Testing and documentation

Computing Methods for Experimental Physics and Data Analysis

L. Baldini

luca.baldini@pi.infn.it

Università and INFN–Pisa

October 20, 2019

How do I make sure my program is correct?

- The short answer is: in real life you don't!
  - Especially if your code is asynchronous
- That is not the same a saying there is nothing you can do
- For compiled langauges the compiler will flag all obvious (and a whola lotta of non-obvious) mistakes
  - This doesn't really apply to Python, since Python is interpreted
  - Although the interpreter will stop upon syntax errors
- Besides paying attention, there are two things that you can do even in interpreted languages:
  1. Unit testing
  2. Static analysis
- Generally people hate both, but they should come right next to version control in your work-flow toolbox

Unit testing naïve example

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/unit_test_naive.py

```
1  def square(x):
2      """Function returning the suare of x.
3      """
4      return x**2.
5
6  def test():
7      """Dumb unit test---make sure that the square of 2. is 4.
8      """
9      assert square(2.) == 4.
10     print('Passed---cool!')
11
12
13 if __name__ == '__main__':
14     test()
15
16 [Output]
17 Passed---cool!
```

Unit testing in a nutshell

- Break up your program in many small pieces
  - Each piece should encapsulate a well-defined and (possibly) simple functionality
- This is usually accomplished by means of a sensible hierarchy of functions and classes
  - And this is typically the hardest task when structuring your code
  - And the code will evolve with time, so you will find yourself refactoring code from time to time
  - Remember to be dry: don't repeat youself
- Unit testing is: make sure that each single piece is correct by implementing a series of basic checks
  - You know what each elementary piece of code is suppose to be
  - Make sure it does
  - And make sure it does with any valid input
- This is much simpler that testing the whole program at once
  - Although you have to do that, too
- Test-Driven Development (TDD)
  1. Write an empty placeholder for your new function
  2. Write all the unit tests (they will fail)
  3. Implement your function and tweak it until all the tests pass

Back to our naïve example

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/unit_test_naive.py

```
1  def square(x):
2      """Function returning the suare of x.
3      """
4      return x**2.
5
6  def test():
7      """Dumb unit test---make sure that the square of 2. is 4.
8      """
9      assert square(2.) == 4.
10     print('Passed---cool!')
11
12
13 if __name__ == '__main__':
14     test()
15
16 [Output]
17 Passed---cool!
```

- This is fine, but everything happens manually
  - You have to run the script yourself
  - You have to inspect the output yourself
- As your code grows in complexity, this is not very effective

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/unit_test.py

```
1  import unittest
2
3  def square(x):
4      """Function returning the suare of x.
5
6      In real life this would be in a differnt module!
7      """
8      return x**2.
9
10
11 class TestSquare(unittest.TestCase):
12
13     def test(self):
14         """Dumb unit test---make sure that the square of 2. is 4.
15         """
16         self.assertAlmostEqual(square(2.), 4.)
17
18
19 if __name__ == '__main__':
20     unittest.main()
21
22 [Output]
23 .
24 ----------------------------------------------------------------------
25 Ran 1 test in 0.000s
26
27 OK
```

Wait a moment...

How is this different?

- This is much better!
- The base TestCase class offers all the goodies for unit testing
  - assertTrue(), assertFalse(), assertEqual(), assertAlmostEqual()...
- The execution can be easily made automatic:
  - Put all your unit test modules into a test folder
  - Run `python -m unittest discover`
  - (Or, even better, write a small Makefile or .bat script to do that)
  - That's it—all your tests are run in sequence
- Did you just find a bug in your code?
  - Make sure you add a unit test along with the fix, so that you'll never be hurt again by that particular bug
- Are you adding a new feature?
  - Make sure the new code is covered by unit tests
  - You should not be obsessed by the coverage, but you should definitely aim for it to be as large as possible
- You should always make sure that all the unit tests are passing before merging stuff on the master
- More about this in a bit (we'll be talking about continuous integration)

Static code analysis

- By its very nature, Python will show you all the errors at runtime
- Say you have a bug in a part of the code that is exercised very rarely, and not covered by unit tests
  - Python might crash the first time you exercise it...
  - or Python might happily do *something* that is not what you intended
- It might take years for even realizing that there is a bug
- Many common mistakes can be found by just looking at the code
  - And in fact all of them can, at least in principle
- Part of it can be done programmatically
  - Generally, a program will not *understand* your program
  - But a program can be trained to spot some kind of errors and inconsistencies
- Pylint and pyflakes are good examples of such tools

Static analysis: an example

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/linting1.py

```
1  x = 1.
2  y = 2.
3  very_uncommon_condition = False
4  if very_uncommon_condition:
5      print(x + z)
6  else:
7      print(x + y)
8
9  [Output]
10 3.0
```

▷ And here is the pylint output

```
1  [lbaldini@nbbaldini latex]$ pylint snippets/linting1.py
2  ************* Module snippets.linting1
3  snippets/linting1.py:1:0: C0111: Missing module docstring (missing-docstring)
4  snippets/linting1.py:1:0: C0103: Constant name "x" doesn't conform to UPPER_CASE
5                            naming style (invalid-name)
6  snippets/linting1.py:2:0: C0103: Constant name "y" doesn't conform to UPPER_CASE
7                            naming style (invalid-name)
8  snippets/linting1.py:3:0: C0103: Constant name "very_uncommon_condition" doesn't
9                            conform to UPPER_CASE naming style (invalid-name)
10 snippets/linting1.py:5:14: E0602: Undefined variable 'z' (undefined-variable)
11
12 ------------------------------------------------------------------
13 Your code has been rated at -5.00/10 (previous run: -5.00/10, +0.00)
```

Static code analysis

- You should consider using static code analysis routinely
- Static analysis tools tend to be quite verbose
  - And often times verbose is the same as annoying
- They try and enforce many different (good!) things at once
  - Formal correctness
  - Efficiency
  - Avoiding anti-patterns
  - Style guides
  - Generic conventions
- They also are typically highly customizable
  - i.e., you can mute errors you don't care about
  - But be advised: you most of the times you should probably care
- Finding a good balance is generally not too hard
- And trust me: it will help you in the long run

Digression: optional static typing in Python

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/type_annotation.py

```
1  def square(x):
2      """Return the square of a number.
3      """
4      return x**2.
5
6  def annotated_square(x: float) -> float:
7      """Return the square of a number.
8      """
9      return x**2.
10
11 print(square(2.))
12 print(annotated_square(2.))
13
14 [Output]
15 4.0
16 4.0
```

- Recent Python 3 versions support type annotations
- The Python interpreter recognizes but does nothing with annotations
  - And so what?
- Well. . . they are handy (as comments are)
  - The code is easier to read
  - Even more checks wrt un-annotated code can be done by tools such as mypy

Continuous integration

- Imagine for a second. . .
  - Wouldn't it be nice if sombody run all the unit tests of my package every time I push on the master or make a pull request?
  - And, since we are at it, sent me an email if any of the tests fail?
- . . . Well, such a thing exists and it is standard practice in code dvelopment
- People even made a name for it: Continuous Integration (CI)
- CI cloud-base services exists just like code-hostng services exist
  - Travis-CI and circleci are two good examples
- They interoperates seamlessly with github, gitlab or bitbucket
- Setting up CI for your package is usually fairly simple
- One-sentence summary: go ahead and do it. Always.

One last thing: documentation

How do the hell they do that?

SciPy.org

SciPy.org | Docs | SciPy v1.3.1 Reference Guide | Optimization and Root Finding (scipy.optimize) | index | modules | next | previous

scipy.optimize.curve_fit

scipy.optimize.curve_fit(*f*, *xdata*, *ydata*, *p0=None*, *sigma=None*, *absolute_sigma=False*, *check_finite=True*, *bounds=(-inf, inf)*, *method=None*, *jac=None*, ***kwargs*) [source]

Use non-linear least squares to fit a function, f, to data.

Assumes `ydata = f(xdata, *params) + eps`

Parameters:

f : *callable*

The model function, f(x, ...). It must take the independent variable as the first argument and the parameters to fit as separate remaining arguments.

xdata : *array_like or object*

The independent variable where the data is measured. Should usually be an M-length sequence or an (k,M)-shaped array for functions with k predictors, but can actually be any object.

ydata : *array_like*

The dependent data, a length M array - nominally `f(xdata, ...)`.

p0 : *array_like, optional*

Initial guess for the parameters (length N). If None, then the initial values will all be 1 (if the number of parameters for the function can be determined using introspection, otherwise a ValueError is raised).

sigma : *None or M-length sequence or MxM array, optional*

Determines the uncertainty in *ydata*. If we define residuals as `r = ydata - f(xdata, *popt)`, then the interpretation of *sigma* depends on its number of dimensions:

- A 1-d *sigma* should contain values of standard deviations of errors in *ydata*. In this case, the optimized function is `chisq = sum((r / sigma) ** 2)`.
- A 2-d *sigma* should contain the covariance matrix of errors in *ydata*. In this case, the optimized function is `chisq = r.T @ inv(sigma) @ r`.
  New in version 0.19.

None (default) is equivalent of 1-d *sigma* filled with ones.

absolute_sigma : *bool, optional*

If True, *sigma* is used in an absolute sense and the estimated parameter covariance *pcov* reflects these absolute values.
If False, only the relative magnitudes of the *sigma* values matter. The returned parameter covariance matrix *pcov* is based on scaling *sigma* by a constant factor. This constant is set by demanding that the reduced *chisq* for the optimal parameters *popt* when using the *scaled sigma* equals unity. In other words, *sigma* is scaled to match the sample variance of the residuals after the

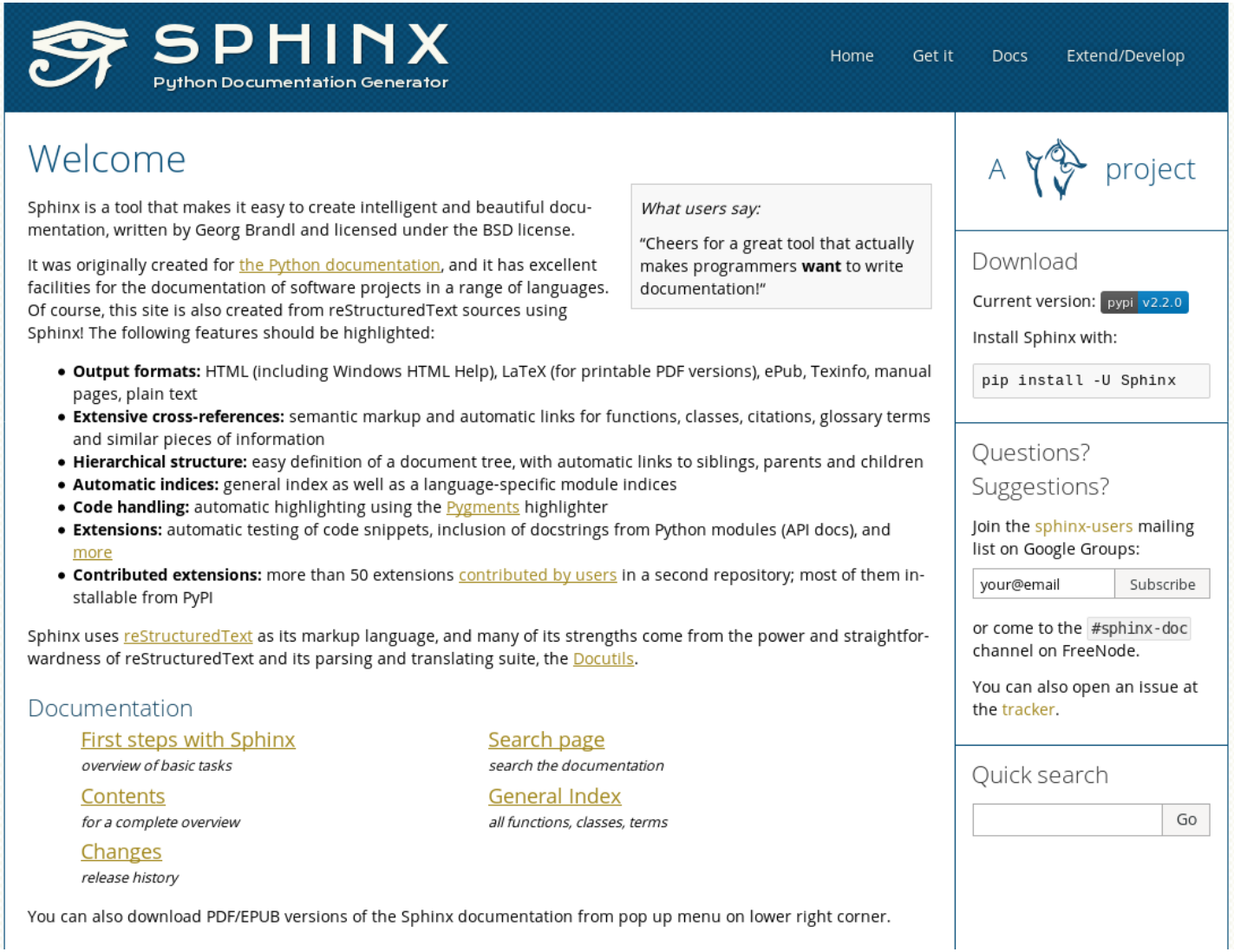
Previous topic
scipy.optimize.lsq_linear
Next topic
scipy.optimize.root_scalar
Quick search
search

One last thing: documentation

Ah—the documentation is embedded in the code. . .

```
def curve_fit(f, xdata, ydata, p0=None, sigma=None, absolute_sigma=False,
              check_finite=True, bounds=(-np.inf, np.inf), method=None,
              jac=None, **kwargs):
    """
    Use non-linear least squares to fit a function, f, to data.

    Assumes ``ydata = f(xdata, *params) + eps``

    Parameters
    ----------
    f : callable
        The model function, f(x, ...).  It must take the independent
        variable as the first argument and the parameters to fit as
        separate remaining arguments.
    xdata : array_like or object
        The independent variable where the data is measured.
        Should usually be an M-length sequence or an (k,M)-shaped array for
        functions with k predictors, but can actually be any object.
    ydata : array_like
        The dependent data, a length M array - nominally ``f(xdata, ...)``.
    p0 : array_like, optional
        Initial guess for the parameters (length N).  If None, then the
        initial values will all be 1 (if the number of parameters for the
        function can be determined using introspection, otherwise a
        ValueError is raised).
    sigma : None or M-length sequence or MxM array, optional
        Determines the uncertainty in `ydata`. If we define residuals as
        ``r = ydata - f(xdata, *popt)``, then the interpretation of `sigma`
        depends on its number of dimensions:

            - A 1-d `sigma` should contain values of standard deviations of
              errors in `ydata`. In this case, the optimized function is
              ``chisq = sum((r / sigma) ** 2)``.

            - A 2-d `sigma` should contain the covariance matrix of
              errors in `ydata`. In this case, the optimized function is
              ``chisq = r.T @ inv(sigma) @ r``.

              .. versionadded:: 0.19

        None (default) is equivalent of 1-d `sigma` filled with ones.
    absolute_sigma : bool, optional
        If True, `sigma` is used in an absolute sense and the estimated parameter
        covariance `pcov` reflects these absolute values.

        If False, only the relative magnitudes of the `sigma` values matter.
        The returned parameter covariance matrix `pcov` is based on scaling
        `sigma` by a constant factor. This constant is set by demanding that the
        reduced `chisq` for the optimal parameters `popt` when using the
```

Sphinx: the documentation tool for Python



SPHINX
Python Documentation Generator

Home Get it Docs Extend/Develop

Welcome

Sphinx is a tool that makes it easy to create intelligent and beautiful documentation, written by Georg Brandl and licensed under the BSD license.

It was originally created for the Python documentation, and it has excellent facilities for the documentation of software projects in a range of languages. Of course, this site is also created from reStructuredText sources using Sphinx! The following features should be highlighted:

What users say:

"Cheers for a great tool that actually makes programmers **want** to write documentation!"

- **Output formats:** HTML (including Windows HTML Help), LaTeX (for printable PDF versions), ePub, Texinfo, manual pages, plain text
- **Extensive cross-references:** semantic markup and automatic links for functions, classes, citations, glossary terms and similar pieces of information
- **Hierarchical structure:** easy definition of a document tree, with automatic links to siblings, parents and children
- **Automatic indices:** general index as well as a language-specific module indices
- **Code handling:** automatic highlighting using the Pygments highlighter
- **Extensions:** automatic testing of code snippets, inclusion of docstrings from Python modules (API docs), and more
- **Contributed extensions:** more than 50 extensions contributed by users in a second repository; most of them installable from PyPI

Sphinx uses reStructuredText as its markup language, and many of its strengths come from the power and straightforwardness of reStructuredText and its parsing and translating suite, the Docutils.

Documentation

First steps with Sphinx
overview of basic tasks

Contents
for a complete overview

Changes
release history

Search page
search the documentation

General Index
all functions, classes, terms

You can also download PDF/EPUB versions of the Sphinx documentation from pop up menu on lower right corner.

A project

Download

Current version: pypi v2.2.0

Install Sphinx with:

```
pip install -U Sphinx
```

Questions? Suggestions?

Join the sphinx-users mailing list on Google Groups:

your@email Subscribe

or come to the #sphinx-doc channel on FreeNode.

You can also open an issue at the tracker.

Quick search

Go

Sphinx basics

- Process all the relevant information to produce several types of output
  - Most notably html and LaTeX
- Two different sources:
  1. The doctrings in the Python modules
  2. Additional markup files (in reStructuredText) containing auxiliary information
- Typical workflow:
  - Use `sphinx-quickstart` once when you setup your project
  - Tweak the generated `conf.py` file to suit your needs
  - Go ahead and have fun!
- Sphinx is *very* powerful
  - e.g., `https://docs.python-guide.org/` is written in Sphinx, and so is all the Python documentation

Ok, I have the documentation compiled

Now what do I do with it?

- Wouldn't it be nice if the documentation was automatically compiled and uploaded on the web each time I push on the master?
- This is possible and is called readthedocs.com
  - And, again, this is a cloud-based service that can interoperate easily with github, gitlab or bitbucket

References

- https://docs.python.org/3/library/unittest.html
- https://www.pylint.org/
- https://pypi.org/project/pyflakes/
- http://mypy-lang.org/
- https://circleci.com/
- https://travis-ci.org/
- http://www.sphinx-doc.org/en/master/