sloth

Release 0.3.dev0

Legorooj, FluffyKoalas

Jun 16, 2020

API REFERENCE

CHAPTER

ONE

THE PYTHON SPEEDTESTING LIBRARY

`sloth` is a Python package for speedtesting python code and functions with as little code as necessary. It's easy to use and, unlike many projects, has decent documentation.

The idea behind this API is:

```
>>> from sloth import compare_sloth
>>> import timeit
>>> compare_sloth(timeit)
'sloth is loads better than timeit!'
```

See? described in 3 lines. *Everything* that `timeit` can do, `sloth` can do better. And *most* (speedtest-related) things `timeit` *can't* do `sloth` can do anyway.

Or, for the bash ninjas and command liners:

```
$ sloth compare timeit
sloth is loads better than timeit!
$ python3 -m sloth compare timeit
sloth is loads better than timeit!
```

CHAPTER

TWO

A QUICK EXAMPLE

```
>>> from sloth.simple import time_callable
>>> import time
>>> def my_func(a, b, c):
...     time.sleep(1)
...     print(a, b, c)
>>> time_callable(my_func, 2, 'a', 'b', 'c')
a b c
a b c
1.0150199500000028
```

CHAPTER

THREE

INSTALLATION

You can install sloth with pip:

```
pip install sloth-speedtest
```

Please see *Installation* for more information.

CHAPTER

FOUR

USAGE

Please see the *The sloth API* for API usage guidance. Please run `sloth -h` or `sloth --help` for help with the command line tool.

CHAPTER

FIVE

LINKS

- PyPI
- GitHub
- Docs

CHAPTER

SIX

INDEX

6.1 The `sloth` API

Most of the code is kept in submodules, not the `sloth` namespace. Eg timers are in the `sloth.timers` module. The only function in the main sloth namespace is `compare_sloth`, which is documented below.

6.1.1 `sloth.timers`: Timing functions and classes

The `sloth.timers` module contains functions and classes for timing code.

class sloth.timers.**StopWatch**
Simple stopwatch for capturing code execution time.

start()
Starts the *StopWatch*.

stop()
Clears the stopwatch and returns the time elapsed since the *start()* method has been called. This method will return 0 if start has not been called.

Returns The time - in seconds - elapsed since *start()* was called

Return type float

lap()
Returns the time elapsed since the *start()* method was called *without* clearing the stopwatch. This method will return 0 if start has not been called.

Returns The time - in seconds - elapsed since *start()* was called

Return type int

Timer(seconds, func, args=None, kwargs=None):
Simple timer that executes a function after a timed interval

Parameters

- **seconds** (*int*) – The number of seconds to call *func* after.
- **func** (*function*) – The function to call after *seconds* seconds have elapsed.
- **args** (*list or tuple or None*) – Positional arguments to pass to *func*.
- **kwargs** (*dict or None*) – Keyword arguments to pass to func.

Raises **TypeError** – if any of the arguments have incorrect types

sloth.timers.**start**()
Start the timer in the background. Eg:

```
>>> from sloth.timers import Timer
>>> from time import sleep
>>> def f():
...     print('Timer finished')
...
>>> def a():
...     t = Timer(5, f)
...     t.start()
...     print('Doing stuff')
...     sleep(3)
...     print('Doing more stuff')
...     sleep(4)
...     print('Finished doing stuff while the timer has executed in the␣
↪background')
...
>>> a()
Doing stuff
Doing more stuff
Timer
Finished doing stuff while the timer has executed in the background
```

sloth.timers.**stop**()
Cancels the timer. If this method is called *before* *start()*, then the timer *will not be run*.

sloth.timers.**join**(*timeout=None*)
Wait until the timer finishes or *timeout*, if not None, has elapsed.

Parameters **timeout** (*int or float or None*) – number of seconds to wait before returning. If None, then it returns when the timer has finished.

sloth.timers.**run**()
Run the timer in the main thread. This is the same as calling *start()* followed immediately by *join()*

sloth.timers.**daemon**

Warning: Please do not set this attribute unless you know what you are doing.

Controls whether or not the underlying thread that runs the timer is daemonic. This must be set *before* calling *start()*. This value defaults to *True*, meaning that the timer will be canceled if the program ends before completion.

Type bool

Default True

6.1.2 `sloth.simple`: Simple timing functions

`sloth.simple` is a module providing helpful functions that are generally simpler to use than the classes they wrap in the main API.

`sloth.simple.`**`call_after`**(*seconds*, *func*, *args=None*, *kwargs=None*)
Call *func* after *seconds* have elapsed.

Parameters

- **`seconds`** (*`int`*) – The number of seconds to call *func* after.
- **`func`** (*`function`*) – The function to call after *seconds* seconds have elapsed.
- **`args`** (*`list or tuple or None`*) – Positional arguments to pass to *func*.
- **`kwargs`** (*`dict or None`*) – Keyword arguments to pass to func.

Raises **`TypeError`** – if any of the arguments have incorrect types

`sloth.simple.`**`time_callable`**(*callable*, *n*, **args*, ***kwargs*)
Time how long it takes to execute *_callable*, run *iterations* times and averaged.

Parameters

- **`func`** (*`function`*) – The callable object to time
- **`n`** (*`int`*) – Number of times to run and average *_callable*
- **`args`** – Positional arguments to be passed directly to *_callable*
- **`kwargs`** – Keyword arguments to be passed directly to *_callable*

Returns how long it took for the callable to run, averaged

Return type float

Raises **`TypeError`** – if any of the arguments have incorrect types

`sloth.simple.`**`time_eval`**(*snippet*, *n*, *gbls=None*, *lcls=None*)
Speedtest `eval(statement, gbls, lcls)`. See the eval docs docs for more info.

Parameters

- **`snippet`** (*`str or bytes or code`*) – The code statement to evaluate.
- **`n`** (*`int`*) – Number of times to run and average the code.

Returns How long the evaluation took to run

Return type float

`sloth.simple.`**`time_exec`**(*snippet*, *n*, *gbls=None*, *lcls=None*)
Speedtest `exec(statement, gbls, lcls)`. See the exec docs docs for more info.

Parameters

- **`snippet`** (*`str or bytes or code`*) – The code statement to execute.
- **`n`** (*`int`*) – Number of times to run and average the code.

Returns How long the execute took to run

Return type float

6.1.3 sloth.raw: The base API

sloth.raw.tests: Standard tests classes

class sloth.raw.complex.tests.**TestCallable**(*_callable*)
This test tests a callable object; any object that `callable(obj)` is `True`.

Parameters **_callable** (*callable*) – The callable object to speedtest

run()
Speedtest the callable and return how long it took to execute.

Returns How long the callable took to run

Return type float

class sloth.raw.complex.tests.**TestCallableWithArgs**(*_callable*, **args*, ***kwargs*)
This test is similar to *TestCallable*, but the callable has args when using this test.

Parameters

- **_callable** (*callable*) – The callable object to speedtest
- **args** – Arguments to pass to the callable object.
- **kwargs** – Keyword arguments to pass to the callable object.

```
>>> from sloth.raw.complex.tests import TestCallableWithArgs
>>> def my_func(a, b, c, d='foo', e='bar'):
...     print(a, b, c, d, e)
>>> my_test = TestCallableWithArgs(my_func, 1, 2, 3, 'bar', 'foo')
>>> my_test.run()
1 2 3 bar foo
0.0
>>>
```

run(**args*, ***kwargs*)
Speedtest the callable and return how long it took to execute. `args` and `kwargs` can be used to override those passed in at creation.

Returns How long the callable took to run

Return type float

class sloth.raw.complex.tests.**TestEval**(*statement*, *gbls=None*, *lcls=None*)
Speedtest `eval(statement, gbls, lcls)`. See the eval docs docs for more info.

Parameters **statement** (*str or bytes or code*) – The code statement to evaluate.

run(*gbls=None*, *lcls=None*)
Evaluate the statement, and return how long it took to execute. See the eval docs docs for more info.

Returns How long the evaluation took to run

Return type float

class sloth.raw.complex.tests.**TestExec**(*statement*, *gbls=None*, *lcls=None*)
Speedtest `exec(statement, gbls, lcls)`. See the exec docs for more info.

Parameters **statement** (*str or bytes or code*) – The code statement to execute.

run(*gbls=None*, *lcls=None*)
Execute the statement, and return how long it took to execute. See the exec docs for more info.

Returns How long the evaluation took to run

Return type float

sloth.raw.runners: Multiple test runners

This module provides utilities for running multiple tests, or averaging tests.

class sloth.raw.runners.**TestRunner**(*tests*)
Convenience class to run multiple tests.

Parameters **tests** (*list or tuple or set*) – List of *sloth.raw.base.Test* instances to run

run()
Generator, returns the results of running each test.

Return type generator

Returns Time it took to run each tests.

class sloth.raw.runners.**AverageTest**(*test*, *n=None*)
Run test *n* times and return the average time it took to run it.

Parameters

- **test** (Test) – Test to average
- **n** (*int*) – Number of times to run the test - the higher the more accurate. Defaults to 2.

run(*n=None*)
Run the average test, and return how long it took to run it, averaged.

Parameters **n** (*int*) – Use this to override the *n* parameter passed in upon creation.

Returns How long it took to run the test, averaged

Return type float

sloth.raw.base: The abstract base classes for testing

This module provides the abstract base classes for tests. They are abstract classes, and *must* be subclassed.

class sloth.raw.base.**Test**(*metaclass=abc.ABCMeta*)
This is the base class for *Tests*. That also includes sloth.runners.AverageRunner(), though that may be more accurately classed as a test *runner*, not a test.

abstract run()
This method is abstract, to be overridden on subclasses. This is the method called by the test runners to run the test.

sloth.**compare_sloth**(*against*)
Returns a string which is the result of comparing sloth to *against*.

Parameters **against** (*module*) – Module/Package to compare sloth to.

Returns String describing sloth vs *against*

Return type str

6.2 Installation

`sloth` can be installed in two ways:

- Through `pip` and PyPI:

```
pip install sloth-speedtest
```

- From source:
  1. Download the latest stable release from our Releases Page.
  2. Decompress it, and navigate into the directory in a command line.
  3. Install with `pip install .`. Make sure you are using the correct python interpreter.

6.2.1 Installing the development release

- `pip install https://github.com/fluffykoalas/sloth/archive/dev.tar.gz`

6.2.2 Dependencies

`sloth` needs *click>=7.0*, and *setuptools>=40.0* to work. These are installed automatically when you install `sloth`.

6.3 Changelog

6.3.1 v0.3 (In development)

Features

- The planned extension API has arrived. See the Extension API for more details.

Incompatible changes

- Support for python 3.5 has been dropped.

6.3.2 v0.2

Bug Fixes

- Stopped arguments that were passed in on TestEval from being ignored

Features

- Added command line interface

6.4 Development

Seeing as `sloth` is an open source project, contributions are welcome! This allows us to provide features that the community want to be included.

If you've found a bug, or would like to request a feature, please submit it on our issue tracker on GitHub.

6.4.1 Quickstart

1 Our repository is at https://github.com/fluffykoalas/sloth.

- Development is done on the `dev` branch. Pull Requests should be filed against this branch, not the master branch.
- Stable releases reside on the `master` branch.

2. Fork the repository into your own account.
3. Create a new branch for you patch/feature
4. If you're going to setup a local environment, see *Setting up a local development environment* for details.
5. Commit as often as you'd like, but:

- Make sure the commits are logical and precise *before* asking for code review. Please see the *Commit guidelines* section for more info.
- Adhere to *Our styling guide*.
- If applicable, provide tests for your code. We aim to have full code coverage.

6. For new files, add the copyright header. It can be found in the `sloth.__init__` file.
7. Add a changelog entry - see *Changelog entries*.
8. Update the documentation - see *Updating the documentation*.
9. Squish, squash, rebase and revert commits as asked by reviewers.

6.4.2 Setting up a local development environment

First of all, make sure you've forked the repository on GitHub.

1. `git clone https://github.com/<your-account-here>/sloth.git`
2. Navigate into the directory with `cd sloth`, then install the package with *pip install -e .*.
3. Install the testing dependencies with `pip install -r requirements-test.txt`.
4. DEVELOP! Please remember that any changes to entry-points will require you to re-run `pip install -e .`

6.4.3 Changelog entries

Please add the entries for the changelog under the correct section in `CHANGELOG.rst`. The sections are `Features`, `BugFixes`, with subsections of `CLI` and `API`.

Say I added a new class to `sloth.raw.complex.tests`, called `TestFoo`. The changelog entry would be something like:

```
vX.Y.Z
------

Features
++++++++

* Added a new test class (``TestFoo``) to ``sloth.raw.complex.tests``. This class␣
↪allows you to speedtest your fooing of
  bars.
```

Which would render as:

vX.Y.Z

Features

- Added a new test class (`TestFoo`) to `sloth.raw.complex.tests`. This class allows you to speedtest your fooing of bars.

6.4.4 Commit guidelines

TL;DR

A commit:

- Stands alone as a single, complete, logical change.
- Has a descriptive commit message
- Has no extraneous modifications - eg fixing a typo in an unrelated file

Avoid committing several unrelated changes in one go. It makes merging difficult, and also makes it harder to determine which change is the culprit if a bug crops up.

If you did several unrelated changes before committing, `git gui` makes committing selected parts and even selected lines easy. Try the context menu within the window's diff area.

This results in a more readable history, which makes it easier to understand why a change was made - and which changes caused a bug.

In detail

A commit should be one (and just one) logical unit. If you'd made the following changes:

```
@@ -4,11 +4,11 @@
 # This file and all others in this project are licensed under the MIT license.
 # Please see the LICENSE file in the root of this repository for more details.
 # --------------------------------------------------------------------------

 __all__ = [
-    '__author__', '__author__', '__maintainer__', '__license__', '__uri__', '__
↪version__', 'CompareSloth'
+    '__author__', '__author__', '__maintainer__', '__license__', '__uri__', '__
↪version__', 'compare_sloth'
 ]

 __author__ = 'Legorooj'
 __maintainer__ = 'Legorooj, FluffyKoalas'

@@ -17,12 +17,10 @@ __license__ = 'MIT'
 __uri__ = 'https://github.com/FluffyKoalas/sloth'

 __version__ = '0.1.dev0'


-class CompareSloth:
-
-    def __or__(self, other):
-        return str(self)
-
-    def __str__(self):
-        return 'sloth is far better'
+def compare_sloth(against):
+    if hasattr(against, 'dummy_src_name') and getattr(against, 'dummy_src_name') == '
↪<timeit-src>':
+        return 'sloth is loads better than timeit!'
+    else:
+        return 'sloth is definitely better... assuming that\'s used for timing.'
```

Then the commit message would be:

```
Replace the CompareSloth class with a function
```

You can, in fact, view this commit on github: 0d8abc.

Use `git rebase -i` to sort, squash, and fix-up commits prior to submitting the pull-request. Make it a readable history, easy to understand what you've done.

Commit messages should provide enough information to enable a third party to decide if the change is relevant to them and if they need to read the change itself.

Also please set the correct author and email if using `git` on the CLI. You can set these like:

```
git config --global user.name "Firstname Lastname"
git config --global user.email "your_email@youremail.com"
```

Optionally remove the `--global` flag to set them for just the `sloth` repository.

6.4.5 Updating the documentation

`sloth`'s documentation is build with Sphinx. We use Sphinx's default, reStructuredText, as our markup language.

The documentation is maintained in the main repository, under the `docs` folder.

Once you've made your changes, call `make clean && make html` to build the docs, then verify that the generated pages are valid. Please look out for and fix any errors that show up in the build.

To reference commits in the docs, please only have the first 5-7 chars of the commit, and format the link as follows:

```
`cdaf638 <https://github.com/fluffykoalas/sloth/commit/cdaf638>`_
```

To reference an issue or pull request, please do so like:

```
`Issue #ISSUE-ID <https://github.com/fluffykoalas/sloth/issues/ISSUE-ID>`_
`Pull Request #PR-ID <https://github.com/fluffykoalas/sloth/pull/PR-ID>`_
```

6.4.6 Our styling guide

We adhere to PEP8, with the exception of the line limit, which we set to 120. Our CI tests code styling via `flake8`. If your code doesn't adhere to (our version of) PEP8, then please reformat it.

We also insist that you:

- Add an `__all__` statement to your file, or edit the existing one to include your function. *All* python modules/submodules must have `__all__` definitions, even private modules like `_types` and `_utils`.

6.4.7 Running and adding to the tests

- Our tests are kept in the `tests` directory, and use `pytest`. We aim for 101% coverage.
- You can install the testing requirements with `pip install -r tests/test-requirements.txt`.
- The tests can be run with:

```
pytest --cov=. tests
```

- If you're not getting 100% coverage, see why with `coverage html` and open the html files (under `htmlcov`) in your browser to see which lines aren't being tested. The config is stored in `.coveragerc`.
- `flake8` style testing can be run with just `flake8 sloth` to check the directory. The config is stored in `.flake8`.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex

PYTHON MODULE INDEX

S

INDEX