Writing Sturdy Python

In Three Parts:

I. Unit Testing

II. Static Analysis (pylint, etcetera)

III. Gradual Typing (mypy, etcetera)

Where To Use These

- You likely want some or all of this in your text editors and/or IDE’s
- However, the place to really pile this on is in your “build”, like in your Continuous Integration software (Jenkins, Hudson, whatever)

I. Unit Testing

- Very Important Without Static Analysis and Gradual Typing
- Rather Important With Them – Similar To Statically, Manifestly Typed Languages
- We had a good presentation on unit testing recently, so I’m not going into detail.

Definitions Related To Typing

- Statically Typed: Types are determined at Compile Time
- Manifestly Typed: Types are explicitly declared by keyboarding in the name of each variable’s type
- Type Inference: Types are figured out from context, and are not manifestly spelled out
- Strongly Typed: Few to no implicit type conversions

- Java is statically, manifestly typed, and mostly strongly typed, the chief exception perhaps being that you can add a number to a string
- Out of the box, Python is dynamically and mostly strongly typed, the chief exception perhaps being that you can use almost anything in a boolean context

II. Static Analysis – What is it?

- Examines your code, ignoring high dynamicity, looking for bugs
- For example:
  - Syntax errors
  - Variables set but not used
  - Variables used but not set
  - Formatted string mismatches
  - Bad number of arguments to a callable
  - No such module, or name not found in module
  - No such symbol in object
  - And more

Static Analysis Tools

- Pylint – very stringent, also checks style
  - The author prefers this one
  - Although it warns about *many* things, undesired warnings can be turned off via comments and pylintrc files
  - Uses a limited form of type inference to typecheck code
- PyChecker – imports everything, does not check style
- Pyflakes – avoids overnotification, does not check style
- PyCharm (an IDE with Static Analysis features)
- Pycodestyle (formerly pep8, style only)
- Flake8 (combination of Pyflakes and pep8 (now Pycodestyle))
- Pymode / Syntastic (vim plugins)
- Bandit (security oriented)
- Tidypy (collects many static analyzers into a single tool)

Pylint example invocation

- pylint file1.py file2.py … filen.py
- Gives back a flood of style warnings on most code, and sometimes some real errors
- EG:

```
$ /usr/local/cpython-3.6/bin/pylint --max-line-length=132 equivs3e bufsock.py python2x3.py
readline0.py

No config file found, using default configuration

************* Module bufsock

W:233, 0: FIXME: This could be sped up a bit using slicing (fixme)

C: 49, 0: Invalid class name "rawio" (invalid-name)

C:119, 0: Invalid class name "bufsock" (invalid-name)

************* Module python2x3

R: 51, 8: Unnecessary "else" after "return" (no-else-return)


------------------------------------------------------------------

Your code has been rated at 9.95/10 (previous run: 9.00/10, +0.95)
```

this-pylint

- Something the author wrote
- It runs pylint twice: Once for Python 2.x, Once for Python 3.x. You can optionally turn off one of them.
- It eliminates all pylint output, unless something relevant is found, to keep your “build” quiet
- It also exits (negative logic) False iff a problem is found
- It also has a final fallback means of disabling a warning, in case comments and/or pylintrc aren’t enough.
- http://stromberg.dnsalias.org/~strombrg/this-pylint/

Writing to Take Full Advantage of Static Analysis (Part 1)

- Disable unimportant warnings, whether by comments (# pylint: disable=) or pylintrc
- EG:

  ```
  # pylint: disable=wrong-import-position
  ```

  Can appear at:

  - The end of a line of code
  - On a callable (function, method)
  - On an entire class
  - Or at the top of an entire python file
- Or generate an rcfile:

  ```
  pylint --generate-rcfile > pylintrc
  ```

  ...and edit it; handles an entire project.

Writing to Take Full Advantage of Static Analysis (Part 2)

- Avoid things that your static analyzer does not understand well, EG:
  - Inheritance (use composition instead where practical)
  - Named tuples (use a class)
  - Argparse (do manual command-line argument parsing)
  - Metaclasses
- There’s a philosophical issue here: Should you change how you code to get the best error checking, even at the expense of a little more keyboarding? Some say yes, some say no.

Complementary Tech

- Static analysis combines well with unit testing
- Unit testing is best for testing your code’s “happy path”, plus some but not all sad paths
- Static analysis can do some happy path, but it also scrutinizes things like error reporting that are impractical to fully unit test.

III. Gradual Typing – What is it?

- Allows the developer to manifestly declare *some* variables and not others
- Can effectively be used just for function signatures, and perhaps a couple of collection types in callable bodies
- ...or more, if you feel like it.

Gradual Typing Tools

- Mypy – The author uses this one
- Pytype – from Google, but less well known
- Pylint – someday, not today. I believe it’s on their roadmap
- PyCharm – or so I’ve been told

Mypy and Python Version

- Type annotations available in Python 3.0 and up
- typing module comes with Python 3.[56]
- typing module available as a backport for 2.7 and 3.[234]
- Can be made to work with Python 2.x, but it requires separate files for type declarations

Mypy Sample Invocation

- /usr/local/cpython-3.6/bin/mypy equivs3e
- A good typecheck is shown by no output

Example Formal Parameter with Type: Python 3.x

```
def get_mtime(filename: str) -> float:
    """Return the modification time of filename."""
    stat_buf = os.stat(filename)
    return stat_buf.st_mtime
```

Declaring the Type of a Variable: Python 3.5

- from typing import List, Dict
- size_dict = {} # type: Dict[int, List[str]]
- The comment does it
- Sometimes helps mypy
- Also works on Python 3.6

Declaring the Type of a Variable: Python 3.6

- from typing import List, Dict
- size_dict: Dict[int, List[str]] = {}
- Sometimes helps mypy
- No weird comment involved
- Confuses pylint 1.7 and before? Pylint 1.8 should be able to deal.

Declaring Types Used Before They Have Been (Fully) Defined

```
class Fraction:
    def __init__(self):
        self.numerator = 0
        self.denominator = 1
    def __lt__(self, other: ‘Fraction’) -> bool:
        ...
```

What uses type annotations

- The CPython interpreter itself treats type annotations as mere documentation; they have no other meaning to Python
- CPython needs an external tool like mypy or PyCharm to do type *checking*
- Cython has true type declarations, but they are on a cdef, not a def with type annotations – otherwise it would be difficult to tell a Python int from a C int

The End

- Questions?
- Comments?