pyexperiment Documentation

Release 0.6.0

Peter Duerr

August 14, 2015

Contents

1 Motivating Example **3**
- 1.1 CLI . . . 3
- 1.2 State . . . 4
- 1.3 Logging . . . 4
- 1.4 Configuration . . . 5
- 1.5 Timing . . . 6
- 1.6 Loading State . . . 6
- 1.7 Plotting . . . 6

2 Code **9**
- 2.1 pyexperiment . . . 9
- 2.2 pyexperiment.Config . . . 9
- 2.3 pyexperiment.Logger . . . 10
- 2.4 pyexperiment.State . . . 11
- 2.5 pyexperiment.experiment . . . 12
- 2.6 pyexperiment.utils.HierarchicalMapping . . . 13
- 2.7 pyexperiment.utils.DelegateCall . . . 14
- 2.8 pyexperiment.utils.Singleton . . . 15
- 2.9 pyexperiment.utils.interactive . . . 15
- 2.10 pyexperiment.utils.plot . . . 16
- 2.11 pyexperiment.utils.printers . . . 16
- 2.12 pyexperiment.utils.stdout_redirector . . . 17
- 2.13 pyexperiment.utils . . . 17

3 Indices and tables **19**

Python Module Index **21**

For a brief introduction and installation instructions, check the README on github.

The main idea behind pyexperiment is to make starting a short experiment as simple as possible without sacrificing the comfort of some basic facilities like a reasonable CLI, logging, persistent state, configuration management, plotting and timing.

CHAPTER 1

Motivating Example

Let's assume we need to write a quick and clean script that reads a couple of files with time series data and computes the average value. We also want to generate a plot of the data.

1.1 CLI

To be efficient, we split our script into three functions. *read* should read one or multiple raw data files, *average* should compute the average of the data in the read files and *plot* should plot the data over time. Moreover, we want to add a test for the average function to make sure it's working correctly. In pyexperiment, we can achieve a CLI with these functions very easily. Let's write the basic structure of our script to a file, say 'analyzer.py'

```
#!/usr/bin/env python
from pyexperiment import experiment

def read(*filenames):
    pass

def average():
    pass

def plot():
    pass

class AverageTest(unittest.TestCase):
    """Tests the average function
    """
    pass

if __name__ == 'main':
    experiment.main(commands=[load, average, plot],
                    tests=[AverageTest])
```

Without any further code, the call to `pyexperiment.experiment.main()` will set up a command line interface for our application that allows executing the three functions *read*, *average*, and *plot* by calling `analyzer read ./datafile1 ./datafile2`, *analyzer average*, and *analyzer plot* respectively. A call to `analyzer test` will run our (yet unimplemented) unittests.

1.2 State

Next, let's write the *read* function and save the loaded data to a persistent state file. To this end, we can use pyexperiment's `pyexperiment.State` which we get by adding `from pyexperiment import state` and `from pyexperiment.experiment import save_state` to the top of 'analyzer.py'. Then, assuming the data files consist of comma separated values, we can achieve this by defining *load* as

```
def read(*filenames):
    """Reads data files and stores their content
    """
    # Initialize the state with an empty list for the data
    state['data'] = []
    for filename in filenames:
        with open(filename) as f:
            state['data'] += [float(data)
                              for data in f.readlines()]
    save_state()
```

Note that internally, the implementation of `pyexperiment.State` uses a `pyexperiment.utils.Singleton.Singleton` wrapped by `pyexperiment.utils.Singleton.delegate_single` so that wherever you access the state you are accessing the same underlying data structure (in a thread safe way).

1.3 Logging

In order to better understand our results, it would be nice to have a logger to print some debug output, e.g., printing the names of the files we load and how many data points they contain. A few calls to pyexperiment's `pyexperiment.log` will do the job - simply add `from pyexperiment import log` and add logging calls at the desired level:

```
def read(*filenames):
    """Reads data files and stores their content
    """
    # Initialize the state with an empty list for the data
    state['data'] = []
    for filename in filenames:
        log.info("Reading file %s", filename)
        with open(filename) as f:
            data = [float(data)
                    for data in f.readlines()]
            if len(data) == 0:
                log.warning("Datafile %s does not contain any data",
                            filename)
            log.debug("Read %i datapoints", len(data))
            state['data'] += data
    save_state()
```

At this point, let's factor out a method that reads a single file to make our code more readable

```
def read_file(filenam):
    """Read a file and return the data
    """
    log.info("Reading file %s", filename)
    with open(filename) as f:
        data = [float(data)
        for data in f.readlines()]
        if len(data) == 0:
```

```
            log.warning("Datafile %s does not contain any data",
                        filename)
        log.debug("Read %i datapoints", len(data))
        return data


def read(*filenames):
    """Reads data files and stores their content
    """
    # Initialize the state with an empty list for the data
    state['data'] = []
    for filename in filenames:
        state['data'] += read_file(filename)
    save_state()
```

1.4 Configuration

You will notice that by default, pyexperiment does not log to a file and it will only print messages at, or above the 'WARNING' level. If you would like to see more (or less) messages, you can change the logging level by running the analyzer with an additional argument e.g., `--verbosity DEBUG`. In general, any configuration option can be set from the command line with `-o [level[.level2.[...]]].key value`.

The *verbosity* configuration value is predefined by pyexperiment, but we can use the same configuration mechanism for our own parameters. This is achieved by defining a specification for the configuration and passing it as the `config_spec` argument to the `pyexperiment.experiment.main()` call. For example, we may want to add an option to ignore data files longer than a certain length:

```
CONFIG_SPEC = ("[read]\n"
               "max_length = integer(min=1, default=100)\n")

if __name__ == '__main__':
    experiment(commands=[load, average, plot],
               tests=[AverageTest],
               config_spec=CONFIG_SPEC)
```

We can then access the parameters by adding `from pyexperiment import conf` at the top of 'analyzer.py' and calling `pyexperiment.conf` like a dictionary with the levels of the configuration separated by dots:

```
def read(*filenames):
    """Reads data files and stores their content
    """
    # Initialize the state with an empty list for the data
    state['data'] = []

    # Get the max length from the configuration
    max_length = conf['read.max_length']

    for filename in filenames:
        data = read_file(filename)
        if len(data < max_length):
            state['data'] += data
    save_state()
```

By default, pyexperiment will try to load a file called 'config.ini' (if necessary, one can of course override this default filename). To generate an initial configuration file with the default options, simply run `analyzer save_config ./config.ini`. Any options set in the resulting file will be used in future runs.

1.5 Timing

If we are loading big data files, we may also be interested to learn how much time it takes to load an individual file - there may be some room for optimization. To measure the time it takes to load a file and compute statistics, we can use pyexperiment's timing function from the `pyexperiment.Logger`.

```
def read(*filenames):
    """Reads data files and stores their content
    """
    # Initialize the state with an empty list for the data
    state['data'] = []

    # Get the max length from the configuration
    max_length = conf['read.max_length']

    for filename in filenames:
        with log.timed("read_file"):
            data = read_file(filename)
        if len(data < max_length):
            state['data'] += data
    save_state()
    log.print_timings()
```

1.6 Loading State

To average over our data, we will need the state from when we called our script with the *read* command. By default, pyexperiment does not load the state saved in previous runs, but we can load it manually with the `pyexperiment.State.load()` function.

```
def average():
    """Returns the average of the data stored in state
    """
    state.load(conf['pyexperiment.state_filename'])
    data = state['data']
    return sum(data)/len(data)
```

We can now call `analyzer.py load file1 file2` followed by `analyzer.py average` to get the average of the data points in our files. If you add timing calls you will notice that `pyexperiment.state.load()` returns almost immediately. By default, pyexperiment loads entries in the `pyexperiment.State` only when they are needed.

1.7 Plotting

Finally, let's add the *setup_figure* function with `from pyexperiment.utils.plot import setup_figure` as well as pyplot (with `from matplotlib import pyplot as plt`) and write the plotter:

```
def plot():
    """Plots the data saved in the state
    """
    state.load(conf['pyexperiment.state_filename'])
    data = state['data']
```

```
    fig = setup_figure('Time Series Data')
    plt.plot(data)
```

With this code in place, we can now call *analyze.py plot* which will open an window with the plotted data. To make the window fullscreen, press the 'f' key on your keyboard, to close the window press 'q'.

CHAPTER 2

Code

pyexperiment	The pyexperiment module - quick and clean experiments with Python.
pyexperiment.Config([filename, spec, ...])	Represents a singleton configuration object.
pyexperiment.Logger	
pyexperiment.State([filename])	Represents persistent state of an experiment.
pyexperiment.experiment	
pyexperiment.utils.HierarchicalMapping	
pyexperiment.utils.DelegateCall	
pyexperiment.utils.Singleton	
pyexperiment.utils.interactive	
pyexperiment.utils.plot	
pyexperiment.utils.printers	
pyexperiment.utils.stdout_redirector	
pyexperiment.utils	

2.1 pyexperiment

The pyexperiment module - quick and clean experiments with Python.

2.2 pyexperiment.Config

class pyexperiment.**Config**(*filename=None*, *spec=u'configspec.ini'*, *options=None*, *default_spec=None*)

Represents a singleton configuration object.

__init__(*filename=None*, *spec=u'configspec.ini'*, *options=None*, *default_spec=None*)

Initializer

Methods

__init__([filename, spec, options, default_spec])	Initializer
base_keys()	Returns the keys of the first level of the mapping
clear() -> None. Remove all items from D.)	
get(key[, default])	Get the key or return the default value if provided
get_instance()	Get the singleton instance if its initialized.

Continued on next page

Table 2.2 – continued from previous page

get_or_set(key, value)	Either gets the value associated with key or set it
initialize(*args, **kwargs)	Initializes the singleton.
items(() -> list of D's (key, value) pairs, ...)	
iteritems(() -> an iterator over the (key, ...)	
iterkeys(() -> an iterator over the keys of D)	
itervalues(...)	
keys(() -> list of D's keys)	
load(filename[, spec, options])	Loads a configuration from filename (or string).
merge(other)	Merge in another mapping, giving precedence to self
override_with_args(options)	Override configuration with option dictionary
pop((k[,d]) -> v, ...)	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem(() -> (k, v), ...)	as a 2-tuple; but raise KeyError if D is empty.
reset_instance()	Overloads reset_instance to reset the DEFAULT_CONFIG
save(filename)	Write configuration to file
section_keys()	Returns the keys of the sections (and subsections) of the mapping
setdefault((k[,d]) -> D.get(k,d), ...)	
show()	Pretty-prints the content of the mapping
update(([E, ...)	If E present and has a .keys() method, does: for k in E: D[k] = E[k]
validate_config(config)	Validate configuration
values(() -> list of D's values)	

Attributes

CONFIG_SPEC_PATH	Path of the file with the specification for configurations.
DEFAULT_CONFIG	Default configuration, later used by initialize
SECTION_SEPARATOR	

2.3 pyexperiment.Logger

Provides a multiprocessing-safe logger with colored console output, rotating log files, and easy block-timing based on the logging module.

Logging made easy. Just use the module level functions debug, warning, info etc. to log messages. At some point, preferably at the start of your program, call init_main to initialize the logger and choose at which level you want to log to the console and or rotating log files.

Timing made easy. Just use the module level function timed in a 'with' block to log timing of the code in the block. You can also get a summary with statistics of all your timed blocks by calling print_timings.

Written by Peter Duerr, inspired by zzzeek's and airmind's examples on Stackoverflow (http://stackoverflow.com/a/894284/2481888, http://stackoverflow.com/a/384125/2481888).

pyexperiment.Logger.**CONSOLE_FORMAT = u'[%(levelname)-19s] [%(relativeCreated)s] $BOLD%(message)s$RESET'**
The format used for logging to the console

pyexperiment.Logger.**CONSOLE_STREAM_HANDLER = <logging.StreamHandler object at 0x7f6f0ceb7cd0>**
The stream handler for the console (can be mocked for testing)

class pyexperiment.Logger.**ColorFormatter**(*msg*, *use_color=True*)
Formats logged messages with optional added color for the log level

format(*record*)
Format the log

pyexperiment.Logger.**FILE_FORMAT = u'[%(relativeCreated)10.5fs] [%(levelname)-1s] %(message)-50s (%(filename)s:%**
The format used for logging to file

pyexperiment.Logger.**FILE_FORMAT_STD_MSG_LEN = 50**
How much space should be reserved for a normal message in the log file.

class pyexperiment.Logger.**Logger**(*console_level=20*, *filename=None*, *file_level=10*, *no_backups=5*)
Implements a multiprocessing-safe logger with timing and colored console output.

close()
Close the logger

class pyexperiment.Logger.**MPRotLogHandler**(*filename*, *level=10*, *no_backups=0*)
Multiprocessing-safe handler for rotating log files

emit(*record*)
Emits logged message by delegating it

setFormatter(*formatter*)
Overload the setFormatter method

class pyexperiment.Logger.**PreInitLogHandler**
Handles messages before the main logger is initialized.

emit(*msg*)
Catch logs and store them for later

class pyexperiment.Logger.**TimingLogger**(*console_level=20*, *filename=None*, *file_level=10*, *no_backups=5*)
Provides a logger with a *timed* context.

Calling code in the *timed* context will collect execution timing statistics.

close()
Make sure the delegated calls are all done...

print_timings()
Prints a summary of the timings collected with 'timed'.

timed(**args*, ***kwds*)
Timer to be used in a with block. If the level is not None, logs the timing at the specified level. If save_result is True, captures the result in the timings dictionary.

2.4 pyexperiment.State

class pyexperiment.**State**(*filename=None*)
Represents persistent state of an experiment.

__init__(*filename=None*)
Initializer

Methods

__init__([filename])	Initializer
base_keys()	Returns the keys of the first level of the mapping
clear(() -> None. Remove all items from D.)	

Continued on next page

Table 2.4 – continued from previous page

do_rollover(filename[, rotate_n_state_files])	Rotate state files (as in logging module).
get(key[, default])	Get the key or return the default value if provided
get_instance()	Get the singleton instance
get_or_set(key, value)	Either gets the value associated with key or set it
items(() -> list of D's (key, value) pairs, ...)	
iteritems(() -> an iterator over the (key, ...)	
iterkeys(() -> an iterator over the keys of D)	
itervalues(...)	
keys(() -> list of D's keys)	
load([filename, lazy, raise_error])	Loads state from a h5f file
merge(other)	Merge in another mapping, giving precedence to self
need_saving()	Checks if state needs to be saved
pop((k[,d]) -> v, ...)	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem(() -> (k, v), ...)	as a 2-tuple; but raise KeyError if D is empty.
reset_instance()	Reset the singleton
save(filename[, rotate_n_state_files, ...])	Saves state to a h5f file, rotating if necessary
section_keys()	Returns the keys of the sections (and subsections) of the mapping
setdefault((k[,d]) -> D.get(k,d), ...)	
show()	Shows the state
update(([E, ...)	If E present and has a .keys() method, does: for k in E: D[k] = E[k]
values(() -> list of D's values)	

Attributes

SECTION_SEPARATOR

2.5 pyexperiment.experiment

Framework for quick and clean experiments with python.

For a simple example to adapt to your own needs, check the example file.

Written by Peter Duerr.

pyexperiment.experiment.**COMMANDS = []**
List of all commands for the experiment. Filled by main.

pyexperiment.experiment.**DEFAULT_CONFIG_FILENAME = u'./config.ini'**
Default name for the configuration file

pyexperiment.experiment.**DEFAULT_CONFIG_SPECS = u"[pyexperiment]\nverbosity = option('DEBUG','INFO','WAR**
Default specification for the experiment's configuration

pyexperiment.experiment.**TESTS = []**
List of all tests for the experiment. Filled by main.

pyexperiment.experiment.**activate_autocompletion**()
Activate auto completion for your experiment with zsh or bash.

Call with eval "$(script_name activate_autocompletion)". In zsh you may need to call *autoload bashcompinit* and *bashcompinit* first.

pyexperiment.experiment.**collect_commands**(*commands*)
Add default commands

pyexperiment.experiment.**configure**(*commands*, *config_specs*, *description*)
Load configuration from command line arguments and optionally, a configuration file. Possible command line arguments depend on the list of supplied commands, the configuration depends on the supplied configuration specification.

pyexperiment.experiment.**format_command_help**(*commands*)
Format the docstrings of the commands.

pyexperiment.experiment.**help**(**args*)
Shows help for a specified command.

pyexperiment.experiment.**init_log**()
Initialize the logger based on the configuration

pyexperiment.experiment.**main**(*commands=None*, *config_spec=u''*, *tests=None*, *description=None*)
Parses command line arguments and configuration, then runs the appropriate command.

pyexperiment.experiment.**save_config**(*filename*)
Save a configuration file to a filename

pyexperiment.experiment.**setup_arg_parser**(*commands*, *description*)
Setup the argument parser for the experiment

pyexperiment.experiment.**show_config**()
Print the configuration

pyexperiment.experiment.**show_state**(**arguments*)
Shows the contents of the state loaded by the configuration or from the file specified as an argument.

pyexperiment.experiment.**show_tests**(**args*)
Show available tests for the experiment

pyexperiment.experiment.**test**(**args*)
Run tests for the experiment

2.6 pyexperiment.utils.HierarchicalMapping

Provide flat, point separated interface to nested mappings

As the zen of python says, flat is better than nested. In many cases, however, it makes sense to store data in a nested data structure. To bridge the gap, the HierarchicalMapping defines an abstract base class for data structures that can be treated like an ordinary mapping from strings to values, but with the advantage that the values for keys containing a level separator, e.g., "level1.level2.level3" are stored in a nested hierarchy of mappings.

Written by Peter Duerr

class pyexperiment.utils.HierarchicalMapping.**HierarchicalDict**
Instance of the HierarchicalMapping based on dict

class pyexperiment.utils.HierarchicalMapping.**HierarchicalMapping**
ABC for flat mutable mappings where all keys are strings.

Levels of hierarchy are indicated by a separator character and the storage is implemented as a hierarchy of nested Mutable mappings.

SECTION_SEPARATOR = u'.'
Separates the hierarchy levels

__delitem__(*key*)
Delete an item

__getitem__(*key*)
Get an item

__iter__()
Need to define __iter__ to make it a MutableMapping

__len__()
Returns the number of entries in the mapping

__repr__()
Get a representation of the mapping

__setitem__(*key*, *value*)
Set an item

base_keys()
Returns the keys of the first level of the mapping

get(*key*, *default=None*)
Get the key or return the default value if provided

get_or_set(*key*, *value*)
Either gets the value associated with key or set it This can be useful as an easy way of

merge(*other*)
Merge in another mapping, giving precedence to self

section_keys()
Returns the keys of the sections (and subsections) of the mapping

show()
Pretty-prints the content of the mapping

class pyexperiment.utils.HierarchicalMapping.**HierarchicalOrderedDict**
Instance of the HierarchicalMapping based on an OrderedDict.

2.7 pyexperiment.utils.DelegateCall

Provides a multiprocessing-safe way to aggregate results from multiple function calls.

In multi-process programs, it is oft often useful to delegate a function call - e.g., writing a log message to a file - to another process to avoid conflicts. pyexperiment.utils.DelegateCall implements a functor that, when called, passes the argument data to a function running in a thread of the process that created the DelegateCall object. The callback itself must be thread-safe though.

Written by Peter Duerr

class pyexperiment.utils.DelegateCall.**DelegateCall**(*callback*)
Helper class that provides a multiprocessing-safe way to aggregate results from multiple function calls.

The arguments to the __call__ function are passed through a multiprocessing.Queue to the process where the class was initialized (i.e., all arguments must be serializable).

__call__(*data*)
Send data, can be called from any process

join()
Returns true if there are currently no pending callbacks

2.8 pyexperiment.utils.Singleton

Provides data structures for unique objects.

The *Singleton* class whose implementation is inspired by Tornado's singleton (tornado.ioloop.IOLoop.instance()), can be inherited by classes which only need to be instantiated once (for example a global settings class such as `pyexperiment.Config`). This design pattern is often criticized, but it is hard to beat in terms of simplicity.

A variant of the *Singleton*, the *DefaultSingleton* provides an abstract base class for classes that are only instantiated once, but need to provide an instance before being properly initialized (such as *pyexperiment.Logger.Logger*).

The function *delegate_singleton* 'thunks' a *Singleton* so that calls to the singleton instance's methods don't need to be ugly chain calls.

Written by Peter Duerr (Singleton inspired by Tornado's implementation)

class `pyexperiment.utils.Singleton.`**`DefaultSingleton`**
ABC for singleton that does not automatically initialize

If get_instance is called on an uninitialized DefaultSingleton, a pseudo-instance is returned.

Sub-classes need to implement the function *_get_pseudo_instance* that returns a pseudo instance.

classmethod **`get_instance`**`()`
Get the singleton instance if its initialized. Returns, the pseudo instance if not.

classmethod **`initialize`**`(`**args*, ***kwargs*`)`
Initializes the singleton. After calling this function, the real instance will be used.

classmethod **`reset_instance`**`()`
Reset the singleton instance if its initialized.

class `pyexperiment.utils.Singleton.`**`Singleton`**
Singleton base-class (or mixin)

classmethod **`get_instance`**`()`
Get the singleton instance

classmethod **`reset_instance`**`()`
Reset the singleton

`pyexperiment.utils.Singleton.`**`delegate_singleton`**`(`*singleton*`)`
Creates an object that delegates all calls to the singleton

`pyexperiment.utils.Singleton.`**`delegate_special_methods`**`(`*singleton*`)`
Decorator that delegates special methods to a singleton

2.9 pyexperiment.utils.interactive

Provides helper functions for interactive prompts

Written by Peter Duerr

`pyexperiment.utils.interactive.`**`embed_interactive`**`(`***kwargs*`)`
Embed an interactive terminal into a running python process

2.10 pyexperiment.utils.plot

Provides setup utilities for matplotlib figures.

The *setup_plotting* function will configure basic plot options, such as font size, line width, etc. Calls after the first call are ignored unless the override flag is set to True. The *setup_figure* function will call *setup_plotting* without overriding an existing setup, and then return the handle to a new figure, pre-configured with the 'q' key bound to close the figure.

The *AsyncPlot* class provides a simple way to plot some datapoints in a separate process without blocking the execution of the main program. Just create an *AsyncPlot* object and use the *plot* method. By default, the window created by the *AsyncPlot* will stay open until you close it. To close the window programatically, call the *close* method on the *AsyncPlot* object.

Written by Peter Duerr.

class pyexperiment.utils.plot.**AsyncPlot**(*name=u'pyexperiment'*, *labels=None*, *x_scale=u'linear'*, *y_scale=u'linear'*)

Plot asynchronously in a different process

close()

Close the figure, join the process

plot(**args*, ***kwargs*)

Plots the data in the separate process

static plot_process(*queue*, *name*, *labels=None*, *x_scale=u'linear'*, *y_scale=u'linear'*)

Grabs data from the queue and plots it in a named figure

pyexperiment.utils.plot.**quit_figure_on_key**(*key*, *figure=None*)

Add handler to figure (defaults to current figure) that closes it on a key press event.

pyexperiment.utils.plot.**setup_figure**(*name=u'pyexperiment'*)

Setup a figure that can be closed by pressing 'q' and saved by pressing 's'.

pyexperiment.utils.plot.**setup_plotting**(*options=None*, *override_setup=True*)

Setup basic style for matplotlib figures

2.11 pyexperiment.utils.printers

Provides printing in color

Written by Peter Duerr, inspired by a stackoverflow comment by airmind (http://stackoverflow.com/a/384125/2481888).

pyexperiment.utils.printers.**COLOR_SEQ = u'\x1b[1;%dm'**

Sequence used to set color for console formatting

pyexperiment.utils.printers.**RESET_SEQ = u'\x1b[0m'**

Sequence used to reset console formatting

pyexperiment.utils.printers.**colorize**(*string*, *color_s*)

Colorize a string

pyexperiment.utils.printers.**create_printer**(*color*)

Creates the printer for the corresponding color

pyexperiment.utils.printers.**print_examples**(*message=None*, **args*)

Print an example message with every available printer

2.12 pyexperiment.utils.stdout_redirector

Context to redirect stdout (inspired by a tutorial by Eli Bendersky)

Adapted by Peter Duerr

pyexperiment.utils.stdout_redirector.**stdout_err_redirector**(**args*, ***kwds*)
Redirect standard out and err to a buffer

pyexperiment.utils.stdout_redirector.**stdout_redirector**(**args*, ***kwds*)
Redirects standard out to a buffer

2.13 pyexperiment.utils

Some utility functions for pyexperiment

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*

p

pyexperiment, 9
pyexperiment.experiment, 12
pyexperiment.Logger, 10
pyexperiment.utils, 17
pyexperiment.utils.DelegateCall, 14
pyexperiment.utils.HierarchicalMapping, 13
pyexperiment.utils.interactive, 15
pyexperiment.utils.plot, 16
pyexperiment.utils.printers, 16
pyexperiment.utils.Singleton, 15
pyexperiment.utils.stdout_redirector, 17

Symbols

__call__() (pyexperiment.utils.DelegateCall.DelegateCall method), 14
__delitem__() (pyexperiment.utils.HierarchicalMapping.HierarchicalMapping method), 13
__getitem__() (pyexperiment.utils.HierarchicalMapping.HierarchicalMapping method), 13
__init__() (pyexperiment.Config method), 9
__init__() (pyexperiment.State method), 11
__iter__() (pyexperiment.utils.HierarchicalMapping.HierarchicalMapping method), 14
__len__() (pyexperiment.utils.HierarchicalMapping.HierarchicalMapping method), 14
__repr__() (pyexperiment.utils.HierarchicalMapping.HierarchicalMapping method), 14
__setitem__() (pyexperiment.utils.HierarchicalMapping.HierarchicalMapping method), 14

A

activate_autocompletion() (in module pyexperiment.experiment), 12
AsyncPlot (class in pyexperiment.utils.plot), 16

B

base_keys() (pyexperiment.utils.HierarchicalMapping.HierarchicalMapping method), 14

C

close() (pyexperiment.Logger.Logger method), 11
close() (pyexperiment.Logger.TimingLogger method), 11
close() (pyexperiment.utils.plot.AsyncPlot method), 16
collect_commands() (in module pyexperiment.experiment), 12
COLOR_SEQ (in module pyexperiment.utils.printers), 16
ColorFormatter (class in pyexperiment.Logger), 10
colorize() (in module pyexperiment.utils.printers), 16
COMMANDS (in module pyexperiment.experiment), 12
Config (class in pyexperiment), 9
configure() (in module pyexperiment.experiment), 12
CONSOLE_FORMAT (in module pyexperiment.Logger), 10
CONSOLE_STREAM_HANDLER (in module pyexperiment.Logger), 10
create_printer() (in module pyexperiment.utils.printers), 16

D

DEFAULT_CONFIG_FILENAME (in module pyexperiment.experiment), 12
DEFAULT_CONFIG_SPECS (in module pyexperiment.experiment), 12
DefaultSingleton (class in pyexperiment.utils.Singleton), 15
delegate_singleton() (in module pyexperiment.utils.Singleton), 15
delegate_special_methods() (in module pyexperiment.utils.Singleton), 15
DelegateCall (class in pyexperiment.utils.DelegateCall), 14

E

embed_interactive() (in module pyexperiment.utils.interactive), 15
emit() (pyexperiment.Logger.MPRotLogHandler method), 11
emit() (pyexperiment.Logger.PreInitLogHandler method), 11

F

FILE_FORMAT (in module pyexperiment.Logger), 10
FILE_FORMAT_STD_MSG_LEN (in module pyexperiment.Logger), 11
format() (pyexperiment.Logger.ColorFormatter method), 10
format_command_help() (in module pyexperiment.experiment), 13

G

get() (pyexperiment.utils.HierarchicalMapping.HierarchicalMapping method), 14
get_instance() (pyexperiment.utils.Singleton.DefaultSingleton class method), 15
get_instance() (pyexperiment.utils.Singleton.Singleton class method), 15
get_or_set() (pyexperiment.utils.HierarchicalMapping.HierarchicalMapping method), 14

H

help() (in module pyexperiment.experiment), 13
HierarchicalDict (class in pyexperiment.utils.HierarchicalMapping), 13
HierarchicalMapping (class in pyexperiment.utils.HierarchicalMapping), 13
HierarchicalOrderedDict (class in pyexperiment.utils.HierarchicalMapping), 14

I

init_log() (in module pyexperiment.experiment), 13
initialize() (pyexperiment.utils.Singleton.DefaultSingleton class method), 15

J

join() (pyexperiment.utils.DelegateCall.DelegateCall method), 14

L

Logger (class in pyexperiment.Logger), 11

M

main() (in module pyexperiment.experiment), 13
merge() (pyexperiment.utils.HierarchicalMapping.HierarchicalMapping method), 14
MPRotLogHandler (class in pyexperiment.Logger), 11

P

plot() (pyexperiment.utils.plot.AsyncPlot method), 16
plot_process() (pyexperiment.utils.plot.AsyncPlot static method), 16
PreInitLogHandler (class in pyexperiment.Logger), 11
print_examples() (in module pyexperiment.utils.printers), 16
print_timings() (pyexperiment.Logger.TimingLogger method), 11
pyexperiment (module), 9
pyexperiment.experiment (module), 12
pyexperiment.Logger (module), 10
pyexperiment.utils (module), 17
pyexperiment.utils.DelegateCall (module), 14
pyexperiment.utils.HierarchicalMapping (module), 13
pyexperiment.utils.interactive (module), 15
pyexperiment.utils.plot (module), 16
pyexperiment.utils.printers (module), 16
pyexperiment.utils.Singleton (module), 15
pyexperiment.utils.stdout_redirector (module), 17

Q

quit_figure_on_key() (in module pyexperiment.utils.plot), 16

R

reset_instance() (pyexperiment.utils.Singleton.DefaultSingleton class method), 15
reset_instance() (pyexperiment.utils.Singleton.Singleton class method), 15
RESET_SEQ (in module pyexperiment.utils.printers), 16

S

save_config() (in module pyexperiment.experiment), 13
section_keys() (pyexperiment.utils.HierarchicalMapping.HierarchicalMapping method), 14
SECTION_SEPARATOR (pyexperiment.utils.HierarchicalMapping.HierarchicalMapping attribute), 13
setFormatter() (pyexperiment.Logger.MPRotLogHandler method), 11
setup_arg_parser() (in module pyexperiment.experiment), 13
setup_figure() (in module pyexperiment.utils.plot), 16
setup_plotting() (in module pyexperiment.utils.plot), 16
show() (pyexperiment.utils.HierarchicalMapping.HierarchicalMapping method), 14
show_config() (in module pyexperiment.experiment), 13
show_state() (in module pyexperiment.experiment), 13
show_tests() (in module pyexperiment.experiment), 13
Singleton (class in pyexperiment.utils.Singleton), 15
State (class in pyexperiment), 11
stdout_err_redirector() (in module pyexperiment.utils.stdout_redirector), 17
stdout_redirector() (in module pyexperiment.utils.stdout_redirector), 17

T

test() (in module pyexperiment.experiment), 13
TESTS (in module pyexperiment.experiment), 12
timed() (pyexperiment.Logger.TimingLogger method), 11
TimingLogger (class in pyexperiment.Logger), 11