $ python project.py script | /bin/bash
```

Executing the `script` command again, we see that it would now execute both the `store_volume_in_document` and the `store_volume_in_json_file` operation, since both share the same pre-conditions:

```
~/ideal_gas_project $ python project.py script -n 2
set -e
set -u

cd /Users/csadorf/ideal_gas_project

# Operation 'store_volume_in_document' for job '03585df0f87fada67bd0f540c102cce7':
python project.py exec store_volume_in_document 03585df0f87fada67bd0f540c102cce7
# Operation 'store_volume_in_json_file' for job '03585df0f87fada67bd0f540c102cce7':
python project.py exec store_volume_in_json_file 03585df0f87fada67bd0f540c102cce7
```

If we wanted to customize the script generation, we could either extend the base template or simply replace the default template with our own. To replace the default template, we can put a template script called `script.sh` into a directory called templates within the project root directory. A simple template script might look like this:

```
cd {{ project.config.project_dir }}

{% for operation in operations %}
{{ operation.cmd }}
{% endfor %}
```

Storing the above template within a file called templates/script.sh will now change the output of the script command to:

```
~/ideal_gas_project $ python project.py script -n 2
cd /Users/csadorf/ideal_gas_project

python project.py exec store_volume_in_document 03585df0f87fada67bd0f540c102cce7
python project.py exec store_volume_in_json_file 03585df0f87fada67bd0f540c102cce7
```

Please see $ python project.py script --template-help to get more information on how to write and use custom templates.

Submit operations to a scheduling system

In addition to executing operations directly on the command line and generating scripts, **signac** can also submit operations to a scheduler such as SLURM. This is essentially equivalent to generating a script as described in the previous section, but in this case the script will also contain the relevant scheduler directives such as the number of processors to request. In addition, **signac** will also keep track of submitted operations in addition to workflow progress, which almost completely automates the submission process as well as preventing the accidental repeated submission of operations.

To use this feature, make sure that you are on a system with any of the supported schedulers and then run the $ python project.py submit command.

1.5 Reference

A complete reference to all major components of the **signac** framework.

This section is primarily targeted at users who want to learn *in-depth* about the individual components. New users should go through the *Tutorial* first.

Contents:

1.5.1 Projects

Introduction

For a full reference of the Project API, please see the Python API.

A **signac** project is a conceptual entity consisting of three components:

1. a **data space**,
2. **scripts and routines** that operate on that space, and
3. the project's **documentation**.

This division corresponds largely to the definition of a computational project outlined by Wilson et al. The primary function of **signac** is to provide a single interface between component **(2)**, the scripts encapsulating the project logic, and component **(1)**, the underlying data generated and manipulated by these operations. By maintaining a clearly defined data space that can be easily indexed, **signac** can provide a consistent, homogeneous data access mechanism. In the process, **signac**'s maintainance of the data space also effectively functions as an implicit part of component **(3)**, the project's documentation.

Project Initialization

In order to use **signac** to manage a project's data, the project must be **initialized** as a **signac** project. After a project has been initialized in **signac**, all shell and Python scripts executed within or below the project's root directory have access to **signac**'s central facility, the **signac** project interface. The project interface provides simple and consistent access to the project's underlying *data space*.[1]

To initialize a project, simply execute `$ signac init <project-name>` on the command line inside the desired project directory (create a new project directory if needed). For example, to initialize a **signac** project named *MyProject* in a directory called `my_project`, execute:

```
$ mkdir my_project
$ cd my_project
$ signac init MyProject
```

You can alternatively initialize your project within Python with the `init_project()` function:

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

This will create a configuration file which contains the name of the project. The directory that contains this configuration file is the project's root directory.

The Data Space

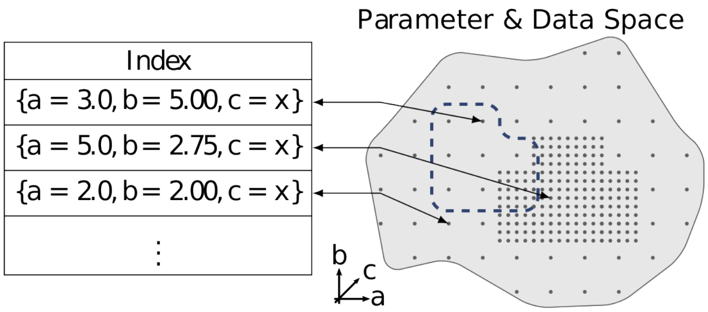
The project data space is stored in the *workspace directory*. By default this is a sub-directory within the project's root directory named *workspace*. Once a project has been initialized, any data inserted into the data space will be stored within this directory. This association is not permanent; a project can be reassociated with a new workspace at any time, and it may at times be beneficial to maintain multiple separate workspaces for a single project. You can access your signac `Project` and the associated *data space* from within your project's root directory or any subdirectory from the command line:

```
$ signac project
MyProject
```

Or with the `get_project()` function:

```
>>> import signac
>>> project = signac.get_project()
>>> print(project)
MyProject
```

[1] You can access a project interface from other locations by explicitly specifying the root directory.


Parameter & Data Space
Index
{a = 3.0, b = 5.00, c = x}
{a = 5.0, b = 2.75, c = x}
{a = 2.0, b = 2.00, c = x}
⋮
b
c
a

Jobs

For a full reference of the Job API, please see the Python API.

The central assumption of the **signac** data model is that the *data space* is divisible into individual data points, consisting of data and metadata, which are uniquely addressable in some manner. In the context of **signac**, each data point is called a *job*, and its unique address is referred to as a *state point*. A job can consist of any type of data, ranging from a single value to multiple terabytes of simulation data; **signac**'s only requirement is that this data can be encoded in a file.

State Points

A *state point* is a simple mapping of key-value pairs containing metadata describing the job. The state point is then used to compute a hash value, called the *job id*, which serves as the unique id for the job. The **signac** framework keeps track of all data and metadata by associating each job with a *workspace directory*, which is just a subdirectory of the project workspace. This subdirectory is named by the *job id*, therefore guaranteeing a unique file system path for each *job* within the project's *workspace* directory.

Note: Because **signac** assumes that the state point is a unique identifier, multiple jobs cannot share the same state point. A typical remedy for scenarios where, *e.g.*, multiple replicas are required, is to append the replica number to the state point to generate a unique state point.

Both the state point and the job id are equivalent addresses for jobs in the data space. To access or modify a data point, obtain an instance of `Job` by passing the associated metadata as a mapping of key-value pairs (for example, as an instance of `dict`) into the `open_job()` method.

```
# Define a state point:
>>> statepoint = {'a': 0}
# Get the associated job:
>>> job = project.open_job(statepoint)
>>> print(job.get_id())
9bfd29df07674bc4aa960cf661b5acd2
```

In general an instance of `Job` only gives you a handle to a python object. To create the underlying workspace directory and thus make the job part of the data space, you must *initialize* it. You can initialize a job **explicitly**, by calling the `init()` method, or **implictly**, by either accessing the job's *job document* or by switching into the job's workspace directory.

```
>>> job = project.open_job({'a': 2})
# Job does not exist yet
>>> job in project
```

(continues on next page)

(continued from previous page)

```
False
>>> job.init()
# Job now exists
>>> job in project
True
```

Once a job has been initialized, it may also be *opened by id* as follows (initialization is required because prior to initialization the job id has not yet been calculated):

```
>>> job.init()
>>> job2 = project.open_job(id=job.get_id())
>>> job == job2
True
```

Whether a job is opened by state point or job id, an instance of `Job` can always be used to retrieve the associated *state point*, the *job id*, and the *workspace* directory with the `statepoint()`, `get_id()`, and `workspace()` methods, respectively:

```
>>> print(job.statepoint())
{'a': 0}
>>> print(job.get_id())
9bfd29df07674bc4aa960cf661b5acd2
>>> print(job.workspace())
'/home/johndoe/my_project/workspace/9bfd29df07674bc4aa960cf661b5acd2'
```

Evidently, the job's workspace directory is a subdirectory of the project's workspace and is named by the job's id. We can use the `signac.Job.fn()` convenience function to prepend the this workspace path to a file name; `job.fn(filename)` is equivalent to `os.path.join(job.workspace(), filename)`. This function makes it easy to create or open files which are associated with the job:

```
>>> print(job.fn('newfile.txt'))
'/home/johndoe/my_project/workspace/9bfd29df07674bc4aa960cf661b5acd2/newfile.txt'
```

For convenience, the *state point* may also be accessed via the `statepoint` or `sp` attributes, e.g., the value for `a` can be printed using either `print(job.sp.a)` or `print(job.statepoint.a)`. This also works for **nested** *state points*: `print(job.sp.b.c)`! An additional advantage of accessing the statepoint via the attributes is that these can be directly modified, triggering a recalculation of the job id and a renaming of the job's workspace directory.

Modifying the State Point

As just mentioned, the state point of a job can be changed after initialization. A typical example where this may be necessary, is to add previously not needed state point keys. Modifying a state point entails modifying the job id which means that the state point file needs to be rewritten and the job's workspace directory is renamed, both of which are computationally cheap operations. The user is nevertheless advised **to take great care when modifying a job's state point** since errors may render the data space **inconsistent**.

There are three main options for modifying a job's state point:

1. Directly via the job's `statepoint` and `sp` attributes,
2. via the job's `update_statepoint()` method, and
3. via the job's `reset_statepoint()` method.

The `update_statepoint()` method provides safeguards against accidental overwriting of existing *state point* values, while `reset_statepoint()` will simply reset the whole *state point* without further questions. The statepoint and sp attributes provide the greatest flexibility, but similar to reset_statepoint() they provide no additional protection.

Important: Regardless of method, **signac** will always raise a DestinationExistsError if a *state point* modification would result in the overwriting of an existing job.

The following examples demonstrate how to **add**, **rename** and **delete** *state point* keys using the sp attribute:

To **add a new key** b to all existing *state points* that do not currently contain this key, execute:

```
for job in project:
    job.sp.setdefault('b', 0)
```

Renaming a state point key from b to c:

```
for job in project:
    assert 'c' not in job.sp
    job.sp.c = job.statepoint.pop('b')
```

To **remove** a state point key c:

```
for job in project:
    if 'c' in job.sp:
        del job.sp['c']
```

You can modify **nested** *state points* in-place, but you will need to use dictionaries to add new nested keys, e.g.:

```
>>> job.statepoint()
{'a': 0}
>>> job.sp.b.c = 0   # <-- will raise an AttributeError!!

# Instead:
>>> job.sp.b = {'c': 0}

# Now you can modify in-place:
>>> job.sp.b.c = 1
```

The Job Document

In addition to the state point, additional metadata can be associated with your job in the form of simple key-value pairs using the job document. This *job document* is automatically stored in the job's workspace directory in JSON format. You can access it via the document or the doc attribute.

```
>>> job = project.open_job(statepoint)
>>> job.doc['hello'] = 'world'
# or equivalently
>>> job.doc.hello = 'world'
```

Just like the job *state point*, individual keys may be accessed either as attributes or through a functional interface, *e.g.*. The following examples are all equivalent:

```
>>> print(job.document().get('hello'))
world
>>> print(job.document.hello)
```

(continues on next page)

(continued from previous page)

```
world
>>> print(job.doc.hello)
world
```

Tip: Use the `Job.document.get()` method to return `None` or another specified default value for missing values. This works exactly like with python's built-in dictionaries.

Use cases for the **job document** include, but are not limited to:

1. **storage** of *lightweight* data,
2. Tracking of **runtime information**
3. **labeling** of jobs, e.g. to identify error states.

Finding jobs

In general, you can iterate over all initialized jobs using the following idiom:

```
for job in project:
    pass
```

This notation is shorthand for the following snippet of code using the `find_jobs()` method:

```
for job in project.find_jobs():
    pass
```

However, the `find_jobs()` interface is much more powerful in that it allows filtering for subsets of jobs. For example, to iterate over all jobs that have a *state point* parameter `b=0`, execute:

```
for job in project.find_jobs({'b': 0}):
    pass
```

For more information on how to search for specific jobs in Python and on the command line, please see the *Query API* chapter.

Grouping

Grouping operations can be performed on the complete project data space or the results of search queries, enabling aggregated analysis of multiple jobs and state points.

The return value of the `signac.Project.find_jobs()` method is an iterator over all jobs (or all jobs matching an optional filter if one is specified). This iterator is an instance of `JobsCursor` and allows us to group these jobs by state point parameters, the job document values, or even arbitrary functions.

Note: The `groupby()` method is very similar to Python's built-in `itertools.groupby()` function.

Basic Grouping by Key

Grouping can be quickly performed using a statepoint or job document key.

If *a* was a state point variable in a project's parameter space, we can quickly enumerate the groups corresponding to each value of *a* like this:

```
for a, group in project.groupby('a'):
    print(a, list(group))
```

Similarly, we can group by values in the job document as well. Here, we group all jobs in the project by a job document key *b*:

```
for b, group in project.groupbydoc('b'):
    print(b, list(group))
```

Grouping by Multiple Keys

Grouping by multiple state point parameters or job document values is possible, by passing an iterable of fields that should be used for grouping. For example, we can group jobs by state point parameters *c* and *d*:

```
for (c, d), group in project.groupby(('c', 'd')):
    print(c, d, list(group))
```

Searching and Grouping

We can group a data subspace by combining a search with a group-by function. As an example, we can first select all jobs, where the state point key *e* is equal to 1 and then group them by the state point parameter *f*:

```
for f, group in project.find_jobs({'e': 1}).groupby('f'):
    print(f, list(group))
```

Custom Grouping Functions

We can group jobs by essentially arbitrary functions. For this, we define a function that expects one argument and then pass it into the `groupby()` method. Here is an example using an anonymous *lambda* function as the grouping function:

```
for (d, count), group in project.groupby(lambda job: (job.sp['d'], job.document['count
↪'])):
    print(d, count, list(group))
```

Moving, Copying and Removal

In some cases it may desirable to divide or merge a project data space. To **move** a job to a different project, use the `move()` method:

```
other_project = get_project(root='/path/to/other_project')

for job in jobs_to_move:
    job.move(other_project)
```

Copy a job from a different project with the `clone()` method:

```
project = get_project()

for job in jobs_to_copy:
    project.clone(job)
```

Trying to move or copy a job to a project which has already an initialized job with the same *state point*, will trigger a `DestinationExistsError`.

> **Warning:** While **moving** is a cheap renaming operation, **copying** may be much more expensive since all of the job's data will be copied from one workspace into the other.

To **clear** all data associated with a specific job, call the `clear()` method. Note that this function will do nothing if the job is uninitialized; the `reset()` method will also clear all data associated with a job, but it will also automatically initialize the job if it was not originally initialized. To **permanently delete** a job and its contents use the `remove()` method:

```
job = project.open_job(statepoint)
job.remove()
assert job not in project
```

Centralized Project Data

To support the centralization of project-level data, **signac** offers simple facilities for placing data at the project level instead of associating it with a specific job. For one, **signac** provides a *project document* analogous to the *job document*. The project document is stored in JSON format in the project root directory and can be used to store similar types of data to the job document.

```
>>> project = signac.get_project()
>>> project.doc['hello'] = 'world'
>>> print(project.doc().get('hello'))
'world'
>>> print(project.doc.hello)
'world'
```

In addition, **signac** also provides the `signac.Project.fn()` method, which is analogous to the `signac.Job.fn()` method described above:

```
>>> print(project.root_directory())
'/home/johndoe/my_project/'
>>> print(project.fn('foo.bar'))
'/home/johndoe/my_project/foo.bar'
```

Schema Detection

While **signac** does not require you to specify an *explicit* state point schema, it is always possible to deduce an *implicit* semi-structured schema from a project's data space. This schema is comprised of the set of all keys present in all state points, as well as the range of values that these keys are associated with.

Assuming that we initialize our data space with two state point keys, `a` and `b`, where `a` is associated with some set of numbers and `b` contains a boolean value:

```
for a in range(3):
    for b in (True, False):
        project.open_job({'a': a, 'b': b}).init()
```

Then we can use the `signac.Project.detect_schema()` method to get a basic summary of keys within the project's data space and their respective range:

```
>>> print(project.detect_schema())
{
 'a': 'int([0, 1, 2], 3)',
 'b': 'bool([False, True], 2)',
}
```

This functionality is also available directly from the command line:

```
$ signac schema
{
 'a': 'int([0, 1, 2], 3)',
 'b': 'bool([False, True], 2)',
}
```

Importing and Exporting Data

Data archival is important to preserving the integrity, utility, and shareability of a project. To this end, **signac** provides interfaces for importing workspaces from and exporting workspaces to directories, zip-files, and tarballs. The exported project archives are useful for publishing data, *e.g.*, for researchers wishing to make an original data set available alongside a publication.

Exporting a Workspace

Exporting a project could be as simple as zipping the project files and workspace paths (`$ zip -r project_archive.zip /data/my_project/`). The functionality provided by `signac export` is a bit more fine-grained and allows the use of a custom path structure or the export of a subset of the jobs based on state point or document filters or by job id.

For example, suppose we have a project stored locally in the path `/data/my_project` and want to export it to a directory `/data/my_project_archive`. The project's jobs are assumed to have state point keys *a* and *b* with integer values. We would first change into the root directory of the project that we want to export and then call `signac export` with the target path:

```
$ cd /data/my_project
$ signac export /data/my_project_archive
```

This would **copy** data from the source project to the export directory with the following directory structure:

```
/data/my_project_archive/a/0/b/0/
/data/my_project_archive/a/0/b/1/
/data/my_project_archive/a/0/b/2/
# etc.
```

The default path function is based on the implicit schema of all exported jobs, but we can also **optionally** specify a specific export path, for example like this:

```
$ signac export /data/my_project_archive "a_{a}/b_{b}"
```

It is possible to directly export to a zip-file or tarball by simply providing the path to the archive-file as target (*e.g.* `$ signac export /data/my_project_archive.zip`). For more details on how to use `signac export`, type `$ signac export --help` or see the documentation for the `export_to()` method.

Importing a Data Space

The import of data spaces into a **signac** workspace means to map all directories as part of an arbitrary directory structure to signac job state points. That is easiest when one imports a previously exported workspace, which will still contain all state point files.

For example, we could first export our workspace in `~/my_project` to `~/data/` with

```
~/my_project $ signac export ~/data/
```

and then import the exported data into a second project:

```
~/my_new_project $ signac import ~/data/
```

Since the imported data space was previously exported with **signac**, all state point metadata is automatically determined from the state point manifest files.

In the case that we want to import a data space that was not previously exported with **signac**, we need to provide a schema-function. In the simplest case, that is just a function based on the data space paths, *e.g.*,

```
$ signac import /data/non_signac_archive "a_{a:int}/b_{b:int}"
```

The command above will copy all data from the the `/data/non_signac_archive` directory and use the paths of sub-directories to identify the associated state points. For example, the path `a_0/b_1` will be interpreted as `{'a': 0, 'b': 1}`. The type specification – here `int` for both *a* and *b* – is optional and means that these values are converted to type `int`; the default type is `str`.

Importing from zip-files and tarballs works similarly, by specifying that path as the origin. For more details on how to use `signac import`, type `$ signac import --help` or see the documentation for `import_from()`.

Linked Views

Data space organization by job id is both efficient and flexible, but the obfuscation introduced by the job id makes inspecting the workspace on the command line or *via* a file browser much harder. A *linked view* is a directory hierarchy with human-interpretable names that link to to the actual job workspace directories. Unlike the default mode for *data export*, no data is copied for the generation of linked views.

To create views from the command line use the `$ signac view` command.

Important: When the project data space is changed by adding or removing jobs, simply update the view, by executing `create_linked_view()` or `signac view` for the same view directory again.

You can limit the *linked view* to a specific data subset by providing a set of *job ids* to the `create_linked_view()` method. This works similar for `$ signac view` on the command line, but here you can also specify a filter directly:

```
$ signac view -f a 0
```

will create a linked view for all jobs, where `a=0`.

Synchronization

In some cases it may be necessary to store a project at more than one location, perhaps for backup purposes or for remote execution of data space operations. In this case there will be a regular need to synchronize these data spaces.

Synchronization of two projects can be accomplished by either using `rsync` to directly synchronize the respective workspace directories, or by using `signac sync`, a tool designed for more fine-grained synchronization of project data spaces. Users who are familiar with `rsync` will recognize that most of the core functionality and API of `rsync` is replicated in `signac sync`.

As an example, let's assume that we have a project stored locally in the path `/data/my_project` and want to synchronize it with `/remote/my_project`. We would first change into the root directory of the project that we want to synchronize data into. Then we would call `signac sync` with the path of the project that we want to *synchronize with*:

```
$ cd /data/my_project
$ signac sync /remote/my_project
```

This would copy data *from the remote project to the local project*. For more details on how to use `signac sync`, type `$ signac sync --help`.

Projects can also be synchronized using the Python API:

```
project.sync('/remote/my_project')
```

1.5.2 Query API

As briefly described in *Finding jobs*, the `find_jobs()` method provides much more powerful search functionality beyond simple selection of jobs with specific state point values. More generally, all **find()** functions within the framework accept filter arguments that will return a selection of jobs or documents. One of the key features of **signac** is the possibility to immediately search managed data spaces to select desired subsets as needed. Internally, all search operations are processed by an instance of `Collection` (see *Collections*). Therefore, they all follow the same syntax, so you can use the same type of filter arguments in `find_jobs()`, `find_statepoints()`, and so on.

Note: The **signac** framework query API is a subset of the MongoDB query API!

Basics

Filter arguments are a mapping of expressions, where a single expression consists of a key-value pair. All selected documents must match these expressions.

The simplest expression is an *exact match*. For example, in order to select all jobs whose state point key *a* has the value 42, you would use the following expression: `{'a':  42}` as follows:

```
project.find_jobs({'a': 42})
```

Select All

If you want to select the complete data set, don't provide any filter argument at all. The default argument of `None` or an empty expression `{}` will select all jobs or documents. As was previously demonstrated, iterating over all jobs in a project or all documents in a collection can be accomplished directly without using any *find* method at all:

```
for job in project:
    pass

for doc in collection:
    pass
```

Simple Selection

To select documents by one or more specific key-value pairs, simply provide these directly as filter arguments. For example, assuming that we have a list of documents with values *N*, *kT*, and *p*, as such:

```
1: {'N': 1000, 'kT': 1.0, 'p': 1}
2: {'N': 1000, 'kT': 1.2, 'p': 2}
3: {'N': 1000, 'kT': 1.3, 'p': 3}
...
```

We can select the 2nd document with `{'p': 2}`, but also `{'N': 1000, 'p': 2}` or any other matching combination.

Nested Keys

To match **nested** keys, avoid nesting the filter arguments, but instead use the .-operator. For example, if the documents shown in the example above were all nested like this:

```
1: {'statepoint': {'N': 1000, 'kT': 1.0, 'p': 1}}
2: {'statepoint': {'N': 1000, 'kT': 1.2, 'p': 2}}
3: {'statepoint': {'N': 1000, 'kT': 1.3, 'p': 3}}
...
```

Then we would use `{'statepoint.p': 2}` instead of `{'statepoint': {'p': 2}}` as filter argument. This is not only easier to read, but also increases compatibility with MongoDB database systems.

Operator Expressions

In addition to simple exact value matching, **signac** also provides **operator-expressions** to execute more complicated search queries.

Arithmetic Expressions

If we wanted to match all documents where *p is greater than 2*, we would use the following filter argument:

```
{'p': {'$gt': 2}}
```

Note that we have replaced the value for p with the expression `{'$gt': 2}` to select *all all jobs withe p values greater than 2*. Here is a complete list of all available **arithmetic operators**:

- `$eq`: equal to
- `$ne`: not equal to
- `$gt`: greater than
- `$gte`: greater or equal than

- `$lt`: less than
- `$lte`: less or equal than

Near Operator

The `$near` operator is used to find jobs with state point parameters that are near a value, where floating point precision may make it difficult to match the exact value. The behavior of `$near` matches that of python's math.isclose function. The "reference" value and tolerances are passed in as a list in the order `[reference, [relative_tolerance, [absolute_tolerance]]]`, where the inner `[]`s denote optional values. Note that default values are `relative_tolerance = 1e-09` and `absolute_tolerance = 0`.

```
signac find theta.\$near 0.6  # easier than typing 0.600000001
signac find '{"p.$near": [100, 0.05]}'  # p within 5% of 100
signac find '{"p.$near": [100, 0.05, 2]}'  # abs(p-100)/max(p, 100) < 0.05 or abs(p-
↪100) < 2
```

Logical Operators

There are two supported logical operators: `$and` and `$or`. To querying with a logical expression, we construct a mapping with the logical-operator as the key and a list of expressions as the value. As usual, the `$and` operator matches documents where all the expressions are true, while the `$or` expression matches if any documents satisfy the provided expression. For example, we can match all documents where *p is greater than 2* **or** *kT=1.0* we could use the following (split onto multiple lines for clarity):

```
{
   '$or': [
            {'p': {'$gt': 2}},    # either match this
            {'kT': 1.0}           # or this
          ]
}
```

Logical expressions may be nested, but cannot be the *value* of a key-value expression.

Exists Operator

If you want to check for the existence of a specific key but do not care about its actual value, use the `$exists`-operator. For example, the expression `{'p': {'$exists': True}}` will return all documents that *have a key p* regardless of its value. Likewise, using `False` as argument would return all documents that have no key with the given name.

Array Operator

This operator may be used to determine whether specific keys have values, that are **in** (`$in`), or **not in** (`$nin`) a given array, e.g.:

```
{'p': {'$in': [1, 2, 3]}}
```

This would return all documents where the value for *p* is either 1, 2, or 3. The usage of `$nin` is equivalent, and will return all documents where the value is *not in* the given array.

Regular Expression Operator

This operator may be used to search for documents where the value of type `str` matches a given *regular expression*. For example, to match all documents where the value for *protocol* contains the string “assembly”, we could use:

```
{'protocol': {'$regex': 'assembly'}}
```

This operator internally applies the `re.search()` function and will never match if the value is not of type `str`.

Type Operator

This operator may be used to search for documents where the value is of a specific type. For example, to match all documents, where the value of the key *N* is of integer-type, we would use:

```
{'N': {'$type': 'int'}}
```

Other supported types include *float*, *str*, *bool*, *list*, and *null*.

Where Operator

This operator allows us to apply a *custom function* to each value and select based on its return value. For example, instead of using the regex-operator, as shown above, we could write the following expression:

```
{'protocol': {'$where': 'lambda x: "assembly" in x'}}
```

Simplified Syntax on the Command Line

It is possible to use search expressions directly on the command line, for example in combination with the `$ signac find` command. In this case filter arguments are expected to be provided as valid JSON expressions. However, for simple filters you can also use a *simplified syntax*. For example, instead of `{'p': 2}`, you can simply type `p 2`.

A simplified expression consists of key-value pairs in alternation. The first argument will then be interpreted as the first key, the second argument as the first value, the third argument as the second key, and so on. If you provide an odd number of arguments, the last value will default to `{'$exists': True}`. Querying via operator is supported using the .-operator. Finally, you can use `/<regex>/` intead of `{'$regex': '<regex>'}` for regular expressions.

The following list shows simplified expressions on the left and their equivalent standard expression on the right.

```
simplified              standard
--------------------    ------------------------------------


p                       {'p': {'$exists': True}}
p 2                     {'p': 2}
p 2 kT                  {'p': 2, 'kT': {'$exists': True}}
p 2 kT.$gte 1.0         {'p': 2, 'kT': {'$gte': 1.0}}
protocol /assembly/     {'protocol': {'$regex': 'assembly'}}
```

Important: The `$` character used in operator-expressions must be escaped in many terminals, that means for example instead of `$ signac find p.$gt 2`, you would need to write `$ signac find p.\$gt 2`.

1.5.3 Indexing

Concept

Data spaces managed with **signac** on the file system are immediately searchable because **signac** creates an index of all relevant files *on the fly* whenever a search operation is executed. This data index contains all information about the project's files, their location and associated metadata such as the *signac id* and the *state point*.

A file index has *one entry per file* and each document has the following fields:

- `id`: a unique value which serves as a primary key
- `root`: The root path of the file
- `filename`: The filename of the file
- `md5`: A MD5-hash value of the file content
- `file_id`: A number identifying the file content[1]
- `format`: A format definition (optional)

The **signac** project interface is specifically designed to assist with processes related to data curation. However, especially when working with a data set comprised of multiple projects or sources that are not managed with **signac**, it might be easier to work with a data index directly.

For example, this is how we would access files related to a specific data subset using the project interface:

```
for job in project.find_jobs({"a": 42}):
    with open(job.fn('hello.txt')) as file:
        print(file.read())
```

And this is how we would do the same, but operating directly with an index:

```
index = signac.Collection(project.index(".*\.txt"))

for doc in index.find({
        "statepoint.a": 42,
        "filename": {"$regex": "hello.txt"}}):
    with signac.fetch(doc) as file:
        print(file.read())
```

Here, we first generate the index with the `Project.index()` function and stored the result in a `Collection` container. Then, we search the index collection for a specific state point and use `fetch()` to open the associated file. The `fetch()` functions works very similar to Python's built-in `open()` function to open files, but in addition will be able to fetch a file from multiple different sources if necessary.

The next few sections are a more detailed outline of how such a workflow can be realized.

Indexing a signac Project

As shown in the previous section, a **signac** project index can be generated directly with the `Project.index()` function in Python. Alternatively, we can generate the index on the command line with `$ signac project --index`.

A signac project index is like a regular file index, but contains the following additional fields:

- `signac_id`: The state point id the document is associated with.

[1] Identical to the `md5` value in the current implementation.

- statepoint: The state point mapping associated with the file.

Each signac project index will have *at least one* entry for each initialized job. This special index document is associated with the job's *document* file and contains not only the signac_id and the statepoint, but also the data stored in the job document. This means the following code snippet would be valid:

```
for job in project:
    job.document['foo'] = 'bar'

for doc in project.index():
    assert doc['foo'] == 'bar'
```

By default, no additional files are indexed; the user is expected to *explicitly* specify which files should be part of the index as described in the next section.

Indexing files

Indexing specific files as part of a project index requires using regular expressions. For instance, in the initial example we used the expression ".*\.txt" to specify that all files with a filename ending with ".txt" should be part of the index.

We can extract metadata directly from the filename by using regular expressions with *named groups*. For example, if we have a filename pattern: a_0.txt, a_1.txt and so on, where the number following a_ is to be extracted as the a field, we can use the following expression:

```
for doc in project.index('.*a_(?P<a>\d+)'):
    print(doc['a'])
```

To further simplify the selection of different files from the index, we may provide multiple patterns with an optional *format definition*. Let's imagine we would like to classify the text files with the a field from the previous example *as well as* PDF-files that adhere to the following pattern: init.pdf or final.pdf. This is how we could generate this index:

```
formats = {
    '.*a_(?P<a>\d+)\.txt': 'TextFile',
    '.*(?P<class>init|final)\.pdf': 'PDFFile'}

for doc in project.index(formats):
    print(doc)
```

Tip: To generate regular expressions for the filename patterns in your data space, copy & paste a few representative filenames into the excellent regex101 online app. That will allow you to work out your expressions while getting direct graphical feedback.

If we want to file an arbitrary directory structure that is not managed by **signac**, we can use the index_files() function, that expects the root path as the first argument, and indexes **all files** by default.

```
for doc in signac.index_files('/data'):
    pass
```

Fetching Data

Index documents can be used to directly fetch associated data. The `signac.fetch()` function is essentially equivalent to python's built-in `open()` function, but instead of a file path it uses an index document[2] to locate and open the file.

```
# Search for specific documents:
for doc in index.find({'statepoint.a': 42, 'format': 'TextFile'}):
    with signac.fetch(doc) as file:
        do_something_with_file(file)
```

The `fetch()` function will attempt to retrieve data from more than one source if data was *mirrored*. Overall, this enables us to operate on indexed project data in a way which is more agnostic to its actual source.

Deep Indexing

We may want to add additional metadata to the index that is neither based on neither the state point, the job document, or the filename, but instead is directly extracted from the data. Such a pattern is typically referred to as *deep indexing* and can be easily implemented with **signac**.

As an example, imagine that we wanted add the number of lines within a file as an additional metadata field in our data index. For this, we use Python's built-in `map()` function, which allows us to apply a function to all index entries:

```
def add_num_lines(doc):
    if 'filename' in doc:
        with signac.fetch(doc) as file:
            doc['num_lines'] = len(list(file))
    return doc

index = map(add_num_lines, project.index())
```

The `index` variable now contains an index, where each index entry has an additional `num_lines` field.

Tip: We are free to apply multiple *deep indexing* functions in succession; the functions are only executed when the `index` iterable is actually evaluated.

Searching an Index

An index generated with the `Project.index()` method or any other index function is just an iterable over the index documents. To be able to **search** the index, we need to either implement routines to select specific documents or use containers that implement such routines, such as the `Collection` class that **signac** uses internally for all search operations.

For example, if we are looking for all files that correspond to a state point variable `a=42`, we could implement the following for-loop:

```
index = project.index()

docs = []
for doc in index:
    if doc['statepoint']['a'] == 42:
      docs.append(doc)
```

[2] or a file id

This is the same logic implemented more concisely as a list comprehension:

```
docs = [doc for doc in index if doc['statepoint']['a'] == 42]
```

Using loops is a very viable approach as long as the index is not too large and the search queries are relatively simple. Alternatively, we can manage the index using a `Collection` container, which then allows us to search the index with the query expressions that we are used to elsewhere using **signac**. For example, to execute the same search operation from above, we could use the `find()` method:

```
index = Collection(signac.index())

docs = index.find({'statepoint.a': 42})
```

Tip

You can search a collection on the command line by calling it's `main()` method.

Unless they are very small, searching collections is usually **much more efficient** than the *pure python* approach, especially when searching multiple times within the same session. Furthermore, since a collection may be saved to and loaded from a file, we only have to generate an index once, saving us the effort of regenerating it each time we use it:

```
with Collection.open('index.txt') as index:
    if update_index:
        index.update(signac.index())
    docs = index.find({'statepoint.a': 42})
```

Since **signac**'s decentralized approach is not designed to automatically keep track of changes, it is up to the user to determine when a particular index needs to be updated. To automatically identify and remove stale documents[3], use the `signac.export()` function:

```
with Collection.open('index.txt') as index:
    signac.export(signac.index(), index, update=True)
```

Tip: The `Collection` class has the same interface as a `pymongo.collection.Collection` class. That means you can use these two types of collections interchangeably.

Master Indexes

Generating a Master Index

A master index is a compilation of multiple indexes that simplifies operating on a larger data space. To make a signac project part of a master index, we simply create a file called `signac_access.py` in its root directory. The existance of this file tells **signac** that the projects in those directories should be indexed as part of a master index.

Imagine that we have two projects in two different directories `~/project_a` and `~/project_b` within our home directory. We create the `signac_access.py` file in each respective project directory like this:

```
$ touch ~/project_a/signac_access.py
$ touch ~/project_b/signac_access.py
```

[3] A *stale* document is associated with a file or state point that has been removed.

Executing the `index()` function for the home directory

```
for doc in signac.index('~'):
    print(doc)
```

will now yield a joint index for both projects in `~/project_a` and `~/project_b`.

For more information on how to have more control over the index creation, see the *signac access module* section.

Tip: By typing `$ signac index` you can directly generate a signac master index on the command line and then pipe it into a file:

```
$ signac index > index.txt
```

The *signac_access.py* Module

We can use the `signac_access.py` module to control the index generation across projects. An **empty** module is equivalent to a module which contains the following directives:

```
import signac

def get_indexes(root):
    yield signac.get_project(root).index()
```

This means that any index yielded from a `get_indexes()` function defined within the access module will be compiled into the master index.

By putting this code explicitly into the module, we have full control over the index generation. For example, to specify that all files with filenames ending with `.txt` should be added to the index, we would put the following code into the module:

```
import signac

def get_indexes(root):
    yield signac.get_project(root).index(formats='.*\.txt')
```

You can generate a basic access module for a **signac** project using the `create_access_module()` method.

Tip: The `signac_access.py` module is perfectly suited to implement deep indexing patterns.

Database Integration

Database access

After *configuring* one or more database hosts you can access a database with the `signac.get_database()` function.

Mirroring of Data

Using the `signac.fetch()` function it is possible retrieve files that are associated with index documents. Those files will preferably be opened directly via a local system path. However, in some cases it may be desirable to mirror files at a different location, e.g., in a database or a different path, to increase the accessibility of files.

Use the mirrors argument in the `signac.export()` function to automatically mirror all files associated with exported index documents. **signac** provides handlers for a local file system and the MongoDB GridFS database file system.

```
from signac import fs, export, get_database

db = get_database('mirror')

localfs = fs.LocalFS('/path/to/mirror')
gridfs = fs.GridFS(db)

export(crawler.crawl(), db.index, mirrors=[localfs, gridfs])
```

To access the data, provide the mirrors argument to the `signac.fetch()` function:

```
for doc in index:
    with signac.fetch(doc, mirrors=[localfs, gridfs]) as file:
        do_something_with_file(file)
```

Note: File systems are used to fetch data in the order provided, starting with the native data path.

Using Tags to Control Access

It may be desirable to only index select projects for a specific *master index*, e.g., to distinguish between public and private indexes. For this purpose, it is possible to specify **tags** that are **required** by a *crawler* or *index*. This means that an index **requiring** tags will be ignored during a master index compilation, unless at least one of the tags is also **provided**.

For example, you can define **required** tags for indexes returned from the `get_indexes()` function, by attaching them to the function like this:

```
def get_indexes(root):
    yield signac.get_project(root).index()

get_indexes.tags = {'public', 'foo'}
```

Similarly, you can require tags for specific crawlers:

```
class MyCrawler(SignacProjectCrawler):
    tags = {'public', 'foo'}
```

Unless you **provide** *at least one* of these tags (`public` or `foo`), the examples above would be ignored during the master index compilation. This means only the second one of the following two lines would **not ignore** the examples above:

```
index = signac.index()                    # examples above are ignored
index = signac.index(tags={'public'})    # includes examples above
```

Similarly on the command line:

```
$ signac index                  # examples above are ignored
$ signac index --tags public   # includes examples above
```

In summary, there must be an overlap between the **requested** and the **provided** tags.

How to publish an index

Here we demonstrate how to compile a master index with data mirroring, which is designed to be publicly accessible. The index will be stored in a document collection called `index` as part of a database called `public_db`. All data files will be mirrored within the same database. That means everybody with access to the `public_db` database will have access to the index as well as to the associated files.

```
import signac

db = signac.get_database('public_db')

# We define two mirrors
file_mirrors = [
  # The GridFS database file system is stored in the
  # same database, that we use to publish the index.
  # This means that anyone with access to the index,
  # will be able to access the associated files as well.
  signac.fs.GridFS(db),

  # The second mirror is on the local file system.
  # It can be downloaded and made available locally,
  # for example to reduce the amount of required
  # network traffic.
  signac.fs.LocalFS('/path/to/mirror')
  ]

# Only crawlers which have been explicitly cleared for
# publication with the `public` tag will be compiled and exported.
index = signac.index('/path/to/projects', tags={'public'})

# The export() function pushes the index documents to the database
# collection and copies all associated files to the file mirrors.
signac.export(index, db.index, file_mirrors, update=True)
```

1.5.4 Collections

An instance of `Collection` is a *container* for multiple documents, where a document is an associative array of key-value pairs. Examples are the job state point, or the job document.

The `Collection` class is used internally to manage and search data space indexes which are generated on-the-fly. But you can also use such a container explicitly for managing document data.

Creating collections

To create an empty collection, simply call the default constructor:

```
from signac import Collection

collection = Collection()
```

You can then add documents with the signac.Collection.insert_one() method. Alternatively you can pass an iterable of documents as the first argument, such as the return value of the signac.Project.index() method:

```
index_collection = Collection(project.index())
```

By default, the collection is stored purely in memory. But you can use the signac.Collection container also to manage collections **directly on disk**. For this, simply *open* a file like this:

```
with Collection.open('my_collection.txt') as collection:
    pass
```

A collection file by default is openend in *append plus* mode, that means it is opened for both reading and writing. The open() function accepts all standard file open modes, such as *r* for *read-only*, etc.

Searching collections

To search a collection, use the signac.Collection.find() method. As an example, to search all documents where the value a is equal to 42, execute:

```
for doc in collection.find({"a": 42}):
    pass
```

The signac.Collection.find() method uses the framework-wide *query* API.

Command Line Interface

To manage and search a collection file directly from the command line, create a python script with the following content:

```
from signac import Collection

with Collection.open("my_collection.txt") as c:
    c.main()
```

Storing the code above in a file called find.py and then executing it will allow you to search for all or specific documents within the collection, directly from the command line $ python find.py.

For more information on how to use the command line interface, execute: $ python find.py --help.

1.5.5 Operations and Conditions

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

Data Space Operations

Concept

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

We will demonstrate this concept with a simple example. Let's initialize a project with a few jobs, by executing the following script within a ~/my_project directory:

```
# init.py
import signac

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

A very simple *operation*, which creates a file called `hello.txt` within a job's workspace directory could then be implemented like this:

```
# operations.py

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

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

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

The *flow.run()*-interface

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

The *run*-interface

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

```
# operations.py

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

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

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

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

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

Parallelized Execution

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

```
from time import sleep

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

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

Conditions

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

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

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

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

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

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

Executing this workflow in an ad-hoc manner could be accomplished like this:

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

This approach is fine for simple workflows, but would become very cumbersome for even slightly more complex workflows and is not very flexible. In the next chapter, we will demonstrate how to integrate operations and conditions into a well-defined workflow using the `FlowProject` class.

1.5.6 The FlowProject

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

Setup and Interface

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

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

class Project(FlowProject):
    pass

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

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

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

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

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

Defining a workflow

We will reproduce the simple workflow introduced in the previous section by first copying both the `greeted()` condition function and the `hello()` *operation* function into the `project.py` module. We then use the `operation()` and the `post()` decorator functions to specify that the `hello()` operation function is part of our workflow and that it should only be executed if the `greeted()` condition is not met.

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


class Project(FlowProject):
    pass


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


@Project.operation
@Project.post(greeted)
def hello(job):
```

(continues on next page)

(continued from previous page)

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


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

We can define both *pre* and *post* conditions, which allow us to define arbitrary workflows as an acyclic graph. A operation is only executed if **all** pre-conditions are met, and at *at least one* post-condition is not met.

Tip: Cheap conditions should be placed before expensive conditions as they are evaluated lazily! That means for example, that given two pre-conditions, the following order of definition would be preferable:

```
@Project.operation
@Project.pre(cheap_condition)
@Project.pre(expensive_condition)
def hello(job):
    pass
```

The same holds for *post*-conditions.

We can then execute this workflow with:

```
~/my_project $ python project.py run
Execute operation 'hello(15e548a2d943845b33030e68801bd125)'...
hello 15e548a2d943845b33030e68801bd125
Execute operation 'hello(288f97857257baee75d9d84bf0e9dfa8)'...
hello 288f97857257baee75d9d84bf0e9dfa8
Execute operation 'hello(2b985fa90138327bef586f9ad87fc310)'...
hello 2b985fa90138327bef586f9ad87fc310
# ...
```

If we implemented and integrated the operation and condition functions correctly, calling the `run` command twice should produce no output the second time, since the `greeted()` condition is met for all jobs and the `hello()` operation should therefore not be executed.

The Project Status

The `FlowProject` class allows us to generate a **status** view of our project. The status view provides information about which conditions are met and what operations are pending execution.

A condition function which is supposed to be shown in the **status** view is called a *label-function*. We can convert any condition function into a label function by adding the `label()` decorator:

```
# project.py
# ...

@Project.label
def greeted(job):
    return job.isfile('hello.txt')

# ...
```

We will reset the workflow for only a few jobs to get a more interesting *status* view:

```
~/my_project $ signac find a.\$lt 5 | xargs -I{} rm workspace/{}/hello.txt
```

We then generate a *detailed* status view with:

```
~/my_project.py status --detailed
Collect job status info: 100%|| 10/10
# Overview:
Total # of jobs: 10

label    ratio
-------  -------------------------------------------------
greeted  |####################--------------------| 50.00%

# Detailed View:
job_id                            labels
--------------------------------  --------
0d32543f785d3459f27b8746f2053824  greeted
14fb5d016557165019abaac200785048
└ hello [U]
2af7905ebe91ada597a8d4bb91a1c0fc
└ hello [U]
2e6ba580a9975cf0c01cb3c3f373a412  greeted
42b7b4f2921788ea14dac5566e6f06d0
└ hello [U]
751c7156cca734e22d1c70e5d3c5a27f  greeted
81ee11f5f9eb97a84b6fc934d4335d3d  greeted
9bfd29df07674bc4aa960cf661b5acd2
└ hello [U]
9f8a8e5ba8c70c774d410a9107e2a32b
└ hello [U]
b1d43cd340a6b095b41ad645446b6800  greeted
Legend: :ineligible :eligible :active :running :completed
```

This view provides information about what labels are met for each job and what operations are eligible for execution. If we did things right, then only those jobs without the `greeted` label should have the `hello` operation pending.

As shown before, all *eligible* operations can then be executed with:

```
~/my_project $ python project.py run
```

Generating Execution Scripts:

Instead of executing operations directly we can also create a script for execution. If we have any pending operations, a script might look like this:

```
~/my_project $ python project.py script

set -e
set -u

cd /Users/csadorf/my_project

# Operation 'hello' for job '14fb5d016557165019abaac200785048':
/Users/csadorf/miniconda3/bin/python project.py exec hello␣
↪14fb5d016557165019abaac200785048
# Operation 'hello' for job '2af7905ebe91ada597a8d4bb91a1c0fc':
```

(continues on next page)

(continued from previous page)

```
/Users/csadorf/miniconda3/bin/python project.py exec hello␣
↪2af7905ebe91ada597a8d4bb91a1c0fc
# Operation 'hello' for job '42b7b4f2921788ea14dac5566e6f06d0':
/Users/csadorf/miniconda3/bin/python project.py exec hello␣
↪42b7b4f2921788ea14dac5566e6f06d0
# Operation 'hello' for job '9bfd29df07674bc4aa960cf661b5acd2':
/Users/csadorf/miniconda3/bin/python project.py exec hello␣
↪9bfd29df07674bc4aa960cf661b5acd2
# Operation 'hello' for job '9f8a8e5ba8c70c774d410a9107e2a32b':
/Users/csadorf/miniconda3/bin/python project.py exec hello␣
↪9f8a8e5ba8c70c774d410a9107e2a32b
```

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

This script is generated from a default jinja2 template, which is shipped with the package. We can extend this default template or write our own to cutomize the script generation process.

Here is an example for such a template, that would essentially generate the same output:

```
cd {{ project.config.project_dir }}

{% for operation in operations %}
operation.cmd
{% endfor %}
```

Note: Unlike the default template, this exemplary template would not allow for `parallel` execution.

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

1.5.7 Cluster Submission

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

In addition, **signac-flow** uses *environment profiles* to select which **base template** to use for the cluster job script generation. All base templates are in essence the same, but will be slightly adapted to the current cluster environment. That is because different cluster environments offer different resources and therefore require slightly different options for submission. You can check out the available options with the `python project.py submit --help` command.

The *submit* interface

In general, we submit operations through the primary interface of the `FlowProject`. We assume that we use the same `project.py` module as shown in the previous section.

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

```
~/my_project $ python project.py submit
```

This will submit all *eligible* job-operations to the cluster scheduler and block that specific job-operation from resubmission.

In some cases you can provide additional arguments to the scheduler, such as which partition to submit to, which will then be used by the template script. In addition you can always forward any arguments directly to the scheduler as positional arguments. For example, if we wanted to specify an account name with a *torque* scheduler, we could use the following command:

```
~/my_project $ python project.py submit -- -l A:my_account_name
```

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

Unless you have one of the *supported schedulers* installed, you will not be able to submit any operations in your local environment. However, **signac-flow** comes with a simple scheduler for testing purposes. You can execute it with `$ simple-scheduler run` and then follow the instructions on screen.

Submitting specific Operations

The submission process consists of the following steps:

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

The first step is largely determined by your project *workflow*. You can see which operation might be submitted by looking at the output of `$ python project.py status --detailed`. You may further reduce the operations to be submitted by selecting specific jobs (*e.g.* with the `-j` or the `-f` options), specific operations (`-o`), or generally reduce the total number of operations to be submitted (`-n`). For example the following command would submit up to 5 `hello` operations, where *a is less than 5*.

```
~/my_project $ python project.py submit -o hello -n 5 -f a.\$lt 5
```

The submission scripts are generated using the same templating system like the `script` command.

Tip: Use the `--pretend` or `--test` option to pre-view the generated submission scripts on screen instead of submitting them.

Parallelization and Bundling

By default all eligible job-operations will be submitted as separate cluster jobs. This is usually the best model for clusters that provide shared compute partitions. However, sometimes it is beneficial to execute multiple operations within one cluster job, especially if the compute cluster can only make reservation for full nodes.

You can place multiple job-operations within one cluster submission with the `--bundle` option. For example, the following command will bundle up to 5 job-operations to be executed in parallel into a single cluster submission:

```
~/my_project $ python project.py submit --bundle=5 --parallel
```

Without any argument the `--bundle` option will bundle **all** eligible job-operations into a single cluster job.

Tip: Recognizing that `--bundle=1` is the default option might help you to better understand the bundling concept.

For more information on managing different environments, see the *next section*.

1.5.8 Manage Environments

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

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

How to Use Environments

Environments are defined by subclassing from the `ComputeEnvironment` class. The `ComputeEnvironment` class is a *meta-class* that ensures that all subclasses are automatically globally registered when they are defined. This enables us to use environments simply by defining them or importing them from a different module. The `flow.get_environment()` function will go through all defined `ComputeEnvironment` classes and return the one where the `is_present()` class method returns `True`.

Packaged Environments

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

In addition, **signac-flow** comes with some environments tailored to specific compute clusters that are defined in the `flow.environments` module. These environments are not automatically available. Instead, you need to *explictly import* the `flow.environments` module.

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

Defining New Environments

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

Those are the steps usually required to define a new environment:

1. Subclass from `flow.ComputeEnvironment`.
2. Determine a host name pattern that would match the output of `socket.gethostname()`.
3. Create a template and specify the template name as `template` class variable.

This is an example for a typical environment class definition:

```
class MyUniversityCluster(flow.DefaultTorqueEnvironment):

    hostname_pattern = 'mycluster.*.university.edu'
    template = 'mycluster.myuniversity.sh'
```

Then, add the `mycluster.myuniversity.sh` template script to the *templates/* directory within your project root directory.

Important: The new environment will be automatically registered and used as long as it is either defined within the same module as your `FlowProject` class or imported into the same module.

As an example on how to write a submission script template, this would be a viable template to define the header for a SLURM scheduler:

```
{% extends "base_script.sh" %}
{% block header %}
#!/bin/bash
#SBATCH --job-name="{{ id }}"
#SBATCH --partition={{ partition }}
#SBATCH -t {{ walltime|format_timedelta }}
{% block tasks %}
#SBATCH --ntasks={{ np_global }}
{% endblock %}
{% endblock %}
```

All templates, which are shipped with the package, are within the *flow/templates/* directory within the package source code.

Contributing Environments to the Package

Users are **highly encouraged** to contribute environment profiles that they developed for their local environments. In order to contribute an environment, either simply email them to the package maintainers (see the README for contact information) or create a pull request.

1.5.9 Templates

The **signac-flow** package uses jinja2 templates to generate scripts for execution and submission to cluster scheduling systems. Templates for simple bash execution and submission to popular schedulers and compute clusters are shipped with the package. To customize the script generation, a user can replace the default template or customize any of the provided ones.

Replacing the default template

The default `script.sh` template simply extends from another script according to the `base_script` variable, which is dynamically set by **signac-flow**:

```
{% extends base_script %}
```

This `base_script` variable provides a way to inject specific templates for, *e.g.*, different environments. However, by default any templates placed within your project root directory in a file called `templates/script.sh` will be used instead of the defaults provided by **signac-flow**. This makes it easy to completely replace the scripts provided by signac-flow; to use your own custom script, simply place a new `script.sh` in a `templates` directory within your project root directory. This is an example for a basic template that would be sufficient for the simple serial execution of multiple operations:

```
cd {{ project.config.project_dir }}

{% for operation in operations %}
{{ operation.cmd }}
{% endfor %}
```

Customize provided templates

Instead of simply replacing the template as shown above, we can also customize the provided templates. One major advantage is that we can still use the template scripts for cluster submission.

Assuming that we wanted to write a time stamp to some log file before executing operations, we could provide a custom template such as this one:

```
{% extends base_script %}
{% block body %}
date >> execution.log
{{ super() }}
{% endblock %}
```

The first line again indicates that this template extends an existing template based on the value of `base_script`; how this variable is set is explained in more detail in the next section. The second and last line indicate that the enclosed lines are to be placed in the *body* block of the base template. The third line is the actual command that we want to add and the fourth line ensures that the code provided by the base template within the body block is still added.

The base template

The **signac-flow** package will select a different base script template depending on whether you are simply generating a script using the `script` command or whether you are submitting to a scheduling system with `submit`. In the latter case, the base script template is selected based on whether you are on any of the officially supported environments, and if not, whether one of the known scheduling system (torque or slurm) is available. This is a short illustration of that heuristic:

```
# The `script` command always uses the same base script template:
project.py script --> base_script='base_script.sh'

# On system with SLURM scheduler:
project.py submit --> base_script='slurm.sh' (extends 'base_script.sh')

# On XSEDE Comet
project.py submit --> base_script='comet.sh' (extends 'slurm.sh')
```

Regardless of which *base script template* you are actually extending from, all templates shipped with **flow** follow the same basic structure:

resources Calculation of the total resources required for the execution of this (submission) script.

header Directives for the scheduling system such as the cluster job name and required resources. This block is empty for shell script templates.

project_header Commands that should be executed once before the execution of operations, such as switching into the project root directory or setting up the software environment.

body All commands required for the actual execution of operations.

footer Any commands that should be executed at the very end of the script.

Execution Directives

Available directives

Any `FlowProject` *operation* can be amended with so called *execution directives*. For example, to specify that we want to parallelize a particular operation on **4** processing units, we would provide the `np=4` directive:

```python
from flow import FlowProject, directives
from multiprocessing import Pool

@FlowProject.operation
@directives(np=4)
def hello(job):
    with Pool(4) as pool:
      print("hello", job)
```

All directives are essentially conventions, the `np` directive in particular means that this particular operation requires 4 processors for execution. The following directives are respected by all base templates shipped with **signac-flow**:

executable Specify which Python executable should be used to execute this operation. Defaults to the one used to generate the script (`sys.executable`).

np The total number of processing units required for this operation.

nranks The number of MPI ranks required for this operation. The command will be prefixed with environment specific MPI command, e.g.: `mpiexec -n 4`. The value for *np* will default to *nranks* unless specified separately.

omp_num_threads The number of OpenMP threads. The value for *np* will default to: "nranks x omp_num_threads" unless otherwise specified.

ngpu The number of GPUs required for this operation.

Execution Modes

Using these directives and their combinations allows us to realize the following essential execution modes:

serial: `@flow.directives()`

This operation is a simple serial process, no directive needed.

parallelized: `@flow.directives(np=4)`

This operation requires 4 processing units.

MPI parallelized: `@flow.directives(nranks=4)`

This operation will be executed on 4 MPI ranks.

MPI/OpenMP Hybrid: `@flow.directives(nranks=4, omp_num_threads=2)`

This operation will be executed on 4 MPI ranks with 2 OpenMP threads per rank.

GPU: `@flow.directives(ngpu=1)`

The operation requires one GPU for execution.

1.5.10 Configuration

Overview

The **signac** framework is configured with configuration files, which are named either `.signacrc` or `signac.rc`. These configuration files are searched for at multiple locations in the following order:

1. in the current working directory,
2. in each directory above the current working directory until a project configuration file is found,
3. and the user's home directory.

The configuration file follows the standard "ini-style". Global configuration options, should be stored in the home directory, while project-specific options should be stored *locally* in a project configuration file.

This is an example for a global configuration file in the user's home directory:

```
# ~/.signarc
[hosts]
[[localhost]]
url = mongodb://localhost
```

You can either edit these configuration files manually, or execute `signac config` on the command line. Please see `signac config --help` for more information.

Project configuration

A project configuration file is defined by containing the keyword *project*. Once **signac** found a project configuration file it will stop to search for more configuration files above the current working directory.

For example, to initialize a project named *MyProject*, navigate to the project's root directory and either execute `$ signac init MyProject` on the command line, use the `signac.init_project()` function or create the project configuration file manually. This is an example for a project configuration file:

```
# signac.rc
project = MyProject
workspace_dir = $HOME/myproject/workspace
```

project The name is required for the identification of the project's root directory.

workspace_dir The path to your project's workspace, which defaults to `$project_root_dir/workspace`. Can be configured relative to the project's root directory or as absolute path and may contain environment variables.

Host configuration

The current version of **signac** supports MongoDB databases as a backend. To use **signac** in combination with a MongoDB database, make sure to install `pymongo`.

Configuring a new host

To configure a new MongoDB database host, create a new entry in the `[hosts]` section of the configuration file. We can do so manually or by using the `signac config host` command.

Assuming that we a have a MongoDB database reachable via *example.com*, which requires a username and a password for login, execute:

```
$ signac config host example mongodb://example.com -u johndoe -p
Configuring new host 'example'.
Password:
Configured host 'example':
[hosts]
[[example]]
        url = mongodb://example.com
        username = johndoe
        auth_mechanism = SCRAM-SHA-1
        password = ***
```

The name of the configured host (here: *example*) can be freely chosen. You can omit the `-p/--password` argument, in which case the password will not be stored and you will prompted to enter it for each session.

We can now connect to this host with:

```
>>> import signac
>>> db = signac.get_database('mydatabase', hostname='example')
```

The `hostname` argument defaults to the first configured host and can always be omitted if there is only one configured host.

Note: To prevent unauthorized users from obtaining your login credentials, **signac** will update the configuration file permissions such that it is only readable by yourself.

Changing the password

To change the password for a configured host, execute

```
$ signac host example --update-pw -p
```

Warning: By default, any password set in this way will be **encrypted**. This means that the actual password is different from the one that you entered. However, while it is practically impossible to guess what you entered, a stored password hash will give any intruder access to the database. This means you need to **treat the hash like a password!**

Copying a configuration

In general, in order to copy a configuration from one machine to another, you can simply copy the `.signacrc` file as is. If you only want to copy a single host configuration, you can either manually copy the associated section or use the `signac config host` command for export:

```
$ signac config host example > example_config.rc
```

Then copy the `example_config.rc` file to the new machine and rename or append it to an existing `.signacrc` file. For security reasons, any stored password is not directly copied in this way. To copy the password, follow:

```
# Copy the password from the old machine:
johndoe@oldmachine $ signac config host example --show-pw
XXXX
```

(continues on next page)

(continued from previous page)

```
# Enter it on the new machine:
johndoe@newmachine $ signac config host example -p
```

Manual host configuration

You can configure one or multiple hosts in the `[hosts]` section, where each subsection header specifies the host's name.

url The url specifies the MongoDB host url, e.g. `mongodb://localhost`.

authentication_method (default=none) Specify the authentication method with the database, possible choices are: `none` or `SCRAM-SHA-1`.

username A username is required if you authenticate via `SCRAM-SHA-1`.

password The password to authenticate via `SCRAM-SHA-1`.

db_auth (default=admin) The database to authenticate with.

password_config In case that you update, but not store your password, the configuration file will contain only meta hashing data, such as the salt. This allows to authenticate by entering the password for each session, which is generally more secure than storing the actual password hash.

> **Warning:** **signac** will automatically change the file permissions of the configuration file to *user read-write only* in case that it contains authentication credentials. In case that this fails, you can set the permissions manually, e.g., on UNIX-like operating systems with: `chmod 600 ~/.signacrc`.

1.6 Examples

> **License**
>
> All the code shown here can be downloaded from the signac-docs repository, and is released into the public domain.

This is a collection of example projects which are designed to illustrate how to implement certain applications and solutions with **signac**. Unlike the tutorial, the examples consist mainly of complete, immediately executable source code with less explanation.

1.6.1 Ideal Gas

This example is based on the *Tutorial* and assumes that we want to model a system using the ideal gas law:

$$pV = Nk_BT$$

The data space is initialized for a specific system size N, thermal energy kT, and pressure p in a script called `init.py`:

```
# init.py
import signac

project = signac.init_project('ideal-gas-project')

for p in range(1, 11):
    sp = {'p': p, 'kT': 1.0, 'N': 1000}
    job = project.open_job(sp)
    job.init()
```

The workflow consists of a `compute_volume` operation that computes the volume based on the given parameters and stores it within a file called `V.txt` within each job's workspace directory. The two additional operations copy the result into a JSON file called `data.json` and into the job document under the `volume` key respectively. All operations are defined in `project.py`:

```
# project.py
from flow import FlowProject


@FlowProject.label
def volume_computed(job):
    return job.isfile("volume.txt")


@FlowProject.operation
@FlowProject.post(volume_computed)
def compute_volume(job):
    volume = job.sp.N * job.sp.kT / job.sp.p
    with open(job.fn("volume.txt"), "w") as file:
        file.write(str(volume) + "\n")


@FlowProject.operation
@FlowProject.pre.after(compute_volume)
@FlowProject.post.isfile("data.json")
def store_volume_in_json_file(job):
    import json
    with open(job.fn("volume.txt")) as textfile:
        with open(job.fn("data.json"), "w") as jsonfile:
            data = {"volume": float(textfile.read())}
            jsonfile.write(json.dumps(data) + "\n")


@FlowProject.operation
@FlowProject.pre.after(compute_volume)
@FlowProject.post(lambda job: 'volume' in job.document)
def store_volume_in_document(job):
    with open(job.fn("volume.txt")) as textfile:
        job.document.volume = float(textfile.read())



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

The complete workflow can be executed on the command line with `$ python project.py run`.

1.6.2 MD with HOOMD-blue

This example demonstrates how to setup and analyze the simulation of a Lennard-Jones fluid with molecular dynamics (MD) using HOOMD-blue. The project data space is initialized in a `src/init.py` script with explicit random seed:

```python
#!/usr/bin/env python
"""Initialize the project's data space.

Iterates over all defined state points and initializes
the associated job workspace directories."""
import logging
import argparse
from hashlib import sha1

import signac
import numpy as np


def main(args, random_seed):
    project = signac.init_project('Ideal-Gas-Example-Project')
    for replication_index in range(args.num_replicas):
        for p in np.linspace(0.5, 5.0, 10):
            statepoint = dict(
                    # system size
                    N=512,

                    # Lennard-Jones potential parameters
                    sigma=1.0,
                    epsilon=1.0,
                    r_cut=2.5,

                    # random seed
                    seed=random_seed*(replication_index + 1),

                    # thermal energy
                    kT=1.0,
                    # presure
                    p=p,
                    # thermostat coupling constant
                    tau=1.0,
                    # barostat coupling constant
                    tauP=1.0)
            project.open_job(statepoint).init()


if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description="Initialize the data space.")
    parser.add_argument(
        'random',
        type=str,
        help="A string to generate a random seed.")
    parser.add_argument(
        '-n', '--num-replicas',
        type=int,
        default=1,
        help="Initialize multiple replications.")
    args = parser.parse_args()
```

(continues on next page)

(continued from previous page)

```
    # Generate an integer from the random str.
    try:
        random_seed = int(args.random)
    except ValueError:
        random_seed = int(sha1(args.random.encode()).hexdigest(), 16) % (10 ** 8)

    logging.basicConfig(level=logging.INFO)
    main(args, random_seed)
```

Using this script, one replica set (for a given random seed, e.g., 42) can then be initialized with:

```
$ python src/init.py 42
```

The simulation and analysis workflow is broken into three operations:

1. **init**: Initialize the simulation configuration.
2. **estimate**: Use the ideal gas law to estimate the expected volume.
3. **sample**: Carry out the simulation with HOOMD-blue.

Those three operations and corresponding condition functions are defined and implemented within a `src/project.py` module:

```
"""This module contains the operation functions for this project.

The workflow defined in this file can be executed from the command
line with

    $ python src/project.py run [job_id [job_id ...]]

See also: $ python src/project.py --help
"""
from flow import FlowProject


class Project(FlowProject):
    pass


@Project.operation
@Project.post.isfile('init.gsd')
def initialize(job):
    "Initialize the simulation configuration."
    import hoomd
    from math import ceil
    if hoomd.context.exec_conf is None:
        hoomd.context.initialize('')
    with job:
        with hoomd.context.SimulationContext():
            n = int(ceil(pow(job.sp.N, 1.0/3)))
            assert n**3 == job.sp.N
            hoomd.init.create_lattice(unitcell=hoomd.lattice.sc(a=1.0), n=n)
            hoomd.dump.gsd('init.gsd', period=None, group=hoomd.group.all())


@Project.operation
```

(continues on next page)

(continued from previous page)

```
@Project.post(lambda job: 'volume_estimate' in job.document)
def estimate(job):
    "Ideal-gas estimate operation."
    # Calculate volume using ideal gas law
    V = job.sp.N * job.sp.kT / job.sp.p
    job.document.volume_estimate = V


def current_step(job):
    import gsd.hoomd
    if job.isfile('dump.gsd'):
        with gsd.hoomd.open(job.fn('dump.gsd')) as traj:
            return traj[-1].configuration.step
    return -1


@Project.label
def sampled(job):
    return current_step(job) >= 5000


@Project.label
def progress(job):
    return '{}/4'.format(int(round(current_step(job) / 5000) * 4))


@Project.operation
@Project.pre.after(initialize)
@Project.post(sampled)
def sample(job):
    "Sample operation."
    import logging
    import hoomd
    from hoomd import md
    if hoomd.context.exec_conf is None:
        hoomd.context.initialize('')
    with job:
        with hoomd.context.SimulationContext():
            hoomd.init.read_gsd('init.gsd', restart='restart.gsd')
            group = hoomd.group.all()
            gsd_restart = hoomd.dump.gsd(
                'restart.gsd', truncate=True, period=100, phase=0, group=group)
            lj = md.pair.lj(r_cut=job.sp.r_cut, nlist=md.nlist.cell())
            lj.pair_coeff.set('A', 'A', epsilon=job.sp.epsilon, sigma=job.sp.sigma)
            md.integrate.mode_standard(dt=0.005)
            md.integrate.npt(
                group=group, kT=job.sp.kT, tau=job.sp.tau,
                P=job.sp.p, tauP=job.sp.tauP)
            hoomd.analyze.log('dump.log', ['volume'], period=100)
            hoomd.dump.gsd('dump.gsd', period=100, group=hoomd.group.all())
            try:
                hoomd.run_upto(5001)
            except hoomd.WalltimeLimitReached:
                logging.warning("Reached walltime limit.")
            finally:
                gsd_restart.write_restart()
                job.document['sample_step'] = hoomd.get_step()
```

(continues on next page)

(continued from previous page)

```


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

There are two additional label functions, which show whether the simulation has finished (**sampled**) and one that shows the rough progress in quarters (**progress**).

Execute the initialization and simulation with:

```
$ python src/project.py run
```

1.6.3 Integration with Sacred

Integrating a sacred experiment with **signac-flow** is very simple. Assuming the following sacred experiment defined in a `experiment.py` module:

```
from sacred import Experiment

ex = Experiment()


@ex.main
def hello(foo):
    print('hello', foo)


if __name__ == '__main__':
    ex.run_commandline()
```

Then we can integrate that experiment on a *per job* basis like this:

```
from flow import FlowProject
from sacred.observers import FileStorageObserver

from experiment import ex


class SacredProject(FlowProject):
    pass


@SacredProject.operation
def run_experiment(job):
    ex.add_config(** job.sp())
    ex.observers[:] = [FileStorageObserver.create(job.fn('my_runs'))]
    ex.run()


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

1.7 Recipes

This is a collection of recipes on how to solve typical problems using **signac**.

1.7.1 How to migrate (change) the data space schema.

Adding/renaming/deleting keys

Oftentimes, one discovers at a later stage that important keys are missing from the metadata schema. For example, in the tutorial we are modeling a gas using the ideal gas law, but we might discover later that important effects are not captured using this overly simplistic model and decide to replace it with the van der Waals equation:

$$\left(p + \frac{N^2 a}{V^2}\right)(V - Nb) = Nk_BT$$

Since the ideal gas law can be considered a special case of the equation above with $a = b = 0$, we could migrate all jobs with:

```
>>> for job in project:
...     job.sp.setdefault('a', 0)
...     job.sp.setdefault('b', 0)
...
```

The `setdefault()` function sets the value for a and b to 0 in case that they are not already present.

- To *delete* a key use `del job.sp['key_to_be_removed']`.
- To *rename* a key, use `job.sp.new_name = job.sp.pop('old_name')`.

Note: The `job.sp` and `job.doc` attributes provide all basic functions of a regular Python dict.

Apply document-wide changes

The safest approach to apply multiple document-wide changes is to replace the document in one operation. Here is an example on how we could recursively replace all dot (.)-characters with the underscore-character in **all** keys[1]:

```
import signac
from collections.abc import Mapping


def migrate(doc):
    if isinstance(doc, Mapping):
        return {k.replace('.', '_'): migrate(v) for k, v in doc.items()}
    else:
        return doc

for job in signac.get_project():
    job.sp = migrate(job.sp)
    job.doc = migrate(job.doc)
```

This approach makes it also easy to compare the pre- and post-migration states before actually applying them.

[1] The use of dots in keys is deprecated. Dots will be exclusively used to denote nested keywords in the future.

1.7.2 How to integrate signac-flow with MATLAB or other software without Python interface

The easiest way to integrate software that has no native Python interface is to implement **signac-flow** operations in combination with the `flow.cmd` decorator. Assuming that we have a MATLAB script called `prog.m` within the project root directory:

```
% prog.m
function []=prog(arg1, arg2)

display(arg1);
display(arg2);

exitcode = 0;
```

Then, we could implement a simple operation that passes it some metadata parameters like this:

```
@FlowProject.operation
@flow.cmd
def compute_volume(job):
    return "matlab -r 'prog {job.sp.foo} {job.sp.bar}' > {job.ws}/output.txt"
```

Executing this operation will store the output of the matlab script within the job's workspace within a file called `output.txt`.

1.7.3 How to implement MPI-parallelized operations

There are basically two strategies to implement `FlowProject` operations that are MPI-parallelized, one for external programs and one for Python scripts.

Tip: Fully functional scripts can be found in the signac-docs repository under `examples/MPI`.

MPI-operations with mpi4py or similar

Assuming that your operation is using mpi4py or similar, you do not have to change your code:

```
@FlowProject.operation
def hello_mpi(job):
    from mpi4py import MPI
    print("Hello from rank", MPI.COMM_WORLD.Get_rank())
```

You could run this operation directly with: `mpiexec -n 2 python project.py run -o hello_mpi`.

Note: This strategy might fail in cases where you cannot ensure that the MPI communicator is initialized *within* the operation function.

Danger: Read and write operations to the **job-/ and project-document** are not protected against race-conditions and should only be executed on one rank at a time. This can be ensured for example like this:

```
from mpi4py import MPI
comm = MPI.COMM_WORLD

if comm.Get_rank() == 0:
    job.doc.foo = 'abc'
comm.barrier()
```

MPI-operations with flow.cmd

Alternatively, you can implement an MPI-parallelized operation with the flow.cmd decorator, optionally in combination with the flow.directives decorator. This strategy lets you define the number of ranks directly within the code and is also the only possible strategy when integrating external programs without a Python interface.

Assuming that we have an MPI-parallelized program named my_program, which expects an input file as its first argument and which we want to run on two ranks, we could implement the operation like this:

```
@FlowProject.operation
@flow.cmd
@flow.directives(np=2)
def hello_mpi(job):
    return "mpiexec -n 2 mpi_program {job.ws}/input_file.txt"
```

The flow.cmd decorator instructs **signac-flow** to interpret the operation as a command rather than a Python function. The flow.directives decorator provides additional instructions on how to execute this operation and is not strictly necessary for the example above to work. However, some script templates, including those designed for HPC cluster submissions, will use the value provided by the np key to compute the required compute ranks for a specific submission.

Tip: You do not have to *hard-code* the number of ranks, it may be a function of the job, *e.g.*: flow.directives(np=lambda job: job.sp.system_size // 1000).

MPI-operations with custom script templates

Finally, instead of modifying the operation implementation, you could use a custom script template, such as this one:

```
{% extends base_script %}
{% block body %}
{% for operation in operations %}
mpiexec -n {{ operation.directives.np }} operation.cmd
{% endfor %}
{% endblock %}
```

Storing the above template in a file called templates/script.sh within your project root directory will prepend *every* operation command with mpiexec and so on.

1.7.4 How to run in containerized environments

Using **signac-flow** in combination with container systems such as docker or singularity is easily achieved by modifying the executable *directive*. For example, assuming that we wanted to use a singularity container named software.

simg, which is placed within the project root directory, we use the following directive to specify that a given operation is to be executed within then container:

```
@Project.operation
@flow.directives(executable='singularity exec software.simg python')
def containerized_operation(job):
    pass
```

If you are using the run command for execution, simply execute the whole script in the container:

```
$ singularity exec software.simg python project.py run
```

Attention: Many cluster environments will not allow you to **submit** jobs to the scheduler using the container image. This means that the actual submission, (e.g. python project.py submit or similar) will need to be executed with a **local** Python executable.

To avoid issues with dependencies that are only available in the container image, move imports into the operation function. Condition functions will be executed during the submission process to determine *what* to submit, so depedencies for those must be installed into the local environment as well.

Tip: You can define a decorator that can be reused like this:

```
def on_container(func):
    return flow.directives(executable='singularity exec software.simg python')(func)


@on_container
@Project.operation
def containerized_operation(job):
    pass
```

1.8 Tips and Tricks

This is a collection of miscellaneous tips that might help new users avoid common mistakes or provide useful hints to more experienced users.

1.8.1 How to design a good schema.

There is really no good answer on how to *generally* design a good schema because it is heavily dependent on the domain and the specific application. Nonetheless, there are some basic rules worth following:

1. Be descriptive. Although we are using short variable names in the tutorial, in general metadata keys should be as long as necessary for a third party to understand their meaning without needing to ask someone.
2. Any parameter which is likely to be *varied* at some point during the study should be part of the metadata right from the start to avoid needing to modify the schema later.
3. Take advantage of grouping keys! The job metadata mapping may be nested, just like any other Python dict.
4. Even if you don't use "official" schemas, consider to work out standardized schemas among your peers or with your collaborators.

1.8.2 Do not replicate job metadata in file names.

Many users, especially those new to **signac**, fall into the trap of storing metadata in filenames within a job's workspace even though that metadata is already encoded in the job itself.

Using the *Tutorial* project as an example, we might have stored the volume corresponding to the job at pressure 4 in a file called `volume_pressure_4.txt`. However, this is completely unnecessary since that information can already be accessed through the job *via* `job.sp.p`. Furthermore, creating files this way causes additional complications, such as the need to modify filenames whenever we operate on the data space. For example, extracting the volume from a particular job originally consisted of doing this:

```
volume = float(open(job.fn('volume.txt')).read())
```

Now, we instead need to adjust the filename for each job:

```
volume = float(open(job.fn('volume_pressure_{}.txt'.format(job.sp.p))).read())
```

In general, it is desirable to keep the filenames across the workspace as uniform as possible.

1.8.3 Do not hard-code job ids in your scripts.

While it is perfectly fine to copy & paste job ids during interactive work or for small tests, hard-coded job ids within code are almost always a bad sign. One of the main advantages of using **signac** for data management is that the schema is flexible and may be migrated at any time without too much hassle. That also means that existing ids will change and scripts that used them in a hard-coded fashion will fail.

Whenever you find yourself hard-coding ids into your code, consider replacing it with a function that uses the `find_jobs()` function instead.

1.9 Packages (API)

The **signac** framework is currently comprised of three packages. You only have to install those that provide the functionality you need, however both **signac-flow** and **signac-dashboard** require the **signac** core package.

The links below lead to the package-specific documentation, including a complete API documentation and changelogs.

signac (core) The *core* **signac** package implements a simple, serverless, distributed database directly on the file system. It allows you manage files on the file system and associate them with JSON-encoded metadata.

This metadata is immediately searchable, which allows you to find and select data for specific data sub spaces.

signac-flow The **signac-flow** package allows us to implement workflows that operate on a *data space* managed with signac. These workflows range from simple, linear workflows, to large workflows with complex dependencies between operations. These workflows can be executed directly on the command line or submitted to a cluster scheduling system, which is relevant for users who work in high-performance computing (HPC) environments.

signac-dashboard The **signac-dashboard** allows users to browse their **signac**-managed data spaces through a web-based GUI. The dashboard can be used for visualization and analysis and is very helpful when sharing data with collaborators.

1.10 Community

1.10.1 Chat Support

To best way to get support for **signac** is to join the signac-gitter channel. The developers and other users are usually able to help within a few minutes. Alternatively, you can send an email to signac-support@umich.edu.

Please use the issue tracker of the individual *packages* to file bug reports or request new features!

1.10.2 Contributions

Contributions to **signac** are very welcome! We highly appreciate contributions in the form of **user feedback** and **bug reports** on the gitter channel, the issue trackers of individual *packages*, or via email. Developers are invited to contribute to the framework by pull request to the appropriate package repository. The source code for all packages is hosted on bitbucket. We recommend discussing new features in form of a proposal on the issue tracker for the appropriate project prior to development.

All code contributed via pull request needs to adhere to the following guidelines:

1. Most signac packages follow the git-flow branching model. Bug fixes should be implemented in a branch based on `master`, while new features should be developed within a branch based on `develop`.
2. All code needs to adhere to the PEP8 style guide, with the exception that a line may have up to 100 characters.
3. New features must be properly documented and tested with automated unit tests.
4. Non-obvious code passages should be extensively documented.
5. Changes must generally be backwards-compatible.
6. All packages targeted to be used within high-performance computing environments should support Python versions 2.7+ and 3.4+ and keep the number of *hard* dependencies to a miminum.

Tip: During continuous integration, the code is checked automatically with Flake8. Run the following commands to set up a pre-commit hook that will ensure your code is compliant before committing:

```
flake8 --install-hook git
git config --bool flake8.strict true
```

Note: Please see the individual package documentation for detailed guidelines on how to contribute to a specific package.

1.11 License

Important: Plase see the individual package documentation for detailed licensing information.

In general, all packages that are part of the **signac** framework are licensed under the **BSD-3-Clause License**:

```
    BSD 3-Clause License for the software signac.

    Copyright (c) 2016-2018, The Regents of the University of Michigan
    All rights reserved.

    Redistribution and use in source and binary forms, with or without modification,
    are permitted provided that the following conditions are met:

    1. Redistributions of source code must retain the above copyright notice, this
       list of conditions and the following disclaimer.

    2. Redistributions in binary form must reproduce the above copyright notice,
       this list of conditions and the following disclaimer in the documentation
       and/or other materials provided with the distribution.

    3. Neither the name of the copyright holder nor the names of its contributors may
       be used to endorse or promote products derived from this software without
       specific prior written permission.

    THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
    ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
    WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
    IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,
    INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
    BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
    OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
    WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
    ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
    POSSIBILITY OF SUCH DAMAGE.
```

Most source code shown as part of the documentation is released into the public domain:

```
This is free and unencumbered software released into the public domain.

Anyone is free to copy, modify, publish, use, compile, sell, or
distribute this software, either in source code form or as a compiled
binary, for any purpose, commercial or non-commercial, and by any
means.

In jurisdictions that recognize copyright laws, the author or authors
of this software dedicate any and all copyright interest in the
software to the public domain. We make this dedication for the benefit
of the public at large and to the detriment of our heirs and
successors. We intend this dedication to be an overt act of
relinquishment in perpetuity of all present and future rights to this
software under copyright law.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES OR
OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
OTHER DEALINGS IN THE SOFTWARE.

For more information, please refer to <https://unlicense.org>
```

1.12 How to cite signac

Please acknowledge the use of this software within the body of your publication for example by copying or adapting the following formulation:

The computational workflow in general and data management in particular for this publication was primarily supported by the signac data management framework [1, 2].

[1] C. S. Adorf, P. M. Dodd, V. Ramasubramani, and S. C. Glotzer. Simple data and workflow management with the signac framework. Comput. Mater. Sci., 146(C):220-229, 2018. DOI:10.1016/j.commatsci.2018.01.035.

[2] C. S. Adorf, P. M. Dodd, V. Ramasubramani, B. Swerdlow, J. Glaser, and B. Dice. csadorf/signac v0.9.2. dec 2017. URL: https://doi.org/10.5281/zenodo.1117952, DOI:10.5281/zenodo.1117952.

References for other specific release versions can be found here. A preprint of the paper published in the Journal of Computational Materials Science is available on the arXiv:.

To cite these references, you can use the following BibTeX entries:

```
@article{signac_commat,
  author        = {Carl S. Adorf and
                   Paul M. Dodd and
                   Vyas Ramasubramani and
                   Sharon C. Glotzer},
  title         = {Simple data and workflow management with the signac framework},
  journal       = {Comput. Mater. Sci.},
  volume        = {146},
  number        = {C},
  year          = {2018},
  pages         = {220-229},
  doi           = {10.1016/j.commatsci.2018.01.035}
}
@misc{signac_0_9_2,
  author       = {Carl S. Adorf and
                  Paul M. Dodd and
                  Vyas Ramasubramani and
                  Benjamin Swerdlow and
                  Jens Glaser and
                  Bradley Dice},
  title        = {csadorf/signac v0.9.2},
  month        = {dec},
  year         = {2017},
  doi          = {10.5281/zenodo.1117952},
  url          = {https://doi.org/10.5281/zenodo.1117952}
}
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index