Object-Oriented Programming in Python

Jack Bennetto

January 10, 2017

DSI Standards

- Given the code for a python class, instantiate a python object and call the methods.
- Design a program in object-oriented fashion.
- Write the python code for a simple class.
- Compare and contrast functional and object-oriented programming.
- Match key "magic" methods to their syntactic sugar.

Objectives

Morning objectives:

- Define key object-oriented (OO) concepts
- Use object-oriented approach to programming
- Design and implement a basic class
- Instantiate an object

Afternoon objectives:

- List key magic methods
- Use basic decorators
- Verify code using test-driven development (TDD) and the Python debugger (PDB)

Agenda

Today's plan:

1. Introduction to OOP
2. Core OOP using Python
3. Advanced OOP using Python
4. Verification, unit tests, and debugging

Recommended Reading for Beginners

A few helpful references, arranged by increasing difficulty:

- Writing Idiomatic Python by Jeff Knupp
- Python 3 Object-Oriented Programming by Dusty Phillips
- Effective Python will help you raise your Python game
- Head First Design Patterns
- Design Patterns: Elements of Reusable Object-Oriented Software is the canonical reference
- Large-Scale C++ Software Design

Plus your favorite Python reference for language syntax. . .

Overview: What is OOP?

Object-oriented programming is an approach that organizes data together with the code that can access it.

Contrast to

- Procedural programming: data scoped by procedure/function
- Functional programming: declaritive; data is immutable

Overview: goals of OOP

Object-Oriented Programming was developed to:

- Facilitate building large-scale software with many developers
- Promote software reuse:
  - Build software components (libraries)
  - Improved code quality by using debugged components
- Decouple code, improving maintainability and stability of code
- Promote separation of concerns
- Avoid common mistakes, such as forgetting to initialize or deallocate a resource

Science and OOP

Sometimes, OOP is not the best fit for doing science:

- Science is inherently linear:
  - Projects tend to build a pipeline
  - Most applications:
    1. Load data
    2. Compute something
    3. Serialize result to disk
  - Should be able to combine steps, similar to Unix's filters + pipes model
- But, need to know OOP:
  - To use libraries which have OO design
  - To build large-scale software

Class vs. object/instance

A *class*:

- Defines a *user-defined type*, i.e., a concept with data and actions
- A full class type, on par with `float`, `str`, etc.
- Consists of:
  - Attributes (data fields)
  - Methods (operations you can perform on the object)

An *object*:

- Is an instance of a class
- Can create multiple instances of the same class
- In python, *everything is an object*

Example: sci-kit learn

All regression models – `LinearRegression`, `LogisticRegression`, `Lasso`, `Ridge`, etc. – support the same interface:

Method	Action
`.fit(X, y)`	Train a model
`.predict(X)`	Predict target/label for new data
`.score(X, y)`	Compute accuracy given data and true labels

Huge benefits for user:

- Use same interface for every model
- Minimizes cognitive load
- Other tools (GridSearch, Pipeline) can use them interchangeably

The big three

OO revolves around three key concepts:

- Encapsulation
- Inheritance
- Polymorphism

Encapsulation

Encapsulation forces code to manipulate an object's internal state only through method calls:

- You should always program this way, regardless of language:
  - Write a library to manage a resource
  - Only access the resource via the library
  - Avoid errors from unexpected interactions
  - This is basic 'defensive programming'
- **Python will not enforce encapsulation**:
  - Malicious code can directly access an object's data
  - Violating encapsulation makes code impossible to maintain
  - *'We are all consenting adults'*

Public vs. protected vs. private

Some languages (C++, Java) enforce encapsulation by making attributes public, protected, or private:

- *Public*: accessible by any external code, e.g., a public interface
- *Protected*: access depends on the language, typically inaccessible by external code and accessible by derived classes
- *Private*: accessible only by code from the same class, but not derived classes
- In Python, start the name with _ if it is private

Inheritance

Derive a *child* class from a *base* class:

- Base class defines general or basic behavior
- Child class specializes or extends behavior
  - Child gets all the functionality of Base class for free
  - Child methods override Base methods of the same name

Example: Inheritance

```
class Distribution(object):
    def logpdf(self, x):
        return np.log(self.pdf(x))

class NormalDistribution(Distribution):
    def __init__(self, mean, std):
        ...
    def pdf(self, x):
        ...
```

Polymorphism

OO code enables polymorphism:

- Treat multiple objects the same if they support same interface
- In most languages, objects must instantiate classes with a common base class
- Python uses *duck-typing*:
  - *'If it looks like a duck and quacks like a duck, it is a duck'*
  - Python does *not* require that classes are related via inheritance
  - Polymorphism works if object instantiates a class which defines the necessary attribute or method

More on typing

Static typing:

- Used in C, C++, C#, Java, Fortran, etc.
- Types are known at compile time
- Can be explicit or inferred
- Can catch bugs early
- Allows optimization based on type
- Inheritance required for polymorphism

Dynamic typing

- Used in Python, Perl (mostly), Ruby, JavaScript, etc.
- Type of an object checked at run time
- Much more flexible
- Easier to code simple scripts

Most languages are a mixture.

Getting Started

Define classes to embody concepts:

- Use `class` keyword
- Always derive your class from object:
- Capitalize name of each class

Example of a simple class

```
import math
class Vector(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def norm(self):
        return math.sqrt(self.x**2 + self.y**2)

    def add(self, other):
        return Vector(self.x+other.x, self.other.y)
```

`self`

Use `self` to refer to an instance's own, unique data:

- I.e., use `self` for 'self-reference'
- Use `self` in a class's member functions to access instance-specific data
- Like `this` in C++
- Start each member function's argument list with `self`
  - ... unless it is a static or class member function

Afternoon

Very basic OOP design

Decompose your problem into nouns and verbs:

- Noun $\Rightarrow$ implement as a class
- Verb $\Rightarrow$ implement as a method

Basic OO design

Build classes via:

- Composition/aggregation:
  - Class contains an object of a class with the desired functionality
  - Often, just basic types: `str`, `float`, `list`, `dict`, etc.
  - *HasA* $\Rightarrow$ use aggregation
- Inheritance
  - Class specializes behavior of a base class
  - *IsA* $\Rightarrow$ use inheritance
  - In some cases, derived class uses a *mix-in* base class only to provide functionality, not polymorphism

An interface is a contract

An interface is a contract between the client and the service provider:

- Isolates client from details of implementation
- Client must satisfy preconditions to call method/function
- Respect boundary of interface:
  - Library/module provides a service
  - Clients only access resource/service via library
  - Then bugs arise from arise from incorrect access or defect in library

Testing an interface

Make sure your interface is intuitive and friction-free:

- Use unit test or specification test
  - To verify interface is good before implementation
  - To exercise individual functions or objects before application is complete
  - Framework can setup and tear-down necessary test fixture
- Stub out methods using `pass`
- Test Driven Development (TDD):
  - Red/Green/Green
  - Write unit tests
  - Verify that they fail (red)
  - Implement code (green)
  - Refactor code (green)
- Use a unit test framework – `unittest` (best), doctest, or `nose`

Verification and debugging

Verifying your code is correct, and finding and fixing bugs are critical skills:

- Just because your code runs, doesn't mean it is correct
- Write unit tests to exercise your code:
  - Ensures interfaces satisfy their contracts
  - Exercise key paths through code as well as corner cases
  - Identify any bugs introduced by future changes which break existing code
  - Test code before implementing entire program
- When unit tests fail, use a debugger to examine how code executes
- Both are critical skills and will save you hours of time
- Verification and Validation in Scientific Computing discusses rigorous framework to ensure correctness

Separation of concerns (SoC)

Try to keep 'concerns' separate:

- Use different layers for each concern
- A *concern* is a set of information or a resource that affects the program
- Keep layers distinct, i.e., write modular code
- Think Unix:
  - Each layer does one thing and does it well
  - Easy to combine
- Avoid cyclic dependencies
- SoC is crucial when building distributed applications

Advanced OOP using Python

Key advanced OOP features in Python

Key features:

- Magic methods
- Decorators
- Class data and static methods
- `*args` and `**kwargs`
- Some popular patterns
- Callables

Magic methods (1/2)

Add support to your class for *magic methods*:

- To support iteration
- To support math and relational operators
- To make your class callable, like a function with state (i.e., a functor)
- To create a new container, e.g., support `len()`

See: magic methods

Magic methods (2/2)

Popular magic methods:

Method	Purpose
`__init__`	Constructor, i.e., initialize the class
`__str__`	Define behavior for `str(obj)`
`__repr__`	Define behavior for `repr(obj)`
`__len__`	Return number of elements in object
`__call__`	Call instance like a function
`__cmp__`	Compare two objects
`__iter__`	Returns an iterable (which supports `__iter__` and `next()`)

Plus methods for order relations ($==$, $!=$, $<$, $>$), attribute access, math, type conversion, custom containers, context managers, ...

Decorators

A *decorator* is a function which wraps another function:

- Looks like the original function, i.e., `help(myfunc)` works correctly
- But, decorator code runs before and after decorated function
- Lecture focuses on using existing decorators
- To write a custom decorator:
  - See Effective Python
  - Use `functools.wrap` to get correct behavior

Common decorators:

Some common decorators are:

- `@staticmethod` - group functions under class namespace
- `@classmethod` - can access class specific data
- `@property` often with `@<NameOfYourProperty>.setter`
- `@abstractmethod` - define a method in an ABC
- Can also find decorators for logging, argument checking, and more

Static methods and data

Static methods are normal functions which live in a class's namespace:

- Do not access class or instance data
- No `self` argument
- Just access by prepending name with the class or object name:

```
class PlantFromOuterSpace(object):
    @staticmethod
    def speak():
        print 'Feed me, Seymour!'

In [18]: StaticExample.call_me()
Feed me, Seymour!
```

Class methods and data

Can have class-specific data:

- Example: number of instances of class which have been created
- Decorate member function with `@classmethod`
- Use `cls` instead of `self` to refer class data
- . . . except in a method which already refers to instance data

Example

```
class ObjCounter(object):
    obj_list = []
    def __init__(self):
        self.obj_list.append(self)

    @classmethod
    def n_created(cls):
        return len(cls.obj_list)

In [14]: oc1 = ObjCounter()

In [15]: oc2 = ObjCounter()

In [16]: ObjCounter.n_created()
Out[16]: 2
```

Properties

Properties look like member data:

- Actually returned by a function which has been decorated with `@property`
- Cannot modify the field unless you also create a setter, by decorating with `@<field_name>.setter`
- Gives you flexibility to change implementation later

Example: @property

```
class Card(object):
    """
    Playing card.

    Note: Class always start with a capital letter.
    Note: New classes inherit from `object` or another class.
    """
    def __init__(self, rank, suit):
        """Create a new playing card with a rank and a suit."
        self.rank = rank
        self.suit = suit

    @property
    def color(self):
        suit_colors = {'S': 'black', 'C': 'black',
                       'H': 'red', 'D': 'red'}
        return suit_colors.get(self.suit)
```

*args and **kwargs

Shorthand to refer to a variable number of arguments:

- For regular arguments, use `*args`:
  - `*args` is a `list`
  - `def genius_func(*args):` to define a function which takes multiple arguments
  - Can also call function using a `list`, if you dereference
- For keyword arguments, use `**kwargs`:
  - `**kwargs` is a `dict`
  - `def genius_func(**kwargs):` to define a function which takes multiple keyword arguments
  - Can also call function using a `dict`, if you dereference

Design patterns

Many design patterns exist to standardize best practice:

- Worth learning if you regularly develop software
- See references
- Key patterns we will use:
  - Callable (Functor) for use with MapReduce
  - Resource Acquisition is Initialization (RAII)

Callable pattern

Class behaves like a function but can store state and other information

- Implement `__call__()`
- Acts like a Functor in C++, i.e., like a function which can store state
- Often used with MapReduce because serializable and more flexible than a lambda or free function

Example

Often, it is best practice to pass a *callable* to map or reduce:

```
class MyMapper(object):
    def __init__(self, state):
        self.state = state

    def __call__(self, elem):
        '''Perform map operation on an element'''
        return self._impl(elem)

    def _impl(self, elem)
        ...
```

Using PDB

When unit tests fail, use the debugger to find a bug:

- If working in ipython, will display line of code which caused exception
- For complex bugs, debug via PDB
- To depug an exception, run from ipython and run `%pdb` first
- To start PDB, at a specific point in your code, add:

```
import pdb

...
pdb.set_trace()  # Start debugger here
...
```

- See PDB's `help` for details
- Learn how to use a debugger. It will save you a lot of pain...

Essential debugging

Once you have mastered one debugger, you have mastered them all:

Command	Action
`h`	help
`b`	set a break-point
`where`	show call stack
`s`	execute next line, stepping into functions
`n`	execute next line, step over functions
`c`	continue execution
`u`	move up one stack frame
`d`	move down one stack frame

Debugging tricks

Some hard-won debugging tips:

- When starting any project ask, 'How will I debug this?'
- Program defensively; write code which facilitates debugging
- If you cannot figure out what is wrong with your code, something you think is true most likely isn't
- Explain your problem to a rubber duck . . . or friend
- Try to produce the smallest, reproducible test case
- If it used to work, ask yourself, 'What changed?'
- Add logging, but beware of Heisenberg: when you measure a system, you perturb it . . .

Summary

- What is the difference between a class and an object?
- What are the three key components of OOP? How do they lead to better code?
- How should I implement my code if the relationship is *IsA*? What if the relationship is *HasA*?
- What is duck typing?
- What should you do ensure an object is initialized correctly?
- What are magic methods?
- What are the benefits of TDD? What does Red/Green/Green mean?