signac Documentation

Release 1.1.0

Carl Simon Adorf

May 19, 2019

Contents

Note: This is documentation for the **signac** *core package*, which is part of the signac framework. See here for a comprehensive introduction to the **signac** *framework*.

CHAPTER 1

Contents

1.1 Installation

The recommended installation method for **signac** is via conda or pip. The software is tested for Python versions 2.7.x and 3.4+ 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.

1.1.1 Install with conda

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

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

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

```
$ conda update signac
```

1.1.2 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 installing optional dependencies:

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

1.1.3 Source Code Installation

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

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

Consider installing *optional dependencies*.

1.1.4 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`

1.2 API Reference

This is the API for the **signac** (core) application.

1.2.1 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`.

```
usage: signac [-h] [--debug] [--version] [-v] [-y]
              {init,project,job,statepoint,document,rm,move,clone,index,find,view,
↪schema,shell,sync,import,export,update-cache,config}
              ...

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

positional arguments:
  {init,project,job,statepoint,document,rm,move,clone,index,find,view,schema,shell,
↪sync,import,export,update-cache,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.
  -v, --verbosity       Set level of verbosity.
  -y, --yes             Answer all questions with yes. Useful for scripted
                        interaction.
```

1.2.2 The Project

class signac.**Project**(*config=None*)

The handle on a signac project.

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

Attributes

Project.build_job_search_index(index[, _trust])	Build a job search index.
Project.build_job_statepoint_index([...])	Build a statepoint index to identify jobs with specific parameters.
Project.check([job_ids])	Check the project's workspace for corruption.
Project.clone(job[, copytree])	Clone job into this project.
Project.config	The project's configuration.
Project.create_access_module([filename, master])	Create the access module for indexing
Project.create_linked_view([prefix, ...])	Create or update a persistent linked view of the selected data space.
Project.detect_schema([exclude_const, ...])	Detect the project's state point schema.
Project.data	The data associated with this project.
Project.doc	The document associated with this project.
Project.document	The document associated with this project.
Project.dump_statepoints(statepoints)	Dump the statepoints and associated job ids.
Project.export_to(target[, path, copytree])	Export all jobs to a target location, such as a directory or a (compressed) archive file.
Project.find_job_ids([filter, doc_filter, index])	Find the job_ids of all jobs matching the filters.
Project.find_jobs([filter, doc_filter])	Find all jobs in the project's workspace.
Project.fn(filename)	Prepend a filename with the project's root directory path.
Project.get_id()	Get the project identifier.
Project.get_statepoint(jobid[, fn])	Get the statepoint associated with a job id.
Project.groupby([key, default])	Groups jobs according to one or more statepoint parameters.
Project.groupbydoc([key, default])	Groups jobs according to one or more document values.
Project.import_from([origin, schema, sync, ...])	Import the data space located at origin into this project.
Project.index([formats, depth, skip_errors, ...])	Generate an index of the project's workspace.
Project.isfile(filename)	True if a file with filename exists in the project's root directory.
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, id])	Get a job handle associated with a statepoint.
Project.read_statepoints([fn])	Read all statepoints from a file.
Project.repair([fn_statepoints, index, job_ids])	Attempt to repair the workspace after it got corrupted.
Project.reset_statepoint(job, new_statepoint)	Reset the state point of job.
Project.root_directory()	Returns the project's root directory.

Continued on next page

Table 1 – continued from previous page

Project.sync(other[, strategy, exclude, ...])	Synchronize this project with the other project.
Project.update_cache()	Update the persistent state point cache.
Project.update_statepoint(job, update[, ...])	Update the statepoint of this job.
Project.workspace()	Returns the project's workspace directory.
Project.write_statepoints([statepoints, fn, ...])	Dump statepoints to a file.

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.

FN_CACHE = '.signac_sp_cache.json.gz'
The default filename for the state point cache file.

FN_DOCUMENT = 'signac_project_document.json'
The project's document filename.

FN_STATEPOINTS = 'signac_statepoints.json'
The default filename to read from and write statepoints to.

KEY_DATA = 'signac_data'
The project's datastore key.

build_job_search_index(*index*, *_trust=False*)
Build a job search index.

Parameters **index** (*list*) – A document index.

Returns A job search index based on the provided index.

Return type JobSearchIndex

build_job_statepoint_index(*exclude_const=False*, *index=None*)
Build a statepoint index to identify jobs with specific parameters.

This method generates pairs of state point keys and mappings of values to a set of all corresponding job ids. The pairs are ordered by the number of different values. Since state point keys may be nested, they are represented as a tuple. For example:

```
>>> for i in range(4):
...     project.open_job({'a': i, 'b': {'c': i % 2}}).init()
...
>>> for key, value in project.build_job_statepoint_index():
...     print(key)
...     pprint.pprint(value)
...
('b', 'c')
defaultdict(<class 'set'>,
            {0: {'3a530c13bfaf57517b4e81ecab6aec7f',
                 '4e9a45a922eae6bb5d144b36d82526e4'},
             1: {'d49c6609da84251ab096654971115d0c',
                 '5c2658722218d48a5eb1e0ef7c26240b'}})
('a',)
defaultdict(<class 'set'>,
```

(continues on next page)

(continued from previous page)

```
            {0: {'4e9a45a922eae6bb5d144b36d82526e4'},
             1: {'d49c6609da84251ab096654971115d0c'},
             2: {'3a530c13bfaf57517b4e81ecab6aec7f'},
             3: {'5c2658722218d48a5eb1e0ef7c26240b'}})
```

Values that are constant over the complete data space can be optionally ignored with the exclude_const argument set to True.

Parameters

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

Yields Pairs of state point keys and mappings of values to a set of all corresponding job ids.

check (*job_ids=None*)
Check the project's workspace for corruption.

Parameters **job_ids** – The ids of jobs to check, defaults to all jobs.

Raises ***JobsCorruptedError*** – When one or more jobs are identified as corrupted.

clone (*job*, *copytree=<function copytree>*)
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.

config
The project's configuration.

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

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

Similar to *export_to()*, this function expands the data space for the selected jobs, but instead of copying data will create symbolic links to the individual job workspace directories. This is primarily useful for browsing through the data space using a file-browser with human-interpretable directory paths.

By default, the paths of the view will be based on variable state point keys as part of the *implicit* schema of the selected jobs that we create the view for. For example, creating a linked view for a data space with schema

```
>>> print(project.detect_schema())
{
 'foo': 'int([0, 1, 2, ..., 8, 9], 10)',
}
```

by calling `project.create_linked_view('my_view')` will look similar to:

```
my_view/foo/0/job -> workspace/b8fcc6b8f99c56509eb65568922e88b8
my_view/foo/1/job -> workspace/b6cd26b873ae3624653c9268deff4485
...
```

It is possible to control the paths using the `path` argument, which behaves in the exact same manner as the equivalent argument for *`export_to()`*.

Note: The behavior of this function is almost equivalent to `project.export_to('my_view', copytree=os.symlink)` with the major difference, that view hierarchies are actually *updated*, that means no longer valid links are automatically removed.

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.
- **`path`** – The path (function) used to structure the linked data space.

Returns A dict that maps the source directory paths, to the linked directory paths.

`data`

The data associated with this project.

Equivalent to:

return project.store[‘signac_data’]

Returns An HDF5-backed datastore.

Return type `H5Store`

`detect_schema`(*exclude_const=False*, *subset=None*, *index=None*)

Detect the project’s state point schema.

Parameters

- **`exclude_const`** (*`bool`*) – Exclude all state point keys that are shared by all jobs within this project.
- **`subset`** – A sequence of jobs or job ids specifying a subset over which the state point schema should be detected.
- **`index`** – A document index.

Returns The detected project schema.

Return type *signac.contrib.schema.ProjectSchema*

doc
The document associated with this project.

Alias for *document*.

Returns The project document handle.

Return type *JSONDict*

document
The document associated with this project.

Returns The project document handle.

Return type *JSONDict*

dump_statepoints(*statepoints*)
Dump the statepoints and associated job ids.

Equivalent to:

```
{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

export_to(*target*, *path=None*, *copytree=None*)
Export all jobs to a target location, such as a directory or a (compressed) archive file.

Use this function in combination with *find_jobs()* to export only a select number of jobs, for example:

```
project.find_jobs({'foo': 0}).export_to('foo_0.tar')
```

The `path` argument enables users to control how exactly the exported data space is to be expanded. By default, the path-function will be based on the *implicit* schema of the exported jobs. For example, exporting jobs that all differ by a state point key *foo* with `project.export_to('data/')`, the exported directory structure could look like this:

```
data/foo/0
data/foo/1
...
```

That would be equivalent to specifying `path=lambda job: os.path.join('foo', job.sp.foo)`.

Instead of a function, we can also provide a string, where fields for state point keys are automatically formatted. For example, the following two path arguments are equivalent: "foo/{foo}" and "foo/{job.sp.foo}".

Any attribute of job can be used as a field here, so `job.doc.bar`, `job._id`, and `job.ws` can also be used as path fields.

A special `{{auto}}` field allows us to expand the path automatically with state point keys that have not been specified explicitly. So, for example, one can provide `path="foo/{foo}/{{auto}}"` to specify that the path shall begin with `foo/{foo}/`, but is then automatically expanded with all other state point key-value pairs. How key-value pairs are concatenated can be controlled *via* the format-specifier, so for example, `path="{{auto:_}}"` will generate a structure such as

```
data/foo_0
data/foo_1
...
```

Finally, providing `path=False` is equivalent to `path="{job._id}"`.

See also:

Previously exported or non-signac data spaces can be imported with *import_from()*.

Parameters

- **target** – A path to a directory to export to. The target can not already exist. Besides directories, possible targets are tar files (*.tar*), gzipped tar files (*.tar.gz*), zip files (*.zip*), bzip2-compressed files (*.bz2*), and xz-compressed files (*.xz*).
- **path** – The path (function) used to structure the exported data space. This argument must either be a callable which returns a path (str) as a function of *job*, a string where fields are replaced using the job-state point dictionary, or *False*, which means that we just use the job-id as path. Defaults to the equivalent of `{{auto}}`.
- **copytree** – The function used for the actual copying of directory tree structures. Defaults to `shutil.copytree()`. Can only be used when the target is a directory.

Returns A dict that maps the source directory paths, to the target directory paths.

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.

find_jobs (*filter=None*, *doc_filter=None*)

Find all jobs in the project's workspace.

The optional filter arguments must be a Mapping of key-value pairs and JSON serializable. The *filter* argument is used to search against job statepoints, whereas the *doc_filter* argument compares against job document keys.

Parameters

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

fn(*filename*)

Prepend a filename with the project's root directory path.

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

Returns The joined path of project root directory and filename.

get_id()

Get the project identifier.

Returns The project id.

Return type str

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

classmethod get_job(*root=None*)

Find a Job in or above the current working directory (or provided path).

Parameters **root** (*str*) – The job root directory. If no root directory is given, the current working directory is assumed to be the job directory.

Returns The job handle.

Raises **LookupError** – If this job cannot be found.

classmethod get_project(*root=None*, *search=True*)

Find a project configuration and return the associated project.

Parameters

- **root** (*str*) – The starting point to search for a project, defaults to the current working directory.
- **search** (*bool*) – If True, search for project configurations inside and above the specified root directory, otherwise only return projects with a root directory identical to the specified root argument.

Returns The project handle.

Raises **LookupError** – If no project configuration can be found.

get_statepoint(*jobid*, *fn=None*)

Get the statepoint associated with a job id.

The state point is retrieved from the internal cache, from the workspace or from a state points file.

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 state point corresponding to jobid.

Return type dict

Raises

- **KeyError** – If the state point associated with jobid could not be found.
- ***JobsCorruptedError*** – If the state point manifest file corresponding to jobid is inaccessible or corrupted.

groupby(*key=None*, *default=None*)

Groups jobs according to one or more statepoint parameters. This method can be called on any JobsCursor such as the one returned by *find_jobs()* or by iterating over a project. Examples:

```
# Group jobs by statepoint parameter 'a'.
for key, group in project.groupby('a'):
    print(key, list(group))

# Find jobs where job.sp['a'] is 1 and group them
# by job.sp['b'] and job.sp['c'].
for key, group in project.find_jobs({'a': 1}).groupby(('b', 'c')):
    print(key, list(group))

# Group by job.sp['d'] and job.document['count'] using a lambda.
for key, group in project.groupby(
    lambda job: (job.sp['d'], job.document['count'])
):
    print(key, list(group))
```

If *key* is None, jobs are grouped by identity (by id), placing one job into each group.

Parameters

- **key** (*str, iterable, or function*) – The statepoint grouping parameter(s) passed as a string, iterable of strings, or a function that will be passed one argument, the job.
- **default** – A default value to be used when a given state point key is not present (must be sortable).

groupbydoc(*key=None*, *default=None*)

Groups jobs according to one or more document values. This method can be called on any JobsCursor such as the one returned by *find_jobs()* or by iterating over a project. Examples:

```
# Group jobs by document value 'a'.
for key, group in project.groupbydoc('a'):
    print(key, list(group))

# Find jobs where job.sp['a'] is 1 and group them
# by job.document['b'] and job.document['c'].
for key, group in project.find_jobs({'a': 1}).groupbydoc(('b', 'c')):
    print(key, list(group))

# Group by whether 'd' is a field in the job.document using a lambda.
for key, group in project.groupbydoc(lambda doc: 'd' in doc):
    print(key, list(group))
```

If *key* is None, jobs are grouped by identity (by id), placing one job into each group.

Parameters

- **key** (*str, iterable, or function*) – The statepoint grouping parameter(s) passed as a string, iterable of strings, or a function that will be passed one argument, `Job.document`.
- **default** – A default value to be used when a given state point key is not present (must be sortable).

import_from(*origin=None*, *schema=None*, *sync=None*, *copytree=None*)

Import the data space located at origin into this project.

This function will walk through the data space located at origin and will try to identify data space paths that can be imported as a job workspace into this project.

The `schema` argument expects a function that takes a path argument and returns a state point dictionary. A default function is used when no argument is provided. The default schema function will simply look for state point manifest files–usually named `signac_statepoint.json`–and then import all data located within that path into the job workspace corresponding to the state point specified in the manifest file.

Alternatively the schema argument may be a string, that is converted into a schema function, for example: Providing `foo/{foo:int}` as schema argument means that all directories under `foo/` will be imported and their names will be interpeted as the value for `foo` within the state point.

Tip: Use `copytree=os.rename` or `copytree=shutil.move` to move dataspaces on import instead of copying them.

Warning: Imports can fail due to conflicts. Moving data instead of copying may therefore lead to inconsistent states and users are advised to apply caution.

See also:

Export the project data space with *export_to()*.

Parameters

- **origin** – The path to the data space origin, which is to be imported. This may be a path to a directory, a zip file, or a tarball archive.
- **schema** – An optional schema function, which is either a string or a function that accepts a path as its first and only argument and returns the corresponding state point as dict.
- **copytree** – Specify which exact function to use for the actual copytree operation. Defaults to `shutil.copytree()`.

Returns A dict that maps the source directory paths, to the target directory paths.

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.

```
for doc in project.index({r'.*\.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 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.

isfile(*filename*)
True if a file with filename exists in the project's root directory.

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

Returns True if a file with filename exists in the project's root directory.

Return type bool

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

num_jobs()
Return the number of initialized jobs.

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.

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()*.

repair(*fn_statepoints=None*, *index=None*, *job_ids=None*)
Attempt to repair the workspace after it got corrupted.

This method will attempt to repair lost or corrupted job state point manifest files using a state points file or a document index or both.

Parameters

- **fn_statepoints** (*str*) – The filename of the file containing the statepoints, defaults to FN_STATEPOINTS.
- **index** – A document index
- **job_ids** – An iterable of job ids that should get repaired. Defaults to all jobs.

Raises ***JobsCorruptedError*** – When one or more corrupted job could not be repaired.

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.

root_directory()
Returns the project's root directory.

stores
Access HDF5-stores associated with this project.

Use this property to access an HDF5 file within the project's root directory using the H5Store dict-like interface.

This is an example for accessing an HDF5 file called 'my_data.h5' within the project's root directory:

project.stores['my_data']['array'] = np.random((32, 4))

This is equivalent to:

H5Store(project.fn('my_data.h5'))['array'] = np.random((32, 4))

Both the *project.stores* and the *H5Store* itself support attribute access. The above example could therefore also be expressed as

project.stores.my_data.array = np.random((32, 4))

Returns The HDF5-Store manager for this project.

Return type :class:'~..core.h5store.H5StoreManager

sync(*other*, *strategy=None*, *exclude=None*, *doc_sync=None*, *selection=None*, ***kwargs*)

Synchronize this project with the other project.

Try to clone all jobs from the other project to this project. If a job is already part of this project, try to synchronize the job using the optionally specified strategies.

Parameters

- **other** (*Project*) – The other project to synchronize this project with.
- **strategy** – A file synchronization strategy.
- **exclude** – Files with names matching the given pattern will be excluded from the synchronization.
- **doc_sync** – The function applied for synchronizing documents.
- **selection** – Only sync the given jobs.
- **kwargs** – This method accepts the same keyword arguments as the *sync_projects()* function.

Raises

- ***DocumentSyncConflict*** – If there are conflicting keys within the project or job documents that cannot be resolved with the given strategy or if there is no strategy provided.
- ***FileSyncConflict*** – If there are differing files that cannot be resolved with the given strategy or if no strategy is provided.
- **SyncSchemaConflict** – In case that the check_schema argument is True and the detected state point schema of this and the other project differ.

temporary_project(*name=None*, *dir=None*)

Context manager for the initialization of a temporary project.

The temporary project is by default created within the root project's workspace to ensure that they share the same file system. This is an example for how this method can be used for the import and synchronization of external data spaces.

```
with project.temporary_project() as tmp_project:
    tmp_project.import_from('/data')
    project.sync(tmp_project)
```

Parameters

- **name** – An optional name for the temporary project. Defaults to a unique random string.
- **dir** – Optionally specify where the temporary project root directory is to be created. Defaults to the project's workspace directory.

Returns An instance of *Project*.

to_dataframe(**args*, ***kwargs*)
Export the project metadata to a pandas dataframe.

The arguments to this function are forwarded to `JobsCursor.to_dataframe()`.

update_cache()
Update the persistent state point cache.

This function updates a persistent state point cache, which is stored in the project root directory. Most data space operations, including iteration and filtering or selection are expected to be significantly faster after calling this function, especially for large data spaces.

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.

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.

---

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()*.

1.2.3 The Job class

class signac.contrib.job.**Job**(*project*, *statepoint*, *_id=None*)

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.

Attributes

Job.clear()	Remove all job data, but not the job itself.
Job.close()	Close the job and switch to the previous working directory.
Job.data	The data store associated with this job.
Job.doc	Alias for *Job.document*.
Job.document	The document associated with this job.
Job.fn(filename)	Prepend a filename with the job's workspace directory path.
Job.get_id()	The unique identifier for the job's statepoint.
Job.init([force])	Initialize the job's workspace directory.
Job.isfile(filename)	Return True if file exists in the job's workspace.
Job.move(project)	Move this job to project.
Job.open()	Enter the job's workspace directory.
Job.remove()	Remove the job's workspace including the job document.
Job.reset()	Remove all job data, but not the job itself.
Job.reset_statepoint(new_statepoint)	Reset the state point of this job.
Job.sp	Alias for *Job.statepoint*.
Job.statepoint	Access the job's state point as attribute dictionary.
Job.stores	Access HDF5-stores associated with this job.
Job.sync(other[, strategy, exclude, doc_sync])	Perform a one-way synchronization of this job with the other job.
Job.update_statepoint(update[, overwrite])	Update the statepoint of this job.
Job.workspace()	Each job is associated with a unique workspace directory.
Job.ws	Alias for *Job.workspace*.

class signac.contrib.job.**Job**(*project*, *statepoint*, *_id=None*)

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'

The job's document filename.

FN_MANIFEST = 'signac_statepoint.json'

The job's manifest filename.

The job manifest, this means a human-readable dump of the job's statepoint is stored in each workspace directory.

KEY_DATA = 'signac_data'

clear()
Remove all job data, but not the job itself.

This function will do nothing if the job was not previously initialized.

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

data
The data store associated with this job.

Equivalent to:

```
return job.stores['signac_data']
```

Returns An HDF5-backed datastore.

Return type *H5Store*

doc
Alias for *Job.document*.

> **Warning:** If you need a deep copy that will not modify the underlying persistent JSON file, use *job.document()* instead of *job.doc*. For more information, see *Job.statepoint* or *JSONDict*.

document
The document associated with this job.

> **Warning:** If you need a deep copy that will not modify the underlying persistent JSON file, use *job.document()* instead of *job.doc*. For more information, see *Job.statepoint* or *JSONDict*.

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(*force=False*)
Initialize the job's workspace directory.

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

Returns the calling job.

Parameters **force** (*bool*) – Overwrite any existing state point's manifest files, e.g., to repair them when they got corrupted.

Returns The job handle.

Return type *Job*

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.
- **RuntimeError** – If the job is not initialized or the destination is on a different device.
- **OSError** – When the move failed due unexpected file system issues.

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()

Remove all job data, but not the job itself.

This function will initialize the job if it was not previously initialized.

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
Alias for *Job.statepoint*.

> **Warning:** As with *Job.statepoint*, use *job.sp()* instead of *job.sp* if you need a deep copy that will not modify the underlying persistent JSON file.

statepoint
Access the job's state point as attribute dictionary.

> **Warning:** The statepoint object behaves like a dictionary in most cases, but because it persists changes to the filesystem, making a copy requires explicitly converting it to a dict. If you need a modifiable copy that will not modify the underlying JSON file, you can access a dict copy of the statepoint by calling it, e.g. *sp_dict = job.statepoint()* instead of *sp = job.statepoint*. For more information, see *JSONDict*.

stores
Access HDF5-stores associated with this job.

Use this property to access an HDF5 file within the job's workspace directory using the *H5Store* dict-like interface.

This is an example for accessing an HDF5 file called 'my_data.h5' within the job's workspace:

```
job.stores['my_data']['array'] = np.random((32, 4))
```

This is equivalent to:

```
H5Store(job.fn('my_data.h5'))['array'] = np.random((32, 4))
```

Both the *job.stores* and the *H5Store* itself support attribute access. The above example could therefore also be expressed as

```
job.stores.my_data.array = np.random((32, 4))
```

Returns The HDF5-Store manager for this job.

Return type *H5StoreManager*

sync(*other*, *strategy=None*, *exclude=None*, *doc_sync=None*, ***kwargs*)
Perform a one-way synchronization of this job with the other job.

By default, this method will synchronize all files and document data with the other job to this job until a synchronization conflict occurs. There are two different kinds of synchronization conflicts:

1. The two jobs have files with the same, but different content.
2. The two jobs have documents that share keys, but those keys are associated with different values.

A file conflict can be resolved by providing a 'FileSync' *strategy* or by *excluding* files from the synchronization. An unresolvable conflict is indicated with the raise of a *FileSyncConflict* exception.

A document synchronization conflict can be resolved by providing a doc_sync function that takes the source and the destination document as first and second argument.

Parameters

- **other** (*.Job*) – The other job to synchronize from.
- **strategy** – A synchronization strategy for file conflicts. If no strategy is provided, a *SyncConflict* exception will be raised upon conflict.
- **exclude** (*str*) – An filename exclude pattern. All files matching this pattern will be excluded from synchronization.
- **doc_sync** – A synchronization strategy for document keys. If this argument is None, by default no keys will be synchronized upon conflict.
- **dry_run** – If True, do not actually perform the synchronization.
- **kwargs** – Extra keyword arguments will be forward to the *sync_jobs()* function which actually excutes the synchronization operation.

Raises ***FileSyncConflict*** – In case that a file synchronization results in a conflict.

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

Alias for *Job.workspace*.

1.2.4 The Collection

class signac.**Collection**(*docs=None*, *primary_key='_id'*, *compresslevel=0*, *_trust=False*)

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:

```
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:

```
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 *Collection.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*.
- **compresslevel** – The level of compression to use. Any positive value implies compression and is used by the underlying gzip implementation. Default value is 0 (no compression).

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:

```
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:

```
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
```

Arithmetic Operators

- *$eq*: equal
- *$neq*: not equal
- *$gt*: greater than
- *$gte*: greater or equal than
- *$lt*: less than
- *$lte*: less or equal than

```
project.find({"a": {"$lt": 5})
```

Matches all docs with *a* less than 5.

Logical Operators

That includes *$and* and *$or*; both expect a list of expressions.

```
project.find({"$or": [{"a": 4}, {"b": {"$gt": 3}}]})
```

Matches all docs, where *a* is 4 or *b* is greater than 3.

Exists operator

Determines whether a specific key exists, or not, e.g.:

```
project.find({"a": {"$exists": True}})
```

Array operator

To determine whether specific elements are in (*$in*), or not in (*$nin*) an array, e.g.:

```
project.find({"a": {"$in": [0, 1, 2]}})
```

Matches all docs, where *a* is either 0, 1, or 2. Usage of *$nin* is equivalent.

Regular expression operator

Allows the "on-the-fly" evaluation of regular expressoions, e.g.:

```
project.find({"protocol": {"$regex": "foo"}})
```

Will match all docs with a protocol that contains the term 'foo'.

$type operator

Matches when a value is of specific type, e.g.:

```
project.find({"protocol": {"$type": str}})
```

Finds all docs, where the value of protocol is of type str. Other types that can be checked are: *int*, *float*, *bool*, *list*, and *null*.

$where operator

Matches an arbitrary python expression, e.g.:

```
project.find({"foo": {"$where": "lambda x: x.startswith('bar')"}}
↪)
```

Matches all docs, where the value for foo starts with the word 'bar'.

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)
```

(continues on next page)

(continued from previous page)

```
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.:

```
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.

```
_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:

```
# 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=None*, *compresslevel=None*)
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:

```
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 *Collection.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')`.

Opening a gzip (**.gz*) file also works as expected. Because gzip does not support a combined read and write mode, *mode=*+* is not available. Be sure to open the file in read, write, or append mode as required. Due to the manner in which gzip works, opening a file in *mode=wt* will effectively erase the current file, so take care using *mode=wt*.

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.

1.2.5 The JSONDict

This class implements the interface for the job's *statepoint* and *document* attributes, but can also be used stand-alone:

class signac.**JSONDict**(*filename=None*, *write_concern=False*, *parent=None*)
A dict-like mapping interface to a persistent JSON file.

The JSONDict is a MutableMapping and therefore behaves similar to a dict, but all data is stored persistently in the associated JSON file on disk.

```
doc = JSONDict('data.json', write_concern=True)
doc['foo'] = "bar"
assert doc.foo == doc['foo'] == "bar"
assert 'foo' in doc
del doc['foo']
```

This class allows access to values through key indexing or attributes named by keys, including nested keys:

```
>>> doc['foo'] = dict(bar=True)
>>> doc
{'foo': {'bar': True}}
>>> doc.foo.bar = False
{'foo': {'bar': False}}
```

Warning: While the JSONDict object behaves like a dictionary, there are important distinctions to remember. In particular, because operations are reflected as changes to an underlying file, copying (even deep copying) a JSONDict instance may exhibit unexpected behavior. If a true copy is required, you should use the *_as_dict* method to get a dictionary representation, and if necessary construct a new JSONDict instance: *new_dict = JSONDict(old_dict._as_dict())*.

Parameters

- **filename** – The filename of the associated JSON file on disk.
- **write_concern** – Ensure file consistency by writing changes back to a temporary file first, before replacing the original file. Default is False.
- **parent** – A parent instance of JSONDict or None.

1.2.6 The H5Store

This class implements the interface to the job's *data* attribute, but can also be used stand-alone:

class signac.**H5Store**(*filename*, ***kwargs*)
An HDF5-backed container for storing array-like and dictionary-like data.

The H5Store is a `MutableMapping` and therefore behaves similar to a `dict`, but all data is stored persistently in the associated HDF5 file on disk.

Supported types include:

- built-in types (int, float, str, bool, NoneType, array)
- numpy arrays
- pandas data frames (requires pandas and pytables),

as well as mappings of values of these types. Values can be accessed as attributes (`h5s.foo`) or via key index (`h5s['foo']`).

Example:

```
with H5Store('file.h5') as h5s:
    h5s['foo'] = 'bar'
    assert 'foo' in h5s
    assert h5s.foo == 'bar'
    assert h5s['foo'] == 'bar'
```

The H5Store can be used as a context manager to ensure that the underlying file is opened, however most built-in types can be read and stored without the need to _explicitly_ open the file. However, to access arrays (reading or writing), the file must always be opened!

To open a file in read-only mode, use the *`open()`* method with `mode=r`:

```
with H5Store('fileh5').open(mode='r') as h5s:
    pass
```

Parameters

- **`filename`** – The filename of the underlying HDF5 file.
- **`kwargs`** – Additional keyword arguments to be forwarded to the `h5py.File` constructor. See the documentation for the h5py.File constructor for more information.

`clear``()`
Remove all data from this store.

> **Danger:** All data will be removed, this action cannot be reversed!

`close``()`
Close the underlying HDF5 file.

`file`
Access the underlying instance of h5py.File.

This property exposes the underlying `h5py.File` object enabling use of functions such as `create_dataset()` or `requires_dataset()`.

> **Note:** The store must be open to access this property!

Returns The `h5py` file-object that this store is operating on.

Return type `h5py.File`

Raises **H5StoreClosedError** – When the store is closed at the time of accessing this property.

flush()
Flush the underlying HDF5 file.

mode
The default opening mode of this store.

open(*mode=None*)
Open the underlying HDF5 file.

Parameters **mode** – The file open mode to use. Defaults to 'a' (append).

Returns This H5Store instance.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

1.2.7 The H5StoreManager

This class implements the interface to the job's *stores* aatribute, but can also be used stand-alone:

class signac.**H5StoreManager**(*prefix*)
Bases: signac.core.dict_manager.DictManager

Helper class to manage multiple instances of *H5Store* within a directory.

Example (assuming that the 'stores/' directory exists):

```
>>> stores = H5StoreManager('stores/')
>>> stores.data
<H5Store(filename=stores/data.h5)>
>>> stores.data.foo = True
>>> dict(stores.data)
{'foo': True}
```

Parameters **prefix** – The directory prefix shared by all stores managed by this class.

cls
alias of *H5Store*

1.2.8 Top-level functions

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.

signac.**TemporaryProject**(*name=None*, *cls=None*, ***kwargs*)
Context manager for the generation of a temporary project.

This is a factory function that creates a Project within a temporary directory and must be used as context manager, for example like this:

```
with TemporaryProject() as tmp_project:
    tmp_project.import_from('/data')
```

Parameters

- **name** – An optional name for the temporary project. Defaults to a unique random string.
- **cls** – The class of the temporary project. Defaults to *Project*.
- **kwargs** – Optional key-word arguments that are forwarded to the TemporaryDirectory class constructor, which is used to create a temporary root directory.

Returns An instance of *Project*.

signac.**get_project**(*root=None*, *search=True*)

Find a project configuration and return the associated project.

Parameters

- **root** (*str*) – The starting point to search for a project, defaults to the current working directory.
- **search** (*bool*) – If True, search for project configurations inside and above the specified root directory, otherwise only return projects with a root directory identical to the specified root argument.

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_job**(*root=None*)

Find a Job in or above the current working directory (or provided path).

Parameters **root** (*str*) – The job root directory. If no root directory is given, the current working directory is assumed to be within the current job workspace directory.

Returns The job handle.

Raises **LookupError** – If this job cannot be found.

For example, when the current directory is a job workspace directory:

```
>>> signac.get_job()
signac.contrib.job.Job(project=..., statepoint={...})
```

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.

```
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.**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.

signac.**export_pymongo**(*docs*, *index*, *mirrors=None*, *update=False*, *num_tries=3*, *timeout=60*, *chunksize=100*)

Optimized *export()* function for pymongo index collections.

The behavior of this function is rougly 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.**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:

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

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

```
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:

```
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.:

```
for doc in signac.index_files(formats=
    {r'.*\.txt': 'TextFile', r'.*\.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:

```
import signac

def get_indexes(root):
    yield signac.index_files(root, r'.*\.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.**buffered**(*buffer_size=33554432*, *force_write=False*)

Enter a global buffer mode for all JSONDict instances.

All future write operations are written to the buffer, read operations are performed from the buffer whenever possible.

All write operations are deferred until the flush_all() function is called, the buffer overflows, or upon exiting the buffer mode.

This context may be entered multiple times, however the buffer size can only be set *once*. Any subsequent specifications of the buffer size are ignored.

Parameters **buffer_size** (*int*) – Specify the maximum size of the read/write buffer. Defaults to DEFAULT_BUFFER_SIZE. A negative number indicates to not restrict the buffer size.

signac.**is_buffered**()

Return true if in buffered read/write mode.

signac.**flush**()
Execute all deferred JSONDict write operations.

signac.**get_buffer_size**()
Returns the current maximum size of the read/write buffer.

signac.**get_buffer_load**()
Returns the current actual size of the read/write buffer.

1.2.9 Submodules

signac.cite module

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.

signac.sync module

Synchronization of jobs and projects.

Jobs may be synchronized by copying all data from the source job to the destination job. This means all files are copied and the documents are synchronized. Conflicts, that means both jobs contain conflicting data, may be resolved with a user defined strategy.

The synchronization of projects is in essence the synchronization of all jobs which are in the destination project with the ones in the source project and the sync synchronization of the project document. If a specific job does not exist yet at the destination it is simply cloned, otherwise it is synchronized.

A sync strategy is a function (or functor) that takes the source job, the destination job, and the name of the file generating the conflict as arguments and returns the decision whether to overwrite the file as Boolean. There are some default strategies defined within this module as part of the *FileSync* class. These are the default strategies:

1. always – Always overwrite on conflict.

2. never – Never overwrite on conflict.
3. time – Overwrite when the modification time of the source file is newer.
4. Ask – Ask the user interactively about each conflicting filename.

For example, to synchronize two projects resolving conflicts by modification time, use:

```
dest_project.sync(source_project, strategy=sync.FileSync.time)
```

Unlike files, which are always either overwritten as a whole or not, documents can be synchronized more fine-grained with a *sync function*. Such a function (or functor) takes the source and the destination document as arguments and performs the synchronization. The user is encouraged to implement their own sync functions, but there are a few default functions implemented as part of the *DocSync* class:

1. NO_SYNC – Do not perform any synchronization.
2. COPY – Apply the same strategy used to resolve file conflicts.
3. update – Equivalent to dst.update(src).
4. ByKey – Synchronize the source document key by key, more information below.

This is how we could synchronize two jobs, where the documents are synchronized with a simple update function:

```
dst_job.sync(src_job, doc_sync=sync.DocSync.update)
```

The *DocSync.ByKey* functor attempts to synchronize the destination document with the source document without overwriting any data. That means this function behaves similar to `update()` for a non-intersecting set of keys, but in addition will preserve nested mappings without overwriting values. In addition, any key conflict, that means keys that are present in both documents, but have differing data, will lead to the raise of a *DocumentSyncConflict* exception. The user may expclitly decide to overwrite certain keys by providing a "key-strategy", which is a function that takes the conflicting key as argument, and returns the decision whether to overwrite that specific key as Boolean. For example, to sync two jobs, where conflicting keys should only be overwritten if they contain the term 'foo', we could execute:

```
dst_job.sync(src_job, doc_sync=sync.DocSync.ByKey(lambda key: 'foo' in key))
```

This means that all documents are synchronized 'key-by-key' and only conflicting keys that contain the word "foo" will be overwritten, any other conflicts would lead to the raise of a *DocumentSyncConflict* exception. A key-strategy may also be a regular expression, so the synchronization above could also be achieved with:

```
dst_job.sync(src_job, doc_sync=sync.DocSync.ByKey('foo'))
```

class signac.sync.**FileSync**

Bases: object

Collection of file synchronization strategies.

class Ask

Bases: object

Ask whether a file should be overwritten interactively.

static always(*src*, *dst*, *fn*)

Always overwrite files on conflict.

classmethod keys()

static never(*src*, *dst*, *fn*)

Never overwrite files on conflict.

static update(*src*, *dst*, *fn*)
Overwrite a file if the source file was modified last (based on timestamp).

class signac.sync.**DocSync**
Bases: object

Collection of document synchronization functions.

class ByKey(*key_strategy=None*)
Bases: object

Synchronize documents key by key.

COPY = 'copy'
Copy (and potentially overwrite) documents like any other file.

NO_SYNC = False
Do not synchronize documents.

static update(*src*, *dst*)
Perform a simple update.

signac.sync.**sync_jobs**(*src*, *dst*, *strategy=None*, *exclude=None*, *doc_sync=None*, *recursive=False*, *follow_symlinks=True*, *preserve_permissions=False*, *preserve_times=False*, *preserve_owner=False*, *preserve_group=False*, *deep=False*, *dry_run=False*)
Synchronize the src job with the dst job.

By default, this method will synchronize all files and document data of dst job with the src job until a synchronization conflict occurs. There are two different kinds of synchronization conflicts:

1. The two jobs have files with the same name, but different content.
2. The two jobs have documents that share keys, but those keys are mapped to different values.

A file conflict can be resolved by providing a 'FileSync' *strategy* or by *excluding* files from the synchronization. An unresolvable conflict is indicated with the raise of a *FileSyncConflict* exception.

A document synchronization conflict can be resolved by providing a doc_sync function that takes the source and the destination document as first and second argument.

Parameters

- **src** (*~.Job*) – The src job, data will be copied from this job's workspace.
- **dst** (*~.Job*) – The dst job, data will be copied to this job's workspace.
- **strategy** – A synchronization strategy for file conflicts. If no strategy is provided, a *errors.SyncConflict* exception will be raised upon conflict.
- **exclude** (*str*) – A filename exclusion pattern. All files matching this pattern will be excluded from the synchronization process.
- **doc_sync** – A synchronization strategy for document keys. The default is to use a safe key-by-key strategy that will not overwrite any values on conflict, but instead raises a *DocumentSyncConflict* exception.
- **recursive** (*bool*) – Recursively synchronize sub-directories encountered within the job workspace directories.
- **follow_symlinks** (*bool*) – Follow and copy the target of symbolic links.
- **preserve_permissions** (*bool*) – Preserve file permissions
- **preserve_times** (*bool*) – Preserve file modification times
- **preserve_owner** (*bool*) – Preserve file owner

- **preserve_group** (*bool*) – Preserve file group ownership
- **dry_run** – If True, do not actually perform any synchronization operations.

signac.sync.**sync_projects**(*source*, *destination*, *strategy=None*, *exclude=None*, *doc_sync=None*, *selection=None*, *check_schema=True*, *recursive=False*, *follow_symlinks=True*, *preserve_permissions=False*, *preserve_times=False*, *preserve_owner=False*, *preserve_group=False*, *deep=False*, *dry_run=False*, *parallel=False*, *collect_stats=False*)

Synchronize the destination project with the source project.

Try to clone all jobs from the source to the destination. If the destination job already exist, try to synchronize the job using the optionally specified strategy.

Parameters

- **source** (*Project*) – The project presenting the source for synchronization.
- **destination** (*Project*) – The project that is modified for synchronization.
- **strategy** – A file synchronization strategy.
- **exclude** – Files with names matching the given pattern will be excluded from the synchronization.
- **doc_sync** – The function applied for synchronizing documents.
- **selection** – Only synchronize the given selection of jobs.
- **check_schema** (*bool*) – If True, only synchronize if this and the other project have a matching state point schema. See also: *detect_schema()*.
- **recursive** (*bool*) – Recursively synchronize sub-directories encountered within the job workspace directories.
- **follow_symlinks** (*bool*) – Follow and copy the target of symbolic links.
- **preserve_permissions** (*bool*) – Preserve file permissions
- **preserve_times** (*bool*) – Preserve file modification times
- **preserve_owner** (*bool*) – Preserve file owner
- **preserve_group** (*bool*) – Preserve file group ownership
- **dry_run** (*bool*) – If True, do not actually perform the synchronization operation, just log what would happen theoretically. Useful to test synchronization strategies without the risk of data loss.

Raises

- ***DocumentSyncConflict*** – If there are conflicting keys within the project or job documents that cannot be resolved with the given strategy or if there is no strategy provided.
- ***FileSyncConflict*** – If there are differing files that cannot be resolved with the given strategy or if no strategy is provided.
- ***SchemaSyncConflict*** – In case that the check_schema argument is True and the detected state point schema of this and the other project differ.

signac.warnings module

exception signac.warnings.**SignacDeprecationWarning**

Bases: UserWarning

Indicates the deprecation of a signac feature, API or behavior.

This class indicates a user-relevant deprecation and is therefore a UserWarning, not a DeprecationWarning which is hidden by default.

signac.errors module

exception signac.errors.**Error**
Bases: Exception

exception signac.errors.**BufferException**
Bases: signac.core.errors.Error

An exception occured in buffered mode.

exception signac.errors.**BufferedFileError**(*files*)
Bases: signac.core.jsondict.BufferException

Raised when an error occured while flushing one or more buffered files.

files
A dictionary of files that caused issues during the flush operation, mapped to a possible reason for the issue or None in case that it cannot be determined.

exception signac.errors.**ConfigError**
Bases: signac.core.errors.Error, RuntimeError

exception signac.errors.**AuthenticationError**
Bases: signac.core.errors.Error, RuntimeError

exception signac.errors.**ExportError**
Bases: signac.core.errors.Error, RuntimeError

exception signac.errors.**FileNotFoundError**
Bases: signac.core.errors.Error, *FileNotFoundError*

exception signac.errors.**FetchError**
Bases: signac.common.errors.FileNotFoundError

exception signac.errors.**DestinationExistsError**(*destination*)
Bases: signac.core.errors.Error, RuntimeError

The destination for a move or copy operation already exists.

destination = None
The destination object causing the error.

exception signac.errors.**JobsCorruptedError**(*job_ids*)
Bases: signac.core.errors.Error, RuntimeError

The state point manifest file of one or more jobs cannot be openend or is corrupted.

job_ids = None
The job id(s) of the corrupted job(s).

exception signac.errors.**SyncConflict**
Bases: signac.core.errors.Error, RuntimeError

Raised when a synchronization operation fails.

exception signac.errors.**FileSyncConflict**(*filename*)
Bases: *signac.errors.SyncConflict*

Raised when a synchronization operation fails due to a file conflict.

filename = None
The filename of the file that caused the conflict.

exception signac.errors.**DocumentSyncConflict**(*keys*)
Bases: *signac.errors.SyncConflict*

Raised when a synchronization operation fails due to a document conflict.

keys = None
The keys that caused the conflict.

exception signac.errors.**SchemaSyncConflict**(*schema_src*, *schema_dst*)
Bases: *signac.errors.SyncConflict*

Raised when a synchronization operation fails due to schema differences.

exception signac.errors.**InvalidKeyError**
Bases: ValueError

Raised when a user uses a non-conforming key.

1.3 Changelog

The **signac** package follows semantic versioning.

1.3.1 Version 1.0

Highlights

- Native integration of HDF5 files with the `H5Store` and `H5StoreManager`, which are exposed as the `job.data`, `job.stores`, `project.data`, and `project.stores` properties respectively.
- The newly added `signac.get_job()` function makes it easier to obtain instances of `Job` by calling the function from within a job's workspace directory or by directly providing the path to the job's workspace directory. This is especially useful for interactive work or when accessing jobs which are outside of the current project.
- Simplified export of project and job data to pandas dataframes *via* the `to_dataframe()` function.
- Projects and job search results are displayed nicely in Jupyter Notebooks.
- Support for compressed Collection files.

[1.1.0] – 2019-05-19

Added

- Add command line options `--sp` and `--doc` for `signac find` that allow users to display key-value pairs of the state point and document in combination with the job id (#97, #146).
- Improve the representation (return value of *repr()*) of instances of `H5Group` and `SyncedAttrDict`.

Fixed

- Fix: Searches for whole numbers will match all numerically matching integers regardless of whether they are stored as decimals or whole numbers (#169).
- Fix: Passing an instance of dict to *H5Store.setdefault()* will return an instance of *H5Group* instead of a dict (#180).
- Fix error with storing numpy arrays and scalars in a synced dictionary (e.g. *job.statepoint*, *job.document*) (#184).
- Fix issue with `ResourceWarning` originating from unclosed instance of `Collection` (#186).
- Fix issue with using the `get_project()` function with a relative path and `search=False` (#191).

Removed

- Support for Python version 3.4 (no longer tested).

[1.0.0] – 2019-02-28

Added

- Official support for Python 3.7.
- The `H5Store` and `H5StoreManager` classes, which are useful for storing (numerical) array-like data with an HDF5-backend. These classes are exposed within the root namespace.
- The `job.data` and `project.data` properties which present an instance of `H5Store` to access numerical data within the job workspace and project root directory.
- The `job.stores` and `project.stores` properties, which present an instance of `H5StoreManager` to manage multiple instances of `H5Store` to store numerical array-like data within the project workspace and project root directory.
- The `signac.get_job()` and the `signac.Project.get_job()` functions that allow users to get a job handle by switching into or providing the job's workspace directory.
- The `job` variable is automatically set when opening a `signac shell` from within a job's workspace directory.
- Add the `signac shell -c` option which allows the direct specification of Python commands to be executed within the shell.
- Automatic cast of `numpy` arrays to lists when storing them within a `JSONDict`, e.g., a `job.statepoint` or `job.document`.
- Enable `Collection` class to manage collections stored in compressed files (gzip, zip, etc.).
- Enable deleting of `JSONDict` keys through the attribute interface, e.g., `del job.doc.foo`.
- Pretty HTML representation of instances of `Project` and `JobsCursor` targeted at Jupyter Notebooks (requires pandas, automatically enabled when installed).
- The `to_dataframe()` function to export the job state point and document data of a `Project` or a `JobsCursor`, e.g., the result of `Project.find_jobs()`, as a `pandas.Dataframe` (requires pandas).

Changed

- Dots (.) in keys are no longer allowed for `JSONDict` and `Collection` keys (previously deprecated).
- The `JSONDict` module is exposed in the root namespace, which is useful for storing text-serializable data with a JSON-backend similar to the `job.statepoint` or `job.document`, etc.
- The `Job.init()` method returns the job to allow one-line job creation and initialization.
- The `search` argument was added to the `signac.get_project()` function, which when True (the default), will cause signac to search for a project within *and above* a specified root directory, not only within the root directory. The behavior without any arguments remains unchanged.

Fixed

- Fix `Collection.update()` behavior such that existing documents with identical primary key are updated. Previously, a `KeyError` would be raised.
- Fix issue where the `Job.move()` would trigger a confusing `DestinationExists` exception when trying to move jobs across devices / file systems.
- Fix issue that caused failures when the `python-rapidjson` package is installed. The `python-rapidjson` package is used as the primary JSON-backend when installed.
- Fix issue where schema with multiple keys would subset incorrectly if the list of jobs or statepoints was provided as an iterator rather than a sequence.

Removed

- Removes the obsolete and deprecated `core.search_engine` module.
- The previously deprecated `Project.find_statepoints()` and `Project.find_job_documents()` functions have been removed.
- The `Project.find_jobs()` no longer accepts the obsolete `index` argument.

1.3.2 Version 0.9

Highlights

- Adds persistent state point index caching, which speeds up all functions that require indexing, for example the `$ signac find` command.
- Adds the `$ signac sync` tool for synchronization of multiple **signac** projects.
- Adds the `$ signac schema` function for the automatic detection of the implicit schema of a **signac** project.
- Adds the `$near` operator to match numbers with up to a specific precision.
- Adds functions for the import and export of data spaces.
- Add functions for the management of data on the project level, as opposed to the job level.

[0.9.5] – 2019-01-31

Fixed

- Ensure that the next() function can be called for a JobsIterator, e.g., *project.find()*.
- Pickling issue that occurs when a _SyncedDict (job.statepoint, job.document, etc.) contains a list.
- Issue with the readline module that would cause signac shell to fail on Windows operating systems.

[0.9.4] – 2018-10-24

Added

- Adds the `$ signac import` command and the `Project.import_from()` method for the import of data spaces into a project workspace, such as a directory, a tarball, or a zip file.
- Adds the `$ signac export` command and the `Project.export_to()` method for the export of project workspaces to an external location, such as a directory, a tarball, or a zip file.
- Adds functionality for the rapid initialization of temporary projects with the `signac.TemporaryProject` context manager.
- Adds the `signac.Project.temporary_project()` context manager which creates a temporary project within the root project workspace.
- Add `signac` to the default namespace when invoking `signac shell`.
- Add option to specify a custom view path for the `signac view`/ `Project.create_linked_view()` function.
- Iterables of documents used to construct a Collection no longer require an `_id` field.

Changed

- The default path for linked views has been adjusted to match the one used for data exports.

Fixed

- Fix issue where differently typed integer values stored within a Collection under the same key would not be indexed correctly. This issue affected the correct function of the *$type* operator for aforementioned cases and would lead to incorrect types in the Project schema detection algorithm for integer values.
- Fix issue where jobs that are migrated (state point change), but are not initialized, were not properly updated.
- Fix issue where changes to lists as part of synchronized dictionary, for example a state point or document would not be saved.
- Fix non-deterministic issue occuring on network file systems when trying to open jobs where the user has no write access to the job workspace directory.

[0.9.3] – 2018-06-14

Added

- Add *$near* operator to express queries for numerical values that match up to a certain precision.
- Add the *$ signac shell* sub command to directly launch a Python interpreter within a project directory.

Fixed

- Fix issue where a job instance would not be properly updated after more than one state point reset.

[0.9.2] – 2017-12-18

Added

- Add provisional feature (persistent state point caching); calling the `Project.update_cache()` method will generate and store a persistent state point cache in the project root directory, which will increase the speed of many project iteration, search, and selection operations.
- Add `Project.check()` method which checks for workspace corruption, but does not make any attempt to repair it.
- The `Project.repair()` method will attempt to repair jobs, that have been corrupted by manually renaming the job's workspace directory.

Changed

- Enable the *write_concern* flag for the `job.document`.
- Allow to omit the specification of an authentication mechanism in the MongoDB host configuration.

Fixed

- Fix critical issue in the `JSONDict` implementation that would previously overwrite the underlying file when attempting to store values that are not JSON serializable.
- Fix issue where the `Project.export()` function would ignore the update argument when the index to export to would be a MongoDB collection.

[0.9.1] – 2017-11-07

Fixed

- Fix critical issue in the `SyncedAttrDict` implementation that would previously overwrite the underlying file if the first operation was a `__setitem__()` operation.

[0.9.0] – 2017-10-28

Added

- Introduction of `$ signac sync`, `Project.sync()`, and `Job.sync()` for the simplified and fine-grained synchronization of multiple project data spaces.
- Introduction of `$ signac schema` and `Project.detect_schema()` for the automatic detection of the implicit and semi-structured state point schema of a project data space.
- Simplified aggregation of jobs over projects and `Project.find_jobs()` results with the `Project.groupby()` function.
- Support for project-centralized data with the `Project.document` attribute and the `Project.fn()` method for the wrapping of filenames within the project root directory.
- Added the `Job.clear()` and the `Job.reset()` methods to clear or reset a job's workspace data.

Changed

- Both `Job.statepoint` and `Job.document` now use the same underlying data structure and provide the exact same API (copy with `()` and access of keys as attributes).
- The `Collection` class uses an internal counter instead of UUIDs for the generation of primary keys (resulting in improved performance).
- Major performance improvements (faster Collection, improved caching)
- Overhaul of the reference documentation.

1.3.3 Version 0.8

Highlights

- Adds boolean and arithmetic operators to search queries.
- Major revision of the indexing system.
- Adds `$ signac document` command line function.
- Add the `signac.Collection` class for the management of persistent document collections.

[0.8.7] – 2017-10-05

Fixed

- Fix an issue where the creation of linked views was non-deterministic in some cases.
- Fix an issue where the creation of linked views would fail when the project contains job with state points that have lists as values.

[0.8.6] – 2017-08-25

Fixed

- Fix Collection append truncation issue (see issue #66).

[0.8.5] – 2017-06-07

Changed

- The signac ids in the *signac find –show* view are no longer enclosed by quotation marks.

Fixed

- Fix compatibility issue that broke the *signac find –view* and all *–pretty* commands on Python 2.7.
- Fix issue where view directories would be incomplete in combination with heterogeneous state point schemas.

[0.8.4] – 2017-05-19

Added

- All search queries on project and collection objects support various operators including: *$and*, *$or*, *$gt*, *$gte*, *$lt*, *$lte*, *$eq*, *$ne*, *$exists*, *$regex*, *$where*, *$in*, *$nin*, and *$type*.
- The `$ signac find` command supports a simple filter syntax, where key value pairs can be provided as individual arguments.
- The `$ signac find` command is extended by a *–show* option, to display the state point and the document contents directly. The contents are truncated to an adjustable depth to reduce output noise.
- The `$ signac view` command has an additional filter option to select a sub data space directly without needing to pipe job ids.
- The new `$ signac document` command can be used to display a job's document directly.

Changed

- Minor performance improvements.

[0.8.3] – 2017-05-10

Changed

- Raise `ExportError` when updating with an empty index.

Fixed

- Fix command line logic issue with `$signac config host`.
- Fix bug, where `Collection.replace_one()` would ignore the upsert argument under specific conditions.

[0.8.2] – 2017-04-19

Fixed

- Fixes a `TypeError` which occurred under specific conditions when calling `Collection.find()` with nested filter arguments.

[0.8.1] – 2017-04-17

Fixed

- Fixes wide-spread typo (*indeces* -> *indexes*).

[0.8.0] – 2017-04-16

Overall major simplification of the generation of indexes and the management and searching of index collections without external database.

Added

- Introduction of the `Collection` class for the management of document collections, such as indexes in memory and on disk.
- Generation of file indexes directly via the `signac.index_files()` function.
- Generation of master indexes directly via the `signac.index()` function and the `$ signac index` command.
- The API of `signac_access.py` files has been simplified, including the possibility to use a blank file for a minimal configuration.
- Use the `$ signac project --access` command to create a minimal access module in addition to `Project.create_access_module()`.
- The update of existing index collections has been simplified by using the `export()` function with the *update=True* argument, which means that stale documents (the associated file or state point no longer exists) are automatically identified and removed.
- Added the `Job.ws` attribute, as short-cut for `Job.workspace()`.
- The `Job.sp` interface has a `get()` function which can be used to specify a default value in case that the requested key is not part of the state point.

Changed (breaking API)

- The `$ signac index` command generates a master index instead of a project index. To generate a project index from the command line use `$ signac project --index` instead.
- The `SignacProjectCrawler` class expects the project's root directory as first argument, not the workspace directory.
- The `get_crawlers()` function defined within a `signac_access.py` access module is expected to yield crawler instances directly, not a mapping of crawler ids and instances.

- The simplification of the `signac_access.py` module API is reflected in a reduction of arguments to the `Project.create_access_module()` method.

Changed (non-breaking)

- The `RegexFileCrawler`, `SignacProjectCrawler` and `MasterCrawler` classes were moved into the root namespace.
- If a `MasterCrawler` object is instantiated with the `raise_on_error` argument set to True, any errors encountered during crawling are raised instead of ignored and skipped; this simplifies the debugging of erroneous access modules.
- Improved error message for invalid configuration files.
- Better error messages for invalid `$ signac find` queries.
- Check a host configuration on the command line via `$ signac host --test`.
- A MongoDB database host configuration defaults to *none* when no authentication method is explicitly specified.
- Using the `--debug` option in combination with `$ signac index` will show the traceback of errors encountered during indexing instead of ignoring them.
- Instances of `Job` are hashable, making it possible to use them as dict keys for instance.
- The representation of `Job` instances via `repr()` can actually serves as copy constructor command.
- The project interface implementation performs all non-trivial search operations on an internally management index collection, which improves performance and simplifies the code base.

Deprecated

- The `DocumentSearchEngine` class has been deprecated, its functionality is now provided by the `Collection` class.

Fixed

- An issue related to exporting documents to MongoDB collections via pymongo in combination with Python 2.7 has been fixed.

1.3.4 Version 0.7

Highlights

- Add support for Python 3.6, PyPy and PyPy3.
- Make any instance of `Project` behave like an iterable (`for job in project`).
- Introduction of the `Job.sp` attribute to access state point variables.
- Revision of the linked view function, which now allows the update of previous views.
- Support for searching by job document keys on the command line.
- Add functions for moving and cloning jobs.
- Add functions for changing a job's state point.

- Enable opening of jobs by *abbreviated id*.

[0.7.1] – 2017-01-09

Added

- When the `python-rapidjson` package is installed, it will be used for JSON encoding/decoding (experimental).

Changed

- All job move-related methods raise `DestinationExistsError` in case of destination conflicts.
- Optimized `$ signac find` command.

Fixed

- Fixed bug in `$ signac statepoint`.
- Suppress 'broken pipe error' message when using `$ signac find` for example in combination with `$ head`.

[0.7.0] – 2017-01-04

Added

- Add support for Python version 3.6.
- Add support for PyPy and PyPy3.
- Simplified iteration over project data spaces.
- An existing linked view can be updated by executing the view command again.
- Add attribute interface for the access and modification of job state points: `Job.sp`.
- Add function for moving and copying of jobs between projects.
- All project related iterators support the `len`-operator.
- Enable iteration over all jobs with: `for job in project:`.
- Make `len(project)` an alias for `project.num_jobs()`.
- Add `in`-operator to determine whether a job is initialized within a project.
- Add `Job.sp` attribute to access and modify a job's state point.
- The `Project.open_job()` method accepts abbreviated job ids.
- Add `Project.min_len_unique_id()` method to determine the minimum length of job ids to be unique within the project's data space.
- Add `Job.move()` method to move jobs between projects.
- Add `Project.clone()` method to copy jobs between projects.
- Add `$ signac move` and `$ signac clone` command line functions.

- Add `Job.reset_statepoint()` method to reset a job's state point.
- Add `Job.update_statepoint()` method to update a job's state point.
- Add a `Job.FN_DOCUMENT` constant which defines the default filename of the job document file
- The `$ signac find` command accepts a `-d/--doc-filter` option to filter by job document contents.
- Add the `Project.create_linked_view()` method as replacement for the previously deprecated `Project.create_view()` method.

Changed

- Linked views use relative paths.
- The *Guide* documentation chapter has been renamed to *Reference* and generally overhauled.
- The *Quick Reference* documentation chapter has been extended.

Fixed

- Fix error when using an instance of `Job` after calling `Job.remove()`.
- A project created in one the standard config directories (such as the home directory) does not take prevalence over project configurations in or above the current working directory.

Removed

- The *signac-gui* component has been removed.
- The `Project.create_linked_view() force` argument is removed.
- The `Project.find_variable_parameters()` method has been removed

1.3.5 Version 0.6

Highlights

- General revision of the indexing and export system.
- General consolidation including the removal of the *conversion framework*.

[0.6.2] – 2017-12-15

Added

- Add instructions on how to acknowledge **signac** in publications to documentation.
- Add cite module for the auto-generation of formatted references and BibTeX entries.

Removed

- Remove SSL authentication support.

[0.6.1] – 2017-11-26

Changed

- The `Project.create_view()` method triggers a `DeprecationWarning` instead of a `PendingDeprecationWarning`.
- The `Project.find_variable_parameters()` method triggers a `DeprecationWarning` instead of a `PendingDeprecationWarning`.

Fixed

- Make package more robust against PySide import errors.
- Fix `Project.__repr__` method.
- Fix critical bug in `fs.GridFS` class, which rendered it unusuable.
- Fix issue in `indexing.fetch()` previously resulting in local paths being ignored.
- Fix error `signac.__all__` namespace directive.

[0.6.0] – 2016-11-18

Added

- Add the `export_to_mirror()` function for mirroring files.
- Introduction of the `signac.fs` namespace to simplify the configuration of mirror filesystems.
- Add *errors* module to root namespace. Many exceptions raised inherit from the base exception types defined within that module, making it easier to catch signac related errors.
- Add the `export_one()` function for the export of a single index document; simplifies the implementation of custom export functions.
- Opening an instance of `Job` with the `open_job()` method multiple times and entering a job context recursively is now well-defined behavior: Opening a job now adds the current working directory onto a stack, closing it switches into the directory on top of the stack.
- The return type of `Project.open_job()` can be configured to make it easier to specialize projects with custom job types.

Changed

- The MasterCrawler logic has been simplified; their primary function is the compilation of index documents from slave crawlers, all export logic, including data mirroring is now provided by the `signac.export()` function.
- Each index document is now uniquely coupled with only one file or data object, which is why `signac.fetch()` replaces `signac.fetch_one()` and the latter one has been deprecated and is currently an alias of the former one.
- The `signac.fetch()` function always returns a file-like object, regardless of format definition.
- The format argument in the crawler `define()` function is now optional and has now very well defined behavior for str types. It is encouraged to define a format with a str constant rather than a file-like object type.

- The `TextFile` file-like object class definition in the formats module has been replaced with a constant of type `str`.
- The `signac.export()` function automatically delegates to specialized implementations such as `export_pymongo()` and is more robust against errors, such as broken connections.
- The `export_pymongo()` function makes multiple automatic restart attempts when encountering errors.
- Documentation: The tutorial is now based on signac-examples jupyter notebooks.
- The `contrib.crawler` module has been renamed to contrib.indexing to better reflect the semantic context.
- The `signac.export()` function now implements the logic for data linking and mirroring.
- Provide default argument for '–indent' option for `$ signac statepoint` command.
- Log, but do not reraise exceptions during `MasterCrawler` execution, making the compilation of master indexes more robust against errors.
- The object representation of `Job` and `Project` instances is simplified.
- The warning verbosity has been reduced when importing modules with optional dependencies.

Removed

- All modules related to the stale *conversion framework* feature have been removed resulting in a removal of the optional networkx dependency.
- Multiple modules related to the *conversion framework* feature have been removed, including: `contrib.formats_network`, `contrib.conversion`, and `contrib.adapters`.

Fixed

- Opening instances of `Job` with the `Job.open()` method multiple times, equivalently entering the job context recursively, does not cause an error anymore, but instead the behavior is well-defined.

1.3.6 Version 0.5

[0.5.0] – 2016-08-31

Added

- New function: `signac.init_project()` simplifies project initialization within Python
- Added optional `root` argument to `signac.get_project()` to simplify getting a project handle outside of the current working directory
- Added optional argument to `signac.get_project()`, to allow fetching of projects outside of the current working directory.
- Added two class factory methods to *Project*: `get_project()` and `init_project()`.

Changed

- The performance of project indexing and crawling has been improved.

1.3.7 Version 0.4

[0.4.0] – 2016-08-05

Added

- The performance of find operations can be greatly improved by using pre-generated job indexes.
- New top-level commands: `$ signac find`, `$ signac index`, `$ signac statepoint`, and `$ signac view`.
- New method: `Project.create_linked_view()`
- New method: `Project.build_job_statepoint_index()`
- New method: `Project.build_job_search_index()`
- The `Project.find_jobs()` method allows to filter by job document.

Changed

- The `SignacProjectCrawler` indexes all jobs, not only those with non-empty job documents.
- The `signac.fetch_one()` function returns `None` if no associated object can be fetched.
- The tutorial is restructured into multiple parts.
- Instructions for installation are separated from the guide.

Removed

- Remove previously deprecated *crawl* keyword argument in index export functions.
- Remove previously deprecated function common.config.write_config().

1.3.8 Version 0.3

[0.3.0] – 2016-06-23

Added

- Add contributing agreement and guidelines.

Changed

- Change license from MIT to BSD 3-clause license.

1.3.9 Version 0.2

[0.2.9] – 2016-06-06

Added

- Addition of the `signac config` command line API.
- Password updates are encrypted with `bcrypt` when `passlib` is installed.
- The user is prompted to enter missing credentials (username/password) in case that they are not stored in the configuration.
- The `$ signac confg` tool provides the `--update-pw` argument, which allows users to update their own password.
- Added MIT license, in addition, all source code files contain a short licensing header.

Changed

- Improved documentation on how to configure signac.
- The OSI classifiers are updated, including an upgrade of the development status to '4 - beta'.

Fixed

- Nested job state points can no longer get corrupted. This bug occurred when trying to operate on nested state point mappings.

Deprecated

- Deprecated pymongo versions 2.x are no longer supported.

[0.2.8] – 2016-04-18

Added

- `Project` is now in the root namespace.
- Add `index()` method to *Project*.
- Add `create_access_module()` method to `Project`.
- Add `find_variable_parameters()` method to `Project`.
- Add `fn()` method to `Job`, which prepends the job's workspace path to a filename.
- The documentation contains a comprehensive tutorial

Changed

- The `crawl()` function yields only the index documents and not a tuple of (_id, doc).
- `export()` and `export_pymongo()` expect the index documents as first argument, not a crawler instance. The old API is still supported, but will trigger a DeprecationWarning.

[0.2.7] – 2016-02-29

Added

- Add `job.isfile()` method

Changed

- Optimize `project.find_statepoints()` and `project.repair()` functions.

[0.2.6] – 2016-02-20

Added

- Add `job.reset_statepoint()` and *job.update_statepoint()*
- Add `job.remove()` function

Changed

- Sanitize filter argument in all `project.find_*()` methods.
- The `job.statepoint()` function accurately represents saved statepoints, e.g. tuples are represented as lists, as there is no difference between tuples and lists in JSON.
- signac-gui does not block on database operations.
- signac-gui allows reload of databases and collections of connected hosts.

Fixed

- `RegexFileCrawler define()` class function only acts upon the actual specialization and not globally on all `RegexFileCrawler` classes.
- signac-gui does not crash when replica sets are configured.

[0.2.5] – 2016-02-10

Added

- Added `signac.get_project()`, `signac.get_database()`, `signac.fetch()` and `signac.fetch_one()` to top-level namespace.
- Added basic shell commands, see `$ signac --help`.

- Allow opening of jobs by id: `project.open_job(id='abc123...')`.
- Mirror data while crawling.
- Use extra sources for `fetch()` and `fetch_one()`.
- Add file system handler: `LocalFS`, handler for local file system.
- Add file system handler: `GridFS`, handler for MongoDB GridFS file system.
- Crawler tags, to control which crawlers are used for a specific index.
- Allow explicit job workspace creation with `job.init()`.
- Forwarding of pymongo host configuration via **signac** configuration.

Changed

- Major reorganization of the documentation, split into: Overview, Guide, Quick Reference and API.
- Documentation: Add notes for system administrators about advanced indexing.
- Warn about outdated pymongo versions.
- Set *zip_safe* flag to true in *setup.py*.
- Remove dependency on six module, by adding it to the common subpackage.

Deprecated

Fixed

- Fixed hard import of pymongo bug (issue #24).
- Crawler issues with malformed documents.

[0.2.4] – 2016-01-11

Added

- Implement `Project.repair()` function for projects with corrupted workspaces.
- Allow environment variables in workspace path definition.
- Check and fix config permission errors.

Changed

- Increase robustness of job manifest file creation.

Fixed

- Fix project crawler deep directory issue (hotfix).

[0.2.3] – 2015-12-09

Fixed

- Fix a few bugs related to project views.

[0.2.2] – 2015-11-30

Fixed

- Fix `SignacProjectCrawler` `super()` bug.

[0.2.1] – 2015-11-29

Added

- Add support for Python 2.7
- Add signac-gui (early alpha)
- Allow specification of relative and default workspace paths
- Add the ability to create project views
- Add `Project.find_*()` functions to search the workspace
- Add function to write and read state point hash tables

[0.2.0] – 2015-11-05

- Major consolidation of the package.
- Remove all hard dependencies, but six.

1.4 Support and Development

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 GitHub and licensed under the open-source BSD 3-Clause license. Please use the repository's issue tracker to report bugs or request new features.

1.4.1 Code contributions

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

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

Set up a development environment

Start by forking the project.

We highly recommend to setup a dedicated development environment, for example with venv:

```
~ $ python -m venv ~/envs/signac-dev
~ $ source ~/envs/signac-dev/bin/activate
(signac-dev) ~ $ pip install six flake8
```

or alternatively with conda:

```
~ $ conda create -n signac-dev python=3 six flake8
~ $ activate signac-dev
```

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

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

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

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

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

With the *flake8* hook, your code will be checked for syntax and style before you make a commit. The continuous integration pipeline for the package will perform these checks as well, so running these tests before committing / pushing will prevent the pipeline from failing due to style-related issues.

The development workflow

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

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

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

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

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

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

Testing

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

```
(signac-dev) signac $ python -m unittest discover tests/
```

Building documentation

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

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

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

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

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

Updating the changelog

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

```
next
----

- Fix issue with launching rockets to the moon.

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

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

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

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

G

H

I

J

K

M

N

O

P

R

S

T

U

W