Sacred Documentation

Release 0.8.4

Klaus Greff

Jan 25, 2023

Contents

Every experiment is sacred
Every experiment is great
If an experiment is wasted
God gets quite irate

Sacred is a tool to configure, organize, log and reproduce computational experiments. It is designed to introduce only minimal overhead, while encouraging modularity and configurability of experiments.

The ability to conveniently make experiments configurable is at the heart of Sacred. If the parameters of an experiment are exposed in this way, it will help you to:

- keep track of all the parameters of your experiment
- easily run your experiment for different settings
- save configurations for individual runs in files or a database
- reproduce your results

In Sacred we achieve this through the following main mechanisms:

1. *Config Scopes* are functions with a `@ex.config` decorator, that turn all local variables into configuration entries. This helps to set up your configuration really easily.
2. Those entries can then be used in *captured functions* via *dependency injection*. That way the system takes care of passing parameters around for you, which makes using your config values really easy.
3. The *command-line interface* can be used to change the parameters, which makes it really easy to run your experiment with modified parameters.
4. Observers log every information about your experiment and the configuration you used, and saves them for example to a Database. This helps to keep track of all your experiments.
5. Automatic seeding helps controlling the randomness in your experiments, such that they stay reproducible.

CHAPTER 1

Contents

1.1 Quickstart

1.1.1 Installation

You can get Sacred directly from pypi like this:

```
pip install sacred
```

But you can of course also clone the git repo and install it from there:

```
git clone https://github.com/IDSIA/sacred.git
cd sacred
[sudo] python setup.py install
```

1.1.2 Hello World

Let's jump right into it. This is a minimal experiment using Sacred:

```
from sacred import Experiment

ex = Experiment()

@ex.automain
def my_main():
    print('Hello world!')
```

We did three things here:

- import `Experiment` from `sacred`
- create an experiment instance `ex`
- decorate the function that we want to run with `@ex.automain`

This experiment can be run from the command-line, and this is what we get:

```
> python h01_hello_world.py
INFO - 01_hello_world - Running command 'my_main'
INFO - 01_hello_world - Started
Hello world!
INFO - 01_hello_world - Completed after 0:00:00
```

This experiment already has a full command-line interface, that we could use to control the logging level or to automatically save information about the run in a database. But all of that is of limited use for an experiment without configurations.

1.1.3 Our First Configuration

So let us add some configuration to our program:

```
from sacred import Experiment

ex = Experiment('hello_config')

@ex.config
def my_config():
    recipient = "world"
    message = "Hello %s!" % recipient

@ex.automain
def my_main(message):
    print(message)
```

If we run this the output will look precisely as before, but there is a lot going on already, so lets look at what we did:

- add the `my_config` function and decorate it with `@ex.config`.
- within that function define the variable `message`
- add the `message` parameter to the function `main` and use it instead of "Hello world!"

When we run this experiment, Sacred will run the `my_config` function and put all variables from its local scope into the configuration of our experiment. All the variables defined there can then be used in the `main` function. We can see this happening by asking the command-line interface to print the configuration for us:

```
> python hello_config.py print_config
INFO - hello_config - Running command 'print_config'
INFO - hello_config - started
Configuration:
  message = 'Hello world!'
  recipient = 'world'
  seed = 746486301
INFO - hello_config - finished after 0:00:00.
```

Notice how Sacred picked up the `message` and the `recipient` variables. It also added a `seed` to our configuration, but we are going to ignore that for now.

Now that our experiment has a configuration we can change it from the *Command-Line Interface*:

```
> python hello_config.py with recipient="that is cool"
INFO - hello_config - Running command 'my_main'
INFO - hello_config - started
```

(continues on next page)

(continued from previous page)

```
Hello that is cool!
INFO - hello_config - finished after 0:00:00.
```

Notice how changing the `recipient` also changed the message. This should give you a glimpse of the power of Sacred. But there is a lot more to it, so keep reading :).

1.2 Experiment Overview

`Experiment` is the central class of the Sacred framework. This section provides an overview of what it does and how to use it.

1.2.1 Create an Experiment

To create an `Experiment` just instantiate it and add main method:

```
from sacred import Experiment
ex = Experiment()

@ex.main
def my_main():
    pass
```

The function decorated with `@ex.main` is the main function of the experiment. It is executed if you run the experiment and it is also used to determine the source-file of the experiment.

Instead of `@ex.main` it is recommended to use `@ex.automain`. This will automatically run the experiment if you execute the file. It is equivalent to the following:

```
from sacred import Experiment
ex = Experiment()

@ex.main
def my_main():
    pass

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

Note: For this to work the `automain` function needs to be at the end of the file. Otherwise everything below it is not defined yet when the experiment is run.

1.2.2 Run the Experiment

The easiest way to run your experiment is to just use the command-line. This requires that you used `automain` (or an equivalent). You can then just execute the experiments python file and use the powerful *Command-Line Interface*.

You can also run your experiment directly from python. This is especially useful if you want to run it multiple times with different configurations. So lets say your experiment is in a file called `my_experiment.py`. Then you can import it from there and run it like this:

```
from my_experiment import ex

r = ex.run()
```

Warning: By default, Sacred experiments **will fail** if run in an interactive environment like a REPL or a Jupyter Notebook. This is an intended security measure since in these environments reproducibility cannot be ensured. If needed, this safeguard can be deactivated by passing `interactive=True` to the experiment like this:

```
ex = Experiment('jupyter_ex', interactive=True)
```

The `run` function accepts `config_updates` to specify how the configuration should be changed for this run. It should be a (possibly nested) dictionary containing all the values that you wish to update. For more information see *Configuration*:

```
from my_experiment import ex

r = ex.run(config_updates={'foo': 23})
```

Note: Under the hood a `Run` object is created every time you run an `Experiment` (this is also the object that `ex.run()` returns). It holds some information about that run (e.g. final configuration and later the result) and is responsible for emitting all the events for the *Observing an Experiment*.

While the experiment is running you can access it by accepting the special *_run* argument in any of your *Captured Functions*. That is also used for *Saving Custom Information*.

1.2.3 Configuration

There are multiple ways of adding configuration to your experiment. The easiest way is through *Config Scopes*:

```
@ex.config
def my_config():
    foo = 42
    bar = 'baz'
```

The local variables from that function are collected and form the configuration of your experiment. You have full access to the power of python when defining the configuration that way. The parameters can even depend on each other.

Note: Only variables that are JSON serializable (i.e. a numbers, strings, lists, tuples, dictionaries) become part of the configuration. Other variables are ignored.

If you think that is too much magic going on, you can always use a plain dictionary to add configuration or, if you prefer, you can also directly load configuration entries from a file.

And of course you can combine all of them and even have several of each kind. They will be executed in the order that you added them, and possibly overwrite each others values.

1.2.4 Capture Functions

To use a configuration value all you have to do is *capture* a function and accept the configuration value as a parameter. Whenever you now call that function Sacred will try to fill in missing parameters from the configuration. To see how that works we need to *capture* some function:

```
from sacred import Experiment
ex = Experiment('my_experiment')

@ex.config
def my_config():
    foo = 42
    bar = 'baz'

@ex.capture
def some_function(a, foo, bar=10):
    print(a, foo, bar)

@ex.main
def my_main():
    some_function(1, 2, 3)      #  1  2   3
    some_function(1)            #  1  42  'baz'
    some_function(1, bar=12)    #  1  42  12
    some_function()             #  TypeError: missing value for 'a'
```

More on this in the *Captured Functions* Section.

Note: Configuration values are preferred over default values. So in the example above, `bar=10` is never used because there is a value of `bar = 'baz'` in the configuration.

1.2.5 Observe an Experiment

Experiments in Sacred collect lots of information about their runs like:

- time it was started and time it stopped
- the used configuration
- the result or any errors that occurred
- basic information about the machine it runs on
- packages the experiment depends on and their versions
- all imported local source-files
- files opened with `ex.open_resource`
- files added with `ex.add_artifact`

To access this information you can use the observer interface. First you need to add an observer like this:

```
from sacred.observers import MongoObserver

ex.observers.append(MongoObserver())
```

`MongoObserver` is one of the default observers shipped with Sacred. It connects to a MongoDB and puts all these information into a document in a collection called `experiments`. You can also add this observer from the *Command-Line Interface* like this:

```
>> python my_experiment.py -m my_database
```

For more information see *Observing an Experiment*

Capturing stdout / stderr

Sacred tries to capture all outputs and transmits that information to the observers. This behaviour is configurable and can happen in three different modes: `no`, `sys` and `fd`. This mode can be *set from the commandline* or in the *Settings*.

In the `no` mode none of the outputs are captured. This is the default behaviour if no observers are added to the experiment.

If the capture mode is set to `sys` then sacred captures all outputs written to `sys.stdout` and `sys.stderr` such as `print` statements, stacktraces and logging. In this mode outputs by system-calls, C-extensions or subprocesses are likely *not captured*. This behaviour is default for Windows.

Finally, the `fd` mode captures outputs on the file descriptor level, and should include all outputs made by the program or any child-processes. This is the default behaviour for Linux and OSX.

The captured output contains all printed characters and behaves like a file and not like a terminal. Sometimes this is unwanted, for example when the output contains lots of live-updates like progressbars. To prevent the captured out from retaining each and every update that is written to the console one can add a *captured out filter* to the experiment like this:

```
from sacred.utils import apply_backspaces_and_linefeeds

ex.captured_out_filter = apply_backspaces_and_linefeeds
```

Here `apply_backspaces_and_linefeeds` is a simple function that interprets all backspace and linefeed characters like in a terminal and returns the modified text. Any function that takes a string as input and outputs a (modified) string can be used as a `captured_out_filter`. For a simple example see examples/captured_out_filter.py.

1.2.6 Interrupted and Failed Experiments

If a run is interrupted (e.g. Ctrl+C) or if an exception occurs, Sacred will gather the stacktrace and the fail time and report them to the observers. The resulting entries will have their status set to `INTERRUPTED` or to `FAILED`. This allows to quickly see the reason for a non-successful run, and enables later investigation of the errors.

Detecting Hard Failures

Sometimes an experiment can fail without an exception being thrown (e.g. power loss, kernel panic, ...). In that case the failure cannot be logged to the database and their status will still be `RUNNING`. Runs that fail in that way are most easily detected by investigating their heartbeat time: each running experiment reports to its observers in regular intervals (default every 10 sec) and updates the heartbeat time along with the captured stdout and the info dict (see *Saving Custom Information*). So if the heartbeat time lies much further back in time than that interval, the run can be considered dead.

Debugging

If an Exception occurs, sacred by default filters the stacktrace by removing all sacred-internal calls. The stacktrace is of course also saved in the database (if appropriate observer is added). This helps to quickly spot errors in your own code. However, if you want to use a debugger, stacktrace filtering needs to be disabled, because it doesn't play well with debuggers like `pdb`.

If you want to use a debugger with your experiment, you have two options:

Disable Stacktrace Filtering

Stacktrace filtering can be deactivated via the `-d` flag. Sacred then does not interfere with the exception and it can be properly handled by any debugger.

Post-Mortem Debugging

For convenience Sacred also supports directly attaching a post-mortem `pdb` debugger via the `-D` flag. If this option is set and an exception occurs, sacred will automatically start `pdb` debugger to investigate the error, and interact with the stack.

Custom Interrupts

Sometimes it can be useful to have custom reasons for interrupting an experiment. One example is if there is a limited time budget for an experiment. If the experiment is stopped because of exceeding that limit, that should be reflected in the database entries.

For these cases, Sacred offers a special base exception *`sacred.utils.SacredInterrupt`* that can be used to provide a custom status code. If an exception derived from this one is raised, then the status of the interrupted run will be set to that code.

For the aforementioned timeout usecase there is the *`sacred.utils.TimeoutInterrupt`* exception with the status code `TIMEOUT`. But any status code can be used by simply creating a custom exception that inherits from *`sacred.utils.SacredInterrupt`* and defines a `STATUS` member like this:

```
from sacred.utils import SacredInterrupt

class CustomInterrupt(SacredInterrupt)
    STATUS = 'MY_CUSTOM_STATUS'
```

When this exception is raised during any run, its status is set to `MY_CUSTOM_STATUS`.

1.2.7 Queuing a Run

Sacred also supports queuing runs by passing the *Queue* flag (`-q`/`--queue`). This will **not** run the experiment, but instead only create a database entry that holds all information needed to start the run. This feature could be useful for having a distributed pool of workers that get configurations from the database and run them. As of yet, however, there is no further support for this workflow.

1.3 Configuration

The configuration of an experiment is the standard way of parametrizing runs. It is saved in the database for every run, and can very easily be adjusted. Furthermore all configuration entries can be accessed by all *Captured Functions*.

There are three different ways of adding configuration to an experiment. Through *Config Scopes*, *Dictionaries*, and *Config Files*

Note: Because configuration entries are saved to the database directly, some restrictions apply. The keys of all dictionaries cannot contain `.`, `=`, or `$`. Furthermore they cannot be `jsonpickle` keywords like `py/object`. If absolutely necessary, these restrictions can be configured in `sacred.settings.SETTINGS.CONFIG`.

Note: Also note - because objects are internally converted to JSON before database storage, python `tuple` objects will be converted to `list` objects when they are stored in a configuration object. Please see Issue #115 for the latest information on this.

1.3.1 Defining a Configuration

Sacred provides several ways to define a configuration for an experiment. The most powerful one are Config Scopes, but it is also possible to use plain dictionaries or config files.

Config Scopes

A Config Scope is just a regular function decorated with `@ex.config`. It is executed by Sacred just before running the experiment. All variables from its local scope are then collected, and become configuration entries of the experiment. Inside that function you have full access to all features of python for setting up the parameters:

```
from sacred import Experiment
ex = Experiment('config_demo')

@ex.config
def my_config():
    """This is my demo configuration"""

    a = 10  # some integer

    # a dictionary
    foo = {
        'a_squared': a**2,
        'bar': 'my_string%d' % a
    }
    if a > 8:
        # cool: a dynamic entry
        e = a/2

@ex.main
def run():
    pass
```

This config scope would return the following configuration, and in fact, if you want to play around with this you can just execute `my_config`:

```
>>> my_config()
{'foo': {'bar': 'my_string10', 'a_squared': 100}, 'a': 10, 'e': 5}
```

Or use the `print_config` command from the *Command-Line Interface*:

```
$ python config_demo.py print_config
INFO - config_demo - Running command 'print_config'
INFO - config_demo - Started
Configuration (modified, added, typechanged, doc):
  """This is my demo configuration"""
  a = 10                                # some integer
  e = 5.0                               # cool: a dynamic entry
  seed = 954471586                      # the random seed for this experiment
  foo:                                  # a dictionary
    a_squared = 100
    bar = 'my_string10'
INFO - config_demo - Completed after 0:00:00
```

Notice how Sacred picked up on the doc-string and the line comments used in the configuration. This can be used to improve user-friendliness of your script.

Warning: Functions used as a config scopes **cannot** contain any `return` or `yield` statements!

Dictionaries

Configuration entries can also directly be added as a dictionary using the `ex.add_config` method:

```
ex.add_config({
  'foo': 42,
  'bar': 'baz
})
```

Or equivalently:

```
ex.add_config(
    foo=42,
    bar='baz'
)
```

Unlike config scopes, this method raises an error if you try to add any object, that is not JSON-Serializable.

Config Files

If you prefer, you can also directly load configuration entries from a file:

```
ex.add_config('conf.json')
ex.add_config('conf.pickle')  # if configuration was stored as dict
ex.add_config('conf.yaml')    # requires PyYAML
```

This will essentially just read the file and add the resulting dictionary to the configuration with `ex.add_config`.

Combining Configurations

You can have multiple Config Scopes and/or Dictionaries and/or Files attached to the same experiment or ingredient. They will be executed in order of declaration. This is especially useful for overriding ingredient default values (more about that later). In config scopes you can even access the earlier configuration entries, by just declaring them as parameters in your function:

```
ex = Experiment('multiple_configs_demo')

@ex.config
def my_config1():
    a = 10
    b = 'test'

@ex.config
def my_config2(a):  # notice the parameter a here
    c = a * 2       # we can use a because we declared it
    a = -1          # we can also change the value of a
    #d = b + '2'    # error: no access to b

ex.add_config({'e': 'from_dict'})
# could also add a config file here
```

As you'd expect this will result in the configuration `{'a': -1, 'b': 'test', 'c': 20, 'e': 'from_dict'}`.

1.3.2 Updating Config Entries

When an experiment is run, the configuration entries can be updated by passing an update dictionary. So let's recall this experiment to see how that works:

```
from sacred import Experiment
ex = Experiment('config_demo')

@ex.config
def my_config():
    a = 10
    foo = {
        'a_squared': a**2,
        'bar': 'my_string%d' % a
    }
    if a > 8:
        e = a/2

@ex.main
def run():
    pass
```

If we run that experiment from python we can simply pass a `config_updates` dictionary:

```
>>> r = ex.run(config_updates={'a': 23})
>>> r.config
{'foo': {'bar': 'my_string23', 'a_squared': 529}, 'a': 23, 'e': 11.5}
```

Using the *Command-Line Interface* we can achieve the same thing:

```
$ python config_demo.py print_config with a=6
INFO - config_demo - Running command 'print_config'
INFO - config_demo - Started
Configuration (modified, added, typechanged, doc):
  a = 6                                 # some integer
  seed = 681756089                      # the random seed for this experiment
  foo:                                  # a dictionary
    a_squared = 36
    bar = 'my_string6'
INFO - config_demo - Completed after 0:00:00
```

Note that because we used a config scope all the values that depend on `a` change accordingly.

Note: This might make you wonder about what is going on. So let me briefly explain: Sacred extracts the body of the function decorated with `@ex.config` and runs it using the `exec` statement. That allows it to provide a `locals` dictionary which can block certain changes and log all the others.

We can also fix any of the other values, even nested ones:

```
>>> r = ex.run(config_updates={'foo': {'bar': 'baobab'}})
>>> r.config
{'foo': {'bar': 'baobab', 'a_squared': 100}, 'a': 10, 'e': 5}
```

or from the commandline using dotted notation:

```
$ config_demo.py print_config with foo.bar=baobab
INFO - config_demo - Running command 'print_config'
INFO - config_demo - Started
Configuration (modified, added, typechanged, doc):
  a = 10                                # some integer
  e = 5.0                               # cool: a dynamic entry
  seed = 294686062                      # the random seed for this experiment
  foo:                                  # a dictionary
    a_squared = 100
    bar = 'baobab'
INFO - config_demo - Completed after 0:00:00
```

To prevent accidentally wrong config updates sacred implements a few basic checks:

- If you change the type of a config entry it will issue a warning
- If you add a new config entry but it is used in some captured function, it will issue a warning
- If you add a new config entry that is not used anywhere it will raise a KeyError.

1.3.3 Named Configurations

With so called *Named Configurations* you can provide a ConfigScope that is not used by default, but can be optionally added as config updates:

```
ex = Experiment('named_configs_demo')

@ex.config
def cfg():
    a = 10
```

(continues on next page)

(continued from previous page)

```
    b = 3 * a
    c = "foo"

@ex.named_config
def variant1():
    a = 100
    c = "bar"
```

The default configuration of this Experiment is `{'a':10, 'b':30, 'c':"foo"}`. But if you run it with the named config like this:

```
$ python named_configs_demo.py with variant1
```

Or like this:

```
>> ex.run(named_configs=['variant1'])
```

Then the configuration becomes `{'a':100, 'b':300, 'c':"bar"}`. Note that the named ConfigScope is run first and its values are treated as fixed, so you can have other values that are computed from them.

Note: You can have multiple named configurations, and you can use as many of them as you like for any given run. But notice that the order in which you include them matters: The ones you put first will be evaluated first and the values they set might be overwritten by further named configurations.

Configuration files can also serve as named configs. Just specify the name of the file and Sacred will read it and treat it as a named configuration. Like this:

```
$ python named_configs_demo.py with my_variant.json
```

or this:

```
>> ex.run(named_configs=['my_variant.json'])
```

Where the format of the config file can be anything that is also supported for *config files*.

1.3.4 Accessing Config Entries

Once you've set up your configuration, the next step is to use those values in the code of the experiment. To make this as easy as possible Sacred automatically fills in the missing parameters of a *captured function* with configuration values. So for example this would work:

```
ex = Experiment('captured_func_demo')

@ex.config
def my_config1():
    a = 10
    b = 'test'

@ex.automain
def my_main(a, b):
    print("a =", a)   # 10
    print("b =", b)   # test
```

Captured Functions

Sacred automatically injects configuration values for captured functions. Apart from the main function (marked by `@ex.main` or `@ex.automain`) this includes all functions marked with `@ex.capture`. So the following example works as before:

```
ex = Experiment('captured_func_demo2')

@ex.config
def my_config1():
    a = 10
    b = 'test'

@ex.capture
def print_a_and_b(a, b):
    print("a =", a)
    print("b =", b)

@ex.automain
def my_main():
    print_a_and_b()
```

Notice that we did not pass any arguments to `print_a_and_b` in `my_main`. These are filled in from the configuration. We can however override these values in any way we like:

```
@ex.automain
def my_main():
    print_a_and_b()           # prints '10' and 'test'
    print_a_and_b(3)          # prints '3'  and 'test'
    print_a_and_b(3, 'foo')   # prints '3'  and 'foo'
    print_a_and_b(b='foo')    # prints '10' and 'foo'
```

Note: All functions decorated with `@ex.main`, `@ex.automain`, and `@ex.command` are also captured functions.

In case of multiple values for the same parameter the priority is:

1. explicitly passed arguments (both positional and keyword)
2. configuration values
3. default values

You will still get an appropriate error in the following cases:

- missing value that is not found in configuration
- unexpected keyword arguments
- too many positional arguments

Note: Be careful with naming your parameters, because configuration injection can hide some missing value errors from you, by (unintentionally) filling them in from the configuration.

Note: Configuration values should not be changed in a captured function because those changes cannot be recorded by the sacred experiment and can lead to confusing and unintended behaviour. Sacred will raise an Exception if you

try to write to a nested configuration item. You can disable this (not recommended) by setting `SETTINGS.CONFIG.READ_ONLY_CONFIG = False`.

Special Values

There are a couple of special parameters that captured functions can accept. These might change, and are not well documented yet, so be careful:

- `_config` : the whole configuration dict that is accessible for this function
- `_seed` : a seed that is different for every invocation (-> Controlling Randomness)
- `_rnd` : a random state seeded with `seed`
- `_log` : a logger for that function
- `_run` : the run object for the current run

Prefix

If you have some function that only needs to access some sub-dictionary of your configuration you can use the `prefix` parameter of `@ex.capture`:

```
ex = Experiment('prefix_demo')

@ex.config
def my_config1():
    dataset = {
        'filename': 'foo.txt',
        'path': '/tmp/'
    }

@ex.capture(prefix='dataset')
def print_me(filename, path):  # direct access to entries of the dataset dict
    print("filename =", filename)
    print("path =", path)
```

That way you have direct access to the items of that dictionary, but no access to the rest of the configuration anymore. It is a bit like setting a namespace for the function. Dotted notation for the prefix works as you would expect.

1.4 Command-Line Interface

Sacred provides a powerful command-line interface for every experiment out of box. All you have to do to use it is to either have a method decorated with `@ex.automain` or to put this block at the end of your file:

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

1.4.1 Configuration Updates

You can easily change any configuration entry using the powerful `with` argument on the command-line. Just put `with config=update` after your experiment call like this:

```
>>> ./example.py with 'a=10'
```

Or even multiple values just separated by a space:

```
>>> ./example.py with 'a=2.3' 'b="FooBar"' 'c=True'
```

Note: The single quotes (') around each statement are to make sure the bash does not interfere. In simple cases you can omit them:

```
>>> ./example.py with a=-1 b=2.0 c=True
```

But be careful especially with strings, because the outermost quotes get removed by bash. So for example all of the following values will be `int`:

```
>>> ./example.py with a=1 b="2" c='3'
```

You can use the standard python literal syntax to set numbers, bools, lists, dicts, strings and combinations thereof:

```
>>> ./example.py with 'my_list=[1, 2, 3]'
>>> ./example.py with 'nested_list=[["a", "b"], [2, 3], False]'
>>> ./example.py with 'my_dict={"a":1, "b":[-.2, "two"]}'
>>> ./example.py with 'alpha=-.3e-7'
>>> ./example.py with 'mask=0b111000'
>>> ./example.py with 'message="Hello Bob!"'
```

Note: Note however, that changing individual elements of a list is not supported now.

Dotted Notation

If you want to set individual entries of a dictionary you can use the dotted notation to do so. So if this is the ConfigScope of our experiment:

```
@ex.config
def cfg():
    d = {
        "foo": 1,
        "bar": 2,
    }
```

Then we could just change the `"foo"` entry of our dictionary to `100` like this:

```
>>> ./example.py with 'd.foo=100'
```

1.4.2 Named Updates

If there are any *Named Configurations* set up for an experiment, then you can apply them using the `with` argument. So for this experiment:

```
ex = Experiment('named_configs_demo')

@ex.config
```

(continues on next page)

(continued from previous page)

```
def cfg():
    a = 10
    b = 3 * a
    c = "foo"

@ex.named_config
def variant1():
    a = 100
    c = "bar"
```

The named configuration `variant1` can be applied like this:

```
>>> ./named_configs_demo.py with variant1
```

Multiple Named Updates

You can have multiple named configurations, and you can use as many of them as you like for any given run. But notice that the order in which you include them matters: The ones you put first will be evaluated first and the values they set might be overwritten by further named configurations.

Combination With Regular Updates

If you combine named updates with regular updates, and the latter have precedence. Sacred will first set an fix all regular updates and then run through all named updates in order, while keeping the regular updates fixed. The resulting configuration is then kept fixed and sacred runs through all normal configurations.

The following will set `a=23` first and then execute `variant1` treating `a` as fixed:

```
>>> ./named_configs_demo.py with variant1 'a=23'
```

So this configuration becomes `{'a':23, 'b':69, 'c':"bar"}`.

Config Files As Named Updates

Config files can be used as named updates, by just passing their name to the `with` argument. So assuming there is a `variant2.json` this works:

```
>>> ./named_configs_demo.py with variant2.json
```

Supported formats are the same as with *Config Files*.

If there should ever be a name-collision between a named config and a config file the latter takes precedence.

1.4.3 Commands

Apart from running the main function (the default command), the command-line interface also supports other (built-in or custom) commands. The name of the command has to be first on the commandline:

```
>>> ./my_demo.py COMMAND_NAME with seed=123
```

If the COMMAND_NAME is omitted it defaults to the main function, but the name of that function can also explicitly used as the name of the command. So for this experiment

```
@ex.automain
def my_main():
    return 42
```

the following two lines are equivalent:

```
>>> ./my_demo.py with seed=123
>>> ./my_demo.py my_main with seed=123
```

Print Config

To inspect the configuration of your experiment and see how changes from the command-line affect it you can use the `print_config` command. The full configuration of the experiment and all nested dictionaries will be printed with indentation. So lets say we added the dictionary from above to the `hello_config.py` example:

```
>>> ./hello_config print_config
INFO - hello_config - Running command 'print_config'
INFO - hello_config - Started
Configuration (modified, added, typechanged):
  message = 'Hello world!'
  recipient = 'world'
  seed = 946502320
  d:
    bar = 2
    foo = 1
INFO - hello_config - Completed after 0:00:00
```

This command is especially helpful to see how `with config=update` statements affect the configuration. It will highlight modified entries in **blue**, added entries in **green** and entries whose type has changed in **red**:

Change	Color
modified	blue
added	green
typechanged	red

But Sacred will also print warnings for all added and typechanged entries, to help you find typos and update mistakes:

```
>> ./hello_config.py print_config with 'recipient="Bob"' d.foo=True d.baz=3
WARNING - root - Added new config entry: "d.baz"
WARNING - root - Changed type of config entry "d.foo" from int to bool
INFO - hello_config - Running command 'print_config'
INFO - hello_config - Started
Configuration (modified, added, typechanged):
  message = 'Hello Bob!'
  recipient = 'Bob'         # blue
  seed = 676870791
  d:                        # blue
    bar = 2
    baz = 3                 # green
    foo = True              # red
INFO - hello_config - Completed after 0:00:00
```

Print Dependencies

The `print_dependencies` command shows the package dependencies, source files, and (optionally) the state of version control for the experiment. For example:

```
>> ./03_hello_config_scope.py print_dependencies
INFO - hello_cs - Running command 'print_dependencies'
INFO - hello_cs - Started
Dependencies:
  numpy                == 1.11.0
  sacred               == 0.7.0

Sources:
  03_hello_config_scope.py                    53cee32c9dc77870f7b39622434aff85

Version Control:
M git@github.com:IDSIA/sacred.git
↪bcdde712957570606ec5087b1748c60a89bb89e0

INFO - hello_cs - Completed after 0:00:00
```

Where the *Sources* section lists all discovered (or added) source files and their md5 hash. The *Version Control* section lists all discovered VCS repositories (ATM only git is supported), the current commit hash. The M at the beginning of the git line signals that the repository is currently dirty, i.e. has uncommitted changes.

Save Configuration

Use the `save_config` command for saving the current/updated configuration into a file:

```
./03_hello_config_scope.py save_config with recipient=Bob
```

This will store a file called `config.json` with the following content:

```
{
  "message": "Hello Bob!",
  "recipient": "Bob",
  "seed": 151625947
}
```

The filename can be configured by setting `config_filename` like this:

```
./03_hello_config_scope.py save_config with recipient=Bob config_filename=mine.yaml
```

The format for exporting the config is inferred from the filename and can be any format supported for *config files*.

Print Named Configs

The `print_named_configs` command prints all available named configurations. Function docstrings for named config functions are copied and displayed colored in **grey**. For example:

```
>> ./named_config print_named_configs
INFO - hello_config - Running command 'print_named_configs'
INFO - hello_config - Started
Named Configurations (doc):
  rude    # A rude named config
INFO - hello_config - Completed after 0:00:00
```

If no named configs are available for the experiment, an empty list is printed:

```
>> ./01_hello_world print_named_configs
INFO - 01_hello_world - Running command 'print_named_configs'
INFO - 01_hello_world - Started
Named Configurations (doc):
  No named configs
INFO - 01_hello_world - Completed after 0:00:00
```

Custom Commands

If you just run an experiment file it will execute the default command, that is the method you decorated with `@ex.main` or `@ex.automain`. But you can also add other commands to the experiment by using `@ex.command`:

```
from sacred import Experiment

ex = Experiment('custom_command')

@ex.command
def scream():
    """
    scream, and shout, and let it all out ...
    """
    print('AAAaaaaaaaahhhhhh...')

# ...
```

This command can then be run like this:

```
>> ./custom_command.py scream
INFO - custom_command - Running command 'scream'
INFO - custom_command - Started
AAAaaaaaaaahhhhhh...
INFO - custom_command - Completed after 0:00:00
```

It will also show up in the usage message and you can get the signature and the docstring by passing it to help:

```
>> ./custom_command.py help scream

scream()
    scream, and shout, and let it all out ...
```

Commands are of course also captured functions, so you can take arguments that will get filled in from the config, and you can use `with config=update` to change parameters from the command-line:

```
@ex.command
def greet(name):
    """
    Print a simple greet message.
    """
    print('Hello %s!' % name)
```

And call it like this:

```
>> ./custom_command.py greet with 'name="Bob"'
INFO - custom_command - Running command 'scream'
```

(continues on next page)

(continued from previous page)

```
INFO - custom_command - Started
Hello Bob!
INFO - custom_command - Completed after 0:00:00
```

Like other *Captured Functions*, commands also accept the `prefix` keyword-argument.

Many commands like `print_config` are helper functions, and should not trigger observers. This can be accomplished by passing `unobserved=True` to the decorator:

```
@ex.command(unobserved=True)
def helper(name):
    print('Running this command will not result in a DB entry!')
```

1.4.4 Flags

Help

`-h`	print usage
`--help`	

This prints a help/usage message for your experiment. It is equivalent to typing just `help`.

Comment

`-c COMMENT`	add a comment to this run
`--comment COMMENT`	

The `COMMENT` can be any text and will be stored with the run.

Logging Level

`-l LEVEL`	control the logging level
`--loglevel=LEVEL`	

With this flag you can adjust the logging level.

Level	Numeric value
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
NOTSET	0

See *Adjusting Log-Levels from command line* for more details.

MongoDB Observer

`-m DB`	add a MongoDB observer
`--mongo_db=DB`	

This flag can be used to add a MongoDB observer to your experiment. `DB` must be of the form `[host:port:]db_name[.collection][!priority]`.

See *Mongo Observer* for more details.

FileStorage Observer

`-F BASEDIR`	add a file storage observer
`--file_storage=BASEDIR`	

This flag can be used to add a file-storage observer to your experiment. `BASEDIR` specifies the directory the observer will use to store its files.

See *File Storage Observer* for more details.

TinyDB Observer

`-t BASEDIR`	add a TinyDB observer
`--tiny_db=BASEDIR`	

This flag can be used to add a TinyDB observer to your experiment. `BASEDIR` specifies the directory the observer will use to store its files.

See *TinyDB Observer* for more details.

Note: For this flag to work you need to have the tinydb, tinydb-serialization, and hashfs packages installed.

SQL Observer

`-s DB_URL`	add a SQL observer
`--sql=DB_URL`	

This flag can be used to add a SQL observer to your experiment. `DB_URL` must be parseable by the sqlalchemy package, which is typically means being of the form `dialect://username:password@host:port/database` (see their documentation for more detail).

Note: For this flag to work you need to have the sqlalchemy package installed.

See *Mongo Observer* for more details.

Debug Mode

`-d`	don't filter the stacktrace
`--debug`	

This flag deactivates the stacktrace filtering. You should usually not need this. It is mainly used for debugging experiments using a debugger (see *Debugging*).

PDB Debugging

`-D`	Enter post-mortem debugging with pdb on failure.
`--pdb`	

If this flag is set and an exception occurs, sacred automatically starts a `pdb` post-mortem debugger to investigate the error and interact with the stack (see *Debugging*).

Beat Interval

`-b BEAT_INTERVAL`	set the interval between heartbeat events
`--beat_interval=BEAT_INTERVAL`	

A running experiment regularly fires a *Heartbeat* event to synchronize the `info` dict (see *Saving Custom Information*). This flag can be used to change the interval from 10 sec (default) to `BEAT_INTERVAL` sec.

Unobserved

`-u`	Ignore all observers for this run.
`--unobserved`	

If this flag is set, sacred will remove all observers from the current run and also silence the warning for having no observers. This is useful for some quick tests or debugging runs.

Queue

`-q`	Only queue this run, do not start it.
`--queue`	

Instead of running the experiment, this will only create an entry in the database (or where the observers put it) with the status `QUEUED`. This entry will contain all the information about the experiment and the configuration. But the experiment will not be run. This can be useful to have some distributed workers fetch and start the queued up runs.

Priority

`-P PRIORITY`	The (numeric) priority for this run.
`--priority=PRIORITY`	

Enforce Clean

`-e`	Fail if any version control repository is dirty.
`--enforce_clean`	

This flag can be used to enforce that experiments are only being run on a clean repository, i.e. with no uncommitted changes.

Note: For this flag to work you need to have the GitPython package installed.

Print Config

`-p`	Always print the config first.
`--print_config`	

If this flag is set, sacred will always print the current configuration including modifications (like the *Print Config* command) before running the main method.

Name

`-n NAME`	Set the name for this run.
`--name=NAME`	

This option changes the name of the experiment before starting the run.

Id

`-i ID`	Set the id for this run.
`--id=ID`	

This option changes the id of the experiment before starting the run.

Capture Mode

`-C CAPTURE_MODE`	Control the way stdout and stderr are captured.
`--capture=CAPTURE_MODE`	

This option controls how sacred captures outputs to stdout and stderr. Possible values for `CAPTURE_MODE` are `no`, `sys` (default under Windows), or `fd` (default for Linux/OSX). For more information see *here*.

1.4.5 Custom Flags

It is possible to add custom flags to an experiment by inheriting from `sacred.cli_option` like this:

```
from sacred import cli_option, Experiment


@cli_option('-o', '--own-flag', is_flag=True)
def my_option(args, run):
    # useless feature: add some string to the info dict
    run.info['some'] = 'prepopulation of the info dict'


ex = Experiment('my pretty exp', additional_cli_options=[my_option])

@ex.run
def my_main():
    ...
```

The name of the flag is taken from the decorator arguments and here would be `-o/--own-flag`. The documentation for the flag is taken from the docstring. The decorated function is called after the `Run` object has been created, but before it has been started.

In this case the `args` parameter will be always be `True`. But it is also possible to add a flag which takes an argument, by turning off the `is_flag` option (which is the default):

```
from sacred import cli_option, Experiment


@cli_option('-o', '--own-flag')  # is_flag=False is the default
def improved_option(args, run):
    """
    This is my even better personal flag
    The cool message that gets saved to info.
    """
    run.info['some'] = args


ex = Experiment('my pretty exp', additional_cli_options=[improved_option])

@ex.run
def my_main():
    ...
```

Here the flag would be `-o MESSAGE` / `--own-flag=MESSAGE` and the `args` parameter of the `apply` function would contain the `MESSAGE` as a string.

1.5 Collected Information

Sacred collects a lot of information about the runs of an experiment and reports them to the observers. This section provides an overview over the collected information and ways to customize it.

1.5.1 Configuration

Arguably the most important information about a run is its *Configuration*. Sacred collects the final configuration that results after incorporating named configs and configuration updates. It also keeps track of information about what changes have occurred and whether they are suspicious. Suspicious changes include adding configuration entries that are not used anywhere, or typechanges of existing entries.

The easiest way to inspect this information is from the commandline using the *Print Config* command or alternatively the `-p / --print_config` *flag*. The config is also passed to the observers as part of the *started_event* or the *queued_event*. It is also available through the *The Run Object* through `run.config` and `run.config_modifications`. Finally the individual values can be directly accessed during a run through *Accessing Config Entries* or also the whole config using the `_config` *special value*.

1.5.2 Experiment Info

The experiment_info includes the name and the base directory of the experiment, a list of source files, a list of dependencies, and, optionally, information about its git repository.

This information is available as a dictionary from the *Run object* through `run.experiment_info`. And it is also passed to (and stored by) the observers as part of the *started_event* or the *queued_event*.

Source Code

To help ensure reproducibility, Sacred automatically discovers the python sources of an experiment and stores them alongside the run. That way the version of the code used for running the experiment is always available with the run.

The auto-discovery is using inspection of the imported modules and comparing them to the local file structure. This process should work in >95% of the use cases. But in case it fails one can also manually add source files using *`add_source_file()`*.

The list of sources is accessible through `run.experiment_info['sources']`. It is a list of tuples of the form `(filename, md5sum)`. It can also be inspected using the *Print Dependencies* command.

Version Control

If the experiment is part of a Git repository, Sacred will also collect the url of the repository, the current commit hash and whether the repository is dirty (has uncommitted changes).

This information can be inspected using the *Print Dependencies* command. But it is also available from `run.experiment_info['repositories']`, as a list of dictionaries of the form `{'url': URL, 'commit': COMMIT_HASH, 'dirty': True}`.

To disable this, pass `save_git_info=False` to the `Experiment` or `Ingredient` constructor.

Dependencies

Sacred also tries to auto-discover the package dependencies of the experiment. This again is done using inspection of the imported modules and trying to figure out their versions. Like the source-code autodiscovery, this should work most of the time. But it is also possible to manually add dependencies using *`add_package_dependency()`*.

The easiest way to inspect the discovered package dependencies is via the *Print Dependencies* command. But they are also accessible from `run.experiment_info['dependencies']` as a list of strings of the form `package==version`.

1.5.3 Host Info

Some basic information about the machine that runs the experiment (the host) is also collected. The default host info includes:

Key	Description
cpu	The CPU model
hostname	The name of the machine
os	Info about the operating system
python_version	Version of python
gpu	Information about NVidia GPUs (if any)
ENV	captured ENVIRONMENT variables (if set)

Host information is available from the *The Run Object* through `run.host_info`. It is sent to the observers by the *started_event*.

The list of captured ENVIRONMENT variables (empty by default) can be extended by appending the relevant keys to `sacred.SETTINGS.HOST_INFO.CAPTURED_ENV`.

It is possible to extend the host information with custom functions decorated by `host_info_gatherer()` like this:

```
from sacred import host_info_gatherer
from sacred import Experiment


@host_info_gatherer('host IP address')
def ip():
    import socket
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.connect(("8.8.8.8", 80))
    ip = s.getsockname()[0]
    s.close()
    return ip


ex = Experiment('cool experiment',
                additional_host_info=[ip])

@ex.main
def my_main():
    ...
```

This example will create an `host IP address` entry in the host_info containing the IP address of the machine.

1.5.4 Live Information

While an experiment is running, sacred collects some live information and reports them in regular intervals (default 10sec) to the observers via the *heartbeat_event*. This includes the captured `stdout` and `stderr` and the contents of the *Info Dict* which can be used to store custom information like training curves. It also includes the current intermediate result if set. It can be set using the `_run` object:

```
@ex.capture
def some_function(_run):
    ...
```

(continues on next page)

(continued from previous page)

```
    _run.result = 42
    ...
```

Output capturing in sacred can be done in different modes. On linux the default is to capture on the file descriptor level, which means that it should even capture outputs made from called c-functions or subprocesses. On Windows the default mode is `sys` which only captures outputs made from within python.

Note that, the captured output behaves differently from a console in that it doesn't by default interpret control characters like backspace (`'\b'`) or carriage return (`'\r'`). As an effect, some updating progressbars or the like might me more verbose than intended. This behavior can be changed by adding a custom filter to the captured output. To interpret control characters like a console this would do:

```
from sacred.utils import apply_backspaces_and_linefeeds

ex.captured_out_filter = apply_backspaces_and_linefeeds
```

Long running and verbose experiments can overload the observer's storage backend. For example, the MongoObserver is limited to 16 MB per run, which can result in experiments being unexpectedly terminated. To avoid this you can turn of output capturing by applying a custom filter like so

```
ex.captured_out_filter = lambda captured_output: "Output capturing turned off."
```

Metrics API

You might want to measure various values during your experiments, such as the progress of prediction accuracy over training steps.

Sacred supports tracking of numerical series (e.g. int, float) using the Metrics API. To access the API in experiments, the experiment must be running and the variable referencing the current experiment or run must be available in the scope. The `_run.log_scalar(metric_name, value, step)` method takes a metric name (e.g. "training.loss"), the measured value and the iteration step in which the value was taken. If no step is specified, a counter that increments by one automatically is set up for each metric.

Step should be an integer describing the position of the value in the series. Steps can be numbered either sequentially 0, 1, 2, 3, ... or they may be given a different meaning, for instance the current iteration round. The earlier behaviour can be achieved automatically when omitting the step parameter. The latter approach is useful when logging occurs only every e.g. 10th iteration: The step can be first 10, then 20, etc. In any case, the numbers should form an increasing sequence.

```
@ex.automain
def example_metrics(_run):
    counter = 0
    while counter < 20:
        counter+=1
        value = counter
        ms_to_wait = random.randint(5, 5000)
        time.sleep(ms_to_wait/1000)
        # This will add an entry for training.loss metric in every second iteration.
        # The resulting sequence of steps for training.loss will be 0, 2, 4, ...
        if counter % 2 == 0:
            _run.log_scalar("training.loss", value * 1.5, counter)
        # Implicit step counter (0, 1, 2, 3, ...)
        # incremented with each call for training.accuracy:
        _run.log_scalar("training.accuracy", value * 2)
        # Another option is to use the Experiment object (must be running)
```

(continues on next page)

(continued from previous page)

```
        # The training.diff has its own step counter (0, 1, 2, ...) too
        ex.log_scalar("training.diff", value * 2)
```

Currently, the information is collected only by two observers: the *Mongo Observer* and the *File Storage Observer*. For the Mongo Observer, metrics are stored in the `metrics` collection of MongoDB and are identified by their name (e.g. "training.loss") and the experiment run id they belong to. For the *File Storage Observer*, metrics are stored in the file `metrics.json` in the run id's directory and are organized by metric name (e.g. "training.loss").

Metrics Records

A metric record is composed of the metric name, the id of the corresponding experiment run, and of the measured values, arranged in an array in the order they were captured using the `log_scalar(...)` function. For the value located in the i-th index (`metric["values"][i]`), the step number can be found in `metric["steps"][i]` and the time of the measurement in `metric["timestamps"][i]`.

Key	Description
`_id`	Unique identifier
`name`	The name of the metric (e.g. training.loss)
`run_id`	The identifier of the run (`_id` in the runs collection)
`steps`	Array of steps (e.g. `[0, 1, 2, 3, 4]`)
`values`	Array of measured values
`timestamps`	Array of times of capturing the individual measurements

1.5.5 Resources and Artifacts

It is possible to add files to an experiment, that will then be added to the database (or stored by whatever observer you are using). Apart from the source files (that are automatically added) there are two more types of files: Resources and Artifacts.

Resources

Resources are files that are needed by the experiment to run, such as datasets or further configuration files. If a file is opened through `open_resource()` then sacred will collect information about that file and send it to the observers. The observers will then store the file, but not duplicate it, if it is already stored.

Artifacts

Artifacts, on the other hand, are files that are produced by a run. They might, for example, contain a detailed dump of the results or the weights of a trained model. They can be added to the run by `add_artifact()` Artifacts are stored with a name, which (if it isn't explicitly specified) defaults to the filename.

1.5.6 Bookkeeping

Finally, Sacred stores some additional bookkeeping information, and some custom meta information about the runs. This information is reported to the observers as soon as it is available, and can also be accessed through the *Run object* using the following keys:

Key	Description
`start_time`	The datetime when this run was started
`stop_time`	The datetime when this run stopped
`heartbeat_time`	The last time this run communicated with the observers
`status`	The status of the run (see below)
`fail_trace`	The stacktrace of an exception that occurred (if so)
`result`	The return value of the main function (if successful)

Note: All stored times are UTC times!

Status

The status describes in what state a run currently is and takes one of the following values:

Status	Description
`QUEUED`	The run was just *queued* and not run yet
`RUNNING`	Currently running (but see below)
`COMPLETED`	Completed successfully
`FAILED`	The run failed due to an exception
`INTERRUPTED`	The run was cancelled with a `KeyboardInterrupt`
`TIMED_OUT`	The run was aborted using a *`TimeoutInterrupt`*
[custom]	A custom py:class:~*sacred.utils.SacredInterrupt* occurred

If a run crashes in a way that doesn't allow Sacred to tell the observers (e.g. power outage, kernel panic, ...), then the status of the crashed run will still be `RUNNING`. To find these *dead* runs, one can look at the `heartbeat_time` of the runs with a `RUNNING` status: If the `heartbeat_time` lies significantly longer in the past than the heartbeat interval (default 10sec), then the run can be considered `DEAD`.

Meta Information

The meta-information is meant as a place to store custom information about a run once in the beginning. It can be added to the run by passing it to `run()`, but some commandline flags or tools also add meta information. It is reported to the observers as part of the *started_event* or the *queued_event*. It can also be accessed as a dictionary through the `meta_info` property of the *Run object*. The builtin usecases include:

Key	Description
`command`	The name of the command that is being run
`options`	A dictionary with all the commandline options
`comment`	A comment for that run (added by the *comment flag*)
`priority`	A priority for scheduling queued runs (added by the *priority flag*)
`queue_time`	The datetime when this run was queued (stored automatically)

1.6 Observing an Experiment

When you run an experiment you want to keep track of enough information, such that you can analyse the results, and reproduce them if needed. Sacred helps you doing that by providing an *Observer Interface* for your experiments. By

attaching an Observer you can gather all the information about the run even while it is still running. Observers have a `priority` attribute, and are run in order of descending priority. The first observer determines the `_id` of the run, or it can be set by the command line option `--id`.

At the moment there are seven observers that are shipped with Sacred:

- The main one is the *Mongo Observer* which stores all information in a MongoDB.
- The *File Storage Observer* stores the run information as files in a given directory and will therefore only work locally.
- The *TinyDB Observer* provides another local way of observing experiments by using tinydb to store run information in a JSON file.
- The *SQL Observer* connects to any SQL database and will store the relevant information there.
- The *S3 Observer* stores run information in an AWS S3 bucket, within a given prefix/directory
- The gcs_observer stores run information in a provided Google Cloud Storage bucket, within a given prefix/directory
- The *Queue Observer* can be used to wrap any of the above observers. It will put the processing of observed events on a fault-tolerant queue in a background process. This is useful for observers that rely on external services such as a database that might be temporarily unavailable.

But if you want the run information stored some other way, it is easy to write your own *Custom Observer*.

1.6.1 Mongo Observer

Note: Requires the pymongo package. Install with `pip install pymongo`.

The MongoObserver is the recommended way of storing the run information from Sacred. MongoDB allows very powerful querying of the entries that can deal with almost any structure of the configuration and the custom info. Furthermore it is easy to set-up and allows to connect to a central remote DB. Most tools for further analysing the data collected by Sacred build upon this observer.

Adding a MongoObserver

You can add a MongoObserver from the command-line via the `-m MY_DB` flag:

```
>> ./my_experiment.py -m MY_DB
```

Here `MY_DB` is just the name of the database inside MongoDB that you want the information to be stored in. To make MongoObserver work with remote MongoDBs you have to pass a URL with a port:

```
>> ./my_experiment.py -m HOST:PORT:MY_DB
>> ./my_experiment.py -m 192.168.1.1:27017:MY_DB
>> ./my_experiment.py -m my.server.org:12345:MY_DB
```

You can also add it from code like this:

```
from sacred.observers import MongoObserver

ex.observers.append(MongoObserver())
```

Or with server and port:

```
from sacred.observers import MongoObserver

ex.observers.append(MongoObserver(url='my.server.org:27017',
                                  db_name='MY_DB'))
```

This assumes you either have a local MongoDB running or have access to it over network without authentication. (See here on how to install)

You can setup MongoDB easily with Docker. See the instructions in *Docker Setup* .

Authentication

If you need authentication a little more work might be necessary. First you have to decide which authentication protocol you want to use. If it can be done by just using the `MongoDB URI` then just pass that, e.g.:

```
from sacred.observers import MongoObserver

ex.observers.append(MongoObserver(
    url='mongodb://user:password@example.com/the_database?authMechanism=SCRAM-SHA-1',
    db_name='MY_DB'))
```

If additional arguments need to be passed to the MongoClient they can just be included:

```
ex.observers.append(MongoObserver(
    url="mongodb://<X.509 derived username>@example.com/?authMechanism=MONGODB-X509",
    db_name='MY_DB',
    ssl=True,
    ssl_certfile='/path/to/client.pem',
    ssl_cert_reqs=ssl.CERT_REQUIRED,
    ssl_ca_certs='/path/to/ca.pem'))
```

Database Entry

The MongoObserver creates three collections to store information. The first, `runs` (that name can be changed), is the main collection that contains one entry for each run. The other two (`fs.files`, `fs.chunks`) are used to store associated files in the database (compare GridFS).

Note: This is the new database layout introduced in version 0.7.0. Before that there was a common prefix *default* for all collections.

So here is an example entry in the `runs` collection:

```
> db.runs.find()[0]
{
    "_id" : ObjectId("5507248a1239672ae04591e2"),
    "format" : "MongoObserver-0.7.0",
    "status" : "COMPLETED",
    "result" : null,
    "start_time" : ISODate("2016-07-11T14:50:14.473Z"),
    "heartbeat" : ISODate("2015-03-16T19:44:26.530Z"),
    "stop_time" : ISODate("2015-03-16T19:44:26.532Z"),
    "config" : {
        "message" : "Hello world!",
```

(continues on next page)

(continued from previous page)

```
        "seed" : 909032414,
        "recipient" : "world"
    },
    "info" : { },
    "resources" : [ ],
    "artifacts" : [ ],
    "captured_out" : "Hello world!\n",
    "experiment" : {
        "name" : "hello_cs",
        "base_dir" : "$(HOME)/sacred/examples/"
        "dependencies" : ["numpy==1.9.1", "sacred==0.7.0"],
        "sources" : [
            [
                "03_hello_config_scope.py",
                ObjectId("5507248a1239672ae04591e3")
            ]
        ],
        "repositories" : [{
            "url" : "git@github.com:IDSIA/sacred.git"
                            "dirty" : false,
                            "commit" : "d88deb2555bb311eb779f81f22fe16dd3b703527"}]
    },
    "host" : {
        "os" : ["Linux",
                "Linux-3.13.0-46-generic-x86_64-with-Ubuntu-14.04-trusty"],
        "cpu" : "Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz",
        "hostname" : "MyAwesomeMachine",
        "python_version" : "3.4.0"
    },
}
```

As you can see a lot of relevant information is being stored, among it the used configuration, automatically detected package dependencies and information about the host.

If we take a look at the `fs.files` collection we can also see, that it stored the sourcecode of the experiment in the database:

```
> db.fs.files.find()[0]
{
    "_id" : ObjectId("5507248a1239672ae04591e3"),
    "filename" : "$(HOME)/sacred/examples/03_hello_config_scope.py",
    "md5" : "897b2144880e2ee8e34775929943f496",
    "chunkSize" : 261120,
    "length" : 1526,
    "uploadDate" : ISODate("2016-07-11T12:50:14.522Z")
}
```

1.6.2 File Storage Observer

The FileStorageObserver is the most basic observer and requires the least amount of setup. It is mostly meant for preliminary experiments and cases when setting up a database is difficult or impossible. But in combination with the template rendering integration it can be very helpful.

Adding a FileStorageObserver

The FileStorageObserver can be added from the command-line via the `-F BASEDIR` and `--file_storage=BASEDIR` flags:

```
>> ./my_experiment.py -F BASEDIR
>> ./my_experiment.py --file_storage=BASEDIR
```

Here `BASEDIR` is the name of the directory in which all the subdirectories for individual runs will be created.

You can, of course, also add it from code like this:

```
from sacred.observers import FileStorageObserver

ex.observers.append(FileStorageObserver('my_runs'))
```

Directory Structure

The FileStorageObserver creates a separate sub-directory for each run and stores several files in there:

```
my_runs/
    run_3mdq4amp/
        config.json
        cout.txt
        info.json
        run.json
    run_zw82a7xg/
        ...
    ...
```

`config.json` contains the JSON-serialized version of the configuration and `cout.txt` the captured output. The main information is stored in `run.json` and is very similar to the database entries from the *Mongo Observer*:

```
{
  "command": "main",
  "status": "COMPLETED",
  "start_time": "2016-07-11T15:35:14.765152",
  "heartbeat": "2016-07-11T15:35:14.766793",
  "stop_time": "2016-07-11T15:35:14.768465",
  "result": null,
  "experiment": {
    "base_dir": "/home/greff/Programming/sacred/examples",
    "dependencies": [
      "numpy==1.11.0",
      "sacred==0.6.9"],
    "name": "hello_cs",
    "repositories": [{
        "commit": "d88deb2555bb311eb779f81f22fe16dd3b703527",
        "dirty": false,
        "url": "git@github.com:IDSIA/sacred.git"}],
    "sources": [
      ["03_hello_config_scope.py",
       "_sources/03_hello_config_scope_897b2144880e2ee8e34775929943f496.py"]]
  },
  "host": {
    "cpu": "Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz",
```

(continues on next page)

(continued from previous page)

```
    "hostname": "Liz",
    "os": ["Linux",
           "Linux-3.19.0-58-generic-x86_64-with-Ubuntu-15.04-vivid"],
    "python_version": "3.4.3"
  },
  "artifacts": [],
  "resources": [],
  "meta": {},
}
```

In addition to that there is an `info.json` file holding *Saving Custom Information* (if existing) and all the *Artifacts*.

The FileStorageObserver also stores a snapshot of the source-code in a separate `my_runs/_sources` directory, and *Resources* in `my_runs/_resources` (if present). Their filenames are stored in the `run.json` file such that the corresponding files can be easily linked to their respective run.

Storing source-code in this way can be disabled by passing `copy_sources=False` when creating the FileStorageObserver. Copying any *Resources* that are already present in *my_runs/*, but not present in *my_runs/_resources/* (for example, a resource that is the output of another run), can be disabled my passing `copy_artifacts=False` when creating the FileStorageObserver.

Template Rendering

In addition to these basic files, the FileStorageObserver can also generate a report for each run from a given template file. The prerequisite for this is that the mako package is installed and a `my_runs/template.html` file needs to exist. The file can be located somewhere else, but then the filename must be passed to the FileStorageObserver like this:

```
from sacred.observers import FileStorageObserver

ex.observers.append(FileStorageObserver('my_runs', template='/custom/template.txt'))
```

The FileStorageObserver will then render that template into a `report.html`/`report.txt` file in the respective run directory. `mako` is a very powerful templating engine that can execute arbitrary python-code, so be careful about the templates you use. For an example see `sacred/examples/my_runs/template.html`.

1.6.3 TinyDB Observer

Note: requires the tinydb, tinydb-serialization, and hashfs packages installed.

The TinyDbObserver uses the tinydb library to provides an alternative to storing results in MongoDB whilst still allowing results to be stored in a document like database. This observer uses TinyDB to store the metadata about an observed run in a JSON file.

The TinyDbObserver also makes use of the hashfs hashfs library to store artifacts, resources and source code files associated with a run. Storing results like this provides an easy way to lookup associated files for a run bases on their hash, and ensures no duplicate files are stored.

The main drawback of storing files in this way is that they are not easy to manually inspect, as their path names are now the hash of their content. Therefore, to aid in retrieving data and files stored by the TinyDbObserver, a TinyDbReader class is provided to allow for easier querying and retrieval of the results. This ability to store metadata and files in a way that can be queried locally is the main advantage of the TinyDbObserver observer compared to the FileStorageObserver.

The TinyDbObserver is designed to be a simple, scalable way to store and query results as a single user on a local file system, either for personal experimentation or when setting up a larger database configuration is not desirable.

Adding a TinyDbObserver

The TinyDbObserver can be added from the command-line via the `-t BASEDIR` and `--tiny_db=BASEDIR` flags:

```
>> ./my_experiment.py -t BASEDIR
>> ./my_experiment.py --tiny_db=BASEDIR
```

Here `BASEDIR` specifies the directory in which the TinyDB JSON file and hashfs filesytem will be created. All intermediate directories are created with the default being to create a directory called `runs_db` in the current directory.

Alternatively, you can also add the observer from code like this:

```
from sacred.observers import TinyDbObserver

ex.observers.append(TinyDbObserver('my_runs'))

# You can also create this observer from a HashFS and
# TinyDB object directly with:
ex.observers.append(TinyDbObserver.create_from(my_db, my_fs))
```

Directory Structure

The TinyDbObserver creates a directory structure as follows:

```
my_runs/
    metadata.json
    hashfs/
```

`metadata.json` contains the JSON-serialized metadata in the TinyDB format. Each entry is very similar to the database entries from the *Mongo Observer*:

```
{
  "_id": "2118c70ef274497f90b7eb72dcf34598",
  "artifacts": [],
  "captured_out": "",
  "command": "run",
  "config": {
    "C": 1,
    "gamma": 0.7,
    "seed": 191164913
  },
  "experiment": {
    "base_dir": "/Users/chris/Dropbox/projects/dev/sacred-tinydb",
    "dependencies": [
      "IPython==5.1.0",
      "numpy==1.11.2",
      "sacred==0.7b0",
      "sklearn==0.18"
    ],
    "name": "iris_rbf_svm",
    "repositories": [],
    "sources": [
```

(continues on next page)

(continued from previous page)

```
      [
        "test_exp.py",
        "6f4294124f7697655f9fd1f7d4e7798b",
        "{TinyFile}:\"6f4294124f7697655f9fd1f7d4e7798b\""
      ]
    ]
  },
  "format": "TinyDbObserver-0.7b0",
  "heartbeat": "{TinyDate}:2016-11-12T01:18:00.228352",
  "host": {
    "cpu": "Intel(R) Core(TM)2 Duo CPU     P8600  @ 2.40GHz",
    "hostname": "phoebe",
    "os": [
      "Darwin",
      "Darwin-15.5.0-x86_64-i386-64bit"
    ],
    "python_version": "3.5.2"
  },
  "info": {},
  "meta": {},
  "resources": [],
  "result": 0.9833333333333333,
  "start_time": "{TinyDate}:2016-11-12T01:18:00.197311",
  "status": "COMPLETED",
  "stop_time": "{TinyDate}:2016-11-12T01:18:00.337519"
}
```

The elements in the above example are taken from a generated JSON file, where those prefixed with `{TinyData}` will be converted into python datetime objects upon reading them back in. Likewise those prefixed with `{TinyFile}` will be converted into a file object opened in read mode for the associated source, artifact or resource file.

The files referenced in either the sources, artifacts or resources sections are stored in a location according to the hash of their contents under the `hashfs/` directory. The hashed file system is setup to create three directories from the first 6 characters of the hash, with the rest of the hash making up the file name. The stored source file is therefore located at

```
my_runs/
    metadata.json
    hashfs/
        59/
            ab/
                16/
                    5b3579a1869399b4838be2a125
```

A file handle, serialised with the tag `{TinyFile}` in the JSON file, is included in the metadata alongside individual source files, artifacts or resources as a convenient way to access the file content.

The TinyDB Reader

To make querying and stored results easier, a TinyDbReader class is provided. Create a class instance by passing the path to the root directory of the TinyDbObserver.

```
from sacred.observers import TinyDbReader

reader = TinyDbReader('my_runs')
```

The TinyDbReader class provides three main methods for retrieving data:

- `.fetch_metadata()` will return all metadata associated with an experiment.
- `.fetch_files()` will return a dictionary of file handles for the sources, artifacts and resources.
- `.fetch_report()` will will return all metadata rendered in a summary report.

All three provide a similar API, allowing the search for records by index, by experiment name, or by using a TinyDB search query. To do so specify one of the following arguments to the above methods:

- `indices` accepts either a single integer or a list of integers and works like list indexing, retrieving experiments in the order they were run. e.g. `indices=0` will get the first or oldest experiment, and `indices=-1` will get the latest experiment to run.
- `exp_name` accepts a string and retrieves any experiment that contains that string in its name. Also works with regular expressions.
- `query` accepts a TinyDB query object and returns all experiments that match it. Refer to the TinyDB documentation for details on the API.

Retrieving Files

To get the files from the last experimental run:

```
results = reader.fetch_files(indices=-1)
```

The results object is a list of dictionaries, each containing the date the experiment started, the experiment id, the experiment name, as well as nested dictionaries for the sources, artifacts and resources if they are present for the experiment. For each of these nested dictionaries, the key is the file name, and the value is a file handle opened for reading that file.

```
[{'date': datetime.datetime(2016, 11, 12, 1, 36, 54, 970229),
  'exp_id': '68b71b5c009e4f6a887479cdda7a93a0',
  'exp_name': 'iris_rbf_svm',
  'sources': {'test_exp.py': <BufferedReaderWrapper name='...'>}}]
```

Individual files can therefore be accessed with,

```
results = reader.fetch_files(indices=-1)
f = results[0]['sources']['test_exp.py']
f.read()
```

Depending on whether the file contents is text or binary data, it can then either be printed to console or visualised in an appropriate library e.g. Pillow for images. The content can also be written back out to disk and inspected in an external program.

Summary Report

Often you may want to see a high level summary of an experimental run, such as the config used the results, and any inputs, dependencies and other artifacts generated. The `.fetch_report()` method is designed to provide these rendered as a simple text based report.

To get the report for the last experiment simple run,

```
results = reader.fetch_report(indices=-1)
print(results[0])
```

```
--------------------------------------------------
Experiment: iris_rbf_svm
--------------------------------------------------
ID: 68b71b5c009e4f6a887479cdda7a93a0
Date: Sat 12 Nov 2016    Duration: 0:0:0.1

Parameters:
    C: 1.0
    gamma: 0.7
    seed: 816200523

Result:
    0.9666666666666667

Dependencies:
    IPython==5.1.0
    numpy==1.11.2
    sacred==0.7b0
    sacred.observers.tinydb_hashfs==0.7b0
    sklearn==0.18

Resources:
    None

Source Files:
    test_exp.py

Outputs:
    None
```

1.6.4 SQL Observer

The SqlObserver saves all the relevant information in a set of SQL tables. It requires the sqlalchemy package to be installed.

Adding a SqlObserver

The SqlObserver can be added from the command-line via the `-s DB_URL` and `--sql=DB_URL` flags:

```
>> ./my_experiment.py -s DB_URL
>> ./my_experiment.py --sql=DB_URL
```

Here `DB_URL` is a url specifying the dialect and server of the SQL database to connect to. For example:

- PostgreSQL: `postgresql://scott:tiger@localhost/mydatabase`
- MySQL: `mysql://scott:tiger@localhost/foo`
- SqlLite: `sqlite:///foo.db`

For more information on the database-urls see the sqlalchemy documentation.

To add a SqlObserver from python code do:

```
from sacred.observers import SqlObserver
```

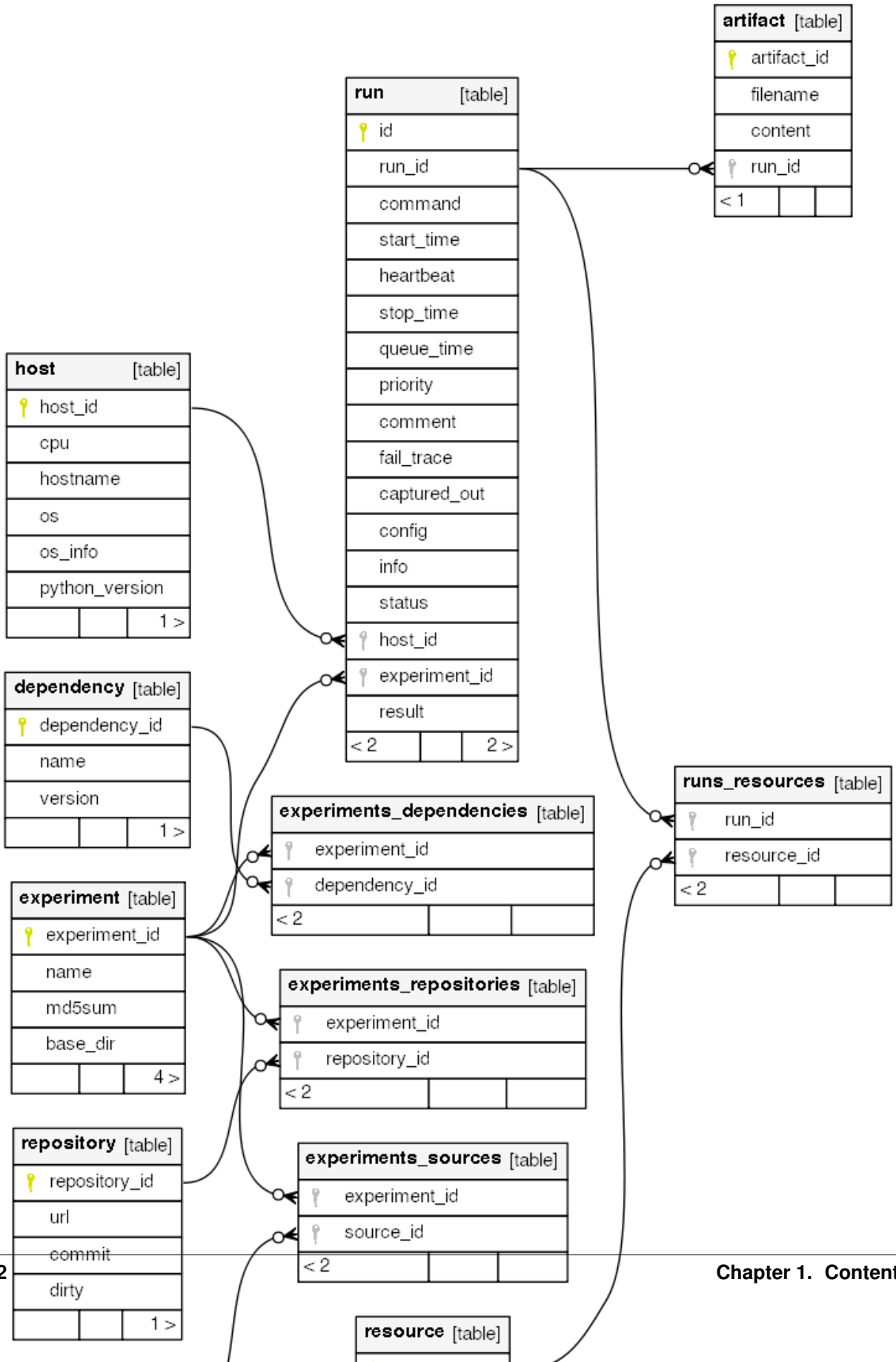
(continues on next page)

(continued from previous page)

```
ex.observers.append(SqlObserver('sqlite:///foo.db'))

# It's also possible to instantiate a SqlObserver with an existing
# engine and session with:
ex.observers.append(SqlObserver.create_from(my_engine, my_session))
```

Schema


artifact [table]
artifact_id
filename
content
run_id
< 1
run [table]
id
run_id
command
start_time
heartbeat
stop_time
queue_time
priority
comment
fail_trace
captured_out
config
info
status
host_id
experiment_id
result
< 2
2 >
host [table]
host_id
cpu
hostname
os
os_info
python_version
1 >
dependency [table]
dependency_id
name
version
1 >
experiments_dependencies [table]
experiment_id
dependency_id
< 2
runs_resources [table]
run_id
resource_id
< 2
experiment [table]
experiment_id
name
md5sum
base_dir
4 >
experiments_repositories [table]
experiment_id
repository_id
< 2
repository [table]
repository_id
url
commit
dirty
1 >
experiments_sources [table]
experiment_id
source_id
< 2
resource [table]

1.6.5 S3 Observer

The S3Observer stores run information in a designated prefix location within a S3 bucket, either by using an existing bucket, or creating a new one. Using the S3Observer requires that boto3 be installed, and also that an AWS config file is created with a user's Access Key and Secret Key. An easy way to do this is by installing AWS command line tools (`pip install awscli`) and running `aws configure`.

Adding a S3Observer

To create an S3Observer in Python:

```
from sacred.observers import S3Observer
ex.observers.append(S3Observer(bucket='my-awesome-bucket',
                               basedir='/my-project/my-cool-experiment/'))
```

By default, an S3Observer will use the region that is set in your AWS config file, but if you'd prefer to pass in a specific region, you can use the `region` parameter of create to do so. If you try to create an S3Observer without this parameter, and with region not set in your config file, it will error out at the point of the observer object being created.

Directory Structure

S3Observers follow the same conventions as FileStorageObservers when it comes to directory structure within a S3 bucket: within `s3://<bucket>/basedir/` numeric run directories will be created in ascending order, and each run directory will contain the files specified within the FileStorageObserver Directory Structure documentation above.

1.6.6 Google Cloud Storage Observer

Note: Requires the google cloud storage package. Install with `pip install google-cloud-storage`.

The Google Cloud Storage Observer allows for experiments to be logged into cloud storage buckets provided by Google. In order to use this observer, the user must have created a bucket on the service prior to the running an experiment using this observer.

Adding a GoogleCloudStorageObserver

To create an GoogleCloudStorageObserver in Python:

```
from sacred.observers import GoogleCloudStorageObserver
ex.observers.append(GoogleCloudStorageObserver(bucket='bucket-name',
                                               basedir='/experiment-name/'))
```

In order for the observer to correctly connect to the provided bucket, The environment variable '' GOOGLE_APPLICATION_CREDENTIALS'' needs to be set by the user. This variable should point to a valid JSON file containing Google authorisation credentials (see: Google Cloud authentication).

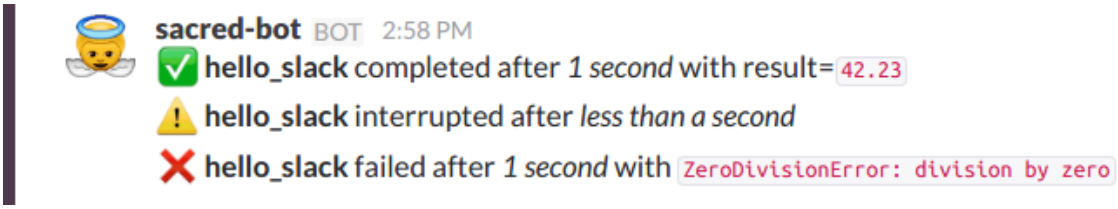
Directory Structure

GoogleCloudStorageObserver follow the same conventions as FileStorageObservers when it comes to directory structure within a bucket: within `gs://<bucket>/basedir/` numeric run directories will be created in ascending

order, and each run directory will contain the files specified within the FileStorageObserver Directory Structure documentation above.

1.6.7 Slack Observer

The `SlackObserver` sends a message to Slack using an incoming webhook everytime an experiment stops:



It requires the requests package to be installed and the `webhook_url` of the incoming webhook configured in Slack. This url is something you shouldn't share with others, so the recommended way of adding a SlackObserver is from a configuration file:

```
from sacred.observers import SlackObserver

slack_obs = SlackObserver.from_config('slack.json')
ex.observers.append(slack_obs)

# You can also instantiate it directly without a config file:
 slack_obs = SlackObserver(my_webhook_url)
```

Where `slack.json` at least specifies the `webhook_url`:

```
# Content of file 'slack.json':
{
    "webhook_url": "https://hooks.slack.com/services/T00000000/B00000000/
↪XXXXXXXXXXXXXXXXXXXXXXXX"
}
```

But it can optionally also customize the other attributes:

```
# Content of file 'slack.json':
{
    "webhook_url": "https://hooks.slack.com/services/T00000000/B00000000/
↪XXXXXXXXXXXXXXXXXXXXXXXX",
    "icon": ":imp:",
    "bot_name": "my-sacred-bot",
    "completed_text": "YAY! {ex_info[name]} completed with result=`{result}`",
    "interrupted_text": null,
    "failed_text": "Oh noes! {ex_info[name]} failed saying `{error}`"
}
```

1.6.8 Telegram Observer

The `TelegramObserver` sends status updates to Telegram using their Python Telegram Bot API which obviously has to be installed to use this observer.

```
pip install --upgrade python-telegram-bot
```

Before using this observer, three steps need to be taken:

- *Create the bot with @BotFather <https://core.telegram.org/bots#6-botfather>*
- Write **to** the newly-created bot, since only users can initiate conversations with telegram bots.
- Retrieve the `chat_id` for the chat the bot will send updates to.

The last step can be accomplished using the following script:

```
import  telegram

TOKEN = 'token obtained from @BotFather'

bot = telegram.Bot(TOKEN)
for u in bot.get_updates():
    print('{}: [{}] {}'.format(u.message.date, u.message.chat_id, u.message.text))
```

As with the `SlackObserver`, the `TelegramObserver` needs to be provided with a json, yaml or pickle file containing...

- `token`: the HTTP API token acquired while
- `chat_id`: the ID (not username) of the chat to write the updates to. This can be a user or a group chat ID
- optionally: a boolean for `silent_completion`. If set to true, regular experiment completions will use no or less intrusive notifications, depending on the receiving device's platform. Experiment starts will always be sent silently, interruptions and failures always with full notifications.
- optionally: a string for `proxy_url`. Specify this field, if Telegram is blocked in the local network or in the country, and you want to use proxy server. Format: `PROTOCOL://PROXY_HOST:[PROXY_PORT]/`. Socks5 and HTTP protocols are supported. These settings also could be received from `HTTPS_PROXY` or `https_proxy` environment variable.
- optionally: `username` for proxy.
- optionally: `password` for proxy.

The observer is then added to the experiment like this:

```
from sacred.observers import TelegramObserver

telegram_obs = TelegramObserver.from_config('telegram.json')
ex.observers.append(telegram_obs)
```

To set the bot's profile photo and description, use @BotFather's commands `/setuserpic` and `/setdescription`. Note that `/setuserpic` requires a *minimum* picture size.

1.6.9 Neptune Observer

Neptune observer sends all the experiment metadata to the Neptune UI. It requires the neptune-sacred package to be installed. You can install it by running:

```
pip install neptune-client neptune-sacred
```

Adding a Neptune Observer

NeptuneObserver can only be added from the Python code. You simply need to initialize it with your project name and (optionally) api token.

```
from neptune.new.integrations.sacred import NeptuneObserver
ex.observers.append(NeptuneObserver(api_token='<YOUR_API_TOKEN>',
                                    project='<YOUR_WORKSPACE/YOUR_PROJECT>'))
```

Warning: Always keep your API token secret - it is like password to the application. It is recommended to pass your token via the environment variable *NEPTUNE_API_TOKEN*. To make things simple you can put *export NEPTUNE_API_TOKEN=YOUR_LONG_API_TOKEN* line to your *~/.bashrc* or *~/.bash_profile* files.

1.6.10 Queue Observer

The *QueueObserver* can be used on top of other existing observers. It runs in a background thread. Observed events are buffered in a queue and the background thread is woken up to process new events at a fixed interval of 20 seconds be default. If the processing of an event fails, the event is put back on the queue and processed next time. This is useful for observers that rely on external services like databases that might become temporarily unavailable. Normally, the experiment would fail at this point, which could result in long running experiments being unnecessarily aborted. The *QueueObserver* can tolerate such temporary problems.

However, the *QueueObserver* has currently no way of declaring an event as finally failed, so if the failure is not due to a temporary unavailability of an external service, the observer will try forever.

Adding a Queue Observer

The `QueueObserver` can be used to wrap any other instantiated observer. For example, the `FileStorageObserver` can be made to use a queue like so

```
from sacred.observers import FileStorageObserver, QueueObserver

fs_observer = FileStorageObserver('my_runs', template='/custom/template.txt')
ex.observers.append(QueueObserver(fs_observer)
```

For wrapping the *Mongo Observer* a convenience class is provided to instantiate the queue based version.

```
from sacred.observers import QueuedMongoObserver

ex.observers.append(
    QueuedMongoObserver(url="my.server.org:27017", db_name="MY_DB")
)
```

1.6.11 Events

A `started_event` is fired when a run starts. Then every 10 seconds while the experiment is running a `heatbeat_event` is fired. Whenever a resource or artifact is added to the running experiment a `resource_event` resp. `artifact_event` is fired. Finally, once it stops one of the three `completed_event`, `interrupted_event`, or `failed_event` is fired. If the run is only being queued, then instead of all the above only a single `queued_event` is fired.

Start

The moment an experiment is started, the first event is fired for all the observers. It contains the following information:

ex_info	Some information about the experiment: • the docstring of the experiment-file • filename and md5 hash for all source-dependencies of the experiment • names and versions of packages the experiment depends on
command	The name of the command that was run.
host_info	Some information about the machine it's being run on: • CPU name • number of CPUs • hostname • Operating System • Python version • Python compiler
start_time	The date/time it was started
config	The configuration for this run, including the root-seed.
meta_info	Meta-information about this run such as a custom comment and the priority of this run.
_id	The ID of this run, as determined by the first observer

The started event is also the time when the ID of the run is determined. Essentially the first observer which sees *_id=None* sets an id and returns it. That id is then stored in the run and also passed to all further observers.

Queued

If a run is only queued instead of being run (see *Queue*), then this event is fired instead of a `started_event`. It contains the same information as the `started_event` except for the `host_info`.

Heartbeat

While the experiment is running, every 10 seconds a Heartbeat event is fired. It updates the **captured stdout and stderr** of the experiment, the custom `info` (see below), and the current result. The heartbeat event is also a way of monitoring if an experiment is still running.

Stop

Sacred distinguishes three ways in which an experiment can end:

Successful Completion: If an experiment finishes without an error, a `completed_event` is fired, which contains the time it completed and the result the command returned.

Interrupted: If a `KeyboardInterrupt` exception occurs (most of time this means you cancelled the experiment manually) instead an `interrupted_event` is fired, which only contains the interrupt time.

Failed: In case any other exception occurs, Sacred fires a `failed_event` with the fail time and the corresponding stacktrace.

Resources

Every time *sacred.Experiment.open_resource()* is called with a filename, an event will be fired with that filename (see *Resources*).

Artifacts

Every time *sacred.Experiment.add_artifact()* is called with a filename and optionally a name, an event will be fired with that name and filename (see *Artifacts*). If the name is left empty it defaults to the filename.

1.6.12 Saving Custom Information

Sometimes you want to add custom information about the run of an experiment, like the dataset, error curves during training, or the final trained model. To allow this sacred offers three different mechanisms.

Info Dict

The `info` dictionary is meant to store small amounts of information about the experiment, like training loss for each epoch or the total number of parameters. It is updated on each heartbeat, such that its content is accessible in the database already during runtime.

To store information in the `info` dict it can be accessed via `ex.info`, but only while the experiment is *running*. Another way is to access it directly through the run with `_run.info`. This can be done conveniently using the special `_run` parameter in any captured function, which gives you access to the current `Run` object.

You can add whatever information you like to `_run.info`. This `info` dict will be sent to all the observers every 10 sec as part of the *heartbeat_event*.

Warning: Many observers will convert the information of `info` into JSON using the jsonpickle library. This works for most python datatypes, but the resulting entries in the database may look different from what you might expect. So only store non-JSON information if you absolutely need to.

If the info dict contains `numpy` arrays or `pandas` Series/DataFrame/Panel then these will be converted to json automatically. The result is human readable (nested lists for `numpy` and a dict for `pandas`), but might be imprecise in some cases.

Resources

Generally speaking a resource is a file that your experiment needs to read during a run. When you open a file using `ex.open_resource(filename)` then a `resource_event` will be fired and the MongoObserver will check whether that file is in the database already. If not it will store it there. In any case the filename along with its MD5 hash is logged.

Artifacts

An artifact is a file created during the run. This mechanism is meant to store big custom chunks of data like a trained model. With *sacred.Experiment.add_artifact()* such a file can be added, which will fire an `artifact_event`. The MongoObserver will then in turn again, store that file in the database and log it in the run entry. Artifacts always have a name, but if the optional name parameter is left empty it defaults to the filename.

1.6.13 Custom Observer

The easiest way to implement a custom observer is to inherit from *sacred.observers.RunObserver* and override some or all of the events:

```
from sacred.observer import RunObserver

class MyObserver(RunObserver):
    def queued_event(self, ex_info, command, queue_time, config, meta_info,
                     _id):
        pass

    def started_event(self, ex_info, command, host_info, start_time,
                      config, meta_info, _id):
        pass

    def heartbeat_event(self, info, captured_out, beat_time, result):
        pass

    def completed_event(self, stop_time, result):
        pass

    def interrupted_event(self, interrupt_time, status):
        pass

    def failed_event(self, fail_time, fail_trace):
        pass

    def resource_event(self, filename):
        pass

    def artifact_event(self, name, filename):
        pass
```

1.7 Controlling Randomness

Many experiments rely on some form of randomness. Controlling this randomness is key to ensure reproducibility of the results. This typically happens by manually seeding the *Pseudo Random Number Generator (PRNG)*. Sacred can help you manage this error-prone procedure.

1.7.1 Automatic Seed

Sacred auto-generates a seed for each run as part of the configuration (You might have noticed it, when printing the configuration of an experiment). This seed has a different value everytime the experiment is run and is stored as part part of the configuration. You can easily set it by:

```
>>./experiment.py with seed=123
```

This root-seed is the central place to control randomness, because internally all other seeds and PRNGs depend on it in a deterministic way.

1.7.2 Global Seeds

Upon starting the experiment, sacred automatically sets the global seed of `random` and (if installed) `numpy.random` (which is with v1.19 mark as legacy), `tensorflow.set_random_seed`, `pytorch.manual_seed` to the auto-generated root-seed of the experiment. This means that even if you don't take any further steps, at least the randomness stemming from those two libraries is properly seeded.

If you rely on any other library that you want to seed globally you should do so manually first thing inside your main function. For this you can either take the argument `seed` (the root-seed), or `_seed` (a seed generated for this call of the main function). In this case it doesn't really matter.

1.7.3 Special Arguments

To generate random numbers that are controlled by the root-seed Sacred provides two special arguments: `_rnd` and `_seed`. You can just accept them as a parameters in any captured function:

```
@ex.capture
def do_random_stuff(_rnd, _seed):
    print(_seed)
    print(_rnd.randint(1, 100))
```

`_seed` is an integer that is different every time the function is called. Likewise `_rnd` is a PRNG that you can directly use to generate random numbers.

Note: If `numpy` is installed `_rnd` will either be a numpy.random.Generator object or a numpy.random.RandomState object. Default behavior is dependen on the numpy version, i. e. with version *v1.19 numpy.random.RandomState* is marked as legacy. To use the legacy numpy random API regardless of the numpy version set *NUMPY_RANDOM_LEGACY_API* to *True*. Otherwise it will be random.Random object.

All `_seed` and `_rnd` instances depend deterministically on the root-seed so they can be controlled centrally.

1.7.4 Resilience to Change

The way Sacred generates these seeds and PRNGs actually offers some amount of resilience to changes in your experiment or your program flow. So suppose for example you have an experiment that has two methods that use randomness: `A` and `B`. You want to run and compare two variants of that experiment:

1. Only call `B`.
2. First call `A` and then `B`.

If you use just a single global PRNG that would mean that for a fixed seed the call to `B` gives different results for the two variants, because the call to `A` changed the state of the global PRNG.

Sacred generates these seeds and PRNGS in a hierarchical way. That makes the calls to `A` and `B` independent from one another. So `B` would give the same results in both cases.

1.8 Logging

Sacred used the python logging module to log some basic information about the execution. It also makes it easy for you to integrate that logging with your code.

1.8.1 Adjusting Log-Levels from command line

If you run the hello_world example you will see the following output:

```
>> python hello_world.py
INFO - hello_world - Running command 'main'
INFO - hello_world - Started
Hello world!
INFO - hello_world - Completed after 0:00:00
```

The lines starting with `INFO` are logging outputs. They can be suppressed by adjusting the loglevel. This can be done via the command-line like with the `--loglevel` (`-l` for short) option:

```
>> python hello_world -l ERROR
Hello world!
```

The specified level can be either a string or an integer:

Level	Numeric value
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
NOTSET	0

1.8.2 Adjusting Log-Levels from python

If you prefer, you can also adjust the logging level from python when running an experiment by passing the long version of the log level command line parameter as an option, as follows:

```
ex.run(options={'--loglevel': 'ERROR'})
```

Note that this can only be done when using `Experiment.run`, not when using `Experiment.main` or `Experiment.automain`.

1.8.3 Integrate Logging Into Your Experiment

If you want to make use of the logging mechanism for your own experiments the easiest way is to use the special `_log` argument in your captured functions:

```
@ex.capture
def some_function(_log):
    _log.warning('My warning message!')
```

This will by default print a line like this:

```
WARNING - some_function - My warning message!
```

The `_log` is a standard Logger object for your function, as a child logger of the experiments main logger. So it allows calls to `debug`, `info`, `warning`, `error`, `critical` and some more. Check out the documentation to see what you can do with them.

1.8.4 Customize the Logger

It is easy to customize the logging behaviour of your experiment by just providing a custom Logger object to your experiment:

```
import logging
logger = logging.getLogger('my_custom_logger')
## configure your logger here
ex.logger = logger
```

The custom logger will be used to generate all the loggers for all captured functions. This way you can use all the features of the logging package. See the `examples/log_example.py` file for an example of this.

1.9 Ingredients

Some tasks have to be performed in many different experiments. One way to avoid code-duplication is of course to extract them to functions and import them. But if those tasks have to be configured, the configuration values would still have to be copied to every single experiment.

Ingredients are a way of defining a configuration with associated functions and possibly commands that can be reused by many different experiments. Furthermore they can influence the configuration and execution of experiments by using certain hooks.

1.9.1 Simple Example

So suppose in many experiments we always load a dataset from a given file and then we might want to normalize it or not. As an Ingredient this could look like that:

```
import numpy as np
from sacred import Ingredient

data_ingredient = Ingredient('dataset')

@data_ingredient.config
def cfg():
    filename = 'my_dataset.npy'
    normalize = True

@data_ingredient.capture
def load_data(filename, normalize):
    data = np.load(filename)
    if normalize:
        data -= np.mean(data)
    return data
```

Now all we have to do to use that in an Experiment is to import that ingredient and add it:

```
from sacred import Experiment

# import the Ingredient and the function we want to use:
from dataset_ingredient import data_ingredient, load_data

# add the Ingredient while creating the experiment
ex = Experiment('my_experiment', ingredients=[data_ingredient])
```

(continues on next page)

(continued from previous page)

```
@ex.automain
def run():
    data = load_data()  # just use the function
```

When you print the config for this experiment you will see an entry for the dataset ingredient:

```
./my_experiment.py print_config
INFO - my_experiment - Running command 'print_config'
INFO - my_experiment - Started
Configuration (modified, added, typechanged):
  seed = 586408722
  dataset:
    filename = 'my_dataset.npy'
    normalize = True
INFO - my_experiment - Completed after 0:00:00
```

And we could of course set these parameters from the command-line using `with 'dataset.filename="other.npy" 'dataset.normalize=False'`.

1.9.2 Overwriting the Default Configuration

You can change the default configuration of an Ingredient in each Experiment by adding *another ConfigScope*:

```
from sacred import Experiment

from dataset_ingredient import data_ingredient, load_data

@data_ingredient.config
def update_cfg():
    filename = 'special_dataset.npy'  # < updated

ex = Experiment('my_experiment', ingredients=[data_ingredient])

# ...
```

1.9.3 Adding Commands

Adding commands to Ingredients works as you would expect:

```
@data_ingredient.command
def stats(filename):
    print('Statistics for dataset "%s":' % filename)
    data = np.load(filename)
    print('mean = %0.2f' % np.mean(data))
```

You can call that command using dotted notation:

```
>> ./my_experiment dataset.stats
INFO - my_experiment - Running command 'dataset.stats'
INFO - my_experiment - Started
Statistics for dataset "my_dataset.npy":
mean = 13.37
INFO - my_experiment - Completed after 0:00:00
```

1.9.4 Nesting Ingredients

It is possible to use Ingredients in other Ingredients

```
data_ingredient = Ingredient('dataset', ingredients=[my_subingredient])
```

In fact Experiments are also Ingredients, so you can even reuse Experiments as Ingredients.

In the configuration of the Experiment there will be all the used Ingredients and sub-Ingredients. So lets say you use an Ingredient called `paths` in the `dataset` Ingredient. Then in the configuration of your experiment you will see two entries: `dataset` and `paths` (`paths` is **not** nested in the `dataset` entry)

Explicit Nesting

If you want nested structure you can do it explicitly by changing the name of the `path` Ingredient to `dataset.path`. Then the path entry will be nested in the dataset entry in the configuration.

1.9.5 Accessing the Ingredient Config

You can access the configuration of any used ingredient from ConfigScopes and from captured functions via the name of the ingredient:

```
@ex.config
def cfg(dataset):  # name of the ingredient here
    abs_filename = os.path.abspath(dataset['filename'])  # access 'filename'

@ex.capture
def some_function(dataset):   # name of the ingredient here
    if dataset['normalize']:  # access 'normalize'
        print("Dataset was normalized")
```

Ingredients with explicit nesting can be accessed by following their path. So for the example of the Ingredient `dataset.path` we could access it like this:

```
@ex.capture
def some_function(dataset):
    path = dataset['path']   # access the configuration of dataset.path
```

The only exception is, that if you want to access the configuration from another Ingredient you can leave away their common prefix. So accessing `dataset.path` from `dataset` you could just directly access `path` in captured functions and ConfigScopes.

1.9.6 Hooks

Hooks are advanced mechanisms that allow the ingredient to affect the normal execution of the experiment.

Pre- and Post-Run Hooks

Configuration Hooks

Configuration hooks are executed during initialization and can be used to update the experiment's configuration before executing any command.

```
ex = Experiment()

@ex.config_hook
def hook(config, command_name, logger):
    config.update({'hook': True})
    return config

@ex.automain
def main(hook, other_config):
    do_stuff()
```

The config_hook function always has to take the 3 arguments *config* of the current configuration, *command_name*, which is the command that will be executed, and *logger*. Config hooks are run after the configuration of the linked Ingredient (in the example above Experiment *ex*), but before any further ingredient-configurations are run. The dictionary returned by a config hook is used to update the config updates. Note that config hooks are not restricted to the local namespace of the ingredient.

1.10 Optional Features

Sacred offers a set of specialized features which are kept optional in order to keep the list of requirements small. This page provides a short description of these optional features.

1.10.1 Git Integration

If the experiment sources are maintained in a git repository, then Sacred can extract information about the current state of the repository. More specifically it will collect the following information, which is stored by the observers as part of the experiment info:

- **url:** The url of the origin repository
- **commit:** The SHA256 hash of the current commit
- **dirty:** A boolean indicating if the repository is dirty, i.e. has uncommitted changes.

This can be especially useful together with the *Enforce Clean* (`-e / --enforce_clean`) commandline option. If this flag is used, the experiment immediately fails with an error if started on a dirty repository.

Note: Git integration can be disabled with `save_git_info` flag in the `Experiment` or `Ingredient` constructor.

1.10.2 Optional Observers

MongoDB

An observer which stores run information in a MongoDB. For more information see *Mongo Observer*.

Note: Requires the pymongo package. Install with `pip install pymongo`.

TinyDB

An observer which stores run information in a tinyDB. It can be seen as a local alternative for the MongoDB Observer. For more information see *TinyDB Observer*.

Note: Requires the tinydb, tinydb-serialization, and hashfs packages. Install with `pip install tinydb tinydb-serialization hashfs`.

SQL

An observer that stores run information in a SQL database. For more information see *SQL Observer*

Note: Requires the sqlalchemy package. Install with `pip install sqlalchemy`.

Template Rendering

The *File Storage Observer* supports automatic report generation using the mako package.

Note: Requires the mako package. Install with `pip install mako`.

1.10.3 Numpy and Pandas Integration

If `numpy` or `pandas` are installed Sacred will automatically take care of a set of type conversions and other details to make working with these packages as smooth as possible. Normally you won't need to know about any details. But for some cases it might be useful to know what is happening. So here is a list of what Sacred will do:

- automatically set the global numpy random seed (`numpy.random.seed()`).
- if `numpy` is installed the *special value* `_rnd` will be a `numpy.random.RandomState` instead of `random.Random`.
- because of these two points having numpy installed actually changes the way randomness is handled. Therefore `numpy` is then automatically added to the dependencies of the experiment, irrespective of its usage in the code.
- ignore typechanges in the configuration from `numpy` types to normal types, such as `numpy.float32` to `float`.
- convert basic numpy types in the configuration to normal types if possible. This includes converting `numpy.array` to `list`.
- convert `numpy.array`, `pandas.Series`, `pandas.DataFrame` and `pandas.Panel` to json before storing them in the MongoDB. This includes instances in the *Info Dict*.

1.10.4 YAML Format for Configurations

If the PyYAML package is installed Sacred automatically supports using config files in the yaml format (see *Config Files*).

Note: Requires the PyYAML package. Install with `pip install PyYAML`.

1.11 Settings

Some of Sacred's general behaviour is configurable via `sacred.SETTINGS`. Its entries can be set simply by importing and modifying it using dict or attribute notation:

```
from sacred import SETTINGS
SETTINGS['HOST_INFO']['INCLUDE_GPU_INFO'] = False
SETTINGS.HOST_INFO.INCLUDE_GPU_INFO = False  # equivalent
```

1.11.1 Settings

Here is a brief list of all currently available options.

- `CAPTURE_MODE` *(default: 'fd' (linux/osx) or 'sys' (windows))* configure how stdout/stderr are captured. ['no', 'sys', 'fd']
- `CONFIG`
  - `ENFORCE_KEYS_MONGO_COMPATIBLE` *(default: True)* Make sure all config keys are compatible with MongoDB.
  - `ENFORCE_KEYS_JSONPICKLE_COMPATIBLE` *(default: True)* Make sure all config keys are serializable with jsonpickle. IMPORTANT: Only deactivate if you know what you're doing.
  - `ENFORCE_VALID_PYTHON_IDENTIFIER_KEYS` *(default: False)* Make sure all config keys are valid python identifiers.
  - `ENFORCE_STRING_KEYS` *(default: False)* Make sure all config keys are strings.
  - `ENFORCE_KEYS_NO_EQUALS` *(default: True)* Make sure no config key contains an equals sign.
  - `IGNORED_COMMENTS` *(default: ['^pylint:', '^noinspection'])* List of regex patterns to filter out certain IDE or linter directives from in-line comments in the documentation.
  - `READ_ONLY_CONFIG` *(default: True)* Make the configuration read-only inside of captured functions. This only works to a limited extend because custom types cannot be controlled.
- `HOST_INFO`
  - `INCLUDE_GPU_INFO` *(default: True)* Try to collect information about GPUs using the nvidia-smi tool. Deactivating this can cut the start-up time of a Sacred run by about 1 sec.
  - `INCLUDE_CPU_INFO` *(default: True)* Try to collect information about the CPU using py-cpuinfo. Deactivating this can cut the start-up time of a Sacred run by about 3 sec.
  - `CAPTURED_ENV` *(default: [])* List of ENVIRONMENT variable names to store in the host-info.
- `COMMAND_LINE`
  - `STRICT_PARSING` *(default: False)* Disallow string fallback if parsing a value from command-line failed. This enforces the usage of quotes in the command-line. Note that this can be very tedious since bash removes one set of quotes, such that double quotes will be needed.

1.12 Examples

You can find these examples in the examples directory (surprise!) of the Sacred sources or in the Github Repository. Look at them for the sourcecode, it is an important part of the examples. It can also be very helpful to run them yourself and play with the command-line interface.

The following is just their documentation from their docstring which you can also get by running them with the `-h`, `--help` or `help` flags.

1.12.1 Hello World

examples/01_hello_world.py

This is a minimal example of a Sacred experiment.

Not much to see here. But it comes with a command-line interface and can be called like this:

```
$ ./01_hello_world.py
WARNING - 01_hello_world - No observers have been added to this run
INFO - 01_hello_world - Running command 'main'
INFO - 01_hello_world - Started
Hello world!
INFO - 01_hello_world - Completed after 0:00:00
```

As you can see it prints 'Hello world!' as expected, but there is also some additional logging. The log-level can be controlled using the `-l` argument:

```
$ ./01_hello_world.py -l WARNING
WARNING - 01_hello_world - No observers have been added to this run
Hello world!
```

If you want to learn more about the command-line interface try `help` or `-h`.

1.12.2 Hello Config Dict

examples/02_hello_config_dict.py

A configurable Hello World "experiment". In this example we configure the message using a dictionary with `ex.add_config`

You can run it like this:

```
$ ./02_hello_config_dict.py
WARNING - 02_hello_config_dict - No observers have been added to this run
INFO - 02_hello_config_dict - Running command 'main'
INFO - 02_hello_config_dict - Started
Hello world!
INFO - 02_hello_config_dict - Completed after 0:00:00
```

The message can also easily be changed using the `with` command-line argument:

```
$ ./02_hello_config_dict.py with message='Ciao world!'
WARNING - 02_hello_config_dict - No observers have been added to this run
INFO - 02_hello_config_dict - Running command 'main'
INFO - 02_hello_config_dict - Started
```

(continues on next page)

(continued from previous page)

```
Ciao world!
INFO - 02_hello_config_dict - Completed after 0:00:00
```

1.12.3 Hello Config Scope

examples/03_hello_config_scope.py

A configurable Hello World "experiment". In this example we configure the message using Sacreds special `ConfigScope`.

As with hello_config_dict you can run it like this:

```
$ ./03_hello_config_scope.py
WARNING - hello_cs - No observers have been added to this run
INFO - hello_cs - Running command 'main'
INFO - hello_cs - Started
Hello world!
INFO - hello_cs - Completed after 0:00:00
```

The message can also easily be changed using the `with` command-line argument:

```
$ ./03_hello_config_scope.py with message='Ciao world!'
WARNING - hello_cs - No observers have been added to this run
INFO - hello_cs - Running command 'main'
INFO - hello_cs - Started
Ciao world!
INFO - hello_cs - Completed after 0:00:00
```

But because we are using a `ConfigScope` that constructs the message from a recipient we can also just modify that:

```
$ ./03_hello_config_scope.py with recipient='Bob'
WARNING - hello_cs - No observers have been added to this run
INFO - hello_cs - Running command 'main'
INFO - hello_cs - Started
Hello Bob!
INFO - hello_cs - Completed after 0:00:00
```

1.12.4 Captured Functions

examples/04_captured_functions.py

In this example the use of captured functions is demonstrated. Like the main function, they have access to the configuration parameters by just accepting them as arguments.

When calling a captured function we do not need to specify the parameters that we want to be taken from the configuration. They will automatically be filled by Sacred. But we can always override that by passing them in explicitly.

When run, this example will output the following:

```
$ ./04_captured_functions.py -l WARNING
WARNING - captured_functions - No observers have been added to this run
This is printed by function foo.
This is printed by function bar.
Overriding the default message for foo.
```

1.12.5 My Commands

examples/05_my_commands.py

This experiment showcases the concept of commands in Sacred. By just using the `@ex.command` decorator we can add additional commands to the command-line interface of the experiment:

```
$ ./05_my_commands.py greet
WARNING - my_commands - No observers have been added to this run
INFO - my_commands - Running command 'greet'
INFO - my_commands - Started
Hello John! Nice to greet you!
INFO - my_commands - Completed after 0:00:00
```

```
$ ./05_my_commands.py shout
WARNING - my_commands - No observers have been added to this run
INFO - my_commands - Running command 'shout'
INFO - my_commands - Started
WHAZZZUUUUUUUUUUP!!!????
INFO - my_commands - Completed after 0:00:00
```

Of course we can also use `with` and other flags with those commands:

```
$ ./05_my_commands.py greet with name='Jane' -l WARNING
WARNING - my_commands - No observers have been added to this run
Hello Jane! Nice to greet you!
```

In fact, the main function is also just a command:

```
$ ./05_my_commands.py main
WARNING - my_commands - No observers have been added to this run
INFO - my_commands - Running command 'main'
INFO - my_commands - Started
This is just the main command. Try greet or shout.
INFO - my_commands - Completed after 0:00:00
```

Commands also appear in the help text, and you can get additional information about all commands using `./05_my_commands.py help [command]`.

1.12.6 Randomness

examples/06_randomness.py

This example showcases the randomness features of Sacred.

Sacred generates a random global seed for every experiment, that you can find in the configuration. It will be different every time you run the experiment.

Based on this global seed it will generate the special parameters `_seed` and `_rnd` for each captured function. Every time you call such a function the `_seed` will be different and `_rnd` will be differently seeded random state. But their values depend deterministically on the global seed and on how often the function has been called.

Here are a couple of things you should try:

- run the experiment a couple of times and notice how the results are different every time
- run the experiment a couple of times with a fixed seed. Notice that the results are the same:

```
:$ ./06_randomness.py with seed=12345 -l WARNING
[57]
[28]
695891797
[82]
```

- run the experiment with a fixed seed and vary the numbers parameter. Notice that all the results stay the same except for the added numbers. This demonstrates that all the calls to one function are in fact independent from each other:

```
:$ ./06_randomness.py with seed=12345 numbers=3 -l WARNING
[57, 79, 86]
[28, 90, 92]
695891797
[82, 9, 3]
```

- run the experiment with a fixed seed and set the reverse parameter to true. Notice how the results are the same, but in slightly different order. This shows that calls to different functions do not interfere with one another:

```
:$ ./06_randomness.py with seed=12345 reverse=True numbers=3 -l WARNING
695891797
[57, 79, 86]
[28, 90, 92]
[82, 9, 3]
```

1.12.7 Less magic

If you are new to Sacred, you might be surprised by the amount of new idioms it introduces compared to standard Python. But don't worry, you don't have to use any of the magic if you don't want to and still benefit from the excellent tracking capabilities.

examples/07_magic.py shows a standard machine learning task, that uses a lot of possible Sacred idioms:

- configuration definition through local variables
- parameter injection through captured functions
- command line interface integration through the `ex.automain` decorator

examples/08_less_magic.py shows the same task without any of those idioms. The recipe for replacing Sacred magic with standard Python is simple.

- define your configuration in a dictionary, alternatively you can use an external `JSON` or `YAML` file
- avoid the `ex.capture` decorator. Instead only pass `_config` to the main function and access all parameters explicitly through the configuration dictionary
- just use `ex.main` instead of `ex.automain` and call `ex.run()` explicitly. This avoids the parsing of command line parameters you did not define yourself.

While we believe that using sacred idioms makes things easier by hard-wiring parameters and giving you a flexible command line interface, we do not enforce its usage if you feel more comfortable with classical Python. At its core Sacred is about tracking computatonal experiments, not about any particular coding style.

1.12.8 Docker Setup

examples/docker

To use Sacred to its full potential you probably want to use it together with MongoDB and dashboards like Omniboard that have been developed for it. To ease getting started with these services you find an exemplary `docker-compose` configuration in examples/docker. After installing Docker Engine and Docker Compose (only necessary for Linux) go to the directory and run:

```
docker-compose up
```

This will pull the necessary containers from the internet and build them. This may take several minutes. Afterwards mongoDB should be up and running. `mongo-express`, an admin interface for MongoDB, should now be available on port `8081`, accessible by the user and password set in the `.env` file (`ME_CONFIG_BASICAUTH_USERNAME` and `ME_CONFIG_BASICAUTH_PASSWORD`). `Sacredboard ``should be available on port ``5000`. `Omniboard` should be available on port `9000`. They will both listen to the the database name set in the `.env` file (`MONGO_DATABASE`) which will allow the boards to listen to the appropriated mongo database name set when creating the MongoObserver with the `db_name` arg. All services will by default only be exposed to `localhost`. If you want to expose them on all interfaces, e.g. for the use on a server, you need to change the port mappings in `docker-compose.yml` from `127.0.0.1:XXXX:XXXX` to `XXXX:XXXX`. However, in this case you should change the authentication information in `.env` to something more secure.

1.13 Projects using Sacred

This is a curated list of projects that make use of Sacred. The list can include code from research projects or show how to integrate sacred with another library.

1.13.1 Sacred_HyperOpt

An example for integrating a general machine learning training script with Sacred and HyperOpt (Distributed Asynchronous Hyperparameter Optimization).

1.14 Integration with Tensorflow

Sacred provides ways to interact with the Tensorflow library. The goal is to provide an API that would allow tracking certain information about how Tensorflow is being used with Sacred. The collected data are stored in `experiment.info["tensorflow"]` where they can be accessed by various *observers*.

1.14.1 Storing Tensorflow Logs (FileWriter)

To store the location of summaries produced by Tensorflow (created by `tensorflow.summary.FileWriter`) into the experiment record specified by the `ex` argument, use the `sacred.stflow.LogFileWriter(ex)` decorator or context manager. Whenever a new `FileWriter` instantiation is detected in a scope of the decorator or the context manager, the path of the log is copied to the experiment record exactly as passed to the FileWriter.

The location(s) can be then found under `info["tensorflow"]["logdirs"]` of the experiment.

Important: The experiment must be in the *RUNNING* state before calling the decorated method or entering the context.

Example Usage As a Decorator

`LogFileWriter(ex)` as a decorator can be used either on a function or on a class method.

```
from sacred.stflow import LogFileWriter
from sacred import Experiment
import tensorflow as tf

ex = Experiment("my experiment")

@ex.automain
@LogFileWriter(ex)
def run_experiment(_run):
    with tf.Session() as s:
        swr = tf.summary.FileWriter("/tmp/1", s.graph)
        # _run.info["tensorflow"]["logdirs"] == ["/tmp/1"]
        swr2 = tf.summary.FileWriter("./test", s.graph)
        #_run.info["tensorflow"]["logdirs"] == ["/tmp/1", "./test"]
```

Example Usage As a Context Manager

There is a context manager available to catch the paths in a smaller portion of code.

```
ex = Experiment("my experiment")
def run_experiment(_run):
    with tf.Session() as s:
        with LogFileWriter(ex):
            swr = tf.summary.FileWriter("/tmp/1", s.graph)
            # _run.info["tensorflow"]["logdirs"] == ["/tmp/1"]
            swr3 = tf.summary.FileWriter("./test", s.graph)
            # _run.info["tensorflow"]["logdirs"] == ["/tmp/1", "./test"]
        # This is called outside the scope and won't be captured
        swr3 = tf.summary.FileWriter("./nothing", s.graph)
        # Nothing has changed:
        # _run.info["tensorflow"]["logdirs"] == ["/tmp/1", "./test"]
```

1.15 API Documentation

This is a construction site...

1.15.1 Experiment

Note: Experiment inherits from *Ingredient*, so all methods from there also available in the Experiment.

class sacred.**Experiment**(*name: Optional[str] = None, ingredients: Sequence[sacred.ingredient.Ingredient] = (), interactive: bool = False, base_dir: Union[str, bytes, pathlib.Path, None] = None, additional_host_info: Optional[List[sacred.host_info.HostInfoGetter]] = None, additional_cli_options: Optional[Sequence[sacred.commandline_options.CLIOption]] = None, save_git_info: bool = True*)

The central class for each experiment in Sacred.

It manages the configuration, the main function, captured methods, observers, commands, and further ingredients.

An Experiment instance should be created as one of the first things in any experiment-file.

__init__(*name: Optional[str] = None*, *ingredients: Sequence[sacred.ingredient.Ingredient] = ()*, *interactive: bool = False*, *base_dir: Union[str, bytes, pathlib.Path, None] = None*, *additional_host_info: Optional[List[sacred.host_info.HostInfoGetter]] = None*, *additional_cli_options: Optional[Sequence[sacred.commandline_options.CLIOption]] = None*, *save_git_info: bool = True*)

Create a new experiment with the given name and optional ingredients.

Parameters

- **name** – Optional name of this experiment, defaults to the filename. (Required in interactive mode)
- **ingredients** (*list[sacred.Ingredient], optional*) – A list of ingredients to be used with this experiment.
- **interactive** – If set to True will allow the experiment to be run in interactive mode (e.g. IPython or Jupyter notebooks). However, this mode is discouraged since it won't allow storing the source-code or reliable reproduction of the runs.
- **base_dir** – Optional full path to the base directory of this experiment. This will set the scope for automatic source file discovery.
- **additional_host_info** – Optional dictionary containing as keys the names of the pieces of host info you want to collect, and as values the functions collecting those pieces of information.
- **save_git_info** – Optionally save the git commit hash and the git state (clean or dirty) for all source files. This requires the GitPython package.

add_artifact(*filename: Union[str, bytes, pathlib.Path], name: Optional[str] = None, metadata: Optional[dict] = None, content_type: Optional[str] = None*) → None

Add a file as an artifact.

In Sacred terminology an artifact is a file produced by the experiment run. In case of a MongoObserver that means storing the file in the database.

This function can only be called during a run, and just calls the *sacred.run.Run.add_artifact()* method.

Parameters

- **filename** – name of the file to be stored as artifact
- **name** – optionally set the name of the artifact. Defaults to the relative file-path.
- **metadata** – optionally attach metadata to the artifact. This only has an effect when using the MongoObserver.
- **content_type** – optionally attach a content-type to the artifact. This only has an effect when using the MongoObserver.

add_config(*cfg_or_file=None*, ***kw_conf*)

Add a configuration entry to this ingredient/experiment.

Can be called with a filename, a dictionary xor with keyword arguments. Supported formats for the config-file so far are: `json`, `pickle` and `yaml`.

The resulting dictionary will be converted into a `ConfigDict`.

Parameters

- **cfg_or_file** (*dict or str*) – Configuration dictionary of filename of config file to add to this ingredient/experiment.
- **kw_conf** – Configuration entries to be added to this ingredient/experiment.

add_named_config(*name*, *cfg_or_file=None*, ***kw_conf*)
Add a **named** configuration entry to this ingredient/experiment.

Can be called with a filename, a dictionary xor with keyword arguments. Supported formats for the config-file so far are: `json`, `pickle` and `yaml`.

The resulting dictionary will be converted into a `ConfigDict`.

See *Named Configurations*

Parameters

- **name** (*str*) – name of the configuration
- **cfg_or_file** (*dict or str*) – Configuration dictionary of filename of config file to add to this ingredient/experiment.
- **kw_conf** – Configuration entries to be added to this ingredient/experiment.

add_package_dependency(*package_name*, *version*)
Add a package to the list of dependencies.

Parameters

- **package_name** (*str*) – The name of the package dependency
- **version** (*str*) – The (minimum) version of the package

add_resource(*filename: Union[str, bytes, pathlib.Path]*) → None
Add a file as a resource.

In Sacred terminology a resource is a file that the experiment needed to access during a run. In case of a MongoObserver that means making sure the file is stored in the database (but avoiding duplicates) along its path and md5 sum.

This function can only be called during a run, and just calls the *sacred.run.Run.add_resource()* method.

Parameters **filename** – name of the file to be stored as a resource

add_source_file(*filename*)
Add a file as source dependency to this experiment/ingredient.

Parameters **filename** (*str*) – filename of the source to be added as dependency

automain(*function*)
Decorator that defines *and runs* the main function of the experiment.

The decorated function is marked as the default command for this experiment, and the command-line interface is automatically run when the file is executed.

The method decorated by this should be last in the file because is equivalent to:

Example

```
@ex.main
def my_main():
    pass

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

capture (*function=None*, *prefix=None*)
Decorator to turn a function into a captured function.

The missing arguments of captured functions are automatically filled from the configuration if possible. See *Captured Functions* for more information.

If a `prefix` is specified, the search for suitable entries is performed in the corresponding subtree of the configuration.

captured_out_filter = None
Filter function to be applied to captured output of a run

command (*function=None*, *prefix=None*, *unobserved=False*)
Decorator to define a new command for this Ingredient or Experiment.

The name of the command will be the name of the function. It can be called from the command-line or by using the run_command function.

Commands are automatically also captured functions.

The command can be given a prefix, to restrict its configuration space to a subtree. (see `capture` for more information)

A command can be made unobserved (i.e. ignoring all observers) by passing the unobserved=True keyword argument.

config (*function*)
Decorator to add a function to the configuration of the Experiment.

The decorated function is turned into a `ConfigScope` and added to the Ingredient/Experiment.

When the experiment is run, this function will also be executed and all json-serializable local variables inside it will end up as entries in the configuration of the experiment.

config_hook (*func*)
Decorator to add a config hook to this ingredient.

Config hooks need to be a function that takes 3 parameters and returns a dictionary: (config, command_name, logger) –> dict

Config hooks are run after the configuration of this Ingredient, but before any further ingredient-configurations are run. The dictionary returned by a config hook is used to update the config updates. Note that they are not restricted to the local namespace of the ingredient.

gather_commands ()
Collect all commands from this ingredient and its sub-ingredients.

Yields

- **cmd_name** (*str*) – The full (dotted) name of the command.
- **cmd** (*function*) – The corresponding captured function.

gather_named_configs () → Generator[Tuple[str, Union[sacred.config.config_scope.ConfigScope, sacred.config.config_dict.ConfigDict, str]], None, None]
Collect all named configs from this ingredient and its sub-ingredients.

Yields

- *config_name* – The full (dotted) name of the named config.
- *config* – The corresponding named config.

get_default_options() → dict
Get a dictionary of default options as used with run.

Returns

- *A dictionary containing option keys of the form '–beat_interval'.*
- *Their values are boolean if the option is a flag, otherwise None or*
- *its default value.*

get_experiment_info()
Get a dictionary with information about this experiment.

Contains:

- *name*: the name
- *sources*: a list of sources (filename, md5)
- *dependencies*: a list of package dependencies (name, version)

Returns experiment information

Return type dict

get_usage(*program_name=None*)
Get the commandline usage string for this experiment.

info
Access the info-dict for storing custom information.

Only works during a run and is essentially a shortcut to:

Example

```
@ex.capture
def my_captured_function(_run):
    # [...]
    _run.info   # == ex.info
```

log_scalar(*name: str*, *value: float*, *step: Optional[int] = None*) → None
Add a new measurement.

The measurement will be processed by the MongoDB* observer during a heartbeat event. Other observers are not yet supported.

Parameters

- **name** – The name of the metric, e.g. training.loss
- **value** – The measured value
- **step** – The step number (integer), e.g. the iteration number If not specified, an internal counter for each metric is used, incremented by one.

main (*function*)
Decorator to define the main function of the experiment.

The main function of an experiment is the default command that is being run when no command is specified, or when calling the run() method.

Usually it is more convenient to use `automain` instead.

named_config (*func*)
Decorator to turn a function into a named configuration.

See *Named Configurations*.

open_resource (*filename: Union[str, bytes, pathlib.Path], mode: str = 'r'*)
Open a file and also save it as a resource.

Opens a file, reports it to the observers as a resource, and returns the opened file.

In Sacred terminology a resource is a file that the experiment needed to access during a run. In case of a MongoObserver that means making sure the file is stored in the database (but avoiding duplicates) along its path and md5 sum.

This function can only be called during a run, and just calls the *sacred.run.Run.open_resource()* method.

Parameters

- **filename** – name of the file that should be opened
- **mode** – mode that file will be open

Returns *The opened file-object.*

option_hook (*function*)
Decorator for adding an option hook function.

An option hook is a function that is called right before a run is created. It receives (and potentially modifies) the options dictionary. That is, the dictionary of commandline options used for this run.

Notes

The decorated function MUST have an argument called options.

The options also contain `'COMMAND'` and `'UPDATE'` entries, but changing them has no effect. Only modification on flags (entries starting with `'--'`) are considered.

post_process_name (*name*, *ingredient*)
Can be overridden to change the command name.

post_run_hook (*func*, *prefix=None*)
Decorator to add a post-run hook to this ingredient.

Post-run hooks are captured functions that are run, just after the main function is executed.

pre_run_hook (*func*, *prefix=None*)
Decorator to add a pre-run hook to this ingredient.

Pre-run hooks are captured functions that are run, just before the main function is executed.

run (*command_name: Optional[str] = None*, *config_updates: Optional[dict] = None*, *named_configs: Sequence[str] = ()*, *info: Optional[dict] = None*, *meta_info: Optional[dict] = None*, *options: Optional[dict] = None*) → sacred.run.Run
Run the main function of the experiment or a given command.

Parameters

- **command_name** – Name of the command to be run. Defaults to main function.
- **config_updates** – Changes to the configuration as a nested dictionary
- **named_configs** – list of names of named_configs to use
- **info** – Additional information for this run.
- **meta_info** – Additional meta information for this run.
- **options** – Dictionary of options to use

Returns *The Run object corresponding to the finished run.*

run_commandline(*argv=None*) → Optional[sacred.run.Run]
Run the command-line interface of this experiment.

If `argv` is omitted it defaults to `sys.argv`.

Parameters **argv** – Command-line as string or list of strings like `sys.argv`.

Returns *The Run object corresponding to the finished run.*

traverse_ingredients()
Recursively traverse this ingredient and its sub-ingredients.

Yields

- **ingredient** (*sacred.Ingredient*) – The ingredient as traversed in preorder.
- **depth** (*int*) – The depth of the ingredient starting from 0.

Raises CircularDependencyError: – If a circular structure among ingredients was detected.

1.15.2 Ingredient

class sacred.**Ingredient**(*path: Union[str, bytes, pathlib.Path], ingredients: Sequence[Ingredient] = (), interactive: bool = False, _caller_globals: Optional[dict] = None, base_dir: Union[str, bytes, pathlib.Path, None] = None, save_git_info: bool = True*)
Ingredients are reusable parts of experiments.

Each Ingredient can have its own configuration (visible as an entry in the parents configuration), named configurations, captured functions and commands.

Ingredients can themselves use ingredients.

__init__(*path: Union[str, bytes, pathlib.Path], ingredients: Sequence[Ingredient] = (), interactive: bool = False, _caller_globals: Optional[dict] = None, base_dir: Union[str, bytes, pathlib.Path, None] = None, save_git_info: bool = True*)
Initialize self. See help(type(self)) for accurate signature.

add_config(*cfg_or_file=None*, ***kw_conf*)
Add a configuration entry to this ingredient/experiment.

Can be called with a filename, a dictionary xor with keyword arguments. Supported formats for the config-file so far are: `json`, `pickle` and `yaml`.

The resulting dictionary will be converted into a `ConfigDict`.

Parameters

- **cfg_or_file** (*dict or str*) – Configuration dictionary of filename of config file to add to this ingredient/experiment.
- **kw_conf** – Configuration entries to be added to this ingredient/experiment.

add_named_config(*name*, *cfg_or_file=None*, ***kw_conf*)
Add a **named** configuration entry to this ingredient/experiment.

Can be called with a filename, a dictionary xor with keyword arguments. Supported formats for the config-file so far are: `json`, `pickle` and `yaml`.

The resulting dictionary will be converted into a `ConfigDict`.

See *Named Configurations*

Parameters

- **name** (*str*) – name of the configuration
- **cfg_or_file** (*dict or str*) – Configuration dictionary of filename of config file to add to this ingredient/experiment.
- **kw_conf** – Configuration entries to be added to this ingredient/experiment.

add_package_dependency(*package_name*, *version*)
Add a package to the list of dependencies.

Parameters

- **package_name** (*str*) – The name of the package dependency
- **version** (*str*) – The (minimum) version of the package

add_source_file(*filename*)
Add a file as source dependency to this experiment/ingredient.

Parameters **filename** (*str*) – filename of the source to be added as dependency

capture(*function=None*, *prefix=None*)
Decorator to turn a function into a captured function.

The missing arguments of captured functions are automatically filled from the configuration if possible. See *Captured Functions* for more information.

If a `prefix` is specified, the search for suitable entries is performed in the corresponding subtree of the configuration.

command(*function=None*, *prefix=None*, *unobserved=False*)
Decorator to define a new command for this Ingredient or Experiment.

The name of the command will be the name of the function. It can be called from the command-line or by using the run_command function.

Commands are automatically also captured functions.

The command can be given a prefix, to restrict its configuration space to a subtree. (see `capture` for more information)

A command can be made unobserved (i.e. ignoring all observers) by passing the unobserved=True keyword argument.

config(*function*)
Decorator to add a function to the configuration of the Experiment.

The decorated function is turned into a `ConfigScope` and added to the Ingredient/Experiment.

When the experiment is run, this function will also be executed and all json-serializable local variables inside it will end up as entries in the configuration of the experiment.

config_hook(*func*)
Decorator to add a config hook to this ingredient.

Config hooks need to be a function that takes 3 parameters and returns a dictionary: (config, command_name, logger) –> dict

Config hooks are run after the configuration of this Ingredient, but before any further ingredient-configurations are run. The dictionary returned by a config hook is used to update the config updates. Note that they are not restricted to the local namespace of the ingredient.

gather_commands()
Collect all commands from this ingredient and its sub-ingredients.

Yields

- **cmd_name** (*str*) – The full (dotted) name of the command.
- **cmd** (*function*) – The corresponding captured function.

gather_named_configs() → Generator[Tuple[str, Union[sacred.config.config_scope.ConfigScope, sacred.config.config_dict.ConfigDict, str]], None, None]
Collect all named configs from this ingredient and its sub-ingredients.

Yields

- *config_name* – The full (dotted) name of the named config.
- *config* – The corresponding named config.

get_experiment_info()
Get a dictionary with information about this experiment.

Contains:

- *name*: the name
- *sources*: a list of sources (filename, md5)
- *dependencies*: a list of package dependencies (name, version)

Returns experiment information

Return type dict

named_config(*func*)
Decorator to turn a function into a named configuration.

See *Named Configurations*.

post_process_name(*name*, *ingredient*)
Can be overridden to change the command name.

post_run_hook(*func*, *prefix=None*)
Decorator to add a post-run hook to this ingredient.

Post-run hooks are captured functions that are run, just after the main function is executed.

pre_run_hook(*func*, *prefix=None*)
Decorator to add a pre-run hook to this ingredient.

Pre-run hooks are captured functions that are run, just before the main function is executed.

traverse_ingredients()
Recursively traverse this ingredient and its sub-ingredients.

Yields

- **ingredient** (*sacred.Ingredient*) – The ingredient as traversed in preorder.
- **depth** (*int*) – The depth of the ingredient starting from 0.

Raises CircularDependencyError: – If a circular structure among ingredients was detected.

1.15.3 The Run Object

The Run object can be accessed from python after the run is finished: `run = ex.run()` or during a run using the `_run` *special value* in a *captured function*.

class sacred.run.**Run**(*config*, *config_modifications*, *main_function*, *observers*, *root_logger*, *run_logger*, *experiment_info*, *host_info*, *pre_run_hooks*, *post_run_hooks*, *captured_out_filter=None*)
Represent and manage a single run of an experiment.

__call__(**args*)
Start this run.

Parameters ***args** – parameters passed to the main function

Returns *the return value of the main function*

add_artifact(*filename*, *name=None*, *metadata=None*, *content_type=None*)
Add a file as an artifact.

In Sacred terminology an artifact is a file produced by the experiment run. In case of a MongoObserver that means storing the file in the database.

See also *sacred.Experiment.add_artifact()*.

Parameters

- **filename** (*str*) – name of the file to be stored as artifact
- **name** (*str, optional*) – optionally set the name of the artifact. Defaults to the filename.
- **metadata** (*dict*) – optionally attach metadata to the artifact. This only has an effect when using the MongoObserver.
- **content_type** (*str, optional*) – optionally attach a content-type to the artifact. This only has an effect when using the MongoObserver.

add_resource(*filename*)
Add a file as a resource.

In Sacred terminology a resource is a file that the experiment needed to access during a run. In case of a MongoObserver that means making sure the file is stored in the database (but avoiding duplicates) along its path and md5 sum.

See also *sacred.Experiment.add_resource()*.

Parameters **filename** (*str*) – name of the file to be stored as a resource

beat_interval = None
The time between two heartbeat events measured in seconds

capture_mode = None
Determines the way the stdout/stderr are captured

captured_out = None
Captured stdout and stderr

captured_out_filter = None
Filter function to be applied to captured output

config = None
The final configuration used for this run

config_modifications = None
A ConfigSummary object with information about config changes

debug = None
Determines whether this run is executed in debug mode

experiment_info = None
A dictionary with information about the experiment

fail_trace = None
A stacktrace, in case the run failed

force = None
Disable warnings about suspicious changes

host_info = None
A dictionary with information about the host

info = None
Custom info dict that will be sent to the observers

log_scalar(*metric_name*, *value*, *step=None*)
Add a new measurement.

The measurement will be processed by the MongoDB observer during a heartbeat event. Other observers are not yet supported.

Parameters

- **metric_name** – The name of the metric, e.g. training.loss
- **value** – The measured value
- **step** – The step number (integer), e.g. the iteration number If not specified, an internal counter for each metric is used, incremented by one.

main_function = None
The main function that is executed with this run

meta_info = None
A custom comment for this run

observers = None
A list of all observers that observe this run

open_resource(*filename*, *mode='r'*)
Open a file and also save it as a resource.

Opens a file, reports it to the observers as a resource, and returns the opened file.

In Sacred terminology a resource is a file that the experiment needed to access during a run. In case of a MongoObserver that means making sure the file is stored in the database (but avoiding duplicates) along its path and md5 sum.

See also *sacred.Experiment.open_resource()*.

Parameters

- **filename** (*str*) – name of the file that should be opened
- **mode** (*str*) – mode that file will be open

Returns *file* – the opened file-object

pdb = None
If true the pdb debugger is automatically started after a failure

post_run_hooks = None
List of post-run hooks (captured functions called after this run)

pre_run_hooks = None
List of pre-run hooks (captured functions called before this run)

queue_only = None
If true then this run will only fire the queued_event and quit

result = None
The return value of the main function

root_logger = None
The root logger that was used to create all the others

run_logger = None
The logger that is used for this run

start_time = None
The datetime when this run was started

status = None
The current status of the run, from QUEUED to COMPLETED

stop_time = None
The datetime when this run stopped

unobserved = None
Indicates whether this run should be unobserved

warn_if_unobserved()

1.15.4 ConfigScope

class sacred.config.config_scope.**ConfigScope**(*func*)

1.15.5 ConfigDict

class sacred.config.config_dict.**ConfigDict**(*d*)

1.15.6 Observers

class sacred.observers.**RunObserver**
Defines the interface for all run observers.

artifact_event(*name*, *filename*, *metadata=None*, *content_type=None*)

completed_event(*stop_time*, *result*)

failed_event(*fail_time*, *fail_trace*)

heartbeat_event(*info*, *captured_out*, *beat_time*, *result*)

interrupted_event(*interrupt_time*, *status*)

join()

log_metrics(*metrics_by_name*, *info*)

priority = 0

queued_event(*ex_info*, *command*, *host_info*, *queue_time*, *config*, *meta_info*, *_id*)

resource_event(*filename*)

started_event(*ex_info*, *command*, *host_info*, *start_time*, *config*, *meta_info*, *_id*)

class sacred.observers.**MongoObserver**(*url: Optional[str] = None*, *db_name: str = 'sacred'*, *collection: str = 'runs'*, *collection_prefix: str = ''*, *overwrite: Union[int*, *str*, *None] = None*, *priority: int = 30*, *client: Optional[pymongo.MongoClient] = None*, *failure_dir: Union[str*, *bytes*, *pathlib.Path*, *None] = None*, ***kwargs*)

COLLECTION_NAME_BLACKLIST = {'_properties', 'fs.chunks', 'fs.files', 'search_space', '

VERSION = 'MongoObserver-0.7.0'

artifact_event(*name*, *filename*, *metadata=None*, *content_type=None*)

completed_event(*stop_time*, *result*)

classmethod create(**args*, ***kwargs*)

classmethod create_from(**args*, ***kwargs*)

failed_event(*fail_time*, *fail_trace*)

final_save(*attempts*)

heartbeat_event(*info*, *captured_out*, *beat_time*, *result*)

initialize(*runs_collection*, *fs*, *overwrite=None*, *metrics_collection=None*, *failure_dir=None*, *priority=30*)

insert()

interrupted_event(*interrupt_time*, *status*)

log_metrics(*metrics_by_name*, *info*)

Store new measurements to the database.

Take measurements and store them into the metrics collection in the database. Additionally, reference the metrics in the info["metrics"] dictionary.

queued_event(*ex_info*, *command*, *host_info*, *queue_time*, *config*, *meta_info*, *_id*)

resource_event(*filename*)

save()

save_sources(*ex_info*)

started_event(*ex_info*, *command*, *host_info*, *start_time*, *config*, *meta_info*, *_id*)

1.15.7 Host Info

Helps to collect information about the host of an experiment.

sacred.host_info.**get_host_info**(*additional_host_info: List[sacred.host_info.HostInfoGetter] = None*)

Collect some information about the machine this experiment runs on.

Returns *dict* – A dictionary with information about the CPU, the OS and the Python version of this machine.

sacred.host_info.**host_info_getter**(*func*, *name=None*)

The decorated function is added to the process of collecting the host_info.

This just adds the decorated function to the global `sacred.host_info.host_info_gatherers` dictionary. The functions from that dictionary are used when collecting the host info using *get_host_info()*.

Parameters

- **func** (*callable*) – A function that can be called without arguments and returns some json-serializable information.
- **name** (*str, optional*) – The name of the corresponding entry in host_info. Defaults to the name of the function.

Returns *The function itself.*

1.15.8 Custom Exceptions

class sacred.utils.**SacredInterrupt**

Base-Class for all custom interrupts.

For more information see *Custom Interrupts*.

class sacred.utils.**TimeoutInterrupt**

Signal that the experiment timed out.

This exception can be used in client code to indicate that the run exceeded its time limit and has been interrupted because of that. The status of the interrupted run will then be set to `TIMEOUT`.

For more information see *Custom Interrupts*.

1.16 Internals of Sacred

This section is meant as a reference for Sacred developers. It should give a high-level description of some of the more intricate internals of Sacred.

1.16.1 Configuration Process

The configuration process is executed when an experiment is started, and determines the final configuration that should be used for the run:

1. Determine the order for running the ingredients
   - topological
   - in the order they where added
2. For each ingredient do:

- gather all config updates that apply (needs `config_updates`)
- gather all named configs to use (needs `named_configs`)
- gather all fallbacks that apply from subrunners (needs `subrunners.config`)
- make the fallbacks read-only
- run all named configs and use the results as additional config updates, but with lower priority than the global ones. (needs `named_configs`, `config_updates`)
- run all normal configs
- update the global `config`
- run the config hook
- update the global `config_updates`

CHAPTER 2

Index

genindex

Python Module Index

e

s

Index

Symbols

A

B

C

D

E

S

T

U

V

W