SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 15

Fabian Wermelinger
Harvard University
CS107 / AC207

Thursday, October 20th 2022

LAST TIME

- Continuous Integration (CI) in Software Development
- GitHub actions
- Testing code
- Test-Driven development

TODAY

Main topics: ***Assess the quality of tests, code coverage, writing documentation***

Details:

- Assessment for quality of tests in your project
- Code coverage
- Documenting Python code

AGENDA CHECK:

- Milestone 1 is due today. This milestone is about *explicit design* (recall explicit/implicit software design approaches lecture 7/8). Try to define a basis you want to build your library on. You can still change design choices later, but keep in mind that this becomes harder and harder as your code starts to become larger and larger.

ASSESSING THE QUALITY OF TESTS

- Once you have written tests, how sure can you be that your tests *cover* all the source lines of code (SLOC) in your code base?
- ***Your tests are only as good for what they test!***
- ***Example:*** if your project has a total of 1000 lines of code (not counting blank lines or comments) and your test suite only uses 200 lines of your source code, then the quality of your tests is low (if you test meaningful things in the test suite).
- There are many reasons for incomplete tests:
  - you may have missed to execute functions
  - you may have not entered all possible branches for `if`-statements
  - for C/C++, some code may have been inlined by the compiler *(will not be an issue for Python)*
  - expansion of macros (C/C++) or use of template engines for code generation

ASSESSING THE QUALITY OF TESTS

- ***Code coverage*** (or test coverage) is a metric that expresses how much of your code base is executed by running your test suite(s).
- The metric usually expresses a *percentage* of covered code, based on these criteria:
  - ***Function/method coverage:*** has each function or method in the program been called?
  - ***Line coverage:*** has each SLOC in the program been executed? *(often used)*
  - ***Branch coverage:*** has each branch path in the program been executed?
- Code coverage tools compute these statistics and convert it into formats like HTML, XML or command line output.
- *Code coverage is easily integrated in the CI pipeline.* Each run generates coverage data which is then uploaded to a service to *track the history* of your code coverage.

CODE COVERAGE IN PYTHON

- Generating coverage reports in Python is easy!
- There are two main tools in Python:
  1. `coverage`: https://pypi.org/project/coverage
  2. `pytest-cov`: https://pypi.org/project/pytest-cov (a plugin that integrates well in the *`pytest` workflow*)
- ***Generating coverage reports involves the following steps:***
  1. Instrumenting the code for coverage. *If the program is compiled*, a special binary is produced for this task (does not apply for Python).
  2. Running the test suites with the instrumented code/binary. This will generate a *database of raw coverage data*.
  3. Post-processing of the data allows to extract several statistics and create *human readable reports*.

CODE COVERAGE IN PYTHON

- `pytest` is used in the following (*examples for `coverage` can be found at https://coverage.readthedocs.io/en/latest/ or in `check_coverage.sh`*)
- Coverage can be computed with

```
1 $ pytest --cov=cs107_package --cov-report=term-missing
```

  - The option `--cov` *is required* to indicate the target code the coverage statistics should be based on. This typically is your Python package (not the test code!).
  - The argument `--cov-report=term-missing` will further indicate the line numbers in your source code that have not been tested. *This is optional but useful.*
  - `pytest` will again attempt to *auto-discover* tests for this invocation.
- `PYTHONPATH` must be setup accordingly for this to work! (*Integrate test coverage in your test harness → see `check_coverage.sh`*)

CODE COVERAGE IN PYTHON

Computing test coverage in our example class package:

```
$ ./run_tests.sh pytest --cov=cs107_package --cov-report=term-missing
============================= test session starts ==============================
----------- coverage: platform linux, python 3.10.6-final-0 -----------
Name                                                      Stmts   Miss  Cover   Missing
---------------------------------------------------------------------------------------
cs107_project/src/cs107_package/__init__.py                   3      0   100%
cs107_project/src/cs107_package/__main__.py                   3      3     0%   1-4
cs107_project/src/cs107_package/subpkg_1/__init__.py          3      0   100%
cs107_project/src/cs107_package/subpkg_1/module_1.py          6      0   100%
cs107_project/src/cs107_package/subpkg_1/module_2.py          4      0   100%
cs107_project/src/cs107_package/subpkg_2/__init__.py          2      0   100%
cs107_project/src/cs107_package/subpkg_2/module_3.py          4      1    75%   36
cs107_project/src/cs107_package/subpkg_2/module_4.py          1      1     0%   1
cs107_project/src/cs107_package/subpkg_2/module_5.py          1      1     0%   1
---------------------------------------------------------------------------------------
TOTAL                                                        27      6    78%
============================== 4 passed in 0.03s ===============================
```

- The tool shows you which source lines in the `cs107_package` are *untested*.
- Total coverage is 78%. ***Typically coverage of 90% or more is a good indicator.***
- It is not too hard to reach 100% in Python. *It can be quite hard in C++ because of template meta-programming and code inlining by the compiler!*

CODE COVERAGE IN PYTHON

- The previous coverage command is useful to locally check coverage in your terminal while developing.
- You often want a record of your coverage to be shared with other developers. There are many third-party providers that offer this service *for a fee*, two better known examples are:
  - https://coveralls.io/ (free for public repositories)
  - https://codecov.io/ (free for public repositories)
- Repositories hosted on https://code.harvard.edu/CS107 are not public (Harvard level 3 data security). We can still engineer our own website with test coverage results using GitHub actions.
- To write you test coverage in HTML, use something similar to

```
$ ./run_tests.sh pytest --cov=cs107_package --cov-report=html:htmlcov
```

→ this will generate HTML output in the directory `htmlcov`

CODE COVERAGE IN PYTHON

Integrate coverage in your CI workflow:

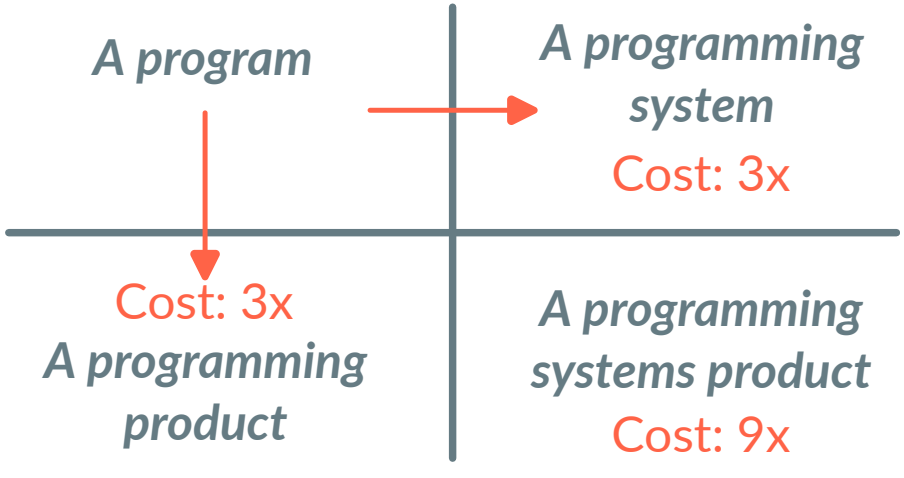
- Integration of test coverage in your CI workflow is now straightforward: *(use another action to deploy webpage!)*

```yaml
jobs:
  test_coverage: # test coverage job
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-python@v3
        with:
          python-version: '3.10'  # let's use a recent version
      - name: Install dependencies
        run: python -m pip install build pytest pytest-cov  # you also need pytest-cov!
      - name: Build and install the cs107_project in the container (using PEP517/518)
        run: (python -m build --wheel && python -m pip install dist/*)
      - name: Run tests and generate coverage html
        run: (cd tests && ./run_tests.sh CI --cov=cs107_package --cov-report=html:htmlcov)
      - name: Clean .gitignore in coverage output
        run: rm -f tests/htmlcov/.gitignore
      - name: Deploy test coverage to GitHub page (gh-pages branch)
        uses: JamesIves/github-pages-deploy-action@v4
```

DOCUMENTATION

Finally, promotion of a program to a programming product requires its thorough documentation, so that anyone may use it, fix it, and extend it. As a rule of thumb, I estimate that a programming product costs at least three times as much as a debugged program with the same function.

Frederick Brooks, The Mythical Man-Month



Credit: Fred Brooks

- ***A program:*** garage product, debugged and runs. ***Example:*** Paul Allen and Bill Gates (Microsoft).
- ***A programming product:*** thoroughly tested, documented and maintainable.
- ***A programming system:*** well defined interfaces, program integration in larger system, modularity.
- ***A programming systems product:*** final product

DOCUMENTATION

- Documentation is an *integral part of any software project* and must follow the Software Requirements Specification (your contract with the customer).
- Once the Software Requirements Specification (SRS) is written and approved, *interfaces and other exposed components are defined and remain* ***invariant***.
- The ***implementation*** of such *invariants is a* ***detail*** *and may change between different releases of the software*.
- ***Therefore:*** the best place to write documentation is the place where you write the code!

DOCUMENTATION

- ***Therefore:*** the best place to write documentation is the place where you write the code!
- We can document code in two ways:
  1. By *commenting* code
     - Comments are intended for the developer/maintainer (and for you). Comments are not part of the end-user documentation
     - Code is more often ***read*** than it is written!
  2. *In-source* documentation:
     - Requires a tool to ***process*** the documentation
     - ***Python:*** `docstrings` following PEP257, type hinting (since Python 3.5), Sphinx (relies on docstrings and markup)
     - ***C++:*** doxygen, breathe

DOCUMENTATION: COMMENTS

Writing good comments is an art like writing good commit messages

Following Jeff Atwood (founder of Stack Overflow):

1. The value of a comment is *directly proportional to the distance* between the comment and the code.
2. Comments with *complex formatting cannot be trusted*.
3. Do *not include redundant information* in the comments.
4. The best kind of comments are the ones you do not need.

   This item refers to *self-documenting* code. Attempt to write simple code that can easily be understood by itself. ***It is not wrong to think about a piece of code you wrote a second, third or fourth time.***

PYTHON DOCSTRINGS

- Python docstrings (documentation strings) are the main means for documentation in Python.
- The main document is PEP257 along with PEP8. *You should be familiar with these two documents*. Additional documentation can be found in PEP258 and PEP287.
- ***What is a docstring:*** a docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. Such a docstring becomes the `__doc__` special attribute of that object.
- All modules should normally have docstrings, and all functions and classes exported by a module should also have docstrings. Public methods (including the `__init__` constructor) should also have docstrings. A package may be documented in the module docstring of the `__init__.py` file in the package directory.
- ***Docstrings ≠ comments!***

PYTHON DOCSTRINGS

- Docstrings are *triple-quoted* strings using *double-quotes*:

```
1 r"""Module docstring.  This is a raw string interpreting `\` literally."""
2
3 import numpy as np
4
5 def f(x):
6     """Function to wrap NumPy np.sin() function."""
7     return np.sin(x)
```

- ***One-line docstrings:*** https://peps.python.org/pep-0257/#one-line-docstrings
  - The closing quotes are on the same line as the opening quotes. This looks better for one-liners.
  - There's no blank line either before or after the docstring.
  - The docstring is a phrase ending in a period. *It is written in an imperative style.*
  - Do not reiterate code (e.g. function signature) that is obvious from introspection *(except function is implemented in C/C++)*.

PYTHON DOCSTRINGS

- ***Multi-line docstrings:*** https://peps.python.org/pep-0257/#multi-line-docstrings
  - Multi-line docstrings consist of a summary line just like a one-line docstring, *followed by a blank line*, followed by a more elaborate description. The summary line may be used by automatic indexing tools → it is important that it fits on one line and is separated from the rest of the docstring by a blank line.
  - The summary line may be on the same line as the opening quotes or on the next line. *The entire docstring is indented the same as the quotes at its first line*.
- Different styles exist for multi-line docstrings. It is important to *choose one and stick with it within a project* for consistency.
  - https://docutils.sourceforge.io/rst.html (reST) is most used and the default for Sphinx (see PEP287).
  - Google format (plugin for Sphinx available)
  - NumPy format (based on Google) (plugin for Sphinx available)

PYTHON DOCSTRINGS

Example: NumPy example.py (see https://numpydoc.readthedocs.io/en/latest/format.html)
(you can find this file in the lecture code handout `CI_tests/src/cs107_package/example.py`)

```
"""Docstring for the example.py module.

Modules names should have short, all-lowercase names.  The module name may
have underscores if this improves readability.

Every module should have a docstring at the very top of the file.  The
module's docstring may extend over multiple lines.  If your docstring does
extend over multiple lines, the closing three quotation marks must be on
a line by itself, preferably preceded by a blank line.

This file is from https://github.com/numpy/numpydoc/blob/main/doc/example.py

"""

import os  # standard library imports first

# Do NOT import using *, e.g. from numpy import *
#
# Import the module using
#
```

PYTHON DOCSTRINGS

- ***Observe:*** the documentation for the example function is in the *same place where the function is defined* ***and*** *it is extensive*!
- A good and well documented docstring style is the NumPy style https://numpydoc.readthedocs.io/en/latest/format.html
- Because docstrings are interpreted by Python, they are accessible in the special `__doc__` attribute.
- There are plenty of tools available to parse the documentation in Python:
  - `pydoc` form the command line:

```
$ pydoc numpy.dot
$ pydoc cs107_package.numpy_example
```

  - From within the interpreter using the `help()` built-in or `pydoc` module:

```
>>> import cs107_package as pkg
>>> help(pkg.numpy_example)
```

PYTHON DOCSTRINGS (SPHINX)

- Once you have written extensive documentation for your software, it is time to show it to the world.
- The main tool for this in Python is Sphinx, a documentation generator (that parses `__doc__` attributes).
- All the NumPy documentation (e.g. `numpy.dot`) is generated this way.
- See the Sphinx quick start section (https://www.sphinx-doc.org/en/master/usage/quickstart.html) and the `shpinx-quickstart` command for the first steps.
- Generating HTML documentation using a tool like Sphinx and deploying the docs on hosting sites like https://readthedocs.org/ *(free for open source projects)* is a further step that is usually added to the CI pipeline.

PYTHON DOCTEST

- `unittest` and `pytest` are the packages you should base your main tests on.
- There is a handy *small-scale* test tool in Python that can be integrated in docstrings called `doctest` (https://docs.python.org/3/library/doctest.html)
- They are useful for providing *examples* in your documentation and serve as a *conceptual test* at the same time.
- A `doctest` *can not accurately capture all corner cases without cluttering your documentation*. Use them appropriately to indicate use cases and adhere to `pytest` and/or `unittest` for proper test suites.

PYTHON DOCTEST

Example: NumPy example.py (see https://numpydoc.readthedocs.io/en/latest/format.html)
(you can find this file in the lecture code handout `CI_tests/src/cs107_package/example.py`)

```
"""
Examples
--------
These are written in doctest format,
and should illustrate how to use the
function.

>>> a = [1, 2, 3]
>>> print([x + 3 for x in a])
[4, 5, 6]
>>> print("a\nb")
a
b
"""
```

Or using `pytest` with auto-discovery:

```
$ pytest --doctest-modules
```

- The docstring of our previous NumPy example function contained the section shown on the left.
- The expressions following `>>>` (including possible output) are *doctests* and resemble simple Python interpreter input/output pairs.
- You can run doctests with

```
$ python -m doctest module_name.py
```

→ if the current code does not generate the expected output, the doctest will fail.

RECAP

- Assessment for quality of tests in your project
- Code coverage
- Documenting Python code

Further reading:

- Code coverage: https://en.wikipedia.org/wiki/Code_coverage
- `pytest-cov`: https://pytest-cov.readthedocs.io/en/latest
- Python developer guide on documentation: https://devguide.python.org/documentation/start-documenting/index.html
- Python docstring conventions (PEP257): https://peps.python.org/pep-0257
- ***Recommended read*** → NumPy docstring style guide: https://numpydoc.readthedocs.io/en/latest/format.html
- Python `doctest`: https://docs.python.org/3/library/doctest.html
- Sphinx: https://www.sphinx-doc.org/en/master
- reStructuredText: https://docutils.sourceforge.io/rst.html
- Readthedocs: https://readthedocs.org/