Laboratory Documentation

Release 1.0.2

Joe Alcorn

Sep 03, 2020

Contents

A library for carefully refactoring critical paths, with support for Python 2.7 & 3.3+

Laboratory is all about sure-footed refactoring achieved through experimentation. By conducting experiments and verifying their results, not only can we see if our refactored code is misbehaving, we have established a feedback loop to help us correct its behaviour.

Note: These docs are a work in progress. Additional documentation can be found in the project's README

CHAPTER 1

Installation

Installing from PyPI is recommended.

If you're unfamiliar with Python packaging tools (such as `pip` and `virtualenv`) see what The Hitchhiker's Guide to Python has to say about them.

```
$ pip install laboratory
```

You can also install a tagged version from Github

```
$ pip install https://github.com/joealcorn/laboratory/archive/v1.0.tar.gz
```

Or the latest development version

```
$ pip install git+https://github.com/joealcorn/laboratory.git
```

Now move on to the *Quickstart*

CHAPTER 2

Publishing results

We saw in the *Quickstart* how to create and run an experiment. Now let's see how we can take the data gathered in that experiment and publish it to make it useful to us.

Laboratory makes no assumptions about how to do this — it's entirely for you to implement to suit your needs. For example, timing data can be sent to graphite, and mismatches could be written to disk for debugging at a later date.

2.1 Publishing

To publish, you must implement the `publish()` method on an `Experiment`.

The publish method is passed a *Result* instance, with control and candidate observations available under `result.control` and `result.candidates` respectively.

`Experiment.`**`publish`**`(`*`result`*`)`

Publish the results of an experiment. This is called after each experiment run. Exceptions that occur during publishing will be caught, but logged.

By default this is a no-op. See *Publishing results*.

Parameters **`result`** (`Result`) – The result of an experiment run

2.2 StatsD implementation

Here's an example implementation for statsd:

```
class StatsdExperiment(laboratory.Experiment):
    def publish(self, result):
        if result.match:
            statsd.incr('experiment.match')
        else:
            statsd.incr('experiment.mismatch')
```

(continues on next page)

(continued from previous page)

```
        statsd.timing('experiment.control', result.control.duration)
        for obs in result.candidates:
            statsd.timing('experiment.%s' % obs.name, obs.duration)
```

CHAPTER 3

API Reference

3.1 Experiment

class laboratory.experiment.**Experiment**(*name='Experiment'*, *context=None*, *raise_on_mismatch=False*)

Experiment base class. Handles running your control and candidate functions. Should be subclassed to add publishing functionality.

Variables

- **name** (*string*) – Experiment name
- **raise_on_mismatch** (*bool*) – Raise `MismatchException` when experiment results do not match

classmethod decorator(*candidate*, **exp_args*, ***exp_kwargs*)

Decorate a control function in order to conduct an experiment when called.

Parameters

- **candidate** (*callable*) – your candidate function
- **exp_args** (*iterable*) – positional arguments passed to *Experiment*
- **exp_kwargs** (*dict*) – keyword arguments passed to *Experiment*

Usage:

```
candidate_func = lambda: True

@Experiment.decorator(candidate_func)
def control_func():
    return True
```

control(*control_func*, *args=None*, *kwargs=None*, *name='Control'*, *context=None*)

Set the experiment's control function. Must be set before `conduct()` is called.

Parameters

- **control_func** (*callable*) – your control function
- **args** (*iterable*) – positional arguments to pass to your function
- **kwargs** (*dict*) – keyword arguments to pass to your function
- **name** (*string*) – a name for your observation
- **context** (*dict*) – observation-specific context

Raises ***LaboratoryException*** – If attempting to set a second control case

candidate(*cand_func*, *args=None*, *kwargs=None*, *name='Candidate'*, *context=None*)

Adds a candidate function to an experiment. Can be used multiple times for multiple candidates.

Parameters

- **cand_func** (*callable*) – your control function
- **args** (*iterable*) – positional arguments to pass to your function
- **kwargs** (*dict*) – keyword arguments to pass to your function
- **name** (*string*) – a name for your observation
- **context** (*dict*) – observation-specific context

conduct(*randomize=True*)

Run control & candidate functions and return the control's return value. `control()` must be called first.

Parameters **randomize** (*bool*) – controls whether we shuffle the order of execution between control and candidate

Raises ***LaboratoryException*** – when no control case has been set

Returns Control function's return value

enabled()

Enable the experiment? If false candidates will not be executed.

Return type bool

compare(*control*, *candidate*)

Compares two `Observation` instances.

Parameters

- **control** (`Observation`) – The control block's `Observation`
- **candidate** (`Observation`) – A candidate block's `Observation`

Raises ***MismatchException*** – If `Experiment.raise_on_mismatch` is True

Return bool match?

publish(*result*)

Publish the results of an experiment. This is called after each experiment run. Exceptions that occur during publishing will be caught, but logged.

By default this is a no-op. See *Publishing results*.

Parameters **result** (`Result`) – The result of an experiment run

get_context()

Return dict Experiment-wide context

3.2 Observation

class laboratory.observation.**Observation**(*name*, *context=None*)

Result of running a single code block.

Variables

- **name** (*string*) – observation name
- **failure** (*bool*) – did the function raise an exception
- **exception** (*Exception*) – exception raised, if any
- **exc_info** – result of sys.exc_info(), if exception raised
- **value** – function return value

duration

How long the function took to execute

Return type timedelta

get_context()

Return observation-specific context

3.3 Result

class laboratory.result.**Result**(*experiment*, *control*, *candidates*)

Variables

- **experiment** (Experiment) – The experiment instance that recorded this Result
- ***control*** (Observation) – The control observation
- **candidates** ([Observation]) – A list of candidate observations
- **match** (*bool*) – Whether all candidates match the control case

3.4 Exceptions

exception laboratory.exceptions.**LaboratoryException**(*message*, **a*, ***kw*)

Base class for all laboratory exceptions

exception laboratory.exceptions.**MismatchException**(*message*, **a*, ***kw*)

CHAPTER 4

Quickstart

See: *Installation* or `pip install laboratory`

With Laboratory you conduct an experiment with your known-good code as the control block and a new code branch as a candidate.

Let's do an experiment together:

```
import laboratory

# create an experiment
experiment = laboratory.Experiment()

# set your control and candidate functions
experiment.control(authorise_control, args=(user,))
experiment.candidate(authorise_candidate, args=(user,))

# conduct the experiment and return the control value
authorised = experiment.conduct()
```

Laboratory just:

- Executed the unproven (candidates) and the existing (control) code
- Compared the return values
- Recorded timing information about all code
- Caught (and logged) exceptions in the unproven code
- Published all of this information (see *Publishing results*)

For the most part that's all there is to it. You'll need to do some work to *publish your results* in order to act on the experiment, but if you've got a metrics solution ready to go it should be straightforward.

If you need to control comparison, you can do that too.

Tip: Your control and candidate functions execute in a random order to help catch ordering issues

CHAPTER 5

Indices and tables

- genindex
- search

Python Module Index

l

Index