Fermilab U.S. DEPARTMENT OF ENERGY | Office of Science

When Python Practices Go Wrong

Marc Paterno

13 January 2021

Section 1

My summary

Speaker's introduction

The talk is not about problems in *Python*, but rather *Python practices*.

- Users of Python can't generally change the language.
- Users *can* control what they use and how they use it.

The talk covers four areas of Python practice:

1. The uses of `eval()`
2. The object model
3. Mutability
4. Object orientation

Compiling code at runtime

- Python `exec` and `eval` allow the compiler to take a string containing source code not available to the compiler when your modules were loading (when most compilation is done), and to compiling the source code contained in the string.
- This is unlike what can be done in a language like C++, where the compiler is not part of the program runtime.

User-level direct use of `eval`

- Generally not needed.
- Python provides ways to manipulate the object model without building code strings.

```
eval('my_object.' + attribute_name) # bad
getattr(my_object, attribute_name)  # better
```

- Why is the `eval` use “bad”?
  - It is more error prone.
  - It is much slower (on my laptop, 5000 ns versus 70 ns).

Where is `eval` used well?

- Inside the Python standard library (*e.g.* it is how `namedtuple` is implemented)
- In `doctest` module:
  - execute the code in documentation as a test of the documentation.
  - Good for testing documentation; not good for testing code!
- The Jupyter[1] notebook uses `eval` to evaluate input cells.

Summary: very limited use in *application* code, but *wide* use in the tools built around application code.

[1] Brandon notes, with some amazement, that Jupyter (and IPython, from which it came), was invented by a *scientist*. This was Ferndando Perez, who has his Ph.D. in particle physics, from Colorado State University. Go Rams!

Python's object model

- *Special methods* == *magic methods* == *dunder methods*
- This is how Python supports operators for non-builtin types, as well as how initialization and finalization is implemented for non-builtin types.

Object model example 1: `__bool__`

- `__bool__` allows a type to participate in `if` statements and `while` loops
- Standard conversions:
  - integers: 0 is `False`, all others `True`
  - floats: 0.0 is `False`, all others `True`
  - sequences: empty is `False`, all others `True`
- PEP 8 recommends relying on all of these; Brandon argues *against* PEP 8 in this.
  - "Python code is always in danger of becoming a type desert". There are few clues in the code about the types of variables.
  - This is more of a problem in large systems than in small bodies of code.
  - "Explicit is better than implicit." Common practice violates this!

```
# What is the type of "users"?
def unfriend(subject, users):
  if not users:
    return
  rm_edges('friend', subject, users)
```

Object model example 1: `__bool__`

- `__bool__` allows a type to participate in `if` statements and `while` loops
- Standard conversions:
  - integers: 0 is `False`, all others `True`
  - floats: 0.0 is `False`, all others `True`
  - sequences: empty is `False`, all others `True`
- PEP 8 recommends relying on all of these; Brandon argues *against* PEP 8 in this.
  - "Python code is always in danger of becoming a type desert". There are few clues in the code about the types of variables.
  - This is more of a problem in large systems than in small bodies of code.
  - "Explicit is better than implicit." Common practice violates this!

```python
# What is the type of "users"?
def unfriend(subject, users):
  if not users:
    return
  rm_edges('friend', subject, users)
```

```python
# Clearly "users" is a count
def unfriend(subject, users):
  if users == 0:
    return
  rm_edges('friend', subject, users)
```

Object model example 2: `__getattr__`

- Used to implement the *Proxy* pattern: make an object appear to have an attribute when it does not have one in the normal sense.
- The `mock` library does this for testing. Allows *any* method or object to be used.
- But what is really being tested? Not the real system under use, but rather the mock objects.
- Brandon cautions against over-used; he wants to see more discussion of *when* to use `mock` rather than *how* to use `mock`.
- Alas, this talk does not contain that discussion.

Object model example 3: `__call__`

- `f()`: is `f` a *function* or an *object*?
  - It could be either; any object of a type with `__call__` defined supports use as if it were a function.
- Overuse can lead to lack of clarity in code.
- Brandon presents an XML templating example from a library that over-uses `__call__`.
- In the following, `t` is an object that represents an XML template, and `args` is a dictionary of values that are used to fill out the template.

```
# What are the intermediate steps?
t(args)()()
```

```
# Each step is clear.
t.bind(args).render().flatten()
```

- Function and method names should be *verbs*. The function call dunder method removes that verb from the place where the reader needs it to understand the code.
- If you are passing an object to an API that demands a callable object, prefer to use a *bound method object* rather than giving your class a dunder call:
  - If `x` is of a type that has method `foo`, `x.foo` is a callable object.

Mutability: modules or classes

- Modules and classes are mutable:

```
class C:
    def method(self):
        print("A")
c1 = C()
c1.method()
C.method = lambda self: print("Wackiness ensues!")
c1.method()
```

- What does this code print?
- In real code, this modification may be done far away from the original code, perhaps in a different module.

Mutability: hiding in module state

- Brandon notes a popular web framework (he does not name it) that uses a module-level global object to hold a `request`, so that it does not have to be passed to each function that needs it.
- The result is that anything that imports your module can modify the request object.
- Data enters the method from two directions: the arguments, and the module global.
- Threads will need thread-local storage.
- Async frameworks will break and will need special knowledge.

Going overboard with DRY
Usually, repetition is a code smell; thus DRY.
But using mutable global state is a worse code smell, and leads to obscurity.

Aside: is this an argument against OOP itself?

- One of the cornerstones of OOP is “information hiding”.
  - Users of objects do not need to know about their implementation details.
  - These implementation details are data, available to all methods.
  - Doesn’t this “hidden state” obscure code, for the same reasons as we just saw?

It is not entirely the same.

- The user of an object should we aware of the logical state of the object (*e.g.* is the file open or closed?)
- *Information hiding* means the user does not need to be aware of the how some feature is implemented.
- *Local* object state is much more readily kept track of than *global* state.

Mutability in tests: `mock.patch`

- Brandon presented a brief but fairly detailed description of how `mock.patch` works.
- The general issue is that this is a technique that is needed to overcome (and its use may even encourage) excessive coupling.
- Primary example is an application in which i/o is done only at the lowest levels of code, with *business logic* at the high levels of the code.
- This means business logic can not be tested without doing i/o – thus the need to patch the i/o code.
- Better is to re-design using the *clean architecture*:
  - The high-level code should handle i/o.
  - The high-level code then passes data to lower-level code containing business logic.
  - Business logic can be tested without needing to deal with i/o.
  - For those familiar with them: this is how both the CMSSW and *art* frameworks function.

Aside: unit testing

He names two reasons for unit testing:

1. Produce an automated alert for broken code.
2. *To apply pressure to decouple code*. Such code is easier to test; it is also easier to maintain (often meaning to extend).

Brandon provided what I think is one of the most succinct descriptions of a unit test that I have heard:

Unit testing
Some people thing unit-testing means each little thing needs its own test. Whereas instead, unit testing means *create* a unit by choosing some *edge* of a smaller system, and only testing things from that edge. But it doesn't have to be as small as one class or function.

Do not allow side effects at import time

- `import` executes the code it reads
  - This code is often mostly class and function definitions.
  - But it can be any code.
- Brandon's main example involves dealing with configuration in Python code.
  - First loading from a file... does not seem to evil.
  - Later modified to read from a Postgres database...
  - Later modified when Zookeepper is used to handle database connection...
- The result is a module that can only be `import`ed if both Zookeepper and Postgres are both available.
- *Rather than executing code at import time*, write functions that do the work and call those functions from your `main` function.

Limit what is done in `__init__.py`

- Having `__init__.py` in a directory makes that *directory* a Python module.
- Having code in `__init__.py` imposes a cost on everyone who imports the module.
- Brandon recommends:
  - Keep `__init__.py` empty.
  - Write another module (*e.g.* `api.py`) that loads everything of interest to the user of the module.
  - Importing the module stays cheap, but users who want the convenience can load everything by using `import nicemodule.api`.

Object orientation

- Major point: prefer composition to specialization (inheritance) to reduce coupling.
  - Example using `Thread`; if make your task by *subclassing* `Thread`, you can not test your logic without first making a new thread of execution. If you instead use composition (creating a `Thread` giving it your `task` as a function to run) you can test your logic *without* dealing with threads.
- Not a new idea; *Design Patterns: Elements of Reusable Object-Oriented Software* (the Gang of Four book) said this in 1994.
  - See the *Bridge* pattern for an example.
  - Decoupling reduces the $m \times n$ problem to a $m + n$ problem.
- Brandon's example is to go a step further: couple with *data*, using the *Pipeline* pattern.
  - *Couple to nouns rather than verbs*.
- Pipeline produces minimal coupling, allows testing of each verb independently of each other one, and allows new steps to be inserted in the middle of the pipeline without modifying the other steps.

Aside: another provocative definition

Composition means putting simple things together.
Specialization means making one thing more complicated.

Use of mixins (1 of 2)

- Brandon claims that Python provides a temptation to not even get as far as the Bridge pattern: mixins.
- The ancient `socketserver` module avoids even the Bridge pattern by using *mixins*. (Invented before the GoF popularized their menu of design patterns).
  - Starting at 52:14, Brandon gives a good short summary of how to use `socketserver`.
  - `socketserver` base class has 13 methods; only 3 are the public API.
  - Some of the remaining methods are for implementing derived classes, not for users of the server.
  - There are 4 layers to the implementation, two of which are for specialization and one of which is the public API.

Use of mixins (2 of 2)

- Brandon draws a distinction between “old” and “new” views of object orientation.
  - Old: classes are *structs* for modeling the world; create one class for one noun.
  - New: interfaces express behavior: create one interface for each behavior.
- *SOLID* “S”: a class should have 1 reason to change. `socketserver` has 3.
  - should be broken into layers, connected more loosely
- Brandon notes that *mixins* provide a way to extend a class in several directions (like the windowing classes we just saw) without solving the actual problem in the design: the class is *too complicated*.
  - If you are using *mixins*, look instead to see whether you can make a better solution by providing classes for the different layers of behavior, and composing those classes.
  - Consider coupling with *data* rather than coupling directly between the classes.