40.317 Lecture 8

Dr. Jon Freeman

11 June 2020

Slido Event #2664

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Agenda

- OOP in Python 3
- Is an OO design always a good design?
- Mid-term survey completion

OOP in Python 3: Topics in Scope

- Constructors and destructors
- Subclassing and `super()`
- Static methods
- Visibility
- Accessors
- `@classmethod`
- `__call__`
- Abstract base classes
- Named tuples
- Mix-ins

But First, the "Big Reveal"

Everything in Python is an object.

Low-level things:

```
isinstance(5, int)
isinstance(True, bool)
isinstance('hello', str)
isinstance(7.2, float)
```

And high-level things like functions, modules, and even class definitions (!).

Python 3: Constructors & Destructors

- The constructor must be `__init__`
- The destructor must be `__del__`
- Both must take `self`, i.e. the current object, as their first argument
- You cannot raise an exception inside `__del__`
- You have no direct control over when `__del__` is called
- Tying the use of a resource to the life of an object has a name: RAII, "Resource Acquisition Is Initialisation"

Python 3: Subclassing and super()

Example: `subclassing_and_super.py`

Python 3: Static Methods

- Indicated via `@staticmethod` decorator.
- Can be called in either of the two obvious ways:
  - `*<ClassName>*.a_static_method()`
  - `*<object>*.a_static_method()`
- Serve little purpose because Python code is organised by module, not class.
- "*Class* methods," discussed later, are different and can serve a useful purpose.

Python 3: Visibility

- I.e., public vs. private inst vars / methods
- Python has no privacy features, really
- A leading _ (single underscore) is a weak "internal use" indicator
  - `from MyModule import *`

    will not import names beginning with _
- A leading __ (double underscore) will trigger name "mangling." Its only purpose: ensure subclasses don't override this thing
  - Not a common use case; you won't need it

Python 3: Accessors

- Also known as "getters and setters."
  - Made popular by Java, where interfaces can only have methods, not inst. vars.

- In any language, use them sparingly.

- In Python, implement using <u>properties</u>.
  - Optional, and recommended: Python's clean "decorator" syntax for properties.

A "Getter" via Python Properties

Example: `property_getter.py`

A "Setter" via Python Properties

Example: `property_setter.py`

Python 3: `@classmethod`

The `@classmethod` decorator produces a method which only has access to the class itself (which is a thing!).

A good use of `@classmethod` is to create a "factory function" which calls `__init__` with a particular set of pre-packaged arguments.

(Presumably you will create several.)

Python 3: `__call__`

- `__call__` lets you treat an instance of a class as if it were a function.

- You can provide arguments as well.

- Consider using it when `myobject()` has an obvious, intuitive meaning.
  - For instance, an object with a state you want to either *increment* or *toggle*.

Re: __init / del / call__, etc.

These are just a few of Python's "special methods", also called "magic methods" or "dunder methods" – "*dunder*" refers to the *d*ouble *under*scores.

Together they form the API for Python's internal data model.

Know them all to start becoming an advanced Python user!

Re: @staticmethod & @classmethod

Along with `@property`, they are a few of Python's built-in "decorators."

A Python decorator transforms (injects or modifies) code in a function or an entire class. They are *not* related to the Decorator Pattern; they are more like "macros".

Look for opportunities to write your own!

Python 3: Abstract Base Classes

The `abc` module, rather than the core language, provides abstract classes. Just:

```
from abc import ABC
```

Then let your abstract class inherit from `ABC`, and tag its abstract methods with the `@abstractmethod` decorator.

You can only instantiate a subclass which overrides *every* abstract method & property.

Python 3: Named Tuples

An easy way – possibly the best way – to create a read-only custom class.

The resulting code is often cleaner and more legible.

Example: `namedtuples.py`

Named Tuples, continued

They are classes themselves, so you can subclass to add methods, properties, etc.

They are a great choice for representing:

- a row in a database table
- a row in a spreadsheet or CSV file

whether reading or writing.

Python 3: Mix-ins

A mix-in is a small class which cannot stand alone. Other classes inherit from it to acquire additional functionality, but *the IS-A relationship does not hold*.

Think of “including” a mix-in rather than “inheriting from” one. Consider them when:

- you want to add many optional features to a class, or
- you want to add one particular feature to many different classes.

Mix-ins, continued

A mix-in is *constrained MI*, i.e. a limited way of using MI. In Python, a mix-in:

- Has no use on its own, yet is *not* an ABC
- Has methods, but no instance variables
- Inherits from `object`

The last two constraints eliminate problems which can otherwise result from using MI.

Mix-ins, continued

An important warning: in Python, the class hierarchy is defined *right to left*, so when there are overridden methods, this will probably *not* do what you want:

```
class MyClass(BaseClass, Mixin1, Mixin2):
```

Instead, declare like this:

```
class MyClass(Mixin2, Mixin1, BaseClass):
```

Mix-ins, continued

An example from Python's own standard library:

https://docs.python.org/3/library/socketserver.html?highlight=mixin#socketserver.ThreadingMixIn

OOP in Python 3: Out of Scope

- Metaclasses
- Metaprogramming via either metaclasses or decorators
- *General* multiple inheritance
  - Mix-ins, a deliberately constrained use of MI, are preferred in Python

Recommended Reading

Python 3 Patterns, Recipes and Idioms, by Bruce Eckel and Friends.

Effective Python, by Brett Slatkin, 2nd Edition. https://effectivepython.com/ provides an overview of its contents.

Is OO Design Good Design?

Can a system built using OO techniques still have a poor overall architecture?

Can its internal engineering also still be poor? If so,

- Why can this still occur?
- What can we do to make it less likely?

OO Design vs. Good Design

Let's start with the obvious point that *objects contain both data and behaviour.*

This means in order to completely implement a class, we must implement both data and behaviour together.

But data design is more important than logic design, and should come first.

OO Design vs. Good Design

Next, let's use the 80/20 rule to establish what a well engineered program looks like.

- 80% of this is due to *comprehensibility*.
  - "Programs are meant to be read by humans and only incidentally for computers to execute." –Donald Knuth
- 80% of this is comprehensibility of a program's *internal state*, which we can divide into <u>mutable</u> vs. <u>immutable</u>.
- The key factor is the *comprehensibility* of the program's *<u>mutable</u> internal state*.

OO Design vs. Good Design

OO proponents argue that *private* mutable state is much cleaner than *shared* mutable state.

But a program filled with objects containing <u>private</u> mutable state is still difficult to understand and debug, i.e. *not enough of an improvement* over shared mutable state stored in pre-OO data structures.

OO Design vs. Good Design

Consider the functions in two extremely useful Python packages, itertools and more-itertools.

- Can the functions in these packages operate on streams of (custom) *objects*?
- Can they operate on streams of *immutable* (custom) objects?

OO Design vs. Good Design

Most programs have plenty of mutable state. We should *isolate* all of it behind a small number of interfaces.

As for the code involving *immutable* state, it can be well written by applying the best practices of functional programming.

– I recommend three packages to start with: itertools, more-itertools, and funcy.

OO Design vs. Good Design

The cleanest, most comprehensible parts of any program involve processing <u>streams of immutable, homogeneous data structures</u>.

The data structures in these streams might, or might not, be (custom) objects.

Thank you

A BETTER WORLD BY DESIGN.

SINGAPORE UNIVERSITY OF
TECHNOLOGY AND DESIGN