# Contents

1 Contents
  1.1 Overview ................................................................. 3
  1.2 Installation ............................................................... 3
  1.3 Tutorial ...................................................................... 5
  1.4 Reference ................................................................. 28
  1.5 Quick Reference ......................................................... 46
  1.6 How to cite signac ....................................................... 51
  1.7 API ........................................................................ 52

2 Support and Contribution .............................................. 125

3 Indices and tables ............................................................ 127

Python Module Index ......................................................... 129
The signac framework aids in the management of large and heterogeneous data spaces. It provides a simple and robust data model to create a well-defined indexable storage layout for data and metadata. This makes it easier to operate on large data spaces, streamlines post-processing and analysis and makes data collectively accessible.
Overview

Signac is designed to assist large-scale multidimensional computational data generation and analysis. It is assumed that the work can be divided into so called projects, where each project is vaguely confined by roughly similar structured data, e.g., a parameter study.

We define the process of generating or manipulating data a job. Every job operates on a set of well-defined unique parameters, which define the job’s context. This means that all data is uniquely addressable from the associated parameters.

You can use signac to manage a project based on this data model and index data for export into a database.

Installation

The recommendend installation method for signac is via conda or pip. The software is tested for python versions 2.7.x and 3.x and does not have any hard dependencies, i.e. there are no packages or libraries required to use the core signac functions. However, some extra features, such as the database integration require additional packages.
Install with conda

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

```
$ conda config --add channels conda-forge
```

Once the conda-forge channel has been enabled, signac can be installed with:

```
$ conda install signac
```

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

```
$ conda update signac
```

Install with pip

To install the package with the package manager pip, execute

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

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

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

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

Consider to install optional dependencies:

```
$ pip install pymongo passlib bcrypt --user
```

Source Code Installation

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

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

Consider to install *optional dependencies*.

Optional dependencies

Unless you install via conda, optional dependencies are not installed automatically. In case you want to use extra features that require external packages, you need to install these manually.

Extra features with dependencies:

**MongoDB database backend** required: pymongo

  recommended: passlib, bcrypt
Tutorial

Requirements

The tutorial assumes a basic proficiency in python. The first chapter will take about 20 to 30 minutes to complete.

This tutorial demonstrates how to implement a basic computational workflow with signac. The signac framework assists us in conducting a computational investigation by managing the data space for us. This means that we do not need to worry about how to organize our data on disk and how to keep track of meta data.

As a beginner, make sure to complete the first chapter. The second chapter deals with more advanced topics that may not be of interest to everyone.

Tip: This tutorial and other examples are available as interactive jupyter notebooks online!

Contents:

1.1 Getting started

Prerequisites

Installation

This tutorial requires signac, so make sure to install the package before starting. The easiest way to do so is using conda:

$ conda config --add channels conda-forge
$ conda install signac

or pip:

pip install signac --user

Please refer to the documentation for detailed instructions on how to install signac.

After successful installation, the following cell should execute without error:

In [1]: import signac
   assert signac.__version__ >= '0.8.0'

We start by removing all data which might be left-over from previous executions of this tutorial.

In [2]: % rm -rf projects/tutorial/workspace

A minimal example

For this tutorial we want to compute the volume of an ideal gas as a function of its pressure and thermal energy using the ideal gas equation

\[ pV = NkT \]

\( N \) refers to the system size, \( p \) to the pressure, \( kT \) to the thermal energy and \( V \) is the volume of the system.

In [3]: def V_idg(N, kT, p):
   return N * kT / p
We can execute the complete study in just a few lines of code. First, we initialize the project directory and get a project handle:

```python
In [4]: import signac
   
   project = signac.init_project('TutorialProject', 'projects/tutorial')
```

We iterate over the variable of interest \(p\) and construct a complete state point \(sp\) which contains all the meta data associated with our data. In this simple example the meta data is very compact, but in principle the state point may be highly complex.

Next, we obtain a job handle and store the result of the calculation within the job document. The job document is a persistent dictionary for storage of simple key-value pairs. Here, we exploit that the state point dictionary \(sp\) can easily be passed into the \(V_{idg}\) function using the keyword expansion syntax (**sp**).

```python
In [5]: for p in 0.1, 1.0, 10.0:
   ...:     sp = {'p': p, 'kT': 1.0, 'N': 1000}
   ...:     job = project.open_job(sp)
   ...:     job.document['V'] = V_idg(**sp)
```

We can then examine our results by iterating over the data space:

```python
In [6]: for job in project:
   ...:     print(job.sp.p, job.document['V'])
   ...:
   ...:
10.0 100.0
0.1 10000.0
1.0 1000.0
```

That’s it.

... Ok, there’s more... Let’s have a closer look at the individual components.

**The Basics**

The signac data management framework assists the user in managing the data space of individual projects. All data related to one or multiple projects is stored in a workspace, which by default is a directory called workspace within the project’s root directory.

```python
In [7]: print(project.root_directory())
   print(project.workspace())
   ...:
/home/johndoe/signac-examples/notebooks/projects/tutorial
/home/johndoe/signac-examples/notebooks/projects/tutorial/workspace
```

The core idea is to tightly couple state points, unique sets of parameters, with their associated data. In general, the parameter space needs to contain all parameters that will affect our data.

For the ideal gas that is a 3-dimensional space spanned by the thermal energy \(kT\), the pressure \(p\) and the system size \(N\). These are the input parameters for our calculations, while the calculated volume \(V\) is the output data. In terms of signac this relationship is represented by an instance of Job.

We use the open_job() method to get a job handle for a specific set of input parameters.

```python
In [8]: job = project.open_job({'p': 1.0, 'kT': 1.0, 'N': 1000})
```

The job handle tightly couples our input parameters \((p, kT, N)\) with the storage location of the output data. You can inspect both the input parameters and the storage location explicitly:

```python
In [9]: print(job.statepoint())
   print(job.workspace())
   ...:
```

...
For convenience, a job’s state point may also be accessed via the short-hand sp attribute. For example, to access the pressure value \( p \) we can use either of the two following expressions:

```python
In [10]:
   print(job.statepoint()['p'])
   print(job.sp.p)
```

1.0
1.0

Each job has a unique id representing the state point. This means opening a job with the exact same input parameters is guaranteed to have the exact same id.

```python
In [11]:
   job2 = project.open_job({'kT': 1.0, 'N': 1000, 'p': 1.0})
   print(job.get_id(), job2.get_id())
```

```
ee617ad585a90809947709a7a45dda9a ee617ad585a90809947709a7a45dda9a
```

The job id is used to uniquely identify data associated with a specific state point. Think of the job as a container that is used to store all data associated with the state point. For example, it should be safe to assume that all files that are stored within the job’s workspace directory are tightly coupled to the job’s statepoint.

```python
In [12]:
   print(job.workspace())
```

```
/home/johndoe/signac-examples/notebooks/projects/tutorial/workspace/ee617ad585a90809947709a7a45dda9a
```

Let’s store the volume calculated for each state point in a file called \( V.txt \) within the job’s workspace.

```python
In [13]:
   import os
   fn_out = os.path.join(job.workspace(), 'V.txt')
   with open(fn_out, 'w') as file:
      V = V_idg(** job.statepoint())
      file.write(str(V) + '\n')
```

Because this is such a common pattern, signac allows you to short-cut this with the job.fn() method.

```python
In [14]:
   with open(job.fn('V.txt'), 'w') as file:
      V = V_idg(** job.statepoint())
      file.write(str(V) + '\n')
```

Sometimes it is easier to temporarily switch the current working directory while storing data for a specific job. For this purpose, we can use the Job object as context manager. This means that we switch into the workspace directory associated with the job after entering, and switch back into the original working directory after exiting.

```python
In [15]:
   with job:
      with open('V.txt', 'w') as file:
         file.write(str(V) + '\n')
```

Another alternative to store light-weight data is the job document as shown in the minimal example. The job document is a persistent JSON storage file for simple key-value pairs.

```python
In [16]:
   job.document['V'] = V_idg(** job.statepoint())
   print(job.statepoint(), job.document)
```

```
'p': 1.0, 'N': 1000, 'kT': 1.0 'V': 1000.0
```

Since we are usually interested in more than one state point, the standard operation is to iterate over all variable(s) of interest, construct the full state point, get the associated job handle, and then either just initialize the job or perform the full operation.

```python
In [17]:
   for pressure in 0.1, 1.0, 10.0:
      statepoint = {'p': pressure, 'kT': 1.0, 'N': 1000}
```
job = project.open_job(statepoint)
job.document['V'] = V_idg(** job.statepoint())

Let’s verify our result by inspecting the data.

In [18]: for job in project:
   print(job.statepoint(), job.document)

'p': 10.0, 'N': 1000, 'kT': 1.0 'V': 100.0
'p': 0.1, 'N': 1000, 'kT': 1.0 'V': 10000.0
'p': 1.0, 'N': 1000, 'kT': 1.0 'V': 1000.0

Those are the basics for using signac for data management. The next section demonstrates how to explore an existing data space.

1.2 Exploring Data

Finding jobs

In section one of this tutorial, we evaluated the ideal gas equation and stored the results in the job document and in a file called V.txt. Let’s now have a look at how we can explore our data space for basic and advanced analysis.

We already saw how to iterate over the complete data space using the “for job in project” expression. This is a short-hand notation for “for job in project.find_jobs()”, meaning: “find all jobs”.

Instead of finding all jobs, we can also find a subset using filters.

Let’s get started by getting a handle on our project using the get_project() function. We don’t need to initialize the project again, since we already did that in section 1.

In [1]: import signac
   project = signac.get_project('projects/tutorial')

Next, we assume that we would like to find all jobs, where \( p = 10.0 \). For this, we can use the find_jobs() method, which takes a dictionary of parameters as filter argument.

In [2]: for job in project.find_jobs({'p': 10.0}):
   print(job.statepoint())

'p': 10.0, 'kT': 1.0, 'N': 1000

In this case, that is of course only a single job.

You can execute the same kind of find operation on the command line with $ signac find, as will be shown later.

While the filtering method is optimized for a simple dissection of the data space, it is possible to construct more complex query routines for example using list comprehensions.

This is an example for how to select all jobs where the pressure \( p \) is greater than 0.1:

In [3]: jobs_p_gt_0_1 = [job for job in project if job.sp.p > 0.1]
   for job in jobs_p_gt_0_1:
       print(job.statepoint(), job.document)

'p': 10.0, 'kT': 1.0, 'N': 1000 'V': 100.0
'p': 1.0, 'kT': 1.0, 'N': 1000 'V': 1000.0

Finding jobs by certain criteria requires an index of the data space. In all previous examples this index was created implicitly, however depending on the data space size, it may make sense to create the index explicitly for multiple uses. This is shown in the next section.
Indexing

An index is a complete record of the data and its associated metadata within our project’s data space. To generate an index for our project’s data space, use the `index()` method:

```python
In [4]: for doc in project.index():
    print(doc)

'root': '/home/johndoe/signac-examples/notebooks/projects/tutorial/workspace', 'V': 100.0, 'signac_id': '5a456c131b0c5897804a4af8e77df5aa', '_id': '5a456c131b0c5897804a4af8e77df5aa', 'statepoint': 'p': 10.0, 'kT': 1.0, 'N': 1000

'root': '/home/johndoe/signac-examples/notebooks/projects/tutorial/workspace', 'V': 10000.0, 'signac_id': '5a6c687f7655319db24de59a2336eff8', '_id': '5a6c687f7655319db24de59a2336eff8', 'statepoint': 'p': 0.1, 'kT': 1.0, 'N': 1000

'root': '/home/johndoe/signac-examples/notebooks/projects/tutorial/workspace', 'V': 1000.0, 'signac_id': 'ee617ad585a90809947709a7a45dda9a', '_id': 'ee617ad585a90809947709a7a45dda9a', 'statepoint': 'p': 1.0, 'kT': 1.0, 'N': 1000
```

Using an index to operate on data is particularly useful in later stages of a computational investigation, where data may come from different projects and the actual storage location of files is less important.

You can store the index wherever it may be useful, e.g., a file, a database, or even just in a variable for repeated find operations within one script. The `signac` framework provides the `Collection` class, which can be utilized to manage indexes in memory and on disk.

```python
In [5]: index = signac.Collection(project.index())
for doc in index.find({'statepoint.p': 10.0}):
    print(doc)

'root': '/home/johndoe/signac-examples/notebooks/projects/tutorial/workspace', 'V': 100.0, 'signac_id': '5a456c131b0c5897804a4af8e77df5aa', '_id': '5a456c131b0c5897804a4af8e77df5aa', 'statepoint': 'p': 10.0, 'kT': 1.0, 'N': 1000
```

Views

Sometimes we want to examine our data on the file system directly. However, the file paths within the workspace are obfuscated by the job id. The solution is to use views, which are human-readable, maximally compact hierarchical links to our data space.

To create a linked view, we simply execute the `create_linked_view()` method within python or the `$ signac view` command on the command line.

```python
In [6]: project.create_linked_view(prefix='projects/tutorial/view')
% ls projects/tutorial/view
p_0.1/   p_l.0/   p_l0.0/
```

The view paths only contain parameters which actually vary across the different jobs. In this example, that is only the pressure $p$.

This allows us to examine the data with highly-compact human-readable path names:

```python
In [7]: % ls 'projects/tutorial/view/p_l.0/job/'
   % cat 'projects/tutorial/view/p_l.0/job/V.txt'
V.txt    signac_job_document.json  signac_statepoint.json
1000.0
```

**NOTE:** Update your view after adding or removing jobs by executing the `view` command for the same prefix again!

Tip: Consider creating a linked view for large data sets on an **in-memory** file system for best performance!

The next section will demonstrate how to implement a basic, but complete workflow for more expensive computations.

### 1.3. A Basic Workflow

This part of the tutorial requires `numpy`. 

#### 1.3. Tutorial
Operations

For this part of the tutorial we will imagine that we are still not convinced of the pressure-volume relations that we just “discovered” and that calculating the volume is actually a very expensive procedure, such as a many particle simulation.

We emulate this by adding an optional cost argument to our volume calculation function:

```
In [1]: from time import sleep
def V_idg(N, p, kT, cost=0):
sleep(cost)
return N * kT / p
```

It is useful to think of each modification of the workspace, that includes addition, modification, and removal of data, in terms of an operation.

An operation should take only one(!) argument: the job handle.

Any additional arguments may represent hidden state point parameters which would lead to a loss of provenance and possibly render our data space inconsistent.

The following function is an example for an operation:

```
In [2]: def compute_volume(job):
   print('compute volume', job)
   V = V_idg(cost=1, **job.statepoint())
   job.document['V'] = V
   with open(job.fn('V.txt'), 'w') as file:
       file.write(str(V) + '

This operation computes the volume solely based on the state point parameters and stores the results such that they are clearly associated with the job, i.e., in the job document and in a file within the job’s workspace.

Please note, that the only reason for storing the same result in two different ways is for demonstration purposes.

Execution

To execute our first data space operation, we simply loop through our project’s data space:

```
In [3]: import signac
project = signac.get_project('projects/tutorial')
for job in project:
    compute_volume(job)
```

Data Space Initialization

Since our operation is now more expensive, it is a good idea to split initialization and execution. Let’s initialize a few more state points in one go:

```
In [4]: import signac
import numpy as np
project = signac.get_project(root='projects/tutorial')
```
```python
def init_statepoints(n):
    for p in np.linspace(0.1, 10.0, n):
        sp = {'p': p, 'kT': 1.0, 'N': 1000}
        job = project.open_job(sp)
        job.init()
        print('initialize', job)
```

```none
initialize 5a6c687f7655319db24de59a2336eff8
initialize d03270cddbabe73c8bb1d9fa0ab370264
initialize 973e29d6a4ed6cf7329c03c77df7f645
initialize 4cf2795722061df825ec9a4d5e31e494
initialize 5a456c131b0c5897804a4af8e77df5aa
```

We see that initializing more jobs and even reinitializing old jobs is no problem. However, since our calculation will be “expensive”, we would want to skip the computation whenever the result is already available.

One possibility is to add a simple check before executing the computation:

```python
In [5]: for job in project:
   if 'V' not in job.document:
       compute_volume(job)
```

```none
compute volume 4cf2795722061df825ec9a4d5e31e494
compute volume 973e29d6a4ed6cf7329c03c77df7f645
compute volume d03270cddbabe73c8bb1d9fa0ab370264
```

**Classification**

It would be even better, if we could get an overview of which state points have been calculated and which not. We call this a project’s *status*.

Before we continue, let’s initialize a few more state points.

```none
In [6]: init_statepoints(10)
initialize 5a6c687f7655319db24de59a2336eff8
initialize 22582e83c6b12336526ed304d4378ff8
initialize c0ab2e09a6f878019a6057175bf718e6
initialize 9110d0837ad93ff6b4013bae30091edd
initialize b45a2485a44a46364cc60134360ea5af
initialize 9110d0837ad93ff6b4013bae30091edd
initialize 5a456c131b0c5897804a4af8e77df5aa
initialize 665547b134fe40de5b2c7ace4204783
initialize 8629822576deb2bfeffaa56787ca348
initialize e8186cb986e18a82f331d51a7b8c8c15
initialize 5a456c131b0c5897804a4af8e77df5aa
```

Next, we implement a `classify()` generator function, which labels a *job* based on certain conditions:

```python
In [7]: def classify(job):
   yield 'init'
       if 'V' in job.document and job.isfile('V.txt'):
           yield 'volume-computed'
```

Our classifier will always yield the *init* label, but the *volume-computed* label is only yielded if the result has been computed and stored both in the *job document* and as a text file. We can then use this function to get an overview of our project’s status.

```python
In [8]: print('Status: {}'.format(project))
   for job in project:
       labels = ','.join(classify(job))
```

1.3. Tutorial

11
Using only simple classification functions, we already get a very good grasp on our project’s overall status.

Furthermore, we can use the classification labels for controlling the execution of operations:

```python
In [9]:
   for job in project:
       if 'volume-computed' not in labels:
           compute_volume(job)
```

```
compute volume 05061d2acea19d2d9a25ac3360f70e04
compute volume 22582e83c6b12336526ed304d4378ff8
compute volume 4cf2795722061df825ec9a4d5e31e494
compute volume 5a456c131b0c5897804a4af8e77df5a
compute volume 5a6c687f765319db24de59a2336eff8
compute volume 665547b1344fe40de5b2c7ace4204783
compute volume 8629822576debc2bfbefaf56787ca348
compute volume 9110d0837ad93ff6b4013bae30091edd
compute volume b45a2485a44a46364cc60134360ea5af
compute volume c0ab2e09a6ff878019a6057175bff718e6
compute volume e8186b9b68e18a82f331d51a7b8c8c15
```

**Parallelization**

So far, we have executed all operations in serial using a simple for-loop. We will now learn how to easily parallelize the execution!

Instead of using a for-loop, we can also take advantage of Python’s built-in map-operator:

```python
In [10]:
   list(map(compute_volume, project))
   print('Done. ')
```

```
compute volume 05061d2acea19d2d9a25ac3360f70e04
compute volume 22582e83c6b12336526ed304d4378ff8
compute volume 4cf2795722061df825ec9a4d5e31e494
compute volume 5a456c131b0c5897804a4af8e77df5a
compute volume 5a6c687f765319db24de59a2336eff8
compute volume 665547b1344fe40de5b2c7ace4204783
compute volume 8629822576debc2bfbefaf56787ca348
compute volume 9110d0837ad93ff6b4013bae30091edd
compute volume b45a2485a44a46364cc60134360ea5af
compute volume c0ab2e09a6ff878019a6057175bff718e6
compute volume e8186b9b68e18a82f331d51a7b8c8c15
```
Done.

Using the `map()` expression makes it trivial to implement parallelization patterns, for example, using a process `Pool`:

```
In [11]: from multiprocessing import Pool

    with Pool() as pool:
        pool.map(compute_volume, project)
```

Or a ThreadPool:

```
In [12]: from multiprocessing.pool import ThreadPool

    with ThreadPool() as pool:
        pool.map(compute_volume, project)
```

Uncomment and execute the following line if you want to remove all data and start over.

```
In [13]: # % rm -r projects/tutorial/workspace
```

In this section we learned how to create a simple, yet complete workflow for our computational investigation. In the next section we will learn how to adjust the data space, e.g., modify existing state point parameters.

## 1.4 Modifying the Data Space

It is very common that we discover at a later stage that we need to revise our computational protocol. In this case we need to carefully update existing job state points and the associated data.

Let’s assume that we realize that the ideal gas law is not sufficiently exact for our needs, so we’re going to use the van der Waals equation (vdW) for a more exact estimation of the volume for each state point.
\[ (p + \frac{N^2a}{V^2})(V - Nb) = NkT, \]

where \( a \) and \( b \) are two additional parameters. For \( a = b = 0 \) this equation is identical to the ideal gas equation.

We start by implementing a function to calculate the volume for a given statepoint.

```python
In [1]: import numpy as np
def V_vdW(p, kT, N, a=0, b=0):
    
    '''Solve the van der Waals equation for V.'''
    coeffs = [p, -(kT * N + p * N * b), a * N**2, - a * N**3 * b]
    V = sorted(np.roots(coeffs))
    return np.real(V).tolist()
```

You will notice that this equation is a cubic polynomial and therefore has 3 possible solutions instead of only one!

```python
In [2]: print(V_vdW(1.0, 1.0, 1000))
[0.0, 0.0, 1000.0]
```

That is because the vdW system has a critical point and up to three possible solutions. These solutions correspond to a liquid, a gaseous and a meta-stable phase.

We want to make the old data compatible with the new protocol, which requires two modifications of the existing data space:

1. We need to add parameters \( a \) and \( b \) to each statepoint and set them to zero.
2. The former value \( V \) needs to be relabeled \( V_{\text{gas}} \) and we add a zero-value for \( V_{\text{liq}} \).

We previously learned that we can use the Job.sp attribute interface to access individual state point values. We can use the same interface to modify existing state point parameters.

```python
In [3]: import signac

project = signac.get_project('projects/tutorial')

    for job in project:
        if 'a' not in job.sp:
            job.sp.a = 0
        if 'b' not in job.sp:
            job.sp.b = 0
```

Please checkout the section on State Point Modifications in the reference documentation for a detailed description on how to modify state points.

Next, we need to update the existing volume data. We check whether the job document has a \( V \) value and replace it with \( V_{\text{liq}} \) and \( V_{\text{gas}} \). The \( V.txt \) files will be rewritten to contain two comma-separated values.

```python
In [4]: for job in project:
    if 'V' in job.document:
        job.document['V_liq'] = 0
        job.document['V_gas'] = job.document.pop('V')
        with open(job.fn('V.txt'), 'w') as file:
            file.write('{},{}
'.format(0, job.document['V_gas']))
```

Let's verify our modifications!

```python
In [5]: for job in project:
    
    print(job.statepoint(), job.document)
    
    'kT': 1.0, 'a': 0, 'N': 1000, 'p': 3.4000000000000004, 'b': 0 'V_gas': 294.1176470588235, 'V_liq': 0
    'kT': 1.0, 'a': 0, 'N': 1000, 'p': 0.1, 'b': 0 'V_gas': 10000.0, 'V_liq': 0
    'kT': 1.0, 'a': 0, 'N': 1000, 'p': 7.525, 'b': 0 'V_gas': 132.8903544850498, 'V_liq': 0
    'kT': 1.0, 'a': 0, 'N': 1000, 'p': 5.6, 'b': 0 'V_gas': 178.57142857142858, 'V_liq': 0
```
Next, we add a few state points with known parameters.

```python
In [6]: vdW = {
    # Source: https://en.wikipedia.org/wiki/Van_der_Waals_constants_(data_page)
    'ideal gas': {'a': 0, 'b': 0},
    'argon': {'a': 1.355, 'b': 0.03201},
    'water': {'a': 5.536, 'b': 0.03049},
}
def calc_volume(job):
    V = V_vdW(** job.statepoint())
    job.document['V_liq'] = min(V)
    job.document['V_gas'] = max(V)
    with open(job.fn('V.txt'), 'w') as file:
        file.write(','.join(str(min(V)), str(max(V))))

for fluid in vdW:
    for p in np.linspace(0.1, 10.0, 10):
        sp = {'N': 1000, 'p': p, 'kT': 1.0}
        sp.update(vdW[fluid])
        job = project.open_job(sp)
        job.document['fluid'] = fluid
        calc_volume(job)
```

The `fluid` label is stored in the job document as a hint, which parameters were used, however they are deliberately not part of the state point, since our calculation is only based on the parameters $N$, $kT$, $p$, $a$, and $b$. In general, all state point variables should be independent of each other.

Let's inspect the results:

```python
In [7]: ps = set((job.statepoint()['p']
    for job in project))
for fluid in sorted(vdW):
    print(fluid)
    for p in sorted(ps):
        jobs = project.find_jobs({'p': p}, doc_filter={'fluid': fluid})
        for job in jobs:
            print(round(p, 2), round(job.document['V_liq'], 4), round(job.document['V_gas']),

argon
0.1 32.8041 8430.94
1.2 32.8034 416.27
2.3 32.8027 216.99
3.4 32.8019 146.66
4.5 32.8012 110.72
5.6 32.8005 88.89
6.7 32.7998 74.23
7.8 32.799 63.71
8.9 32.7983 55.79
10.0 32.7976 49.61
```

1.3. Tutorial
ideal gas
0.1 0.0 10000.0
1.2 0.0 833.33
2.3 0.0 434.78
3.4 0.0 294.12
4.5 0.0 222.22
5.6 0.0 178.57
6.7 0.0 149.25
7.8 0.0 128.36
8.9 0.0 112.36
10.0 0.0 100.0

water
0.1 30.6598 4999.92
1.2 30.6598 416.58
2.3 30.6597 217.31
3.4 30.6597 146.97
4.5 30.6597 111.03
5.6 30.6596 89.2
6.7 30.6596 74.54
7.8 30.6596 64.02
8.9 30.6595 56.1
10.0 30.6595 49.92

We observe that the liquid phase is almost incompressible, while the gas phase is strongly pressure dependent.

The final section of the first chapter demonstrates how to interact with a signac on the command line.

1.5 Command Line Interface (CLI)

The following section demonstrates how to use the signac command line interface (CLI). The CLI allows you to interact with your data space without python, which may be advantageous in various situations, e.g., for scripting or data exploration.

You will find that for many of the functions introduced earlier there is an equivalent CLI command.

The CLI is accessed via the top-level signac command. You can get help about the various functions with the -h or --help argument.

In [1]: %bash
   signac --help

usage: signac [-h] [--debug] [--version] [-y]
   init,project,job,statepoint,move,clone,index,find,view,config ...

signac aids in the management, access and analysis of large-scale computational investigations.

positional arguments:
   init,project,job,statepoint,move,clone,index,find,view,config

optional arguments:
   -h, --help               show this help message and exit
   --debug                 Show traceback on error for debugging.
   --version               Display the version number and exit.
   -y, --yes               Answer all questions with yes. Useful for scripted interaction.
To interact with a project on the command line, the current working directory needs to be within or below the project’s root directory. Let’s start by resetting the designated project root directory for this section of the tutorial.

```
In [2]: % rm -rf projects/tutorial/cli
    % mkdir -p projects/tutorial/cli
```

Next we switch the current working directory to the project root directory.

```
In [3]: % cd projects/tutorial/cli
/home/johndoe/signac-examples/notebooks/projects/tutorial/cli
```

Then we initialize the project.

```
In [4]: %bash
    signac init TutorialCLIProject
```

Initialized project 'TutorialCLIProject'.

We can verify the project configuration using the `signac project` command.

```
In [5]: %bash
    signac project
    signac project --workspace
```

```
TutorialCLIProject
/home/johndoe/signac-examples/notebooks/projects/tutorial/cli/workspace
```

We access a job by providing the state point on the command line in JSON format †).

†) The JSON format requires double quotes for keys.

```
In [6]: %bash
    signac job '{"kT": 1.0, "p": 1.0, "N": 1000}'
```

```
ee617ad585a90809947709a7a45dda9a
```

By default this will print the associated job id to STDOUT.

Instead of the job id, we can also get the path to the job’s workspace.

```
In [7]: %bash
    signac job '{"kT": 1.0, "p": 1.0, "N": 1000}' --workspace
```

```
/home/johndoe/signac-examples/notebooks/projects/tutorial/cli/workspace/ee617ad585a90809947709a7a45dda9a
```

Please not, that obtaining the path in this way does not necessarily mean that the path exists. However, we can initialize the job and create the workspace using the `-c` or `--create` argument.

```
In [8]: %bash
    signac job '{"kT": 1.0, "p": 1.0, "N": 1000}' --create
```

```
ee617ad585a90809947709a7a45dda9a
```

We can use the `signac statepoint` command to get the statepoint associated with the initialized job.

```
In [9]: %bash
    signac statepoint ee617ad585a90809947709a7a45dda9a
```

```
"p": 1.0, "kT": 1.0, "N": 1000
```

Usually we will not provide statepoints on the command line, but read them from a file. Let’s create a statepoint file with one statepoint:

```
In [10]: %bash
    echo '{"kT": 1.0, "p": 0.1, "N": 1000}' > statepoint.txt
    cat statepoint.txt
```

```
"kT": 1.0, "p": 0.1, "N": 1000
```
We can pipe the content of this file into the *signac* CLI to get the corresponding *job id*.

In [11]: ```bash
cat statepoint.txt | signac job
```
Error: expected string or buffer

We will reproduce the ideal gas project from section 1.1 to generate some data for the following examples.

In [12]: ```python
import signac

def V_idg(N, p, kT):
    return N * kT / p

project = signac.get_project()
for p in 0.1, 1.0, 10.0:
    sp = {'p': p, 'kT': 1.0, 'N': 1000}
    job = project.open_job(sp)
    job.document['V'] = V_idg(**sp)
```

We can use the *signac find* command to find all jobs within our project’s workspace.

In [13]: ```bash
signac find
```
```
5a456c131b0c5897804a4af8e77df5aa
5a6c687f7655319db24de59a2336eff8
ee617ad585a90809947709a7a45da9a
```

Just like with `project.find_jobs()` we can provide a filter argument to find a subset of jobs matching the given filter. Here we get all jobs with a pressure of 0.1:

In [14]: ```bash
signac find "{'p': 0.1}'
```
```
5a6c687f7655319db24de59a2336eff8
```

In this example, that is of course only one job.

Similarly, we can also filter based on information in the job document. Here, we find all jobs that have a volume corresponding to a pressure of 1 (volume = 1000*1/1 = 1000).

In [15]: ```bash
signac find --doc-filter "{'V': 1000.0}'
```
```
ee617ad585a90809947709a7a45da9a
```

Once again, this only returns one job in this case.

We can pipe `signac find` results into `signac statepoint` with `xargs` to resolve the statepoints.

In [16]: ```bash
signac find | xargs signac statepoint
```
```
"p": 10.0, "N": 1000, "kT": 1.0
"p": 0.1, "N": 1000, "kT": 1.0
"p": 1.0, "kT": 1.0, "N": 1000
```

You will have noticed that each time we execute a `find` operation the data space is indexed anew.

This is no problem for small data spaces, however especially for larger data spaces, where the indexing process may be substantially expensive it’s advantageous to cache the index in a file.

In [17]: ```bash
signac project --index > index.txt
```
This index file can be used in conjunction with all functions that require a data space index, for example `signac find`:

In [18]: ```bash
signac find -i index.txt "{'p': 0.1}"
```

5a6c687f7655319db24de59a2336eff8

Reading index from file 'index.txt'...

Or for instance when creating a linked view.

In [19]: ```bash
signac view -i index.txt ./view
```

Reading index from file 'index.txt'...

The `signac view` command works exactly like `project.create_linked_view` such that the `./view` directory now contains a linked view to the job workspaces.

In [20]: ```sh
ls view
```

p_0.1/ p_1.0/ p_10.0/

This concludes the first chapter of the tutorial. The next chapter introduces a few more advanced topics.

Return to index

## 2.1 Advanced Indexing

### Indexing files

As was shown earlier, we can create an index of the data space using the `index()` method:

In [1]: ```python
import signac

project = signac.get_project(root='projects/tutorial')
index = list(project.index())

for doc in index[:3]:
    print(doc)

'fluid': 'ideal gas', 'root': '/home/johndoe/signac-examples/notebooks/projects/tutorial/workspace', 'V_liq': 0.0, ...
'statepoint': 'kT': 1.0, 'b': 0, 'p': 3.4000000000000004, 'a': 0, 'N': 1000, '_id': '0e909ffdba496bbb590fbce31f3a4563'

'fluid': 'ideal gas', 'root': '/home/johndoe/signac-examples/notebooks/projects/tutorial/workspace', 'V_liq': 0.0, ...
'statepoint': 'kT': 1.0, 'b': 0, 'p': 0.1, 'a': 0, 'N': 1000, '_id': '10743bc8b95bffab09503bce9abbe627'

'fluid': 'water', 'root': '/home/johndoe/signac-examples/notebooks/projects/tutorial/workspace', 'V_liq': ... 'statepoint': 'kT': 1.0, 'b': 0.03049, 'p': 1.2000000000000002, 'a': 5.536, 'N': 1000, '_id': '11d8997f19b8ba53d2360ee9fb1606fa'
```

We will use the `Collection` class to manage the index directly in-memory:

In [2]: ```python
index = signac.Collection(project.index())
```

This enables us for example, to quickly search for all indexes related to a specific state point:

In [3]: ```python
for doc in index.find({ 'statepoint.p': 0.1}):
    print(doc)

'fluid': 'argon', 'root': '/home/johndoe/signac-examples/notebooks/projects/tutorial/workspace', 'V_liq': ...
'statepoint': 'kT': 1.0, 'b': 0.03201, 'p': 0.1, 'a': 0, 'N': 1000, '_id': 'f803d91519e23a9eee19fd9e789eeb2e'

'fluid': 'ideal gas', 'root': '/home/johndoe/signac-examples/notebooks/projects/tutorial/workspace', 'V_liq': ...
'statepoint': 'kT': 1.0, 'a': 1.355, 'p': 0.1, 'b': 0.03201, 'N': 1000, '_id': '10743bc8b95bffab09503bce9abbe627'

'fluid': 'water', 'root': '/home/johndoe/signac-examples/notebooks/projects/tutorial/workspace', 'V_liq': ...
'statepoint': 'kT': 1.0, 'a': 5.536, 'p': 0.1, 'b': 0.03049, 'N': 1000, '_id': '40405b550e7cc2d127b9758d0e764672'
```

At this point the index contains information about the statepoint and all data stored in the job document. If we want to include the V.txt text files we used to store data in, with the index, we need to tell `signac` the filename pattern and optionally the file format.

1.3. Tutorial
In [4]: index = signac.Collection(project.index('.*\.txt'))
   for doc in index.find(limit=2):
       print(doc)

'root': '/home/johndoe/signac-examples/notebooks/projects/tutorial/workspace', 'signac_id': '30435783', 'fluid': 'argon', 'root': '/home/johndoe/signac-examples/notebooks/projects/tutorial/workspace', 'V_gas': 149.2537313432836, 'statepoint': 'kT': 1.0, 'a': 0, 'p': 6.7, 'b': 0, 'N': 1000, 'format': 'File'

The index contains basic information about the files within our data space, such as the path and the MD5 hash sum. The format field currently says File, which is the default value.

We can specify that all files ending with .txt are to be defined to be of TextFile format:

In [5]: index = signac.Collection(project.index({'.*\.txt': 'TextFile'}))
   print(index.find_one({'format': 'TextFile'}))


Generating a Master Index

A master index is compiled from multiple other indexes, which is useful when operating on data compiled from multiple sources, such as multiple signac projects.

To make a data space part of master index, we need to create a signac_access.py module. We use the access module to define how the index for the particular space is to be generated. We can create a basic access module using the Project.create_access_module() function:

In [6]: # Let's make sure to remove any remnants from previous runs...
   % rm -f projects/tutorial/signac_access.py
   # This will generate a minimal access module:
   project.create_access_module(master=False)
   % cat projects/tutorial/signac_access.py

import signac

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

When compiling a master index, signac will search for access modules named signac_access.py. Whenever it finds a file with that name, it will import the module and compile all indexes yielded from a function called get_indexes() into the master index.

Let's try that!

In [7]: master_index = signac.Collection(signac.index())
   for doc in master_index.find(limit=2):
       print(doc)

'root': '/home/johndoe/signac-examples/notebooks/projects/tutorial/workspace', 'signac_id': '7baa5980', 'fluid': 'argon', 'root': '/home/johndoe/signac-examples/notebooks/projects/tutorial/workspace', 'V_gas': 149.2537313432836, 'statepoint': 'kT': 1.0, 'a': 0, 'p': 6.7, 'b': 0, 'N': 1000, 'format': None

Please note, that we executed the index() function without specifying the project directory. The function crawled through all sub-directories below the root directory in an attempt to find access modules.

We can use the access module to control how exactly the index is generated, for example by adding filename and format definitions. Usually we could edit the file directly, here we will just overwrite the old one:

In [8]: access_module = 
   #***import signac
   
   def get_indexes(root):
       yield signac.get_project(root).index({'.*\.txt': 'TextFile'})
```python
with open('projects/tutorial/signac_access.py', 'w') as file:
    file.write(access_module)

Now files will also be part of the master index!

```}

In [9]:
master_index = signac.Collection(signac.index())
```python
print(master_index.find_one({'format': 'TextFile'}))
```
'root': '/home/johndoe/signac-examples/notebooks/projects/tutorial/workspace', 'signac_id': '7dfb9f29f81c99f38d24b72f1e59f69e', 'statepoint': 'kT': 1.0, 'a': 0, 'p': 5.6, 'b': 0, 'N': 1000, 'format': 'TextFile'

We can use the signac.fetch() function to directly open files associated with a particular index document:

```python
In [10]:
for doc in master_index.find({'format': 'TextFile'}, limit=3):
    with signac.fetch(doc) as file:
        p = doc['statepoint']['p']
        V = [float(v) for v in file.read().strip().split(',')]
    print(p, V)
```

```
5.6 [0.0, 178.57142857142858]
1.2000000000000002 [0.0, 833.3333333333333]
3.4000000000000004 [32.80193336746696, 146.6628568456784]
```

Think of fetch() like the built-in open() function. It allows us to retrieve and open files based on the index document (file id) instead of an absolute file path. This makes it easier to operate on data agnostic to its actual physical location.

Please note that we can specify access modules for any kind of data space, it does not have to be a signac project!

In the next section, we will learn how to use indexes in combination with pandas dataframes.

### 2.2 Integration with pandas data frames

As was shown earlier, we can use indexes to search for specific data points. One way to operate on the data is using pandas data frames.

*Please note: The following steps require the pandas package.*

```python
In [1]:
import signac
import pandas as pd
project = signac.get_project(root='projects/tutorial')

Let’s first create a basic index and use it to construct an index data frame:

```python
In [2]:
df_index = pd.DataFrame(project.index())
df_index.head()
```

```
Out[2]:
   V_gas   V_liq    _id             fluid root
0 294.117647 0.000000 0e909ffda496bbb590fbc31f3a4563 ideal gas
1 10000.000000 0.000000 10743bc9b95bff9ab09503bce9abbe627 ideal gas
2 416.581783 30.659767 11d8997f19b8ba53d2360ee9fb1606fa water
3 132.890365 0.000000 195c26531df79e70d8f50267f67f0e5 NaN
4 110.715506 32.801209 1f147aff97cbbda8aa7e4457a9b51159 argon
```

1.3. Tutorial
It is a good idea to explicitly use the \_id value as index key:

```python
In [3]: df_index = df_index.set_index([\'_id\'])
df_index.head()
```

```
Out[3]:
  V_gas   V_liq   fluid
_id
0e909ffda496bb590fabe31f3a4563 294.117647 0.000000 ideal gas
10743bc8b95b9bfa9b790503c9e9abbe627 10000.000000 0.000000 ideal gas
11d8997f19b8b5a3d2360ee9f1b606fa 416.581783 30.659767 water
195c26531df979e70d8f5026767f0e5 132.890365 0.000000 NaN
1f147aff97cbbda8a7c4457a9b51159 110.715506 32.801209 argon
```

Furthermore, the index would be more useful if each statepoint parameter had its own column.

```python
In [4]: statepoints = {doc[\'_id\']: doc[\'statepoint\'] for doc in project.index()}
df = pd.DataFrame(statepoints).T.join(df_index)
df.head()
```

```
Out[4]:
   N  a   b  kT  p
_id
0e909ffda496bb590fabe31f3a4563 1000.0  0.000  0.00000  1.0  3.400
10743bc8b95b9bfa9b790503c9e9abbe627 1000.0  0.000  0.00000  1.0  0.100
11d8997f19b8b5a3d2360ee9f1b606fa 1000.0  5.536  0.03049  4.5  1.000
```
Now we can select specific data subsets, for example to calculate the mean gas volume of argon for a pressure $p$ between 2.0 and 5.0:

```
In [5]: df[(df.fluid=='argon') & (df.p > 2.0) & (df.p <= 5.0)].V_gas.mean()
Out[5]: 158.12444608049674
```

Or we can plot a p-V phase diagram for argon (requires matplotlib).

```
In [6]: % matplotlib inline
   
   df_water = df[df.fluid=='argon'][['p', 'V_liq', 'V_gas']]
   df_water.sort_values('p').set_index('p').plot(logy=True)
Out[6]: <matplotlib.axes._subplots.AxesSubplot at 0x10fce6c18>
```
Or we group the data by fluid and compare the gas densities for low pressures:

```python
In [7]: from matplotlib import pyplot as plt

for fluid, group in df[df.p < 2].groupby('fluid'):
    d = group.sort_values('p')
    plt.plot(d['p'], d['V_gas'] / d['N'], label=fluid)
plt.xlabel('p')
plt.ylabel(r'$\rho_{gas}$')
plt.legend(loc=0)
```

Out[7]: `<matplotlib.legend.Legend at 0x1130da0b8>`
2.3 Integration with MongoDB databases

Instead of storing the index in a variable or in a plain-text file, we could export it to any tool of our choice. For convenience, signac provides export routines for MongoDB database collections.

Please note: The following steps require pymongo and either a local MongoDB instance or a signac database configuration.

In [1]: import signac

```python
# --- PLEASE UNCOMMENT ONE OF THE FOLLOWING TWO CONFIGURATIONS ---
#
# --- 1) USING A LOCAL MONGODB INSTANCE ---
#from pymongo import MongoClient
#client = MongoClient()
#db = client.testing

# --- 2) USING A SIGNAC DATABASE CONFIGURATION ---
db = signac.get_database('testing')

index = db.signac_tutorial_index
index.drop()
master_index = db.signac_tutorial_master_index
master_index.drop()
```

We can export either the project index directly via:

In [2]: project = signac.get_project(root='projects/tutorial')

```python
    signac.export(project.index(), index)
```

Or we compile and export a master index:
In [3]: signac.export(signac.index(), master_index, update=True)

In both cases we can now use MongoDB’s query engine for advanced search queries. For example, let’s find the mean gas volume of water for the pressure range (2.0 < p <= 5.0):

In [4]: import numpy as np

    query = {
        '$and': [{'statepoint.p': {'$gt': 2.0}}, {'statepoint.p': {'$lte': 5.0}}],
        'fluid': 'water',
        'V_gas': {'$exists': True}}

docs = master_index.find(query)
V = np.array([doc['V_gas'] for doc in docs])
print(V.mean())

158.43561567

2.4 External Tools

The following section demonstrates how to use the signac command line interface (CLI) in conjunction with other tools.

In [1]: %bash
    signac --help

usage: signac [-h] [-d] [-v] [-y] init,project,job,statepoint,move,clone,index,find,view,config ...

signac aids in the management, access and analysis of large-scale computational investigations.

positional arguments:
    init,project,job,statepoint,move,clone,index,find,view,config

optional arguments:
    -h, --help            show this help message and exit
    -d, --debug           Show traceback on error for debugging.
    -v, --version         Display the version number and exit.
    -y, --yes             Answer all questions with yes. Useful for scripted interaction.

To interact with a project on the command line, the current working directory needs to be within or below the project root directory. Let’s start by resetting the designated project root directory for this section of the tutorial.

In [2]: % pwd
    % rm -rf projects/tutorial/cli
    % mkdir -p projects/tutorial/cli
    % cp idg projects/tutorial/cli

Next we switch the current working directory to the project root directory.

In [3]: % cd projects/tutorial/cli

/home/johndoe/signac-examples/notebooks/projects/tutorial/cli

Then we initialize the project.

In [4]: %bash
    signac init TutorialCLIProject

Initialized project 'TutorialCLIProject'.
We can verify the project configuration.

```
In [5]: %%bash
    signac project
    signac project --workspace
```

```text
Tutorial1CLIProject
/home/johndoe/signac-examples/notebooks/projects/tutorial/cli/workspace
```

We access the job handle by providing the state point on the command line in JSON format.

```
In [6]: %%bash
    signac job '{{"kT": 1.0, "p": 1.0, "N": 1000}}'
```

```
e617ad585a9080947709a7a45dda9a
```

The statepoints could also be read from STDIN, e.g., by reading it from a file. Let’s create a statepoints file with three statepoints:

```
In [7]: %%bash
    echo '{{"kT": 1.0, "p": 0.1, "N": 1000}}' > statepoints.txt
    echo '{{"kT": 1.0, "p": 1.0, "N": 1000}}' >> statepoints.txt
    echo '{{"kT": 1.0, "p": 10.0, "N": 1000}}' >> statepoints.txt
```

```
cat statepoints.txt
```

```
"kT": 1.0, "p": 0.1, "N": 1000
"kT": 1.0, "p": 1.0, "N": 1000
"kT": 1.0, "p": 10.0, "N": 1000
```

We can pipe the content of this file into the `signac` CLI to get the corresponding `job id`.

```
In [8]: %%bash
    head -n 1 statepoints.txt | signac job
```

```
Error: expected string or buffer
```

Instead of the `job id`, we can directly obtain the path to the `job workspace`.

```
In [9]: %%bash
    head -n 1 statepoints.txt | signac job --workspace
```

```
Error: expected string or buffer
```

That’s specifically useful in conjunction with external tools. Let’s pretend that we need to use a program called `idg` to calculate the ideal gas equation.

The `idg` program will calculate the volume of an ideal gas given the input parameters `p`, `kT` and `N`, just like in the previous sections.

```
In [10]: %%bash
    ./idg -p 1.0 -N 1000 --kT 1.0
```

```
1000.0
```

We can store the result in a file based on the input arguments using the `-cw` argument, short for `--create --workspace`, which returns the workspace path and creates it if necessary.

```
In [11]: %%bash
    ./idg -p 1.0 --kT 1.0 -N 1000 > $(signac job -cw '{{"p": 1.0, "kT": 1.0, "N": 1000}}')/V.txt
```

```
Obviously, we wouldn’t write these commands all manually, but use a script instead.
```

```
In [12]: import signac
    import json
    project = signac.get_project()
    for p in 0.1, 1.0, 10.0:
        job = project.open_job({'N': 1000, 'p': p, 'kT': 1.0})
```
We can then execute this script...

... and examine the results.

Reference

A complete reference to all major components of the \texttt{signac} framework.

Contents:

28 Chapter 1. Contents
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 back end. 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:

```python
>>> 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.

1.4. Reference
Note: To prevent that unauthorized users can obtain 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
```

Omiting the `-p/--password` argument means that a password will not be stored, requiring to enter it for each session.

**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
# 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`.

### Projects

#### Introduction

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 corresponds largely to the definition of a computational project outlined by Wilson et al. The signac framework
helps to design and implement all three of these components by providing the necessary computational infrastructure
and by being an implicit part of the project’s documentation.

Larger, more complex computational investigations usually demand a division into multiple subprojects, however there
is no simple answer to how exactly to divide routines and data space among individual subprojects. A general rule of
thumb is that if two projects share more than 50% of their routines and the data they operate on, they are probably the
same project. Keep in mind that signac makes it easy to *divide or merge* projects even at a later stage.

#### Project Initialization

To initialize a project, simply create a project directory and execute `$signac init <projectname>` on the
command line. For example, to initialize a project named *MyProject* in a directory called *my_project*, execute:

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

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

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

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

#### The Project Data Space

After a project has been initialized, all shell and Python scripts executed within or below the project’s root directory
have access to signac’s project interface. This allows you to access and manipulate the project’s *data space* in a
simple and consistent manner.

1 You can access a project interface from other locations by explicitly specifying the root directory.
It is assumed that the *data space* is divided into individual data points, where each point may be a single value or multiple terabytes of data, but is always uniquely addressable by its metadata. In the context of *signac*, each data-metadata pair is called a *Job* and the metadata is referred to as a *state point*. A state point is a unique mapping of key-value pairs.

The project’s data space includes all data, such as individual values and files, that are associated with the project. In a broader sense even non-computational data such as hand-written notes are part of the data space. The *signac* framework is designed to help you keep track of all data and metadata, for example, by providing a unique file system path for each *Job* within the project’s *workspace* directory.

You can access your *signac* Project and its *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:

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

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.

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

Once you have the *job* instance you can add it to the project’s data space by *initializing* it, which means to create the corresponding workspace directory. You can initialize a job explicitly by calling the `init()` method or implicitly by accessing the job’s *job document* or by switching into the job’s workspace directory.

Use the *job* instance to retrieve the associated *state point* and *workspace* directory with the `statepoint()` and the `workspace()` methods:

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

**Tip:** For convenience, the *state point* may also be accessed via the `sp` attribute, e.g., the value for *a* can be printed like this: `print(job.sp.a)`.
This also works for nested state points: `print(job.sp.b.c)`!

**signac** ensures that each *job* can be uniquely identified by calculating a hash value for each *state point* called the *job id*. That means you can obtain an instance of *Job* either by its state point or – once a job has been initialized – directly by its id:

```python
>>> job = project.open_job({'a': 0})
>>> print(job.get_id())
9bfd29df07674bc4aa960cf661b5acd2
>>> job.init()
>>> job in project
True
>>> job2 = project.open_job(id=job.get_id())
>>> job == job2
True
```

You can iterate over all initialized jobs using the `find_jobs()` method:

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

Or you can select a subspace by defining a *filter*. For example, to iterate over all jobs that have a *state point* parameter `b=0`, execute:

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

**Tip:** Since iterating over all jobs, that means omitting the *filter* argument or setting it to `None`, is a very common pattern, you can use the following short-hand notation:

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

### The Job Document

To associate simple key-value pairs with your job, you can use the *job document*. The document is automatically stored in the job’s workspace directory in *JSON* format.

```python
>>> job = project.open_job(statepoint)
>>> job.document['hello'] = 'world'
```

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

1. *storage* of *lightweight* data,
2. keeping track of *runtime information* or to
3. *label* jobs, e.g. to identify error states.

In the following example, all job documents contain a field called *user_status*, which contains a list of labels that help to identify the job status.

```python
>>> for job in project:
...    print(job.document['user_status'])
...```
Tip: Just like with Python’s built-in dictionaries, use `get()` to return None or another default value for missing values.

Data Space Operations

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 function with only one argument: an instance of `Job`.

That means, the initialization of a job, either implicitly or explicitly, is always the first data space operation. For demonstration purposes we are going to initialize a data space with two numbers `a` and `b` from 0 to 25, calculate the product of these two numbers and store the result in a file called `product.txt`.

First, we define our product function:

```python
def compute_product(job):
    with job:
        with open('product.txt', 'w') as file:
            file.write(str(job.sp.a * job.sp.b))
```

In this example we use the job as context manager to switch into the job’s workspace directory. Then we access the two numbers `a` and `b` via the `sp state point` interface and write their product to a file called `product.txt` located within the job’s workspace. Alternatively, we could also store the result in the `job document`:

```python
def compute_product(job):
    job.document['product'] = job.sp.a * job.sp.b
```

Next, we are going to initialize the project’s data space by iterating over the two numbers, obtaining the `Job` instance with `open_job()` and calling the `init()` method:

```python
project = signac.get_project()
for i in range(25):
    for j in range(25):
        job = project.open_job({'a': i, 'b': j})
        job.init()
```

We can then execute our operation for the complete data space, for example, like this:

```python
for job in project:
    compute_product(job)
```

Finally, we can now retrieve our pre-calculated products by defining an access function,

```python
def product(a, b):
    job = project.open_job({'a': a, 'b': b})
    with open(job.fn('product.txt')) as file:
        return int(file.read())
```

Here, we first retrieve the corresponding job to our input values and then return the result using the `fn()` convenience method, where `job.fn(filename)` is equivalent to `os.path.join(job.workspace(), filename)`.
Note: In reality, we should account for missing values, for example, by catching FileNotFoundError exceptions, by checking whether the job is actually part of our data space with job in project or using the isfile() method (or any combination thereof).

Parallelization

To execute a data space operation func() for the complete project data space in serial we can either run a for-loop as shown before:

```python
for job in project:
    func(job)
```

or take advantage of python’s built-in map() function for a more concise expression:

```python
list(map(func, project))
```

Of course, this also works for a data subspace: list(map(func, project.find_jobs(a_filter))).

Using the map() function makes it trivial to implement parallelization patterns, for example, using a process Pool:

```python
from multiprocessing import Pool

with Pool() as pool:
    pool.map(func, project)
```

This will execute func() for the complete project data space on as many processing units as there are available.

Tip: Visualize execution progress with a progress bar by wrapping iterables with tqdm:

```python
from tqdm import tqdm

map(func, tqdm(project))
```

We can use the exact same pattern to parallelize using threads:

```python
from multiprocessing.pool import ThreadPool

with ThreadPool() as pool:
    pool.map(func, project)
```

Or even with Open MPI using a MPIPool:

```python
from signac.contrib.mpi_pool import MPIPool

with MPIPool() as pool:
    pool.map(func, tqdm(project))
```

Warning: Make sure to execute write-operations only on one MPI rank, e.g.:

```python
if comm.Get_rank() == 0:
    job.document['a'] = 0
comm.Barrier()
```
Note: Without further knowledge about the exact nature of the data space operation, it is not possible to predict which parallelization method is most efficient. The best way to find out is to run a few benchmarks.

Workspace Views

The workspace structure is organized by job id, which is efficient and flexible for organizing the data. However, inspecting files as part of a job workspace directly on the file system is now harder.

In this case it is useful to create a linked view, that means, a directory hierarchy with human-readable names, that link to the actual job workspace directories. This means that no data is copied, but you can inspect data in a more convenient way.

To create a linked view you can either call the `create_linked_view()` method or execute the `signac view` function on the command line.

Let’s assume the data space contains the following state points:

- a=0, b=0
- a=1, b=0
- a=2, b=0
- ...

where \( b \) is constant for all state points.

We then create the linked view with:

```
$ signac view my_view
Indexing project...
$ ls my_view/
a_0 a_1 a_2 ...
```

As the parameter \( b \) is constant for all jobs within the data space, it is ignored for the creation of the linked views.

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, for example, in combination with `signac find`:

```
$ signac find '(("a": 0))' | xargs signac view my_view -j
```

Tip: Consider creating a linked view for large data sets on an in-memory file system for best performance.

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 permanently delete a job and its contents use the remove() method:

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

State Point Modifications

It may be necessary to change the state point of one or more jobs after initialization—for example, to add previously not needed state point values. 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 sp attribute,
2. via the job’s update_statepoint() method, and
3. via the job’s reset_statepoint() method.

The update_statepoint() method provides safe-guards against accidental overwriting of existing state point values, while reset_statepoint() will simply reset the whole state point without further questions. The sp attribute provides the greatest flexibility, but similar to reset_statepoint() 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, execute:

for job in project:
    if 'b' not in job.sp:
        job.sp.b = 0

1.4. Reference
Renaming a state point key from b to c:

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

To remove a state point key c:

```python
for job in project:
    try:
        del job.sp['c']
    except KeyError:
        pass  # already deleted
```

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

```python
>>> job.statepoint()
{'a': 0}
>>> job.sp.b.c = 0  # <-- will raise a KeyError!!
# Instead:
>>> job.sp.b = {'c': 0}
# Now you can modify in-place:
>>> job.sp.b.c = 1
```

Indexing

Concept

To create a homogeneous data access layer, signac encourages the creation of a data index. The data index contains all information about the project’s data structure and can be stored in a database or data frame which then allows the execution of query and aggregation operations on the data.

While signac’s project interface is specifically useful during the data curation and generation phase, working with indexes may be useful in later stages of an investigation. Especially when data is curated from multiple different projects and sources or if data spaces do not use the signac project schema.

For example, we may want to calculate the average of some values that we read from files associated with a specific data sub space:

```python
def extract_value(doc):
    with signac.fetch(doc) as file:
        return float(file.read())

docs = index.find({'statepoint.a': 42})
average = sum(map(extract_value, docs)) / len(docs)
```

The next few sections will outline in detail how such a workflow can be realized.

Generating a File Index

An index is a collection of index documents, where each index document is an arbitrary collection of metadata describing the data space. In the specific case of a file index, each index document is associated with one file on disk and contains 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
• format: A format definition (optional)

To create a file index, execute the `index_files()` function:

```python
for doc in signac.index_files():
    print(doc)
```

With no arguments, the `index_files()` function will index all files in the current working directory. We can limit the indexing to specific files by specifying the root path and by providing a regular expression pattern that all filenames must match. For example, to index all files in the `/data` directory that end in `.txt`, execute:

```python
for doc in signac.index_files('/data', '.*\.txt'):
    print(doc)
```

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:

```python
for doc in signac.index_files('/data', '.*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 and in addition index PDF-files that adhere to the following pattern: `init.pdf` or `final.pdf`.

This is how we could generate the index:

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

## Indexing a signac Project

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

---

1. Identical with the `md5` value in the current implementation.
This means that we do not have to define regular expressions to extract the state point schema, but take advantage of the signac project schema for state points. To generate a signac project index, execute the `Project.index()` method:

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

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` and contains not only the `signac_id` and the `statepoint`, but also the data stored in the job document:

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

Just like for regular file indexes generated with `index_files()`, we can still define regular expressions to limit the indexing to specific files and to extract additional metadata.

### Generating a Master Index

A master index is a compilation of multiple indexes, which simplifies the operation 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 existence of this file tells signac that the projects in those directories should be indexed as part of a master index.

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

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

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

```python
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 [The signac_access.py Module](#) section.

**Tip:** You can generate a signac master index directly on the command line with `signac index`, which can thus be directly piped into a file:

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

### Managing Index Collections

Once we have generated an index, we can use it to search our data space. 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:
```python
index = project.index()

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

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

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

This is a very viable approach as long as the index is not too large and the search queries are relatively simple. An alternative way to manage an index is to use a `Collection`. For example, to execute the same search operation from above, we could use the `find()` method:

```python
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.

Searching a collection is usually **much more efficient** compared to the `pure-python` approach especially when searching multiple times within the same session. Furthermore, a collection may be saved to and loaded from a file. This allows us to generate a index once and then load it from disk, which is much faster then regenerating it each time we use it:

```python
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 documents², use the `signac.export()` function:

```python
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.

### 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 document³ to locate and open the file.

---

² A stale document is associated with a file or state point that has been removed.
³ or a file id
# Search for specific documents:

```python
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.

## 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:

```python
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 index all files with a `.txt` filename suffix, we would put the following code into the module:

```python
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.

## Advanced Indexing

### The Crawler Concept

`signac` uses crawlers to crawl through a data source to generate an index. A crawler is defined to generate a sequence of index documents (mappings), the data source is arbitrary. Each index document requires at least one `_id` keyword. For example, this would be a valid crawler:

```python
my_source = ['a', 'b', 'c']

class MyCrawler(object):
    def crawl(self):
        for i, x in enumerate(my_source):
            yield dict(_id=i, x=x)
```

This crawler would generate the following documents:

```python
>>> for doc in MyCrawler().crawl():
...    print(doc)
...
{'x': 'a', '_id': 0}
{'x': 'b', '_id': 1}
{'x': 'c', '_id': 2}
```
The `index()`, `index_files()`, or `Project.index()` functions, internally define a `Crawler` class that is then executed to generate the index. These crawlers are subclassed from either `RegexFileCrawler`, `SignacProjectCrawler`, or `MasterCrawler`.

### Customizing Crawlers

Defining our own `Crawler` class provides us with full control over the index creation. For example, imagine we wanted to add an additional field to the index, that contains the length of each indexed file, we could define the following crawler class:

```python
class MyCrawler(signac.RegexFileCrawler):
    
    def process(self, doc, dirpath, fn):
        with open(os.path.join(dirpath, fn)) as file:
            doc['size'] = len(file.read())
        return super(MyCrawler, self).process(doc, dirpath, fn)

MyCrawler.define('.*\.txt')
```

In this example, we define a subclass of `RegexFileCrawler` called `MyCrawler` and redefine the `process()` method to add a `size` field to each generated document. We could put this definition into a `signac_access.py` module and make it part of a master index like this:

```python
import signac

class MyCrawler(signac.RegexFileCrawler):
    
    def get_indexes(root):
        yield MyCrawler(root).crawl()
```

### 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.

```python
from signac import fs, export, get_database

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

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

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

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

```python
def get_indexes(root):
    yield signac.get_project(root).index()
get_indexes.tags = {'public', 'foo'}
```

Similarly, you can require tags for specific crawlers:

```python
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:

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

Similarly on the command line:

```sh
$ 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.

```python
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),
]```
Database Integration

Database access

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

```python
signac.get_database(name, hostname=None, config=None)
```

Get a database handle.

The database handle is an instance of `Database`, which provides access to the document collections within one database.

```python
db = signac.db.get_database('MyDatabase')
docs = db.my_collection.find()
```

Please note, that a collection which did not exist at the point of access, will automatically be created.

**Parameters**

- `name` *(str)* – The name of the database to get.
- `hostname` *(str)* – The name of the configured host. Defaults to the first configured host, or the host specified by `default_host`.
- `config` *(common.config.Config)* – The config object to retrieve the host configuration from. Defaults to the global configuration.

**Returns** The database handle.

**Return type** `pymongo.database.Database`

See also:

- https://api.mongodb.org/python/current/api/pymongo/database.html

Queries and aggregation

To execute queries on a `Collection` instance use the `find()` or `find_one()` methods.

Aggregation pipelines are executed with the `aggregate()` method.

See also:
Basic mapping

Processing data always consists of three steps:

1. Fetch the input.
2. Process the input to produce output.
3. Store the output.

First, we define our processing function:

```python
def process(doc):
    doc['calc_value'] = # ...
    return doc
```

In this case we effectively copy and extend the input document to produce the output document which has the benefit of preserving all meta data, but is not strictly necessary.

We fetch our input documents from a collection that contains the index, in this case called `index`:

```python
db = signac.db.get_database('MyProject')
docs = db.index.find({'a': {'$lt': 100}})
```

We can use the `map()` function to generate the results:

```python
results = map(process, docs)
```

and store them in a result collection called `results`:

```python
results = map(process, docs)
db.results.insert_many(results)
```

By using a different map function, we can trivially parallelize this process, for example with a process pool:

```python
import multiprocessing

with multiprocessing.Pool(8) as pool:
    results = pool.imap(process, docs)
```

or an MPI pool, which is bundled with signac:

```python
with signac.contrib.MPIPool() as pool:
    results = pool.map(process, docs, ntask=docs.count())
```

Quick Reference

Projects

A signac Project is the primary data interface for the generation and management of project data spaces.
Start a new project

Python:

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

Command line:

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

Access the project

Python:

```
# Within or below the project root directory:
project = signac.get_project()

# With explicit path specification:
project = signac.get_project(root='/path/to/project')
```

Command line:

```
$ signac project
MyProject
```

Open a job

Python:

```
# Open with state point
with project.open_job({'a': 0}) as job:
    pass

# Open with job id
with project.open_job(id='9bfd29df07674bc4aa960cf661b5acd2') as job:
    pass

# Open with abbreviated job id
with project.open_job(id='9bfd29') as job:
    pass
```

Command line:

```
$ signac job '{"a": 0}'
9bfd29df07674bc4aa960cf661b5acd2

$ signac job --workspace '{"a": 0}'
/path/to/workspace/9bfd29df07674bc4aa960cf661b5acd2

$ signac statepoint 9bfd29df07674bc4aa960cf661b5acd2
{"a": 0}
```
$ signac statepoint 9bfd29
{"a": 0}

Note: Using an abbreviated job id may result in multiple matches and is primarily designed for interactive use.

Find jobs

Python:

```python
# Iterate over all jobs in the data space
for job in project:
    pass

# Equivalent to
for job in project.find_jobs():
    pass

# Iterate over a data sub space with state point filter
for job in project.find_jobs({'a': 0}):
    pass

# Iterate over a data sub space with document filter
for job in project.find_jobs(doc_filter={'a': 0}):
    pass
```

Command line:

```bash
# Find all jobs
$ signac find

# Find a subset filtered by state point
$ signac find '{"a": 0}'

# Find a subset filtered by job document entries
$ signac find --doc-filter '{"a": 0}'
```

Note: The state point and document filter can be applied in combination.

Dataspace Operations

A dataspace operation in the context of signac projects is defined as any process which creates, modifies or deletes project data as part of the project’s dataspace. Implemented in Python, such a operation should only require one argument, an instance of Job, in order to be well-defined:

```python
def operate(job):
    pass
```

Execute in serial:

```python
for job in project:
    operate(job)
```
Execute in parallel:

```python
from multiprocessing import Pool
with Pool() as pool:
    pool.map(operate, project)

from multiprocessing.pool import ThreadPool
with ThreadPool() as pool:
    pool.map(operate, project)

from signac.contrib.mpipool import MPIPool
with MPIPool() as pool:
    pool.map(operate, project)
```

### Indexing

An index is collection of documents which describe an existing data space.

#### Generate a file index

```python
signac.index_files('/data', '*.txt')
```

#### Generate a signac project index

Python:

```python
project.index('.txt')
```

Command line:

```
$ signac project --index
```

#### Create an access module:

Python:

```python
project.create_access_module()
```

Command line:

```
$ touch signac_access.py
$ # or:
$ signac project --access
```
Generate a master index

Python:

```python
signac.index('/data')
```

Command line:

```
$ signac index
```

Fetch Data

Fetch files from an index document with `fetch()`:

```python
for doc in index:
    with signac.fetch(doc) as file:
        print(file.read())
```

Collections

A `Collection` is a set of documents (mappings of key-value pairs).

Initialize a collection

```python
# Directly in-memory:
collection = signac.Collection(docs)

# Associated with a file object:
with Collection.open('index.txt') as collection:
    pass
```

Setup a command line interface

```python
# find.py
with signac.Collection.open('collection.txt') as collection:
    collection.main()
```

Iterate through a collection

Python:

```python
for doc in collection:
    print(doc)
```

Command line:

```
$ python find.py
```
Search for documents

Python:

```python
for doc in collection.find({'a': 42}):
    print(doc)
```

Command line:

```
$ python find.py 'a': 42'
```

Database Integration

The signac framework allows for the simple integration of databases, for example for the management of index collections.

Access a database

```python
db = signac.get_database('my_database')
```

Search a database collection

```python
# a > 0
docs = db.index.find({'a': {'$gt': 0}})
# a = 2
doc = db.index.find_one({'a': 2})
```

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].*


References for a specific release versions can be found here.

**Tip:** You can auto-generate the first formatted reference and the corresponding BibTeX file via `signac.cite.reference()` and `signac.cite.bibtex()`.
API

The complete signac application interface (API).

Command Line Interface

Some core signac functions are – in addition to the Python interface – accessible directly via the $ signac command. For more information, please see $ signac --help.

Module contents

The signac framework aids in the management of large and heterogeneous data spaces.

It provides a simple and robust data model to create a well-defined indexable storage layout for data and metadata. This makes it easier to operate on large data spaces, streamlines post-processing and analysis and makes data collectively accessible.

class signac.Project (config=None)
    Bases: object

    The handle on a signac project.
    Application developers should usually not need to directly instantiate this class, but use signac.get_project() instead.

class Job (project, statepoint)
    Bases: object

    The job instance is a handle to the data of a unique statepoint.
    Application developers should usually not need to directly instantiate this class, but use open_job() instead.

    FN_DOCUMENT = ‘signac_job_document.json’
    FN_MANIFEST = ‘signac_statepoint.json’

    close()
        Close the job and switch to the previous working directory.
**document**

The document associated with this job.

**Returns** The job document handle.

**Return type** `JSONDict`

**fn (filename)**

Prepend a filename with the job’s workspace directory path.

**Parameters** `filename (str)` – The filename of the file.

**Returns** The full workspace path of the file.

**get_id ()**

The unique identifier for the job’s statepoint.

**Returns** The job id.

**Return type** `str`

**init ()**

Initialize the job’s workspace directory.

This function will do nothing if the directory and the job manifest already exist.

**isfile (filename)**

Return True if file exists in the job’s workspace.

**Parameters** `filename (str)` – The filename of the file.

**Returns** True if file with filename exists in workspace.

**Return type** `bool`

**move (project)**

Move this job to project.

This function will attempt to move this instance of job from its original project to a different project.

**Parameters** `project (Project)` – The project to move this job to.

**Raises** `DestinationExistsError` – If the job is already initialized in project.

**open ()**

Enter the job’s workspace directory.

You can use the `Job` class as context manager:

```python
with project.open_job(my_statepoint) as job:
    # manipulate your job data
```

Opening the context will switch into the job’s workspace, leaving it will switch back to the previous working directory.

**remove ()**

Remove the job’s workspace including the job document.

This function will do nothing if the workspace directory does not exist.

**reset_statepoint (new_statepoint)**

Reset the state point of this job.

**Danger:** Use this function with caution! Resetting a job’s state point, may sometimes be necessary, but can possibly lead to incoherent data spaces.

**Parameters** `new_statepoint (mapping)` – The job’s new state point.

**Raises**

- `DestinationExistsError` – If a job associated with the new state point is already initialized.
• **OSError** – If the move failed due to an unknown system related error.

```python
sp
```

Access the job’s state point as attribute dictionary.

```python
statepoint()
```

The statepoint associated with this job.

- **Returns**: The statepoint mapping.
- **Return type**: dict

```python
update_statepoint(update, overwrite=False)
```

Update the statepoint of this job.

---

**Warning**: While appending to a job’s state point is generally safe, modifying existing parameters may lead to data inconsistency. Use the overwrite argument with caution!

- **Parameters**
  - `update(mapping)` – A mapping used for the statepoint update.
  - `overwrite` – Set to true, to ignore whether this update overwrites parameters, which are currently part of the job’s state point. Use with caution!

- **Raises**
  - **KeyError** – If the update contains keys, which are already part of the job’s state point and overwrite is False.
  - **DestinationExistsError** – If a job associated with the new state point is already initialized.
  - **OSError** – If the move failed due to an unknown system related error.

```python
workspace()
```

Each job is associated with a unique workspace directory.

- **Returns**: The path to the job’s workspace directory.
- **Return type**: str

```python
ws
```

The job’s workspace directory.

```python
Project.build_job_search_index(index)
```

Build a job search index.

- **Parameters**
  - `index(list)` – A document index.
- **Returns**: A job search index based on the provided index.
- **Return type**: JobSearchIndex

```python
Project.build_job_statepoint_index(exclude_const=False, index=None)
```

Build a statepoint index to identify jobs with specific parameters.

This method generates unordered key-value pairs, with complete statepoint paths as keys, encoded in JSON, and a set of job ids of all corresponding jobs, e.g.:

```python
>>> project.open_job({'a': 0, 'b': {'c': 'const'}}).init()
>>> project.open_job({'a': 1, 'b': {'c': 'const'}}).init()
>>> for k, v in project.job_statepoint_index():
...     print(k, v)
...
["a", 1] {'b7568fa73881d27cbf24bf58d226d80e'}
["a", 0] {'54b61a7adbbe004b30b39aa399d04f483'}
["b", "c", "abc"] {'b7568fa73881d27cbf24bf58d226d80e',
                    '54b61a7adbbe004b30b39aa399d04f483'}```
Parameters

• **exclude_const (bool)** – Exclude entries that are shared by all jobs that are part of the index.

• **index** – A document index.

Yields Key-value pairs of JSON-encoded statepoint parameters and and a set of corresponding job ids.

Project.clone (job)
Clone job into this project.

Parameters **job (Job)** – The job to copy into this project.

Returns The job instance corresponding to the copied job.

Return type **Job**

Raises **DestinationExistsError** – In case that a job with the same id is already initialized within this project.

Project.config
The project’s configuration.

Project.create_access_module (filename=None, master=True)
Create the access module for indexing

This method generates the access module required to make this project’s index part of a master index.

Parameters

• **filename (str)** – The name of the access module file. Defaults to the standard name and should usually not be changed.

• **master (bool)** – If True, add directives for the compilation of a master index when executing the module.

Returns The name of the created access module.

Return type **str**

Project.create_linked_view (prefix=None, job_ids=None, index=None)
Create or update a persistent linked view of the selected data space.

This method determines unique paths for each job based on the job’s statepoint and creates symbolic links to the associated workspace directories. This is useful for browsing through the data space in a human-readable manner.

Assuming that the parameter space is

• a=0, b=0
• a=1, b=0
• a=2, b=0
• ...

where b does not vary over all statepoints, this method will create the following symbolic links within the specified view prefix:
view/a/0/job -> /path/to/workspace/7f9fb369851609ce9cb91404549393f3
view/a/1/job -> /path/to/workspace/017d53deb17a290d8b0d2ae02fa8bd9d
...

Note: To maximize the compactness of each view path, $b$ which does not vary over the selected data space, is ignored.

Parameters

- **prefix** *(str)* – The path where the linked view will be created or updated.
- **job_ids** – If None (the default), create the view for the complete data space, otherwise only for the sub space constituted by the provided job ids.
- **index** – A document index.

Project.dump_statepoints *(statepoints)*

Dump the statepoints and associated job ids.

Equivalent to:

```python
{project.open_job(sp).get_id(): sp for sp in statepoints}
```

Parameters **statepoints** *(iterable)* – A list of statepoints.

Returns A mapping, where the key is the job id and the value is the statepoint.

Return type **dict**

Project.find_job_documents *(filter=None)*

Find all job documents in the project’s workspace.

This method iterates through all jobs or all jobs matching the filter and yields each job’s document as a dict. Each dict additionally contains a field ‘statepoint’, with the job’s statepoint and a field ‘_id’, which is the job’s id.

Parameters **filter** *(Mapping)* – If not None, only find job documents matching filter.

Yields Instances of dict.

Raises **KeyError** – If the job document already contains the fields ‘_id’ or ‘statepoint’.

Project.find_job_ids *(filter=None, doc_filter=None, index=None)*

Find the job_ids of all jobs matching the filters.

The optional filter arguments must be a Mapping of key-value pairs and JSON serializable.

Note: Providing a pre-calculated index may vastly increase the performance of this function.

Parameters

- **filter** *(Mapping)* – A mapping of key-value pairs that all indexed job statepoints are compared against.
- **doc_filter** – A mapping of key-value pairs that all indexed job documents are compared against.
index – A document index.

Yields The ids of all indexed jobs matching both filters.

Raises

- **TypeError** – If the filters are not JSON serializable.
- **ValueError** – If the filters are invalid.
- **RuntimeError** – If the filters are not supported by the index.

Project`\texttt{.find\_jobs} (filter=None, doc\_filter=None, index=None)

Find all jobs in the project’s workspace.

The optional filter arguments must be a Mapping of key-value pairs and JSON serializable.

Note: Providing a pre-calculated index may vastly increase the performance of this function.

Parameters

- **filter (Mapping)** – A mapping of key-value pairs that all indexed job statepoints are compared against.
- **doc\_filter** – A mapping of key-value pairs that all indexed job documents are compared against.

Yields Instances of `\texttt{Job}`

Raises

- **TypeError** – If the filters are not JSON serializable.
- **ValueError** – If the filters are invalid.
- **RuntimeError** – If the filters are not supported by the index.

Project`\texttt{.find\_statepoints} (filter=None, doc\_filter=None, index=None, skip\_errors=False)

Find all statepoints in the project’s workspace.

Parameters

- **filter (mapping)** – If not None, only yield statepoints matching the filter.
- **skip\_errors (bool)** – Show, but otherwise ignore errors while iterating over the workspace. Use this argument to repair a corrupted workspace.

Yields statepoints as dict

Project`\texttt{.get\_id}()

Get the project identifier.

Returns The project id.

Return type **str**

Raises **LookupError** – If no project id could be determined.

classmethod Project`\texttt{.get\_project} (root=None)

Find a project configuration and return the associated project.

Parameters **root (str)** – The project root directory. If no root directory is given, the next project found within or above the current working directory is returned.
Returns The project handle.

Raises LookupError – If no project configuration can be found.

Project.get_statepoint (jobid, fn=\(None\))
Get the statepoint associated with a job id.

The statepoint is retrieved from the workspace or from the statepoints file if the former attempt fails.

Parameters

- **jobid** (str) – A job id to get the statepoint for.
- **fn** (str) – The filename of the file containing the statepoints, defaults to FN_STATEPOINTS.

Returns The statepoint.

Return type dict

Raises KeyError – If the statepoint associated with jobid could not be found.

See also dump_statepoints().

Project.index (formats=\(None\), depth=0, skip_errors=False, include_job_document=True)
Generate an index of the project’s workspace.

This generator function indexes every file in the project’s workspace until the specified depth. The job document if it exists, is always indexed, other files need to be specified with the formats argument.

```python
for doc in project.index({'.*\.txt', 'TextFile'}):
    print(doc)
```

Parameters

- **formats** (dict) – The format definitions as mapping.
- **depth** (int) – Specifies the crawling depth. A value of 0 (default) means no limit.
- **skip_errors** (bool) – Skip all errors which occur during indexing. This is useful when trying to repair a broken workspace.
- **include_job_document** (bool) – Include the contents of job documents.

Yields index documents

classmethod Project.init_project (name, root=\(None\), workspace=\(None\), make_dir=True)
Initialize a project with the given name.

It is safe to call this function multiple times with the same arguments. However, a RuntimeError is raised in case where an existing project configuration would conflict with the provided initialization parameters.

Parameters

- **name** (str) – The name of the project to initialize.
- **root** (str) – The root directory for the project. Defaults to the current working directory.
- **workspace** (str) – The workspace directory for the project. Defaults to $project_root/workspace.$
- **make_dir** (bool) – Create the project root directory, if it does not exist yet.

Returns The project handle of the initialized project.

Return type Project
**Raises** **RuntimeError** – If the project root path already contains a conflicting project configuration.

**Project.min_len_unique_id()**
Determine the minimum length required for an id to be unique.

**Project.num_jobs()**
Return the number of initialized jobs.

**Project.open_job(statepoint=None, id=None)**
Get a job handle associated with a statepoint.

This method returns the job instance associated with the given statepoint or job id. Opening a job by a valid statepoint never fails. Opening a job by id, requires a lookup of the statepoint from the job id, which may fail if the job was not previously initialized.

**Parameters**
- **statepoint** *(mapping)* – The job’s unique set of parameters.
- **id** *(str)* – The job id.

**Returns** The job instance.

**Return type** **Job**

**Raises**
- **KeyError** – If the attempt to open the job by id fails.
- **LookupError** – If the attempt to open the job by an abbreviated id returns more than one match.

**Project.read_statepoints(fn=None)**
Read all statepoints from a file.

**Parameters**
- **fn** *(str)* – The filename of the file containing the statepoints, defaults to `FN_STATEPOINTS`.

See also `dump_statepoints()` and `write_statepoints()`.

**Project.repair()**
Attempt to repair the workspace after it got corrupted.

**Project.reset_statepoint(job, new_statepoint)**
Reset the state point of job.

**Danger:** Use this function with caution! Resetting a job’s state point, may sometimes be necessary, but can possibly lead to incoherent data spaces.

**Parameters**
- **job** *(Job)* – The job, that should be reset to a new state point.
- **new_statepoint** *(mapping)* – The job’s new state point.

**Raises**
- **DestinationExistsError** – If a job associated with the new state point is already initialized.
- **OSError** – If the move failed due to an unknown system related error.
Project.root_directory()
Returns the project’s root directory.

Project.update_statepoint(job, update, overwrite=False)
Update the statepoint of this job.

**Warning:** While appending to a job’s state point is generally safe, modifying existing parameters may lead to data inconsistency. Use the overwrite argument with caution!

**Parameters**
- **job (Job)** – The job, whose statepoint shall be updated.
- **update (mapping)** – A mapping used for the statepoint update.
- **overwrite** – Set to true, to ignore whether this update overwrites parameters, which are currently part of the job’s state point. Use with caution!

**Raises**
- **KeyError** – If the update contains keys, which are already part of the job’s state point and overwrite is False.
- **DestinationExistsError** – If a job associated with the new state point is already initialized.
- **OSError** – If the move failed due to an unknown system related error.

Project.workspace()
Returns the project’s workspace directory.

The workspace defaults to `project_root/workspace`. Configure this directory with the `workspace_dir` attribute. If the specified directory is a relative path, the absolute path is relative from the project’s root directory.

**Note:** The configuration will respect environment variables, such as $HOME.

Project.write_statepoints(statepoints=None, fn=None, indent=2)
Dump statepoints to a file.

If the file already contains statepoints, all new statepoints will be appended, while the old ones are preserved.

**Parameters**
- **statepoints (iterable)** – A list of statepoints, defaults to all statepoints which are defined in the workspace.
- **fn (str)** – The filename of the file containing the statepoints, defaults to FN_STATEPOINTS.
- **indent (int)** – Specify the indentation of the json file.

See also `dump_statepoints()`.

signac.get_project(root=None)
Find a project configuration and return the associated project.

**Parameters**
- **root (str)** – The project root directory. If no root directory is given, the next project found within or above the current working directory is returned.
Returns The project handle.

Return type Project

Raises LookupError – If no project configuration can be found.

signac.init_project (name, root=None, workspace=None, make_dir=True)
Initialize a project with the given name.

It is safe to call this function multiple times with the same arguments. However, a RuntimeError is raised in case where an existing project configuration would conflict with the provided initialization parameters.

Parameters

- name (str) – The name of the project to initialize.
- root (str) – The root directory for the project. Defaults to the current working directory.
- workspace (str) – The workspace directory for the project. Defaults to $project_root/workspace.
- make_dir (bool) – Create the project root directory, if it does not exist yet.

Returns The project handle of the initialized project.

Return type Project

Raises RuntimeError – If the project root path already contains a conflicting project configuration.

signac.get_database (name, hostname=None, config=None)
Get a database handle.

The database handle is an instance of Database, which provides access to the document collections within one database.

```python
db = signac.db.get_database('MyDatabase')
docs = db.my_collection.find()
```

Please note, that a collection which did not exist at the point of access, will automatically be created.

Parameters

- name (str) – The name of the database to get.
- hostname (str) – The name of the configured host. Defaults to the first configured host, or the host specified by default_host.
- config (common.config.Config) – The config object to retrieve the host configuration from. Defaults to the global configuration.

Returns The database handle.

Return type pymongo.database.Database

See also:
https://api.mongodb.org/python/current/api/pymongo/database.html

signac.fetch (doc_or_id, mode='r', mirrors=None, num_tries=3, timeout=60, ignore_local=False)
Fetch the file associated with this document or file id.

This function retrieves a file associated with the provided index document or file id and behaves like the built-in open() function, e.g.:
for doc in index:
    with signac.fetch(doc) as file:
        do_something_with(file)

Parameters

• doc_or_id – A file_id or a document with a file_id value.
• mode – Mode to use for opening files.
• mirrors – An optional set of mirrors to fetch the file from.
• num_tries (int) – The number of automatic retry attempts in case of mirror connection errors.
• timeout (int) – The time in seconds to wait before an automatic retry attempt.

Returns The file associated with the document or file id.

Return type A file-like object

signac.fetch_one(doc, *args, **kwargs)
Legacy function, use fetch() instead.

signac.export_one(doc, index, mirrors=None, num_tries=3, timeout=60)
Export one document to index and an optionally associated file to mirrors.

Parameters

• doc – A document with a file_id entry.
• docs – The index collection to export to.
• mirrors – An optional set of mirrors to export files to.
• num_tries (int) – The number of automatic retry attempts in case of mirror connection errors.
• timeout (int) – The time in seconds to wait before an automatic retry attempt.

Returns The id and file id after successful export.

signac.export(docs, index, mirrors=None, update=False, num_tries=3, timeout=60, **kwargs)
Export docs to index and optionally associated files to mirrors.

The behavior of this function is equivalent to:

for doc in docs:
    export_one(doc, index, mirrors, num_tries)

If the update argument is set to True, the export algorithm will automatically identify stale index documents, that means documents that refer to files or state points that have been removed and are no longer part of the data space. Any document which shares the root, but not the _id field with any of the updated documents is considered stale and removed. Using update in combination with an empty docs sequence will raise ExportError, since it is not possible to identify stale documents in that case.

Note: This function will automatically delegate to specialized implementations for special index types. For example, if the index argument is a MongoDB document collection, the index documents will be exported via export_pymongo().
Parameters

- **docs** – The index documents to export.
- **index** – The collection to export the index to.
- **mirrors** – An optional set of mirrors to export files to.
- **update** *(bool)* – If True, remove stale index documents, that means documents that refer to files or state points that no longer exist.
- **num_tries** *(int)* – The number of automatic retry attempts in case of mirror connection errors.
- **timeout** *(int)* – The time in seconds to wait before an automatic retry attempt.
- **kwargs** – Optional keyword arguments to pass to delegate implementations.

** Raises ** *ExportError* – When using the update argument in combination with an empty docs sequence.

**signac.export_to_mirror**(doc, mirror, num_tries=3, timeout=60)

Export a file associated with doc to mirror.

**Parameters**

- **doc** – A document with a file_id entry.
- **mirror** – A file-system object to export the file to.
- **num_tries** *(int)* – The number of automatic retry attempts in case of mirror connection errors.
- **timeout** *(int)* – The time in seconds to wait before an automatic retry attempt.

**Returns** The file id after successful export.

**class signac.Collection**(docs=None, primary_key='_id')

**Bases:** *object*

A collection of documents.

The Collection class manages a collection of documents in memory or in a file on disk. A document is defined as a dictionary mapping of key-value pairs.

An instance of collection may be used to manage and search documents. For example, given a collection with member data, where each document contains a name entry and an age entry, we can find the name of all members that are at age 32 like this:

```python
members = [
    {'name': 'John', 'age': 32},
    {'name': 'Alice', 'age': 28},
    {'name': 'Kevin', 'age': 32},
    # ...
]

member_collection = Collection(members)
for doc in member_collection.find({'age': 32}):
    print(doc['name'])
```

To iterate over all documents in the collection, use:

```python
for doc in collection:
    print(doc)
```
By default a collection object will reside in memory. However, it is possible to manage a collection associated to a file on disk. To open a collection which is associated with a file on disk, use the `open()` class method:

```python
with Collection.open('collection.txt') as collection:
    for doc in collection.find({'age': 32}):
        print(doc)
```

The collection file is by default opened in `a+` mode, which means it can be read from and written to and will be created if it does not exist yet.

### Parameters

- `docs` – Initialize the collection with these documents.
- `primary_key` – The name of the key which serves as the primary index of the collection. Selecting documents by primary key has time complexity of O(N) in the worst case and O(1) on average. All documents must have a primary key value. The default primary key is `_id`.

#### clear()

Remove all documents from the collection.

#### close()

Close this collection instance.

In case that the collection is associated with a file-object, all changes are flushed to the file and the file is closed.

It is not possible to re-open the same collection instance after closing it.

#### delete_many(filter)

Delete all documents that match the filter.

#### delete_one(filter)

Delete one document that matches the filter.

#### dump(file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>)

Dump the collection in JSON-encoding to file.

The file argument defaults to `sys.stdout`, which means the encoded blob will be printed to screen in case that no file argument is provided.

For example, to dump to a file on disk, one could write:

```python
with open('my_collection.txt', 'w') as file:
    collection.dump(file)
```

### Parameters

- `file` – The file to write the encoded blob to.

#### find(filter=None, limit=0)

Find all documents matching filter, but not more than limit.

This function searches the collection for all documents that match the given filter and returns a result vector. For example:

```python
for doc in collection.find(my_filter):
    print(doc)
```

Nested values should be searched using the `.` operator, for example:
docs = collection.find({'nested.value': 42})

will return documents with a nested structure: {'nested': {'value': 42}}.

The result of find() can be stored and iterated over multiple times. In addition, the result vector can be queried for its size:

docs = collection.find(my_filter)
print(len(docs))  # the number of documents matching
for doc in docs:  # iterate over the result vector
    pass

Parameters

- **filter (Mapping)** – All documents must match the given filter.
- **limit (int)** – Do not return more than limit number of documents. A limit value of 0 (the default) means no limit.

Returns A result object that iterates over all matching documents.

Raises **NotFoundError** – In case that the filter argument is invalid.

**find_one (filter=None)**

Return one document that matches the filter or None.

doc = collection.find_one(my_filter)
if doc is None:
    print("No result found for filter", my_filter)
else:
    print("Doc matching filter:", my_filter, doc)

Parameters **filter** – The returned document must match the given filter.

Raises **NotFoundError** – In case that the filter argument is invalid.

Returns A matching document or None.

**flush()**

Write all changes to the associated file.

If the collection instance is associated with a file-object, calling the flush() method will write all changes to this file.

This method is also called when the collection is explicitly or implicitly closed.

**ids**

Return an iterator over the primary key in the collection.

**index (key, build=False)**

Get (and optionally build) the index for a given key.

An index allows to access documents by a specific key with minimal time complexity, e.g.:

```python
age_index = member_collection.index('age')
for _id in age_index[32]:
    print(member_collection[_id]['name'])
```
This means we can access documents by the ‘age’ key in O(1) time on average in addition to the primary key. Using the find() method will automatically build all required indexes for the particular search.

Once an index has been built, it will be internally managed by the class and updated with subsequent changes. An index returned by this method is always current with the latest state of the collection.

**Parameters**

- **key (str)** – The primary key of the requested index.
- **build** – If True, build a non-existing index if necessary, otherwise raise KeyError.

**Raises** KeyError – In case that build is False and the index has not been built yet.

**insert_one (doc)**

Insert one document into the collection

If the document does not have a value for the collection’s primary key yet, it will be assigned one.

```python
_id = collection.insert_one(doc)
assert _id in collection
```

**Note:** The document will be directly updated in case that it has no primary key and must therefore be mutable!

**Parameters** **doc** – The document to be inserted.

**Returns** The _id of the inserted document.

**main ()**

Start a command line interface for this Collection.

Use this function to interact with this instance of Collection on the command line. For example, executing the following script:

```python
# find.py
with Collection.open('my_collection.txt') as c:
    c.main()
```

will enable us to search for documents on the command line like this:

```
$ python find.py '{"age": 32}'
{"name": "John", "age": 32}
{"name": "Kevin", "age": 32}
```

**classmethod open (filename, mode='a+')**

Open a collection associated with a file on disk.

Using this factory method will return a collection that is associated with a collection file on disk. For example:

```python
with Collection.open('collection.txt') as collection:
    for doc in collection:
        print(doc)
```

will read all documents from the collection.txt file or create the file if it does not exist yet.

Modifications to the file will be written to the file when the flush() method is called or the collection is explicitly closed by calling the close() method or implicitly by leaving the with-clause:
with Collection.open('collection.txt') as collection:
    collection.update(my_docs)
# All changes to the collection have been written to collection.txt.

The open-modes work as expected, so for example to open a collection file in read-only mode, use Collection.open('collection.txt', 'r').

**primary_key**

The name of the collection's primary key (default='_id').

**replace_one**(filter, replacement, upsert=False)
Replace one document that matches the given filter.

The first document matching the filter will be replaced by the given replacement document. If the *upsert* argument is True, the replacement will be inserted in case that no document matches the filter.

**Parameters**

- **filter** – A document that should be replaced must match this filter.
- **replacement** – The replacement document.
- **upsert** – If True, insert the replacement document in the case that no document matches the filter.

**Raises** ValueError – In case that the filter argument is invalid.

**Returns** The _id of the replaced (or upserted) documented.

**update**(docs)
Update the collection with these documents.

Any existing documents with the same primary key will be replaced.

**Parameters**

- **docs** – A sequence of documents to be upserted into the collection.

**signac.export_pymongo**(docs, index, mirrors=None, num_tries=3, timeout=60, chunksize=100)
Optimized export() function for pymongo index collections.

The behavior of this function is roughly equivalent to:

```python
for doc in docs:
    export_one(doc, index, mirrors, num_tries)
```

**Note:** All index documents must be JSON-serializable to be able to be exported to a MongoDB collection.

**Parameters**

- **docs** – The index documents to export.
- **index**(pymongo.collection.Collection) – The database collection to export the index to.
- **num_tries**(int) – The number of automatic retry attempts in case of mirror connection errors.
- **timeout**(int) – The time in seconds to wait before an automatic retry attempt.
- **chunksize**(int) – The buffer size for export operations.
signac Documentation, Release 0.8.3

signac.index_files(root='.', formats=None, depth=0)

Generate a file index.

This generator function yields file index documents, where each index document corresponds to one file.

To index all files in the current working directory, simply execute:

```python
for doc in signac.index_files():
    print(doc)
```

A file associated with a file index document can be fetched via the `fetch()` function:

```python
for doc in signac.index_files():
    with signac.fetch(doc) as file:
        print(file.read())
```

This is especially useful if the file index is part of a collection (`Collection`) which can be searched for specific entries.

To limit the file index to files with a specific filename formats, provide a regular expression as the formats argument. To index all files that have file ending `.txt`, execute:

```python
for doc in signac.index_files(formats='.*\.txt'):
    print(doc)
```

We can specify specific formats by providing a dictionary as `formats` argument, where the key is the filename pattern and the value is an arbitrary formats string, e.g.:

```python
for doc in signac.index_files(formats=
    {'.*\.txt': 'TextFile', '.*\.zip': 'ZipFile'}):
    print(doc)
```

Parameters

- `root (str)` – The directory to index, defaults to the current working directory.
- `formats` – Limit the index to files that match the given regular expression and optionally associate formats with given patterns.
- `depth (int)` – Limit the search to the specified directory depth.

Yields

The file index documents as dicts.

signac.index(root='.', tags=None, depth=0, **kwargs)

Generate a master index.

A master index is compiled from other indexes by searching for modules named `signac_access.py` and compiling all indexes which are yielded from a function `get_indexes(root)` defined within that module as well as the indexes generated by crawlers yielded from a function `get_crawlers(root)` defined within that module.

This is a minimal example for a `signac_access.py` file:

```python
import signac

def get_indexes(root):
    yield signac.index_files(root, '.*\.txt')
```

Internally, this function constructs an instance of `MasterCrawler` and all extra key-word arguments will be forwarded to the constructor of said master crawler.
Parameters

- **root** *(str)* – Look for access modules under this directory path.
- **tags** – If tags are provided, do not execute slave crawlers that don’t match the same tags.
- **depth** *(int)* – Limit the search to the specified directory depth.
- **kwargs** – These keyword-arguments are forwarded to the internal MasterCrawler instance.

Yields

The master index documents as instances of dict.

```python
class signac.RegexFileCrawler(root)
Bases: signac.contrib.indexing.BaseCrawler

Generate documents from filenames and associate each file with a data type.

The RegexFileCrawler uses regular expressions to generate data from files. This is a particular easy method to retrieve metadata associated with files. Inherit from this class to configure a crawler for your data structure.

Let’s assume we want to index text files, with a naming pattern, that specifies a parameter \(a\) through the filename, e.g.:

```~my_project/a_0.txt
~my_project/a_1.txt
...
```

A valid regular expression to match this pattern would be: `.*\/(a\_\d+).txt` which may be defined for a crawler as such:

```python
MyCrawler(RegexFileCrawler):
    pass
MyCrawler.define('.*\/(a\_\d+).txt', 'TextFile')
```

classmethod **compute_file_id** *(doc, file)*

Compute the file id for a given doc and the associated file.

Parameters

- **doc** – The index document
- **file** – The associated file

Returns

The file id.

```python
crawl(depth=0)
```

classmethod **define** *(regex, format_=None)*

Define a format for a particular regular expression.

Parameters

- **regex** *(str)* – All files of the specified format must match this regular expression.
- **format** *(object)* – The format associated with all matching files.

```python
definitions = {}
docs_from_file(dirpath, fn)
```

Generate documents from filenames.

This method implements the abstract method `.BaseCrawler.docs_from_file` and yields index documents associated with files.
Note: It is not recommended to reimplement this method to modify documents generated from filenames. See `process()` instead.

Parameters

- **dirpath** – The path of the file relative to root.
- **fn** – The filename of the file.

Yields Index documents.

**fetch**(doc, mode='r')

Fetch the data associated with `doc`.

Parameters **doc**(dict) – A index document.

Returns The file associated with the index document.

Return type A file-like object

**process**(doc, dirpath, fn)

Post-process documents generated from filenames.

Example:

```python
def process(self, doc, dirpath, fn):
    doc['long_name_for_a'] = doc['a']
    return super(MyCrawler, self).process(doc, dirpath, fn)
```

Parameters

- **dirpath**(str) – The path of the file, relative to `root`.
- **fn**(str) – The filename.

Returns An index document, that means an instance of mapping.

Return type mapping

class signac.MasterCrawler(root, raise_on_error=False)

Bases: signac.contrib.indexing.BaseCrawler

Compiles a master index from indexes defined in access modules.

An instance of this crawler will search the data space for access modules, which by default are named `signac_access.py`. Once such a file is found, the crawler will import the module and try to execute two special functions given that they are defined within the module’s global namespace: `get_indexes()` and `get_crawlers()`.

The `get_indexes()` is assumed to yield one or multiple index generator functions, while the `get_crawlers()` function is assumed to yield one or more crawler instances.

This is an example for such an access module:

```python
import signac

def get_indexes(root):
    yield signac.index_files(root, '.*\txt')
```
def get_crawlers(root):
    yield MyCrawler(root)

In case that the master crawler has tags, the `get_indexes()` function will always be ignored while crawlers yielded from the `get_crawlers()` function will only be executed in case that they match at least one of the tags.

In case that the access module is completely empty, it will be executed as if it had the following directives:

```python
import signac

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

Tags for indexes yielded from the `get_indexes()` function can be specified by assigning them directly to the function:

```python
def get_indexes(root):
    yield signac.get_project(root).index()
get_indexes.tags = {'foo'}
```

**Parameters**

- `root` *(str)* – The path to the root directory to crawl through.
- `raise_on_error` *(bool)* – Raise all exceptions encountered during crawling instead of ignoring them.

**FN_ACCESS_MODULE** = ‘signac_access.py’

`docs_from_file(dirpath, fn)`

Compile master index from file in case it is an access module.

**Parameters**

- `dirpath` – The path of the file relative to root.
- `fn` – The filename of the file.

**Yields** Index documents.

class `signac.SignProjectCrawler(root)`

**Bases:** `signac.contrib.indexing.RegexFileCrawler`

Index a signac project workspace.

Without any file format definitions, this crawler yields index documents for each job, including the statepoint and the job document.

**See also:** `RegexFileCrawler`

**Parameters**

- `root` *(str)* – The path to the project’s root directory.

**crawl**(depth=0)

**encoding** = ‘utf-8’

**fn_job_document** = ‘signac_job_document.json’

**fn_statepoint** = ‘signac_statepoint.json’

**get_statepoint**(dirpath)
process($doc$, $dirpath$, $fn$)

$signac_id_alias$ = ‘id’

$statepoint_index$ = ‘statepoint’

Functions to support citing this software.

$signac$.cite.bibtex($file$=None)

Generate bibtex entries for $signac$.

The bibtex entries will be printed to screen unless a filename or a file-like object are provided, in which case they will be written to the corresponding file.

**Note:** A full reference should also include the version of this software. Please refer to the documentation on how to cite a specific version.

Parameters **file** – A str or file-like object. Defaults to `sys.stdout`.

$signac$.cite.reference($file$=None)

Generate formatted reference entries for $signac$.

The references will be printed to screen unless a filename or a file-like object are provided, in which case they will be written to the corresponding file.

**Note:** A full reference should also include the version of this software. Please refer to the documentation on how to cite a specific version.

Parameters **file** – A str or file-like object. Defaults to `sys.stdout`.

Subpackages

$signac$.contrib package

Module contents

class signac.contrib.Project($config$=None)

Bases: `object`

The handle on a signac project.

Application developers should usually not need to directly instantiate this class, but use $signac$.get_project() instead.

class Job($project$, $statepoint$)

Bases: `object`

The job instance is a handle to the data of a unique statepoint.

Application developers should usually not need to directly instantiate this class, but use open_job() instead.

$FN_DOCUMENT$ = ‘signac_job_document.json’

$FN_MANIFEST$ = ‘signac_statepoint.json’
close()
Close the job and switch to the previous working directory.

document
The document associated with this job.
    Returns The job document handle.
    Return type JsonDict

fn(filename)
Prepend a filename with the job’s workspace directory path.
    Parameters filename (str) – The filename of the file.
    Returns The full workspace path of the file.

get_id()
The unique identifier for the job’s statepoint.
    Returns The job id.
    Return type str

init()
Initialize the job’s workspace directory.
This function will do nothing if the directory and the job manifest already exist.

isfile(filename)
Return True if file exists in the job’s workspace.
    Parameters filename (str) – The filename of the file.
    Returns True if file with filename exists in workspace.
    Return type bool

move(project)
Move this job to project.
This function will attempt to move this instance of job from its original project to a different project.
    Parameters project (Project) – The project to move this job to.
    Raises DestinationExistsError – If the job is already initialized in project.

open()
Enter the job’s workspace directory.
You can use the Job class as context manager:

```python
with project.open_job(my_statepoint) as job:
    # manipulate your job data
```
Opening the context will switch into the job’s workspace, leaving it will switch back to the previous working directory.

remove()
Remove the job’s workspace including the job document.
This function will do nothing if the workspace directory does not exist.

reset_statepoint (new_statepoint)
Reset the state point of this job.

Danger: Use this function with caution! Resetting a job’s state point, may sometimes be necessary, but can possibly lead to incoherent data spaces.

    Parameters new_statepoint (mapping) – The job’s new state point.
    Raises
• **DestinationExistsError** – If a job associated with the new state point is already initialized.
• **OSError** – If the move failed due to an unknown system related error.

**sp**

Access the job’s state point as attribute dictionary.

**statepoint()**

The statepoint associated with this job.

Returns The statepoint mapping.

Return type dict

**update_statepoint (update, overwrite=False)**

Update the statepoint of this job.

**Warning:** While appending to a job’s state point is generally safe, modifying existing parameters may lead to data inconsistency. Use the overwrite argument with caution!

**Parameters**

• **update (mapping)** – A mapping used for the statepoint update.
• **overwrite** – Set to true, to ignore whether this update overwrites parameters, which are currently part of the job’s state point. Use with caution!

**Raises**

• **KeyError** – If the update contains keys, which are already part of the job’s state point and overwrite is False.
• **DestinationExistsError** – If a job associated with the new state point is already initialized.
• **OSError** – If the move failed due to an unknown system related error.

**workspace ()**

Each job is associated with a unique workspace directory.

Returns The path to the job’s workspace directory.

Return type str

**ws**

The job’s workspace directory.

**Project.build_job_search_index (index)**

Build a job search index.

Parameters index (list) – A document index.

Returns A job search index based on the provided index.

Return type JobSearchIndex

**Project.build_job_statepoint_index (exclude_const=False, index=None)**

Build a statepoint index to identify jobs with specific parameters.

This method generates unordered key-value pairs, with complete statepoint paths as keys, encoded in JSON, and a set of job ids of all corresponding jobs, e.g.:

```python
>>> project.open_job({'a': 0, 'b': {'c': 'const'}}).init()
>>> project.open_job({'a': 1, 'b': {'c': 'const'}}).init()
>>> for k, v in project.job_statepoint_index():
...     print(k, v)
...
['a', 1] {'b7568fa73881d27cbf24bf58d226d80e'}
```
Parameters

- **exclude_const (bool)** – Exclude entries that are shared by all jobs that are part of the index.
- **index** – A document index.

Yields Key-value pairs of JSON-encoded statepoint parameters and a set of corresponding job ids.

Project.clone (job)

Clone job into this project.

Parameters **job** (*Job*) – The job to copy into this project.

Returns The job instance corresponding to the copied job.

Return type *Job*

Raises *DestinationExistsError* – In case that a job with the same id is already initialized within this project.

Project.config

The project's configuration.

Project.create_access_module (filename=None, master=True)

Create the access module for indexing

This method generates the access module required to make this project's index part of a master index.

Parameters

- **filename (str)** – The name of the access module file. Defaults to the standard name and should usually not be changed.
- **master (bool)** – If True, add directives for the compilation of a master index when executing the module.

Returns The name of the created access module.

Return type *str*

Project.create_linked_view (prefix=None, job_ids=None, index=None)

Create or update a persistent linked view of the selected data space.

This method determines unique paths for each job based on the job’s statepoint and creates symbolic links to the associated workspace directories. This is useful for browsing through the data space in a human-readable manner.

Assuming that the parameter space is

- a=0, b=0
- a=1, b=0
- a=2, b=0
- ...,
where \( b \) does not vary over all statepoints, this method will create the following \textit{symbolic links} within the specified view prefix:

\[
\begin{align*}
\text{view/a/0/job} & \rightarrow /path/to/workspace/7f9fb369851609ce9cb91404549393f3 \\
\text{view/a/1/job} & \rightarrow /path/to/workspace/017d53deb17a290d8b0d2ae02fa8bd9d \\
& \ldots
\end{align*}
\]

\textbf{Note:} To maximize the compactness of each view path, \( b \) which does not vary over the selected data space, is ignored.

\textbf{Parameters}

- \texttt{prefix} (\textit{str}) – The path where the linked view will be created or updated.
- \texttt{job_ids} – If None (the default), create the view for the complete data space, otherwise only for the sub space constituted by the provided job ids.
- \texttt{index} – A document index.

\texttt{Project.dump_statepoints(statepoints)}

Dump the statepoints and associated job ids.

Equivalent to:

\[
\{\text{project.open_job(sp).get_id(): sp for sp in statepoints}\}
\]

\textbf{Parameters} \texttt{statepoints} (\textit{iterable}) – A list of statepoints.

\textbf{Returns} A mapping, where the key is the job id and the value is the statepoint.

\textbf{Return type} \texttt{dict}

\texttt{Project.find_job_documents(filter=None)}

Find all job documents in the project’s workspace.

This method iterates through all jobs or all jobs matching the filter and yields each job’s document as a dict. Each dict additionally contains a field ‘statepoint’, with the job’s statepoint and a field ‘_id’, which is the job’s id.

\textbf{Parameters} \texttt{filter} (\textit{mapping}) – If not None, only find job documents matching filter.

\textbf{Yields} Instances of dict.

\textbf{Raises} \texttt{KeyError} – If the job document already contains the fields ‘_id’ or ‘statepoint’.

\texttt{Project.find_job_ids(filter=None, doc_filter=None, index=None)}

Find the job_ids of all jobs matching the filters.

The optional filter arguments must be a Mapping of key-value pairs and JSON serializable.

\textbf{Note:} Providing a pre-calculated index may vastly increase the performance of this function.

\textbf{Parameters}

- \texttt{filter} (\textit{Mapping}) – A mapping of key-value pairs that all indexed job statepoints are compared against.
- \texttt{doc_filter} – A mapping of key-value pairs that all indexed job documents are compared against.
• **index** – A document index.

**Yields** The ids of all indexed jobs matching both filters.

**Raises**

• **TypeError** – If the filters are not JSON serializable.
• **ValueError** – If the filters are invalid.
• **RuntimeError** – If the filters are not supported by the index.

Project.**find_jobs** *(filter=None, doc_filter=None, index=None)*

Find all jobs in the project’s workspace.

The optional filter arguments must be a Mapping of key-value pairs and JSON serializable.

**Note:** Providing a pre-calculated index may vastly increase the performance of this function.

**Parameters**

• **filter** *(Mapping)* – A mapping of key-value pairs that all indexed job statepoints are compared against.

• **doc_filter** – A mapping of key-value pairs that all indexed job documents are compared against.

**Yields** Instances of **Job**

**Raises**

• **TypeError** – If the filters are not JSON serializable.
• **ValueError** – If the filters are invalid.
• **RuntimeError** – If the filters are not supported by the index.

Project.**find_statepoints** *(filter=None, doc_filter=None, index=None, skip_errors=False)*

Find all statepoints in the project’s workspace.

**Parameters**

• **filter** *(mapping)* – If not None, only yield statepoints matching the filter.

• **skip_errors** *(bool)* – Show, but otherwise ignore errors while iterating over the workspace. Use this argument to repair a corrupted workspace.

**Yields** statepoints as dict

Project.**get_id**()

Get the project identifier.

**Returns** The project id.

**Return type** *str*

**Raises** **LookupError** – If no project id could be determined.

**classmethod Project.** **get_project** *(root=None)*

Find a project configuration and return the associated project.

**Parameters**

• **root** *(str)* – The project root directory. If no root directory is given, the next project found within or above the current working directory is returned.

**Returns** The project handle.
Raises \texttt{LookupError} – If no project configuration can be found.

\texttt{Project.get_statepoint} \texttt{(jobid, fn=None)}
Get the statepoint associated with a job id.

The statepoint is retrieved from the workspace or from the statepoints file if the former attempt fails.

\textbf{Parameters}

\begin{itemize}
  \item \texttt{jobid} (\texttt{str}) – A job id to get the statepoint for.
  \item \texttt{fn} (\texttt{str}) – The filename of the file containing the statepoints, defaults to \texttt{FN\_STATEPOINTS}.
\end{itemize}

\textbf{Returns} The statepoint.
\textbf{Return type} \texttt{dict}

\textbf{Raises} \texttt{KeyError} – If the statepoint associated with jobid could not be found.

See also \texttt{dump\_statepoints()}.  

\texttt{Project.index} \texttt{(formats=None, depth=0, skip\_errors=False, include\_job\_document=True)}
Generate an index of the project’s workspace.

This generator function indexes every file in the project’s workspace until the specified \texttt{depth}. The job document if it exists, is always indexed, other files need to be specified with the formats argument.

\begin{verbatim}
for doc in project.index({'.*\.txt', 'TextFile'}):
    print(doc)
\end{verbatim}

\textbf{Parameters}

\begin{itemize}
  \item \texttt{formats} (\texttt{dict}) – The format definitions as mapping.
  \item \texttt{depth} (\texttt{int}) – Specifies the crawling depth. A value of 0 (default) means no limit.
  \item \texttt{skip\_errors} (\texttt{bool}) – Skip all errors which occur during indexing. This is useful when trying to repair a broken workspace.
  \item \texttt{include\_job\_document} (\texttt{bool}) – Include the contents of job documents.
\end{itemize}

\textbf{Yields} index documents

\textbf{classmethod} \texttt{Project.init\_project} \texttt{(name, root=None, workspace=None, make\_dir=True)}
Initialize a project with the given name.

It is safe to call this function multiple times with the same arguments. However, a \texttt{RuntimeError} is raised in case where an existing project configuration would conflict with the provided initialization parameters.

\textbf{Parameters}

\begin{itemize}
  \item \texttt{name} (\texttt{str}) – The name of the project to initialize.
  \item \texttt{root} (\texttt{str}) – The root directory for the project. Defaults to the current working directory.
  \item \texttt{workspace} (\texttt{str}) – The workspace directory for the project. Defaults to \texttt{$project\_root/workspace}.
  \item \texttt{make\_dir} (\texttt{bool}) – Create the project root directory, if it does not exist yet.
\end{itemize}

\textbf{Returns} The project handle of the initialized project.
\textbf{Return type} \texttt{Project}

\textbf{Raises} \texttt{RuntimeError} – If the project root path already contains a conflicting project configuration.
Project.min_len_unique_id()
Determine the minimum length required for an id to be unique.

Project.num_jobs()
Return the number of initialized jobs.

Project.open_job(statepoint=None, id=None)
Get a job handle associated with a statepoint.

This method returns the job instance associated with the given statepoint or job id. Opening a job by a valid statepoint never fails. Opening a job by id, requires a lookup of the statepoint from the job id, which may fail if the job was not previously initialized.

Parameters

- **statepoint (mapping)** – The job’s unique set of parameters.
- **id (str)** – The job id.

Returns
The job instance.

Return type
**Job**

Raises

- **KeyError** – If the attempt to open the job by id fails.
- **LookupError** – If the attempt to open the job by an abbreviated id returns more than one match.

Project.read_statepoints(fn=None)
Read all statepoints from a file.

Parameters

- **fn (str)** – The filename of the file containing the statepoints, defaults to FN_STATEPOINTS.

See also dump_statepoints() and write_statepoints().

Project.repair()
Attempt to repair the workspace after it got corrupted.

Project.reset_statepoint(job, new_statepoint)
Reset the state point of job.

**Danger:** Use this function with caution! Resetting a job’s state point, may sometimes be necessary, but can possibly lead to incoherent data spaces.

Parameters

- **job (Job)** – The job, that should be reset to a new state point.
- **new_statepoint (mapping)** – The job’s new state point.

Raises

- **DestinationExistsError** – If a job associated with the new state point is already initialized.
- **OSError** – If the move failed due to an unknown system related error.

Project.root_directory()
Returns the project’s root directory.
Project.update_statepoint(job, update, overwrite=False)

Update the statepoint of this job.

**Warning:** While appending to a job’s state point is generally safe, modifying existing parameters may lead to data inconsistency. Use the overwrite argument with caution!

**Parameters**

- **job** *(Job)* – The job, whose statepoint shall be updated.
- **update** *(mapping)* – A mapping used for the statepoint update.
- **overwrite** – Set to true, to ignore whether this update overwrites parameters, which are currently part of the job’s state point. Use with caution!

**Raises**

- **KeyError** – If the update contains keys, which are already part of the job’s state point and overwrite is False.
- **DestinationExistsError** – If a job associated with the new state point is already initialized.
- **OSError** – If the move failed due to an unknown system related error.

Project.workspace()

Returns the project’s workspace directory.

The workspace defaults to `project_root/workspace`. Configure this directory with the `workspace_dir` attribute. If the specified directory is a relative path, the absolute path is relative from the project’s root directory.

**Note:** The configuration will respect environment variables, such as $HOME.

Project.write_statepoints(statepoints=None, fn=None, indent=2)

Dump statepoints to a file.

If the file already contains statepoints, all new statepoints will be appended, while the old ones are preserved.

**Parameters**

- **statepoints** *(iterable)* – A list of statepoints, defaults to all statepoints which are defined in the workspace.
- **fn** *(str)* – The filename of the file containing the statepoints, defaults to `FN_STATEPOINTS`.
- **indent** *(int)* – Specify the indentation of the json file.

See also `dump_statepoints()`.

signac.contrib.get_project(root=None)

Find a project configuration and return the associated project.

**Parameters**

- **root** *(str)* – The project root directory. If no root directory is given, the next project found within or above the current working directory is returned.

**Returns**

The project handle.

**Return type** *Project*
signac Documentation, Release 0.8.3

Raises `LookupError` – If no project configuration can be found.

```
signac.contrib.init_project(name, root=None, workspace=None, make_dir=True)
```

Initialize a project with the given name.

It is safe to call this function multiple times with the same arguments. However, a `RuntimeError` is raised in case where an existing project configuration would conflict with the provided initialization parameters.

**Parameters**

- `name` *(str)* – The name of the project to initialize.
- `root` *(str)* – The root directory for the project. Defaults to the current working directory.
- `workspace` *(str)* – The workspace directory for the project. Defaults to `$project_root/workspace`.
- `make_dir` *(bool)* – Create the project root directory, if it does not exist yet.

**Returns**

The project handle of the initialized project.

**Return type** `Project`

**Raises** `RuntimeError` – If the project root path already contains a conflicting project configuration.

```
class signac.contrib.BaseCrawler(root)
```

**Bases:** `object`

Crawl through `root` and index all files.

The crawler creates an index on data, which can be exported to a database for easier access.

```
crawl(depth=0)
```

Crawl through the `root` directory.

The crawler will inspect every file and directory up until the specified `depth` and call the `docs_from_file()` method.

**Parameters**

- `depth` – Crawl through the directory for the specified depth. A value of 0 specifies no limit.

**Yields** *(id, doc)-tuples*

```
docs_from_file(dirpath, fn)
```

Implement this method to generate documents from files.

**Parameters**

- `dirpath` *(str)* – The path of the file, relative to `root`.
- `fn` *(str)* – The filename.

**Yields** Index documents.

```
fetch(doc, mode='r')
```

Implement this generator method to associate data with a document.

**Returns** object associated with doc

```
process(doc, dirpath, fn)
```

Implement this method for additional processing of generated docs.

The default implementation will return the unmodified `doc`.

**Parameters**

- `dirpath` *(str)* – The path of the file, relative to `root`. 
• **fn** *(str)* – The filename.

Returns  A document, that means an instance of mapping.

Return type  mapping

tags = None

class signac.contrib.RegexFileCrawler *(root)*

    Bases: signac.contrib.indexing.BaseCrawler

Generate documents from filenames and associate each file with a data type.

The *RegexFileCrawler* uses regular expressions to generate data from files. This is a particular easy method to retrieve meta data associated with files. Inherit from this class to configure a crawler for your data structure.

Let’s assume we want to index text files, with a naming pattern, that specifies a parameter *a* through the filename, e.g.:

```
~/my_project/a_0.txt
~/my_project/a_1.txt
...
```

A valid regular expression to match this pattern would be: `.*\/a_(?P<a>[0-9]+)\.txt` which may be defined for a crawler as such:

```
MyCrawler(RegexFileCrawler):
    pass
MyCrawler.define('.*\/a_(?P<a>[0-9]+)\.txt', 'TextFile')
```

classmethod compute_file_id *(doc, file)*

    Compute the file id for a given doc and the associated file.

    Parameters

    • **doc** – The index document
    • **file** – The associated file

    Returns  The file id.

crawl *(depth=0)*

classmethod define *(regex, format=None)*

    Define a format for a particular regular expression.

    Parameters

    • **regex** *(str)* – All files of the specified format must match this regular expression.
    • **format** *(object)* – The format associated with all matching files.

definitions = {}  
docs_from_file *(dirpath, fn)*

    Generate documents from filenames.

    This method implements the abstract :py:meth:`BaseCrawler.docs_from_file` and yields index documents associated with files.

    Note:  It is not recommended to reimplement this method to modify documents generated from filenames. See process() instead.
Parameters

- **dirpath** – The path of the file relative to `root`.
- **fn** – The filename of the file.

Yields Index documents.

**fetch**(doc, mode='r')

Fetch the data associated with `doc`.

Parameters  
**doc** (dict) – A index document.  

Returns The file associated with the index document.

Return type A file-like object

**process**(doc, dirpath, fn)

Post-process documents generated from filenames.

Example:

```python
MyCrawler(signac.indexing.RegexFileCrawler):
    def process(self, doc, dirpath, fn):
        doc['long_name_for_a'] = doc['a']
        return super(MyCrawler, self).process(doc, dirpath, fn)
```

Parameters

- **dirpath** (str) – The path of the file, relative to `root`.
- **fn** (str) – The filename.

Returns An index document, that means an instance of mapping.

Return type mapping

**class** signac.contrib.JSONCrawler(root)

Bases: signac.contrib.indexing.BaseCrawler

docs_from_file(dirpath, fn)
docs_from_json(doc)
encoding = 'utf-8'
fn_regex = '.*\.json'

**class** signac.contrib.SignacProjectCrawler(root)

Bases: signac.contrib.indexing.RegexFileCrawler

Index a signac project workspace.

Without any file format definitions, this crawler yields index documents for each job, including the statepoint and the job document.

See also: RegexFileCrawler

Parameters  
**root** (str) – The path to the project’s root directory.

crawl(depth=0)
encoding = 'utf-8'
fn_job_document = 'signac_job_document.json'
fn_statepoint = 'signac_statepoint.json'
get_statepoint (dirpath)
process (doc, dirpath, fn)
signac_id_alias = 'id'
statepoint_index = 'statepoint'
class signac.contrib.MasterCrawler (root, raise_on_error=False)
Bases: signac.contrib.indexing.BaseCrawler
Compiles a master index from indexes defined in access modules.
An instance of this crawler will search the data space for access modules, which by default are named signac_access.py. Once such a file is found, the crawler will import the module and try to execute two special functions given that they are defined within the module’s global namespace: get_indexes() and get_crawlers().
The get_indexes() function is assumed to yield one or multiple index generator functions, while the get_crawlers() function is assumed to yield one or more crawler instances.

This is an example for such an access module:

```
import signac

def get_indexes(root):
    yield signac.index_files(root, '.*\.txt')

def get_crawlers(root):
    yield MyCrawler(root)
```

In case that the master crawler has tags, the get_indexes() function will always be ignored while crawlers yielded from the get_crawlers() function will only be executed in case that they match at least one of the tags.

In case that the access module is completely empty, it will be executed as if it had the following directives:

```
import signac

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

Tags for indexes yielded from the get_indexed() function can be specified by assigning them directly to the function:

```
def get_indexes(root):
    yield signac.get_project(root).index()
get_indexes.tags = ('foo')
```

Parameters

- root (str) – The path to the root directory to crawl through.
- raise_on_error (bool) – Raise all exceptions encountered during crawling instead of ignoring them.

FN_ACCESS_MODULE = 'signac_access.py'
docs_from_file (dirpath, fn)
Compile master index from file in case it is an access module.
Parameters

- `dirpath` – The path of the file relative to root.
- `fn` – The filename of the file.

Yields

Index documents.

```python
signac.contrib.fetch(doc_or_id, mode='r', mirrors=None, num_tries=3, timeout=60, ignore_local=False)
```

Fetch the file associated with this document or file id.

This function retrieves a file associated with the provided index document or file id and behaves like the built-in `open()` function, e.g.:

```python
for doc in index:
    with signac.fetch(doc) as file:
        do_something_with(file)
```

Parameters

- `doc_or_id` – A file_id or a document with a file_id value.
- `mode` – Mode to use for opening files.
- `mirrors` – An optional set of mirrors to fetch the file from.
- `num_tries (int)` – The number of automatic retry attempts in case of mirror connection errors.
- `timeout (int)` – The time in seconds to wait before an automatic retry attempt.

Returns

The file associated with the document or file id.

Return type

A file-like object

```python
signac.contrib.fetch_one(doc, *args, **kwargs)
```

Legacy function, use `fetch()` instead.

```python
signac.contrib.fetched(docs)
```

Iterate over documents and yield associated files.

```python
signac.contrib.export_one(doc, index, mirrors=None, num_tries=3, timeout=60)
```

Export one document to index and an optionally associated file to mirrors.

Parameters

- `doc` – A document with a file_id entry.
- `docs` – The index collection to export to.
- `mirrors` – An optional set of mirrors to export files to.
- `num_tries (int)` – The number of automatic retry attempts in case of mirror connection errors.
- `timeout (int)` – The time in seconds to wait before an automatic retry attempt.

Returns

The id and file id after successful export.

```python
signac.contrib.export(docs, index, mirrors=None, update=False, num_tries=3, timeout=60, **kwargs)
```

Export docs to index and optionally associated files to mirrors.

The behavior of this function is equivalent to:
If the `update` argument is set to True, the export algorithm will automatically identify stale index documents, that means documents that refer to files or state points that have been removed and are no longer part of the data space. Any document which shares the `root`, but not the `_id` field with any of the updated documents is considered stale and removed. Using `update` in combination with an empty `docs` sequence will raise `ExportError`, since it is not possible to identify stale documents in that case.

**Note:** This function will automatically delegate to specialized implementations for special index types. For example, if the `index` argument is a MongoDB document collection, the index documents will be exported via `export_pymongo()`.

### Parameters
- **docs** – The index documents to export.
- **index** – The collection to export the index to.
- **mirrors** – An optional set of mirrors to export files to.
- **update** *(bool)* – If True, remove stale index documents, that means documents that refer to files or state points that no longer exist.
- **num_tries** *(int)* – The number of automatic retry attempts in case of mirror connection errors.
- **timeout** *(int)* – The time in seconds to wait before an automatic retry attempt.
- **kwargs** – Optional keyword arguments to pass to delegate implementations.

**Raises** `ExportError` – When using the update argument in combination with an empty docs sequence.

```python
for doc in docs:
    export_one(doc, index, mirrors, num_tries)
```

### Export to Mirror

Export a file associated with `doc` to `mirror`.

**Parameters**
- **doc** – A document with a `file_id` entry.
- **mirror** – A file-system object to export the file to.
- **num_tries** *(int)* – The number of automatic retry attempts in case of mirror connection errors.
- **timeout** *(int)* – The time in seconds to wait before an automatic retry attempt.

**Returns** The file id after successful export.

```python
signac.contrib.export_to_mirror(doc, mirror, num_tries=3, timeout=60)
```

### Export Pymongo

Optimized `export()` function for pymongo index collections.

The behavior of this function is roughly equivalent to:

```python
for doc in docs:
    export_one(doc, index, mirrors, num_tries)
```
Note: All index documents must be JSON-serializable to be able to be exported to a MongoDB collection.

Parameters

- **docs** – The index documents to export.
- **index** ([pymongo.collection.Collection](https://docs.mongodb.com/)) – The database collection to export the index to.
- **num_tries** (**int**) – The number of automatic retry attempts in case of mirror connection errors.
- **timeout** (**int**) – The time in seconds to wait before an automatic retry attempt.
- **chunksize** (**int**) – The buffer size for export operations.

signac.contrib.index_files(root='.', formats=None, depth=0)

Generate a file index.

This generator function yields file index documents, where each index document corresponds to one file.

To index all files in the current working directory, simply execute:

```python
for doc in signac.index_files():
    print(doc)
```

A file associated with a file index document can be fetched via the `fetch()` function:

```python
for doc in signac.index_files():
    with signac.fetch(doc) as file:
        print(file.read())
```

This is especially useful if the file index is part of a collection ([Collection](https://docs.mongodb.com/)) which can be searched for specific entries.

To limit the file index to files with a specific filename formats, provide a regular expression as the formats argument. To index all files that have file ending `.txt`, execute:

```python
for doc in signac.index_files(formats='.*\.txt'):  
    print(doc)
```

We can specify specific formats by providing a dictionary as `formats` argument, where the key is the filename pattern and the value is an arbitrary formats string, e.g.:

```python
for doc in signac.index_files(formats=  
    {'.*\.txt': 'TextFile', '*.zip': 'ZipFile'}):
    print(doc)
```

Parameters

- **root** (**str**) – The directory to index, defaults to the current working directory.
- **formats** – Limit the index to files that match the given regular expression and optionally associate formats with given patterns.
- **depth** (**int**) – Limit the search to the specified directory depth.

Yields The file index documents as dicts.
signac.contrib.index(root='.', tags=None, depth=0, **kwargs)

Generate a master index.

A master index is compiled from other indexes by searching for modules named signac_access.py and compiling all indexes which are yielded from a function get_indexes(root) defined within that module as well as the indexes generated by crawlers yielded from a function get_crawlers(root) defined within that module.

This is a minimal example for a signac_access.py file:

```python
import signac
def get_indexes(root):
    yield signac.index_files(root, '.*\.txt')
```

Internally, this function constructs an instance of MasterCrawler and all extra key-word arguments will be forwarded to the constructor of said master crawler.

**Parameters**

- **root (str)** – Look for access modules under this directory path.
- **tags** – If tags are provided, do not execute slave crawlers that don’t match the same tags.
- **depth (int)** – Limit the search to the specified directory depth.
- **kwargs** – These keyword-arguments are forwarded to the internal MasterCrawler instance.

**Yields** The master index documents as instances of dict.

class signac.contrib.Collection(docs=None, primary_key='_id')

Bases: object

A collection of documents.

The Collection class manages a collection of documents in memory or in a file on disk. A document is defined as a dictionary mapping of key-value pairs.

An instance of collection may be used to manage and search documents. For example, given a collection with member data, where each document contains a name entry and an age entry, we can find the name of all members that are at age 32 like this:

```python
members = [
    {'name': 'John', 'age': 32},
    {'name': 'Alice', 'age': 28},
    {'name': 'Kevin', 'age': 32},
    # ...
]
member_collection = Collection(members)
for doc in member_collection.find({'age': 32}):
    print(doc['name'])
```

To iterate over all documents in the collection, use:

```python
for doc in collection:
    print(doc)
```

By default a collection object will reside in memory. However, it is possible to manage a collection associated to a file on disk. To open a collection which is associated with a file on disk, use the open() class method:
with Collection.open('collection.txt') as collection:
    for doc in collection.find({'age': 32}):
        print(doc)

The collection file is by default opened in a+ mode, which means it can be read from and written to and will be created if it does not exist yet.

Parameters

- **docs** – Initialize the collection with these documents.
- **primary_key** – The name of the key which serves as the primary index of the collection. Selecting documents by primary key has time complexity of O(N) in the worst case and O(1) on average. All documents must have a primary key value. The default primary key is `_id`.

`clear()`
Remove all documents from the collection.

`close()`
Close this collection instance.

In case that the collection is associated with a file-object, all changes are flushed to the file and the file is closed.

It is not possible to re-open the same collection instance after closing it.

`delete_many(filter)`
Delete all documents that match the filter.

`delete_one(filter)`
Delete one document that matches the filter.

`dump(file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>)`
Dump the collection in JSON-encoding to file.

The file argument defaults to `sys.stdout`, which means the encoded blob will be printed to screen in case that no file argument is provided.

For example, to dump to a file on disk, one could write:

```python
with open('my_collection.txt', 'w') as file:
    collection.dump(file)
```

Parameters **file** – The file to write the encoded blob to.

`find(filter=None, limit=0)`
Find all documents matching filter, but not more than limit.

This function searches the collection for all documents that match the given filter and returns a result vector. For example:

```python
for doc in collection.find(my_filter):
    print(doc)
```

Nested values should be searched using the . operator, for example:

```python
docs = collection.find({'nested.value': 42})
```

will return documents with a nested structure: `{"nested": {"value": 42}}.  

1.7. API
The result of `find()` can be stored and iterated over multiple times. In addition, the result vector can be queried for its size:

```python
docs = collection.find(my_filter)
print(len(docs))  # the number of documents matching
for doc in docs:  # iterate over the result vector
    pass
```

**Parameters**

- `filter` *(Mapping)* – All documents must match the given filter.
- `limit` *(int)* – Do not return more than limit number of documents. A limit value of 0 (the default) means no limit.

**Returns** A result object that iterates over all matching documents.

**Raises** `ValueError` – In case that the filter argument is invalid.

### find_one(filter=None)

Return one document that matches the filter or None.

```python
doc = collection.find_one(my_filter)
if doc is None:
    print("No result found for filter", my_filter)
else:
    print("Doc matching filter:", my_filter, doc)
```

**Parameters** `filter` – The returned document must match the given filter.

**Raises** `ValueError` – In case that the filter argument is invalid.

**Returns** A matching document or None.

### flush()

Write all changes to the associated file.

If the collection instance is associated with a file-object, calling the `flush()` method will write all changes to this file.

This method is also called when the collection is explicitly or implicitly closed.

### ids

Return an iterator over the primary key in the collection.

### index(key, build=False)

Get (and optionally build) the index for a given key.

An index allows to access documents by a specific key with minimal time complexity, e.g.:

```python
age_index = member_collection.index('age')
for _id in age_index[32]:
    print(member_collection[_id]['name'])
```

This means we can access documents by the ‘age’ key in O(1) time on average in addition to the primary key. Using the `find()` method will automatically build all required indexes for the particular search.

Once an index has been built, it will be internally managed by the class and updated with subsequent changes. An index returned by this method is always current with the latest state of the collection.
Parameters

- **key** *(str)* – The primary key of the requested index.
- **build** – If True, build a non-existing index if necessary, otherwise raise KeyError.

**Raises** **KeyError** – In case that build is False and the index has not been built yet.

**insert_one** *(doc)*

Insert one document into the collection

If the document does not have a value for the collection’s primary key yet, it will be assigned one.

```python
_id = collection.insert_one(doc)
assert _id in collection
```

**Note:** The document will be directly updated in case that it has no primary key and must therefore be mutable!

**Parameters** **doc** – The document to be inserted.

**Returns** The _id of the inserted documented.

**main**()

Start a command line interface for this Collection.

Use this function to interact with this instance of Collection on the command line. For example, executing the following script:

```python
# find.py
with Collection.open('my_collection.txt') as c:
    c.main()
```

will enable us to search for documents on the command line like this:

```bash
$ python find.py '{"age": 32}'
{"name": "John", "age": 32}
{"name": "Kevin", "age": 32}
```

**classmethod** **open** *(filename, mode='a+')*

Open a collection associated with a file on disk.

Using this factory method will return a collection that is associated with a collection file on disk. For example:

```python
with Collection.open('collection.txt') as collection:
    for doc in collection:
        print(doc)
```

will read all documents from the *collection.txt* file or create the file if it does not exist yet.

Modifications to the file will be written to the file when the **flush**() method is called or the collection is explicitly closed by calling the **close**() method or implicitly by leaving the **with**-clause:

```python
with Collection.open('collection.txt') as collection:
    collection.update(my_docs)
    # All changes to the collection have been written to collection.txt.
```
The open-modes work as expected, so for example to open a collection file in read-only mode, use

```
Collection.open('collection.txt', 'r')
```

**primary_key**
The name of the collection’s primary key (default='_id').

**replace_one** (filter, replacement, upsert=False)
Replace one document that matches the given filter.

The first document matching the filter will be replaced by the given replacement document. If the upsert argument is True, the replacement will be inserted in case that no document matches the filter.

**Parameters**
- **filter** – A document that should be replaced must match this filter.
- **replacement** – The replacement document.
- **upsert** – If True, insert the replacement document in the case that no document matches the filter.

**Raises** **ValueError** – In case that the filter argument is invalid.

**Returns** The _id of the replaced (or upserted) documented.

**update** (docs)
Update the collection with these documents.

Any existing documents with the same primary key will be replaced.

**Parameters**
- **docs** – A sequence of documents to be upserted into the collection.

---

**class** **signac.contrib.MPIPool** (comm=None, debug=False, loadbalance=False)

**Bases:** object

A pool that distributes tasks over a set of MPI processes using mpi4py. MPI is an API for distributed memory parallelism, used by large cluster computers. This class provides a similar interface to Python’s multiprocessing Pool, but currently only supports the `map()` method.

Contributed initially by Joe Zuntz.

**Parameters**
- **comm** ([optional]) The mpi4py communicator.
- **debug** [bool (optional)] If True, print out a lot of status updates at each step.
- **loadbalance** [bool (optional)] if True and the number of tasks is greater than the number of processes, tries to loadbalance by sending out one task to each cpu first and then sending out the rest as the cpus get done.

**bcast** (*args, **kwargs)
Equivalent to mpi4py bcast() collective operation.

**close**()
Just send a message off to all the pool members which contains the special _close_pool_message sentinel.

**is_master**()
Is the current process the master?

**map** (function, tasks, ntask=None, callback=None)
Like the built-in `map()` function, apply a function to all of the values in a list and return the list of results.

**Parameters**
- **function** [callable] The function to apply to each element in the list.
tasks : A list of tasks – each element is passed to the input function.

callback [callable (optional)] A callback function to call on each result.

wait ()
If this isn’t the master process, wait for instructions.

Submodules

signac.contrib.indexing module

class signac.contrib.indexing.BaseCrawler (root)
    Bases: object
    Crawl through root and index all files.
    The crawler creates an index on data, which can be exported to a database for easier access.

crawl (depth=0)
    Crawl through the root directory.
    The crawler will inspect every file and directory up until the specified depth and call the
    docs_from_file() method.

    Parameters
        depth – Crawl through the directory for the specified depth. A value of 0 specifies
                  no limit.

    Yields (id, doc)-tuples

docs_from_file(dirpath, fn)
    Implement this method to generate documents from files.

    Parameters
        • dirpath (str) – The path of the file, relative to root.
        • fn (str) – The filename.

    Yields Index documents.

fetch (doc, mode='r')
    Implement this generator method to associate data with a document.

    Returns object associated with doc

process (doc, dirpath, fn)
    Implement this method for additional processing of generated docs.
    The default implementation will return the unmodified doc.

    Parameters
        • dirpath (str) – The path of the file, relative to root.
        • fn (str) – The filename.

    Returns A document, that means an instance of mapping.

    Return type mapping

tags = None

class signac.contrib.indexing.JSONCrawler (root)
    Bases: signac.contrib.indexing.BaseCrawler
**docs_from_file**(dirpath, fn)

**docs_from_json**(doc)

**encoding** = ‘utf-8’

**fn_regex** = ‘.\n\json’

**class** signac.contrib.indexing.MasterCrawler**(root, raise_on_error=False)**

**Bases:** signac.contrib.indexing.BaseCrawler

Compiles a master index from indexes defined in access modules.

An instance of this crawler will search the data space for access modules, which by default are named signac_access.py. Once such a file is found, the crawler will import the module and try to execute two special functions given that they are defined within the module’s global namespace: get_indexes() and get_crawlers().

The get_indexes() is assumed to yield one or multiple index generator functions, while the get_crawlers() function is assumed to yield one or more crawler instances.

This is an example for such an access module:

```python
import signac

def get_indexes(root):
    yield signac.index_files(root, '.*\.txt')

def get_crawlers(root):
    yield MyCrawler(root)
```

In case that the master crawler has tags, the get_indexes() function will always be ignored while crawlers yielded from the get_crawlers() function will only be executed in case that they match at least one of the tags.

In case that the access module is completely empty, it will be executed as if it had the following directives:

```python
import signac

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

Tags for indexes yielded from the get_indexes() function can be specified by assigning them directly to the function:

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

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

**Parameters**

- **root** *(str)* – The path to the root directory to crawl through.
- **raise_on_error** *(bool)* – Raise all exceptions encountered during during crawling instead of ignoring them.

**FN_ACCESS_MODULE** = ‘signac_access.py’

**docs_from_file**(dirpath, fn)

Compile master index from file in case it is an access module.
Parameters

- **dirpath** – The path of the file relative to root.
- **fn** – The filename of the file.

Yields

Index documents.

class signac.contrib.indexing.RegexFileCrawler(root)

Bases: signac.contrib.indexing.BaseCrawler

Generate documents from filenames and associate each file with a data type.

The *RegexFileCrawler* uses regular expressions to generate data from files. This is a particular easy method to retrieve meta data associated with files. Inherit from this class to configure a crawler for your data structre.

Let’s assume we want to index text files, with a naming pattern, that specifies a parameter *a* through the filename, e.g.:

```bash
~/my_project/a_0.txt
~/my_project/a_1.txt
...
```

A valid regular expression to match this pattern would be: `.*\/_a_(?P<a>\d+).txt` which may be defined for a crawler as such:

```python
MyCrawler(RegexFileCrawler):
    pass
MyCrawler.define('.*\/_a_(?P<a>\d+).txt', 'TextFile')
```

classmethod compute_file_id(doc, file)

Compute the file id for a given doc and the associated file.

Parameters

- **doc** – The index document
- **file** – The associated file

Returns

The file id.

crawl(depth=0)

classmethod define(regex, format=None)

Define a format for a particular regular expression.

Parameters

- **regex** *(str)* – All files of the specified format must match this regular expression.
- **format** *(object)* – The format associated with all matching files.

definitions = {}

docs_from_file(dirpath, fn)

Generate documents from filenames.

This method implements the abstract :py:meth:`BaseCrawler.docs_from_file` and yields index documents associated with files.

**Note:** It is not recommended to reimplement this method to modify documents generated from filenames. See *process()* instead.
Parameters

- **dirpath** – The path of the file relative to root.
- **fn** – The filename of the file.

Yields  Index documents.

```python
fetch(doc, mode='r')
```

Fetch the data associated with `doc`.

Parameters

- **doc** *(dict)* – A index document.

Returns  The file associated with the index document.

Return type  A file-like object

```python
process(doc, dirpath, fn)
```

Post-process documents generated from filenames.

Example:

```python
MyCrawler(signac.indexing.RegexFileCrawler):
    def process(self, doc, dirpath, fn):
        doc['long_name_for_a'] = doc['a']
        return super(MyCrawler, self).process(doc, dirpath, fn)
```

Parameters

- **dirpath** *(str)* – The path of the file, relative to `root`.
- **fn** *(str)* – The filename.

Returns  An index document, that means an instance of mapping.

Return type  mapping

class signac.contrib.indexing.SignProjectCrawler(root)

Bases: signac.contrib.indexing.RegexFileCrawler

Index a signac project workspace.

Without any file format definitions, this crawler yields index documents for each job, including the statepoint and the job document.

See also:  `RegexFileCrawler`

Parameters

- **root** *(str)* – The path to the project’s root directory.

```python
crawl(depth=0)
```

encoding = ‘utf-8’

```python
fn_job_document = 'signac_job_document.json'
```

```python
fn_statepoint = 'signac_statepoint.json'
```

```python
get_statepoint(dirpath)
```

```python
process(doc, dirpath, fn)
```

```python
signac_id_alias = '_id'
```

```python
statepoint_index = 'statepoint'
```
signac.contrib.indexing.export (docs, index, mirrors=None, update=False, num_tries=3, timeout=60, **kwargs)

Export docs to index and optionally associated files to mirrors.

The behavior of this function is equivalent to:

```python
for doc in docs:
    export_one(doc, index, mirrors, num_tries)
```

If the `update` argument is set to True, the export algorithm will automatically identify stale index documents, that means documents that refer to files or state points that have been removed and are no longer part of the data space. Any document which shares the `root`, but not the `_id` field with any of the updated documents is considered stale and removed. Using `update` in combination with an empty docs sequence will raise `ExportError`, since it is not possible to identify stale documents in that case.

**Note:** This function will automatically delegate to specialized implementations for special index types. For example, if the index argument is a MongoDB document collection, the index documents will be exported via `export_pymongo()`.

**Parameters**

- `docs` – The index documents to export.
- `index` – The collection to export the index to.
- `mirrors` – An optional set of mirrors to export files to.
- `update (bool)` – If True, remove stale index documents, that means documents that refer to files or state points that no longer exist.
- `num_tries (int)` – The number of automatic retry attempts in case of mirror connection errors.
- `timeout (int)` – The time in seconds to wait before an automatic retry attempt.
- `kwargs` – Optional keyword arguments to pass to delegate implementations.

**Returns** `ExportError` – When using the update argument in combination with an empty docs sequence.

signac.contrib.indexing.export_one (doc, index, mirrors=None, num_tries=3, timeout=60)

Export one document to index and an optionally associated file to mirrors.

**Parameters**

- `doc` – A document with a file_id entry.
- `docs` – The index collection to export to.
- `mirrors` – An optional set of mirrors to export files to.
- `num_tries (int)` – The number of automatic retry attempts in case of mirror connection errors.
- `timeout (int)` – The time in seconds to wait before an automatic retry attempt.

**Returns** The id and file id after successful export.

signac.contrib.indexing.export_pymongo (docs, index, mirrors=None, num_tries=3, timeout=60, chunksize=100)

Optimized `export ()` function for pymongo index collections.

The behavior of this function is roughly equivalent to:
for doc in docs:
    export_one(doc, index, mirrors, num_tries)

**Note:** All index documents must be JSON-serializable to be able to be exported to a MongoDB collection.

**Parameters**

- **docs** – The index documents to export.
- **index** (*pymongo.collection.Collection*) – The database collection to export the index to.
- **num_tries** (*int*) – The number of automatic retry attempts in case of mirror connection errors.
- **timeout** (*int*) – The time in seconds to wait before an automatic retry attempt.
- **chunksize** (*int*) – The buffer size for export operations.

**signac.contrib.indexing.export_to_mirror**

Export a file associated with doc to mirror.

**Parameters**

- **doc** – A document with a file_id entry.
- **mirror** – A file-system object to export the file to.
- **num_tries** (*int*) – The number of automatic retry attempts in case of mirror connection errors.
- **timeout** (*int*) – The time in seconds to wait before an automatic retry attempt.

**Returns** The file id after successful export.

**signac.contrib.indexing.fetch**

Fetch the file associated with this document or file id.

This function retrieves a file associated with the provided index document or file id and behaves like the built-in `open()` function, e.g.:

```python
for doc in index:
    with signac.fetch(doc) as file:
        do_something_with(file)
```

**Parameters**

- **doc_or_id** – A file_id or a document with a file_id value.
- **mode** – Mode to use for opening files.
- **mirrors** – An optional set of mirrors to fetch the file from.
- **num_tries** (*int*) – The number of automatic retry attempts in case of mirror connection errors.
- **timeout** (*int*) – The time in seconds to wait before an automatic retry attempt.

**Returns** The file associated with the document or file id.

**Return type** A file-like object
signac.contrib.indexing.fetch_one(doc, *args, **kwargs)
Legacy function, use fetch() instead.

signac.contrib.indexing.fetched(docs)
Iterate over documents and yield associated files.

signac.contrib.indexing.index(root='.', tags=None, depth=0, **kwargs)
Generate a master index.

A master index is compiled from other indexes by searching for modules named signac_access.py and compiling all indexes which are yielded from a function get_indexes(root) defined within that module as well as the indexes generated by crawlers yielded from a function get_crawlers(root) defined within that module.

This is a minimal example for a signac_access.py file:

```python
import signac

def get_indexes(root):
    yield signac.index_files(root, '.*\.txt')
```

Internally, this function constructs an instance of MasterCrawler and all extra key-word arguments will be forwarded to the constructor of said master crawler.

**Parameters**

- **root** (str) – Look for access modules under this directory path.
- **tags** – If tags are provided, do not execute slave crawlers that don’t match the same tags.
- **depth** (int) – Limit the search to the specified directory depth.
- **kwargs** – These keyword-arguments are forwarded to the internal MasterCrawler instance.

**Yields**

The master index documents as instances of dict.

signac.contrib.indexing.index_files(root='.', formats=None, depth=0)
Generate a file index.

This generator function yields file index documents, where each index document corresponds to one file.

To index all files in the current working directory, simply execute:

```python
for doc in signac.index_files():
    print(doc)
```

A file associated with a file index document can be fetched via the fetch() function:

```python
for doc in signac.index_files():
    with signac.fetch(doc) as file:
        print(file.read())
```

This is especially useful if the file index is part of a collection (Collection) which can be searched for specific entries.

To limit the file index to files with a specific filename formats, provide a regular expression as the formats argument. To index all files that have file ending .txt, execute:

```python
for doc in signac.index_files(formats='.*\.txt'):  
    print(doc)
```
We can specify specific formats by providing a dictionary as `formats` argument, where the key is the filename pattern and the value is an arbitrary formats string, e.g.:

```python
for doc in signac.index_files(formats={'.*\.txt': 'TextFile', '.*\.zip': 'ZipFile'}):
    print(doc)
```

**Parameters**

- **root** *(str)* – The directory to index, defaults to the current working directory.
- **formats** – Limit the index to files that match the given regular expression and optionally associate formats with given patterns.
- **depth** *(int)* – Limit the search to the specified directory depth.

**Yields** The file index documents as dicts.

### signac.contrib.indexing.md5(file)

Calculate and return the md5 hash value for the file data.

### signac.contrib.collection module

**class** signac.contrib.collection.Collection(*(docs=None, primary_key='_id'))

**_bases:** object

A collection of documents.

The Collection class manages a collection of documents in memory or in a file on disk. A document is defined as a dictionary mapping of key-value pairs.

An instance of collection may be used to manage and search documents. For example, given a collection with member data, where each document contains a `name` entry and an `age` entry, we can find the name of all members that are at age 32 like this:

```python
members = [
    {'name': 'John', 'age': 32},
    {'name': 'Alice', 'age': 28},
    {'name': 'Kevin', 'age': 32},
    ...
]
member_collection = Collection(members)
for doc in member_collection.find({'age': 32}):
    print(doc['name'])
```

To iterate over all documents in the collection, use:

```python
for doc in collection:
    print(doc)
```

By default a collection object will reside in memory. However, it is possible to manage a collection associated to a file on disk. To open a collection which is associated with a file on disk, use the `open()` class method:

```python
with Collection.open('collection.txt') as collection:
    for doc in collection.find({'age': 32}):
        print(doc)
```
The collection file is by default opened in `a+` mode, which means it can be read from and written to and will be created if it does not exist yet.

**Parameters**

- **docs** – Initialize the collection with these documents.
- **primary_key** – The name of the key which serves as the primary index of the collection. Selecting documents by primary key has time complexity of \(O(N)\) in the worst case and \(O(1)\) on average. All documents must have a primary key value. The default primary key is `_id`.

**clear()**
Remove all documents from the collection.

**close()**
Close this collection instance.
- In case that the collection is associated with a file-object, all changes are flushed to the file and the file is closed.
- It is not possible to re-open the same collection instance after closing it.

**delete_many(filter)**
Delete all documents that match the filter.

**delete_one(filter)**
Delete one document that matches the filter.

**dump(file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>)**
Dump the collection in JSON-encoding to file.
- The file argument defaults to `sys.stdout`, which means the encoded blob will be printed to screen in case that no file argument is provided.
- For example, to dump to a file on disk, one could write:

```python
with open('my_collection.txt', 'w') as file:
    collection.dump(file)
```

**Parameters** **file** – The file to write the encoded blob to.

**find(filter=None, limit=0)**
Find all documents matching filter, but not more than limit.

This function searches the collection for all documents that match the given filter and returns a result vector. For example:

```python
for doc in collection.find(my_filter):
    print(doc)
```

Nested values should be searched using the . operator, for example:

```python
docs = collection.find({'nested.value': 42})
```

will return documents with a nested structure: `{'nested': {'value': 42}}`.

The result of `find()` can be stored and iterated over multiple times. In addition, the result vector can be queried for its size:
docs = collection.find(my_filter)
print(len(docs))  # the number of documents matching
for doc in docs:  # iterate over the result vector
    pass

Parameters

• filter (Mapping) – All documents must match the given filter.
• limit (int) – Do not return more than limit number of documents. A limit value of 0 (the default) means no limit.

Returns A result object that iterates over all matching documents.

Raises ValueError – In case that the filter argument is invalid.

find_one (filter=None)
Return one document that matches the filter or None.

doc = collection.find_one(my_filter)
if doc is None:
    print("No result found for filter", my_filter)
else:
    print("Doc matching filter:", my_filter, doc)

Parameters filter – The returned document must match the given filter.

Raises ValueError – In case that the filter argument is invalid.

Returns A matching document or None.

flush()
Write all changes to the associated file.

If the collection instance is associated with a file-object, calling the flush() method will write all changes to this file.

This method is also called when the collection is explicitly or implicitly closed.

ids
Return an iterator over the primary key in the collection.

index (key, build=False)
Get (and optionally build) the index for a given key.

An index allows to access documents by a specific key with minimal time complexity, e.g.:

```python
age_index = member_collection.index('age')
for _id in age_index[32]:
    print(member_collection[_id]['name'])
```

This means we can access documents by the ‘age’ key in O(1) time on average in addition to the primary key. Using the find() method will automatically build all required indexes for the particular search.

Once an index has been built, it will be internally managed by the class and updated with subsequent changes. An index returned by this method is always current with the latest state of the collection.

Parameters

• key (str) – The primary key of the requested index.
• **build** – If True, build a non-existing index if necessary, otherwise raise KeyError.

    Raises **KeyError** – In case that build is False and the index has not been built yet.

    **insert_one**(doc)
    Insert one document into the collection
    If the document does not have a value for the collection’s primary key yet, it will be assigned one.
    
    ```python
    _id = collection.insert_one(doc)
    assert _id in collection
    ```

    **Note:** The document will be directly updated in case that it has no primary key and must therefore be mutable!

    **Parameters** doc – The document to be inserted.

    **Returns** The _id of the inserted documented.

    **main()**
    Start a command line interface for this Collection.

    Use this function to interact with this instance of Collection on the command line. For example, executing the following script:
    
    ```python
    # find.py
    with Collection.open('my_collection.txt') as c:
        c.main()
    ```

    will enable us to search for documents on the command line like this:
    
    ```bash
    $ python find.py '{"age": 32}'
    {"name": "John", "age": 32}
    {"name": "Kevin", "age": 32}
    ```

    **classmethod open**(filename, mode='a+')
    Open a collection associated with a file on disk.

    Using this factory method will return a collection that is associated with a collection file on disk. For example:
    
    ```python
    with Collection.open('collection.txt') as collection:
        for doc in collection:
            print(doc)
    ```

    will read all documents from the *collection.txt* file or create the file if it does not exist yet.

    Modifications to the file will be written to the file when the `flush()` method is called or the collection is explicitly closed by calling the `close()` method or implicitly by leaving the `with`-clause:
    
    ```python
    with Collection.open('collection.txt') as collection:
        collection.update(my_docs)
    # All changes to the collection have been written to collection.txt.
    ```

    The open-modes work as expected, so for example to open a collection file in *read-only* mode, use `Collection.open('collection.txt', 'r')`.

    **primary_key**
    The name of the collection’s primary key (default=''_id'').
**replace_one** *(filter, replacement, upsert=False)*

Replace one document that matches the given filter.

The first document matching the filter will be replaced by the given replacement document. If the `upsert` argument is True, the replacement will be inserted in case that no document matches the filter.

**Parameters**

- **filter** – A document that should be replaced must match this filter.
- **replacement** – The replacement document.
- **upsert** – If True, insert the replacement document in the case that no document matches the filter.

**Raises** `ValueError` – In case that the filter argument is invalid.

**Returns** The `_id` of the replaced (or upserted) documented.

**update**(docs)

Update the collection with these documents.

Any existing documents with the same primary key will be replaced.

**Parameters** docs – A sequence of documents to be upserted into the collection.

---

**signac.contrib.errors module**

**exception** `signac.contrib.errors.DestinationExistsError` *(destination)*

Bases: `signac.core.errors.Error`, `RuntimeError`

The destination for a move or copy operation already exists.

---

**signac.contrib.filesystems module**

The file system handlers defined in this module encapsulate the I/O operations required to store and fetch data from different file systems.

**class** `signac.contrib.filesystems.GridFS` *(db, collection='fs')*

Bases: `object`

A file system handler for the MongoDB GridFS file system.

**Note:** If the `database` argument is a `str`, signac will attempt to connect to the database using the global configuration.

**Parameters**

- **db** *(str or pymongo.database.Database)* – The database used to store the grid.

- **AutoRetry**
- **FileExistsError**
- **FileNotFoundError**
- **config()**

Return the file system configuration for this handler.
**get** (*id*, *mode=’r’)

Open the file with the specified id.

**Warning:** To avoid compatibility issues, all files are opened in text-mode (*r*) by default, however for higher efficiency, files should generally be opened in binary mode (*rb*) whenever possible.

**Parameters**

- *id* (*str*) – The file identifier.
- *mode* – The file mode used for opening.

**Returns** A file-like object to read from.

**gridfs**

Instance of `pymongo.gridfs.GridFS`.

**name** = ‘gridfs’

**new_file** (*id*)

Create a new file for *id*.

**Parameters**

- *id* (*str*) – The file identifier.

**Returns** A file-like object to write to.

**class signac.contrib.filesystems.LocalFS (root)**

Bases: `object`

A file system handler for the local file system.

This handler will store all files at the specified root path using a file id based naming scheme.

**Parameters**

- *root* (*str*) – The path to the root directory.

**exception AutoRetry**

Bases: `RuntimeError`

LocalFS.FileExistsError

alias of `OSError`

LocalFS.FileNotFoundError

alias of `OSError`

**LocalFS.config()**

Return the file system configuration for this handler.

**LocalFS.get** (*id*, *mode=’r’)

Open the file with the specified id.

**Parameters**

- *id* (*str*) – The file identifier.

- *mode* – The file mode used for opening.

**Returns** A file-like object to read from.

**LocalFS.name** = ‘localfs’

**LocalFS.new_file** (*id*, *mode=None*)

Create a new file for *id*.

**Parameters**

- *id* (*str*) – The file identifier.
Returns A file-like object to write to.

signac.contrib.filesystems.filesystems_from_config(fs_config)
Generate file system handlers from a configuration.

This function yields file system handler objects from a file system configuration. A configuration is a mapping where the key identifies the type of file system, and the values represent the argument(s) to the constructor of the specified file system handler. Arguments can be provided as mappings, sequences or single values, e.g.:

```python
# The following two function calls are equivalent and both
# generate two file system handler objects:

filesystems_from_config({
    'localfs': '/path/to/storage',
    'gridfs': ('gridfsdb', 'fs'),
})

filesystems_from_config({
    'localfs': {'root': '/path/to/storage'},
    'gridfs': {'db': 'gridfsdb', 'collection': 'fs'}
})
```

See `LocalFS` for an example of a file system class.

Parameters

**fs_config** – A file system configuration.

Yields file system handlers

signac.contrib.filesystems.filesystems_from_configs(fs_configs)
Generate file system handlers.

The `fs_configs` argument may be a sequence of file system handlers, file system configurations or a mix of both.

See also: `filesystems_from_config()`.

Parameters

**fs_configs** – A sequence of file system handlers or configurations.

Yields file system handlers

signac.contribformats module

signac.contrib.hashing module

signac.contrib.hashing.calc_id(spec)
Calculate and return a hash value for the given spec.

signac.contrib.job module

class signac.contrib.job.Job(project, statepoint)
Bases: object

The job instance is a handle to the data of a unique statepoint.

Application developers should usually not need to directly instantiate this class, but use `open_job()` instead.

FN_DOCUMENT = ‘signac_job_document.json’

FN_MANIFEST = ‘signac_statepoint.json’
close()
Close the job and switch to the previous working directory.

document
The document associated with this job.

Returns  The job document handle.

Return type  JSONDict

fn(filename)
Prepend a filename with the job’s workspace directory path.

Parameters  filename (str) – The filename of the file.

Returns  The full workspace path of the file.

get_id()
The unique identifier for the job’s statepoint.

Returns  The job id.

Return type  str

init()
Initialize the job’s workspace directory.

This function will do nothing if the directory and the job manifest already exist.

isfile(filename)
Return True if file exists in the job’s workspace.

Parameters  filename (str) – The filename of the file.

Returns  True if file with filename exists in workspace.

Return type  bool

move(project)
Move this job to project.

This function will attempt to move this instance of job from its original project to a different project.

Parameters  project (Project) – The project to move this job to.

Raises  DestinationExistsError – If the job is already initialized in project.

open()
Enter the job’s workspace directory.

You can use the Job class as context manager:

```
with project.open_job(my_statepoint) as job:
    # manipulate your job data
```

Opening the context will switch into the job’s workspace, leaving it will switch back to the previous working directory.

remove()
Remove the job’s workspace including the job document.

This function will do nothing if the workspace directory does not exist.

reset_statepoint(new_statepoint)
Reset the state point of this job.
Danger: Use this function with caution! Resetting a job’s state point, may sometimes be necessary, but can possibly lead to incoherent data spaces.

Parameters `new_statepoint (mapping)` – The job’s new state point.

Raises

- `DestinationExistsError` – If a job associated with the new state point is already initialized.
- `OSError` – If the move failed due to an unknown system related error.

sp
Access the job’s state point as attribute dictionary.

statepoint()
The statepoint associated with this job.

Returns The statepoint mapping.

Return type dict

update_statepoint (update, overwrite=False)
Update the statepoint of this job.

Warning: While appending to a job’s state point is generally safe, modifying existing parameters may lead to data inconsistency. Use the overwrite argument with caution!

Parameters

- `update (mapping)` – A mapping used for the statepoint update.
- `overwrite` – Set to true, to ignore whether this update overwrites parameters, which are currently part of the job’s state point. Use with caution!

Raises

- `KeyError` – If the update contains keys, which are already part of the job’s state point and overwrite is False.
- `DestinationExistsError` – If a job associated with the new state point is already initialized.
- `OSError` – If the move failed due to an unknown system related error.

workspace()
Each job is associated with a unique workspace directory.

Returns The path to the job’s workspace directory.

Return type str

ws
The job’s workspace directory.

signac.contrib.mpipool module

MPIPool for MPI-based multiprocessing-like process pools.

This 3rd party module is copied from https://github.com/adrn/mpipool.
class `signac.contrib.mpipool.MPIPool` (*comm=None, debug=False, loadbalance=False*)

Bases: `object`

A pool that distributes tasks over a set of MPI processes using mpi4py. MPI is an API for distributed memory parallelism, used by large cluster computers. This class provides a similar interface to Python’s multiprocessing Pool, but currently only supports the `map()` method.

Contributed initially by Joe Zuntz.

Parameters

- **comm** *(optional)* The mpi4py communicator.
- **debug** *(bool (optional))* If True, print out a lot of status updates at each step.
- **loadbalance** *(bool (optional))* if True and the number of tasks is greater than the number of processes, tries to loadbalance by sending out one task to each cpu first and then sending out the rest as the cpus get done.

- **bcast**( *args, **kwargs*)
  Equivalent to mpi4py `bcast()` collective operation.

- **close**()
  Just send a message off to all the pool members which contains the special `_close_pool_message` sentinel.

- **is_master**()
  Is the current process the master?

- **map**( *function, tasks, ntask=None, callback=None*)
  Like the built-in `map()` function, apply a function to all of the values in a list and return the list of results.

  Parameters

  - **function** *(callable)* The function to apply to each element in the list.
  - **tasks** : A list of tasks – each element is passed to the input function.
  - **callback** *(callable (optional))* A callback function to call on each result.

- **wait**()
  If this isn’t the master process, wait for instructions.

**signac.contrib.project module**

class `signac.contrib.project.JobSearchIndex` *(index)*

Bases: `object`

Search for specific jobs with filters.

The JobSearchIndex allows to search for job_ids, that are part of an index, which match specific statepoint filters or job document filters.

Parameters

- **index** – A document index.

- **find_job_ids**( *filter=None, doc_filter=None*)
  Find the job_ids of all jobs matching the filters.

  The optional filter arguments must be a Mapping of key-value pairs and JSON serializable.

  Parameters

  - **filter** *(Mapping)* – A mapping of key-value pairs that all indexed job statepoints are compared against.
• **doc_filter** – A mapping of key-value pairs that all indexed job documents are compared against.

**Yields** The ids of all indexed jobs matching both filters.

**Raises**

- **TypeError** – If the filters are not JSON serializable.
- **ValueError** – If the filters are invalid.
- **RuntimeError** – If the filters are not supported by the index.

```python
class signac.contrib.project.Project(config=None)
    Bases: object

The handle on a signac project.

Application developers should usually not need to directly instantiate this class, but use `signac.get_project()` instead.

class Job(project, statepoint)
    Bases: object

The job instance is a handle to the data of a unique statepoint.

Application developers should usually not need to directly instantiate this class, but use `open_job()` instead.

    FNDOCUMENT = 'signac_job_document.json'
    FMANIFEST = 'signac_statepoint.json'

close()
    Close the job and switch to the previous working directory.

document
    The document associated with this job.
        Returns The job document handle.
        Return type JSONDict

fn(filename)
    Prepend a filename with the job’s workspace directory path.
        Parameters filename (str) – The filename of the file.
        Returns The full workspace path of the file.

get_id()
    The unique identifier for the job’s statepoint.
        Returns The job id.
        Return type str

init()
    Initialize the job’s workspace directory.
        This function will do nothing if the directory and the job manifest already exist.

isfile(filename)
    Return True if file exists in the job’s workspace.
        Parameters filename (str) – The filename of the file.
        Returns True if file with filename exists in workspace.
        Return type bool

move(project)
    Move this job to project.
This function will attempt to move this instance of job from its original project to a different project.

**Parameters**

- **project** *(Project)* – The project to move this job to.

**Raises**

- *DestinationExistsError* – If the job is already initialized in project.

**open()**

Enter the job’s workspace directory.

You can use the Job class as context manager:

```python
with project.open_job(my_statepoint) as job:
    # manipulate your job data
```

Opening the context will switch into the job’s workspace, leaving it will switch back to the previous working directory.

**remove()**

Remove the job’s workspace including the job document.

This function will do nothing if the workspace directory does not exist.

**reset_statepoint** *(new_statepoint)*

Reset the state point of this job.

**Danger:** Use this function with caution! Resetting a job’s state point, may sometimes be necessary, but can possibly lead to incoherent data spaces.

**Parameters**

- **new_statepoint** *(mapping)* – The job’s new state point.

**Raises**

- *DestinationExistsError* – If a job associated with the new state point is already initialized.
- *OSError* – If the move failed due to an unknown system related error.

**sp**

Access the job’s state point as attribute dictionary.

**statepoint()**

The statepoint associated with this job.

**Returns**

The statepoint mapping.

**Return type**

**dict**

**update_statepoint** *(update, overwrite=False)*

Update the statepoint of this job.

**Warning:** While appending to a job’s state point is generally safe, modifying existing parameters may lead to data inconsistency. Use the overwrite argument with caution!

**Parameters**

- **update** *(mapping)* – A mapping used for the statepoint update.
- **overwrite** – Set to true, to ignore whether this update overwrites parameters, which are currently part of the job’s state point. Use with caution!

**Raises**

- *KeyError* – If the update contains keys, which are already part of the job’s state point and overwrite is False.
- *DestinationExistsError* – If a job associated with the new state point is already initialized.
- *OSError* – If the move failed due to an unknown system related error.
workspace()

Each job is associated with a unique workspace directory.

Returns The path to the job’s workspace directory.

Return type str

ws

The job’s workspace directory.

Project.build_job_search_index(index)

Build a job search index.

Parameters index (list) – A document index.

Returns A job search index based on the provided index.

Return type JobSearchIndex

Project.build_job_statepoint_index(exclude_const=False, index=None)

Build a statepoint index to identify jobs with specific parameters.

This method generates unordered key-value pairs, with complete statepoint paths as keys, encoded in JSON, and a set of job ids of all corresponding jobs, e.g.:

```python
>>> project.open_job({'a': 0, 'b': {'c': 'const'}}).init()
>>> project.open_job({'a': 1, 'b': {'c': 'const'}}).init()
>>> for k, v in project.job_statepoint_index():
...     print(k, v)
...     ...
["a", 1] {'b7568fa73881d27cbf24bf58d226d80e'}
["a", 0] {'54b61a7adbe004b30b39aa399d04f483'}
["b", "c", "abc"] {'b7568fa73881d27cbf24bf58d226d80e',
-->'54b61a7adbe004b30b39aa399d04f483'}
```

Parameters

- **exclude_const (bool)** – Exclude entries that are shared by all jobs that are part of the index.

- **index** – A document index.

Yields Key-value pairs of JSON-encoded statepoint parameters and a set of corresponding job ids.

Project.clone(job)

Clone job into this project.

Create an identical copy of job within this project.

Parameters job (Job) – The job to copy into this project.

Returns The job instance corresponding to the copied job.

Return type Job

Raises DestinationExistsError – In case that a job with the same id is already initialized within this project.

Project.config

The project’s configuration.

Project.create_access_module(filename=None, master=True)

Create the access module for indexing

This method generates the access module required to make this project’s index part of a master index.
Parameters

- **filename** *(str)* – The name of the access module file. Defaults to the standard name and should usually not be changed.
- **master** *(bool)* – If True, add directives for the compilation of a master index when executing the module.

Returns  The name of the created access module.

Return type  str

Project.create_linked_view *(prefix=None, job_ids=None, index=None)*

Create or update a persistent linked view of the selected data space.

This method determines unique paths for each job based on the job’s statepoint and creates symbolic links to the associated workspace directories. This is useful for browsing through the data space in a human-readable manner.

Assuming that the parameter space is

- **a=0, b=0**
- **a=1, b=0**
- **a=2, b=0**
- ...,

where *b* does not vary over all statepoints, this method will create the following *symbolic links* within the specified view prefix:

<table>
<thead>
<tr>
<th>View Path</th>
<th>Workspace Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>view/a/0/job</td>
<td>/path/to/workspace/7f9fb369851609ce9cb91404549393f3</td>
</tr>
<tr>
<td>view/a/1/job</td>
<td>/path/to/workspace/017d53deb17a290d8b0d2ae02fa8bd9d</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Note: To maximize the compactness of each view path, *b* which does not vary over the selected data space, is ignored.

Parameters

- **prefix** *(str)* – The path where the linked view will be created or updated.
- **job_ids** – If None (the default), create the view for the complete data space, otherwise only for the sub space constituted by the provided job ids.
- **index** – A document index.

Project.dump_statepoints *(statepoints)*

Dump the statepoints and associated job ids.

Equivalent to:

```python
{project.open_job(sp).get_id(): sp for sp in statepoints}
```

Parameters **statepoints** *(iterable)* – A list of statepoints.

Returns  A mapping, where the key is the job id and the value is the statepoint.

Return type  dict
Project.find_job_documents (filter=None)
Find all job documents in the project’s workspace.
This method iterates through all jobs or all jobs matching the filter and yields each job’s document as a dict. Each dict additionally contains a field ‘statepoint’, with the job’s statepoint and a field ‘_id’, which is the job’s id.

Parameters

- **filter (mapping)** – If not None, only find job documents matching filter.

Yields

- Instances of dict.

Raises

- **KeyError** – If the job document already contains the fields ‘_id’ or ‘statepoint’.

Project.find_job_ids (filter=None, doc_filter=None, index=None)
Find the job_ids of all jobs matching the filters.
The optional filter arguments must be a Mapping of key-value pairs and JSON serializable.

Note: Providing a pre-calculated index may vastly increase the performance of this function.

Parameters

- **filter (Mapping)** – A mapping of key-value pairs that all indexed job statepoints are compared against.

- **doc_filter** – A mapping of key-value pairs that all indexed job documents are compared against.

- **index** – A document index.

Yields

- The ids of all indexed jobs matching both filters.

Raises

- **TypeError** – If the filters are not JSON serializable.

- **ValueError** – If the filters are invalid.

- **RuntimeError** – If the filters are not supported by the index.

Project.find_jobs (filter=None, doc_filter=None, index=None)
Find all jobs in the project’s workspace.
The optional filter arguments must be a Mapping of key-value pairs and JSON serializable.

Note: Providing a pre-calculated index may vastly increase the performance of this function.

Parameters

- **filter (Mapping)** – A mapping of key-value pairs that all indexed job statepoints are compared against.

- **doc_filter** – A mapping of key-value pairs that all indexed job documents are compared against.

Yields

- Instances of Job

Raises

- **TypeError** – If the filters are not JSON serializable.

- **ValueError** – If the filters are invalid.
• **RuntimeError** – If the filters are not supported by the index.

**Project.find_statepoints** *(filter=None, doc_filter=None, index=None, skip_errors=False)*

Find all statepoints in the project’s workspace.

**Parameters**

- **filter** *(mapping)* – If not None, only yield statepoints matching the filter.
- **skip_errors** *(bool)* – Show, but otherwise ignore errors while iterating over the workspace. Use this argument to repair a corrupted workspace.

**Yields** statepoints as dict

**Project.get_id**

Get the project identifier.

**Returns** The project id.

**Return type** str

**Raises** **LookupError** – If no project id could be determined.

**classmethod Project.get_project** *(root=None)*

Find a project configuration and return the associated project.

**Parameters**

- **root** *(str)* – The project root directory. If no root directory is given, the next project found within or above the current working directory is returned.

**Returns** The project handle.

**Raises** **LookupError** – If no project configuration can be found.

**Project.get_statepoint** *(jobid, fn=None)*

Get the statepoint associated with a job id.

The statepoint is retrieved from the workspace or from the statepoints file if the former attempt fails.

**Parameters**

- **jobid** *(str)* – A job id to get the statepoint for.
- **fn** *(str)* – The filename of the file containing the statepoints, defaults to FN_STATEPOINTS.

**Returns** The statepoint.

**Return type** dict

**Raises** **KeyError** – If the statepoint associated with jobid could not be found.

See also **dump_statepoints()**.

**Project.index** *(formats=None, depth=0, skip_errors=False, include_job_document=True)*

Generate an index of the project’s workspace.

This generator function indexes every file in the project’s workspace until the specified depth. The job document if it exists, is always indexed, other files need to be specified with the formats argument.

```python
for doc in project.index({'.*\.txt', 'TextFile'}):
    print(doc)
```
• **skip_errors** *(bool)* – Skip all errors which occur during indexing. This is useful when trying to repair a broken workspace.

• **include_job_document** *(bool)* – Include the contents of job documents.

**Yields** index documents

```
classmethod Project.init_project(name, root=None, workspace=None, make_dir=True)
```

Initialize a project with the given name.

It is safe to call this function multiple times with the same arguments. However, a RuntimeError is raised in case where an existing project configuration would conflict with the provided initialization parameters.

**Parameters**

• **name** *(str)* – The name of the project to initialize.

• **root** *(str)* – The root directory for the project. Defaults to the current working directory.

• **workspace** *(str)* – The workspace directory for the project. Defaults to $project_root/workspace.

• **make_dir** *(bool)* – Create the project root directory, if it does not exist yet.

**Returns** The project handle of the initialized project.

**Return type** **Project**

**Raises** **RuntimeError** – If the project root path already contains a conflicting project configuration.

```
Project.min_len_unique_id()
```

Determine the minimum length required for an id to be unique.

```
Project.num_jobs()
```

Return the number of initialized jobs.

```
Project.open_job(statepoint=None, id=None)
```

Get a job handle associated with a statepoint.

This method returns the job instance associated with the given statepoint or job id. Opening a job by a valid statepoint never fails. Opening a job by id, requires a lookup of the statepoint from the job id, which may fail if the job was not previously initialized.

**Parameters**

• **statepoint** *(mapping)* – The job’s unique set of parameters.

• **id** *(str)* – The job id.

**Returns** The job instance.

**Return type** **Job**

**Raises**

• **KeyError** – If the attempt to open the job by id fails.

• **LookupError** – If the attempt to open the job by an abbreviated id returns more than one match.

```
Project.read_statepoints(fn=None)
```

Read all statepoints from a file.

**Parameters**

• **fn** *(str)* – The filename of the file containing the statepoints, defaults to FN_STATEPOINTS.
See also `dump_statepoints()` and `write_statepoints()`.

**Project.**`repair()`  
Attempt to repair the workspace after it got corrupted.

**Project.**`reset_statepoint (job, new_statepoint)`  
Reset the state point of job.

**Danger:** Use this function with caution! Resetting a job’s state point, may sometimes be necessary, but can possibly lead to incoherent data spaces.

**Parameters**

- **job (Job)** – The job, that should be reset to a new state point.
- **new_statepoint (mapping)** – The job’s new state point.

**Raises**

- **DestinationExistsError** – If a job associated with the new state point is already initialized.
- **OSError** – If the move failed due to an unknown system related error.

**Project.**`root_directory()`  
Returns the project’s root directory.

**Project.**`update_statepoint (job, update, overwrite=False)`  
Update the statepoint of this job.

**Warning:** While appending to a job’s state point is generally safe, modifying existing parameters may lead to data inconsistency. Use the overwrite argument with caution!

**Parameters**

- **job (Job)** – The job, whose statepoint shall be updated.
- **update (mapping)** – A mapping used for the statepoint update.
- **overwrite** – Set to true, to ignore whether this update overwrites parameters, which are currently part of the job’s state point. Use with caution!

**Raises**

- **KeyError** – If the update contains keys, which are already part of the job’s state point and overwrite is False.
- **DestinationExistsError** – If a job associated with the new state point is already initialized.
- **OSError** – If the move failed due to an unknown system related error.

**Project.**`workspace()`  
Returns the project’s workspace directory.

The workspace defaults to `project_root/workspace`. Configure this directory with the `workspace_dir` attribute. If the specified directory is a relative path, the absolute path is relative from the project’s root directory.
Note: The configuration will respect environment variables, such as $HOME.

**Project**. `write_statepoints` *(statepoints=None, fn=None, indent=2)*

Dump statepoints to a file.

If the file already contains statepoints, all new statepoints will be appended, while the old ones are preserved.

**Parameters**

- **statepoints** *(iterable)* — A list of statepoints, defaults to all statepoints which are defined in the workspace.
- **fn** *(str)* — The filename of the file containing the statepoints, defaults to FN_STATEPOINTS.
- **indent** *(int)* — Specify the indentation of the json file.

See also **dump_statepoints()**.

**signac.contrib.project**. `get_project` *(root=None)*

Find a project configuration and return the associated project.

**Parameters**

- **root** *(str)* — The project root directory. If no root directory is given, the next project found within or above the current working directory is returned.

**Returns** The project handle.

**Return type** *Project*

**Raises** **LookupError** — If no project configuration can be found.

**signac.contrib.project**. `init_project` *(name, root=None, workspace=None, make_dir=True)*

Initialize a project with the given name.

It is safe to call this function multiple times with the same arguments. However, a RuntimeError is raised in case where an existing project configuration would conflict with the provided initialization parameters.

**Parameters**

- **name** *(str)* — The name of the project to initialize.
- **root** *(str)* — The root directory for the project. Defaults to the current working directory.
- **workspace** *(str)* — The workspace directory for the project. Defaults to $project_root/workspace.
- **make_dir** *(bool)* — Create the project root directory, if it does not exist yet.

**Returns** The project handle of the initialized project.

**Return type** *Project*

**Raises** **RuntimeError** — If the project root path already contains a conflicting project configuration.

**signac.contrib.utility module**

**class** **signac.contrib.utility**. `EmptyIsTrue` *(option_strings, dest, nargs=None, const=None, default=..., type=..., choices=..., required=False, help=..., metavar=...)*

**Bases:** *argparse.Action*
class signac.contrib.utility.SmartFormatter (prog, indent_increment=2, max_help_position=24, width=None)

Bases: argparse.HelpFormatter
class signac.contrib.utility.VerbosityAction (option_strings, dest, nargs=None, const=None, default=None, type=None, choices=None, required=False, help=None, metavar=None)

Bases: argparse.Action
class signac.contrib.utility.VerbosityLoggingConfigAction (option_strings, dest, nargs=None, const=None, default=None, type=None, choices=None, required=False, help=None, metavar=None)

Bases: signac.contrib.utility.VerbosityAction

signac.contrib.utility.add_verbosity_action_argument (parser, default=0)
Add a verbosity argument to parser.

Note: The argument is ‘-v’. Add multiple ‘-v’ arguments, e.g. ‘-vv’ or ‘-vvv’ to increase the level of verbosity.

Args: parser: A argparse object. default: The default level, defaults to 0.

signac.contrib.utility.add_verbosity_argument (parser, default=0)
Add a verbosity argument to parser.

Note: The argument is ‘-v’ or ‘--verbosity’. Add multiple ‘-v’ arguments, e.g. ‘-vv’ or ‘-vvv’ to increase the level of verbosity.

Args: parser: A argparse object. default: The default level, defaults to 0.

signac.contrib.utility.is_string (s)

signac.contrib.utility.prompt_new_password (prompt='Password: ')

signac.contrib.utility.prompt_password (prompt='Password: ')

signac.contrib.utility.query_yes_no (question, default='yes')
Ask a yes/no question via input() and return their answer.

“question” is a string that is presented to the user. “default” is the presumed answer if the user just hits <Enter>. It must be “yes” (the default), “no” or None (meaning an answer is required of the user).

The “answer” return value is one of “yes” or “no”.

signac.contrib.utility.set_verbosity_level (verbosity, default=None, increment=10)
Set the verbosity level as a function of an integer level.

Args: verbosity: The verbosity level as integer. default: The default verbosity level, defaults to logging.ERROR.

signac.contrib.utility.walkdepth (path, depth=0)

signac.db package

Module contents

signac.db.get_database (name, hostname=None, config=None)
Get a database handle.
The database handle is an instance of `Database`, which provides access to the document collections within one database.

```python
db = signac.db.get_database('MyDatabase')
docs = db.my_collection.find()
```

Please note, that a collection which did not exist at the point of access, will automatically be created.

**Parameters**

- `name` *(str)* – The name of the database to get.
- `hostname` *(str)* – The name of the configured host. Defaults to the first configured host, or the host specified by `default_host`.
- `config` *(common.config.Config)* – The config object to retrieve the host configuration from. Defaults to the global configuration.

**Returns** The database handle.

**Return type** `pymongo.database.Database`

**See also:**

https://api.mongodb.org/python/current/api/pymongo/database.html

**Submodules**

`signac.db.database module`

`signac.db.database.get_database` *(name, hostname=None, config=None)*

Get a database handle.

The database handle is an instance of `Database`, which provides access to the document collections within one database.

```python
db = signac.db.get_database('MyDatabase')
docs = db.my_collection.find()
```

Please note, that a collection which did not exist at the point of access, will automatically be created.

**Parameters**

- `name` *(str)* – The name of the database to get.
- `hostname` *(str)* – The name of the configured host. Defaults to the first configured host, or the host specified by `default_host`.
- `config` *(common.config.Config)* – The config object to retrieve the host configuration from. Defaults to the global configuration.

**Returns** The database handle.

**Return type** `pymongo.database.Database`

**See also:**

https://api.mongodb.org/python/current/api/pymongo/database.html
signac.core package

Submodules

signac.core.jsondict module

Dict implementation with backend json file.

class signac.core.jsondict.JSONDict (filename, synchronized=False, write_concern=False)
    Bases: collections.UserDict
        clear()
        get(key, default=None)
        load()
        save()
        update(mapping)

signac.core.utility module

class signac.core.utility.Version(major=0, minor=0, change=0, postrelease='', prerelease='final')
    Bases: dict
    Utility class to manage revision control numbers.
    to_tuple()

signac.core.utility.get_subject_from_certificate(fn_certificate)

signac.core.utility.parse_version(version_str)
    Parse a version number into a version object.

signac.common package

Submodules

signac.common.config module

class signac.common.config.Config(infile=None, options=None, configspec=None, encoding=None, interpolation=True, raise_errors=False, list_values=True, create_empty=False, file_error=False, stringify=True, indent_type=None, default_encoding=None, unrepr=False, write_empty_values=False, _inspec=False)
    Bases: signac.common.configobj.ConfigObj
        encoding = ‘utf-8’
        has_password()
        verify (validator=None, *args, **kwargs)
        write (outfile=None, section=None)

exception signac.common.config.PermissionsError
    Bases: signac.common.errors.ConfigError
signac.common.config.check_and_fix_permissions(filename)
signac.common.config.check_permissions(filename)
signac.common.config.fix_permissions(filename)
signac.common.config.get_config(infile=None, configspec=None, *args, **kwargs)
signac.common.config.load_config(root=None, local=False)
signac.common.config.read_config_file(filename)
signac.common.config.search_standard_dirs()
signac.common.config.search_tree(root=None)

signac.common.connection module

class signac.common.connection.DBClientConnector(host_config, **kwargs)
  Bases: object
    authenticate()
    client
    config
    connect(host=None)
    host
    logout()

signac.common.connection.get_subject_from_certificate(fn_certificate)

signac.common.connection.raise_unsupported_auth_mechanism(mechanism)

signac.common.errors module

exception signac.common.errors.AuthenticationError
  Bases: signac.core.errors.Error, RuntimeError

exception signac.common.errors.ConfigError
  Bases: signac.core.errors.Error, RuntimeError

exception signac.common.errors.ExportError
  Bases: signac.core.errors.Error, RuntimeError

exception signac.common.errors.FetchError
  Bases: signac.common.errors.FileNotFoundException

exception signac.common.errors.FileNotFoundException
  Bases: signac.core.errors.Error, FileNotFoundException

signac.common.host module

signac.common.host.check_credentials(hostcfg)

signac.common.host.get_client(hostcfg, **kwargs)

signac.common.host.get_connector(hostcfg, **kwargs)
signac.common.host.get_credentials(hostcfg, ask=True)

signac.common.host.get_database(name, hostname=None, config=None, **kwargs)

signac.common.host.get_default_host(config=None)

signac.common.host.get_host_config(hostname=None, config=None)

signac.common.host.make_uri(hostcfg)

signac.common.validate module

signac.common.validate.get_validator()

signac.common.validate.mongodb_uri(value, *args, **kwargs)

signac.common.validate.password(value, *args, **kwargs)

signac.common.validate.version(value, *args, **kwargs)

Overview  An illustration of signac’s data model.

Installation  Instructions on how to install the signac package.

Tutorial  Compact demonstration of the implementation of a computational workflow.

Reference  A complete reference to the framework’s major components.

Quick Reference  Brief overview of the core functions, to serve as reference.

How to cite signac  Instructions on how to acknowledge this software in publications.

API  The complete API reference.
Support and Contribution

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

The signac package is hosted on bitbucket and licensed under the open-source BSD 3-Clause license. Please use the repository’s issue tracker to report bugs or request new features.
CHAPTER 3

Indices and tables

- genindex
- modindex
- search
S

signac, 52
signac.cite, 72
signac.common.config, 121
signac.common.connection, 122
signac.common.errors, 122
signac.common.host, 122
signac.common.validate, 123
signac.contrib.errors, 104
signac.contrib.filesystems, 104
signac.contrib.formats, 106
signac.contrib.hashing, 106
signac.contrib.mpipool, 108
signac.contrib.utility, 118
signac.core.jsondict, 121
signac.core.utility, 121
signac.db.database, 120
Index

A
acknowledge, 123
add_verbosity_action_argument() (in module signac.contrib.utility), 119
add_verbosity_argument() (in module signac.contrib.utility), 119
api, 123
authenticate() (signac.common.connection.DBClientConnector method), 122
AuthenticationError, 122
AutoRetry (signac.contrib.filesystems.GridFS attribute), 104

B
bcast() (signac.contrib.mpipool.MPIPool method), 109
bibtex() (in module signac.cite), 72
build_job_search_index() (signac.Project method), 54
build_job_statepoint_index() (signac.Project method), 54

C
calc_id() (in module signac.contrib.hashing), 106
clear_access_module() (signac.Project method), 55
clear() (signac.core.jsondict.JSonDict method), 121
clear() (signac.contrib.mpipool.MPIPool method), 109
close() (signac.contrib.mpipool.MPIPool method), 109
close() (signac.Project method), 52
Collection (class in signac), 63
compute_file_id() (signac.RegexFileCrawler class method), 69
Config (class in signac.common.config), 121
config (signac.common.connection.DBClientConnector attribute), 122
config (signac.Project attribute), 55
cconfig() (signac.contrib.filesystems.GridFS method), 104
cconfig() (signac.contrib.filesystems.LocalFS method), 105
ConfigError, 122
crawl() (signac.RegexFileCrawler method), 69
crawl() (signac.SignacProjectCrawler method), 71
create_access_module() (signac.Project method), 55
create_linked_view() (signac.Project method), 55

d
DBClientConnector (class in signac.common.connection), 122
define() (signac.RegexFileCrawler class method), 69
definitions (signac.RegexFileCrawler attribute), 69
delete_many() (signac.Collection method), 64
delete_one() (signac.Collection method), 64
DestinationExistsError, 104
docs_from_file() (signac.MasterCrawler method), 71
docs_from_file() (signac.RegexFileCrawler method), 69
document (signac.Project.Job attribute), 52
dump() (signac.Collection method), 64
dump_statepoints() (signac.Project method), 56

e
EmptyIsTrue (class in signac.contrib.utility), 118
encoding (signac.common.config.Config attribute), 121
encoding (signac.SignacProjectCrawler attribute), 71
export() (in module signac), 62
export_one() (in module signac), 62
export_pymongo() (in module signac), 67
export_to_mirror() (in module signac), 63
ExportError, 122

f
fetch() (in module signac), 61
fetch() (signac.RegexFileCrawler method), 70
fetch_one() (in module signac), 62
FetchError, 122
FileExistsError (signac.contrib.filesystems.GridFS attribute), 104
FileExistsError (signac.contrib.filesystems.LocalFS attribute), 105
FileNotFoundError, 122
FileNotFoundError (signac.contrib.filesystems.GridFS attribute), 104
FileNotFoundError (signac.contrib.filesystems.LocalFS attribute), 105
filesystems_from_config() (in module signac.contrib.filesystems), 106
filesystems_from_configs() (in module signac.contrib.filesystems), 106
find() (signac.Collection method), 64
find_job_documents() (signac.Project method), 56
find_job_ids() (signac.Project method), 56
find_jobs() (signac.Project method), 57
find_one() (signac.Collection method), 65
find_statepoints() (signac.Project method), 57
fix_permissions() (in module signac.common.config), 122
flush() (signac.Collection method), 65
fn() (signac.Project.Job method), 53
FN_ACCESS_MODULE (signac.MasterCrawler attribute), 71
FN_DOCUMENT (signac.Project.Job attribute), 52
fn_job_document (signac.SignacProjectCrawler attribute), 71
FN_MANIFEST (signac.Project.Job attribute), 52
fn_statepoint (signac.SignacProjectCrawler attribute), 71
G
get() (signac.contrib.filesystems.GridFS method), 104
get() (signac.contrib.filesystems.LocalFS method), 105
get() (signac.core.jsondict.JSonDict method), 121
get_client() (in module signac.common.host), 122
get_config() (in module signac.common.config), 122
get_connector() (in module signac.common.config), 122
get_credentials() (in module signac.common.config), 122
get_database() (in module signac), 61
get_database() (in module signac.common.host), 122
get_database() (in module signac.db.database), 120
get_default_host() (in module signac.common.host), 123
get_host_config() (in module signac.common.host), 123
get_id() (signac.Project method), 57
get_id() (signac.Project.Job method), 53
get_project() (in module signac), 60
get_project() (signac.Project class method), 57
get_statepoint() (signac.Project method), 58
get_statepoint() (signac.SignacProjectCrawler method), 71
get_subject_from_certificate() (in module signac.common.connection), 122
get_subject_from_certificate() (in module signac.core.utility), 121
get_validator() (in module signac.common.validate), 123
GridFS (class in signac.contrib.filesystems), 104
gridfs (signac.contrib.filesystems.GridFS attribute), 105
H
has_password() (signac.common.config.Config method), 121
host (signac.common.connection.DBClientConnector attribute), 122
I
ids (signac.Collection attribute), 65
index() (in module signac), 68
index() (signac.Collection method), 65
index() (signac.Project method), 58
index_files() (in module signac), 67
init() (signac.Project.Job method), 53
init_project() (in module signac), 61
init_project() (signac.Project class method), 58
insert_one() (signac.Collection method), 66
installation, 123
is_master() (signac.contrib.mpipool.MPIPool method), 109
is_string() (in module signac.contrib.utility), 119
isfile() (signac.Project.Job method), 53
J
JSonDict (class in signac.core.jsondict), 121
L
load() (signac.core.jsondict.JSonDict method), 121
load_config() (in module signac.common.config), 122
LocalFS (class in signac.contrib.filesystems), 105
LocalFS.AutoRetry, 105
logout() (signac.common.connection.DBClientConnector method), 122
M
main() (signac.Collection method), 66
make_uri() (in module signac.common.host), 123
map() (signac.contrib.mpipool.MPIPool method), 109
MasterCrawler (class in signac), 70
min_len_unique_id() (signac.Project method), 59
MongoDB database backend, 4
mongodb_uri() (in module signac.common.validate), 123
move() (signac.Project.Job method), 53
MPIPool (class in signac.contrib.mpipool), 108
N
name (signac.contrib.filesystems.GridFS attribute), 105
Index

A
name (signac.contrib.filesystems.LocalFS attribute), 105
new_file() (signac.contrib.filesystems.GridFS method), 105
new_file() (signac.contrib.filesystems.LocalFS method), 105
num_jobs() (signac.Project method), 59

O
open() (signac.Collection class method), 66
open() (signac.Project.Job method), 53
open_job() (signac.Project.method), 59
overview, 123

P
parse_version() (in module signac.core.utility), 121
password() (in module signac.common.validate), 123
PermissionsError, 121
primary_key (signac.Collection attribute), 67
process() (signac.RegexFileCrawler method), 70
process() (signac.SignacProjectCrawler method), 71
Project (class in signac), 52
Project.Job (class in signac), 52
prompt_new_password() (in module signac.contrib.utility), 119
prompt_password() (in module signac.contrib.utility), 119

Q
query_yes_no() (in module signac.contrib.utility), 119
quickreference, 123

R
raise_unsupported_auth_mechanism() (in module signac.common.connection), 122
read_config_file() (in module signac.common.config), 122
read_statepoints() (signac.Project method), 59
reference, 123
reference() (in module signac.cite), 72
RegexFileCrawler (class in signac), 69
remove() (signac.Project.Job method), 53
repair() (signac.Project method), 59
replace_one() (signac.Collection method), 67
reset_statepoint() (signac.Project method), 59
reset_statepoint() (signac.Project.Job method), 53
root_directory() (signac.Project method), 59

S
save() (signac.core.jsondict.JsonDict method), 121
search_standard_dirs() (in module signac.common.config), 122
search_tree() (in module signac.common.config), 122
set_verbosity_level() (in module signac.contrib.utility), 119
signac (module), 1, 52
signac.cite (module), 72
signac.common.config (module), 121
signac.common.connection (module), 122
signac.common.errors (module), 122
signac.common.host (module), 122
signac.common.validate (module), 123
signac.contrib.errors (module), 104
signac.contrib.filesystems (module), 104
signac.contrib.forms (module), 106
signac.contrib.hashing (module), 106
signac.contrib.mpipool (module), 108
signac.contrib.utility (module), 118
signac.core.jsondict (module), 121
signac.core.utility (module), 121
signac.db.database (module), 120
signac_id_alias (signac.SignacProjectCrawler attribute), 72
SignacProjectCrawler (class in signac), 71
SmartFormatter (class in signac.contrib.utility), 118
sp (signac.Project.Job attribute), 54
statepoint() (signac.Project.Job method), 54
statepoint_index (signac.SignacProjectCrawler attribute), 72

to_tuple() (signac.core.utility.Version method), 121
tutorial, 123

U
update() (signac.Collection method), 67
update() (signac.core.jsondict.JsonDict method), 121
update_statepoint() (signac.Project method), 60
update_statepoint() (signac.Project.Job method), 54

V
VerbosityAction (class in signac.contrib.utility), 119
VerbosityLoggingConfigAction (class in signac.contrib.utility), 119
verify() (signac.common.config.Config method), 121
Version (class in signac.core.utility), 121
version() (in module signac.common.validate), 123

W
wait() (signac.contrib.mpipool.MPIPool method), 109
walkdepth() (in module signac.common.config), 122
workspace() (signac.Project method), 60
workspace() (signac.Project.Job method), 54
write() (signac.common.config.Config method), 121
write_statepoints() (signac.Project method), 60
ws (signac.Project.Job attribute), 54