

Introduction to Object Oriented Programming (in Python)

Benjamin S. Skrainka

July 14, 2015

Objectives

Today's objectives:

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

References

A couple helpful references, arranged by increasing difficulty:

- ▶ [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: goals of OOP

Object-Oriented Programming was developed to:

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

Science and OOP

Often, OOP is not a good 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

Using OOP

OOP requires changing how you think about code:

- ▶ As a library consumer:
 - ▶ Identify the classes with the functionality you need
 - ▶ Compose objects until you have the object you need to provide the service
- ▶ Objects provide a service to clients if they satisfy the interface's contract
- ▶ Class describes behavior and attributes of a type of object

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

Attributes

An *attribute* is a property of a class

- ▶ Usually a variable
- ▶ Could look like a variable, but really be a getter/setter method
 - ▶ Decorate a function with the attribute's name with `@property`
 - ▶ Decorate the setter with `@<my_attribute>.setter`

Example: sci-kit learn

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

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

Huge benefits for user:

- ▶ Just instantiate the model you want
- ▶ Use same interface for every model!
- ▶ Minimizes cognitive load

The big three

OO revolves around three key concepts:

- ▶ Inheritance
- ▶ Polymorphism
- ▶ Encapsulation

Don't worry about templates and generics:

- ▶ Not related to OO
- ▶ But, often used with OO, especially with C++/Java

Inheritance

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

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

Example: Inheritance

```
class Metric(object):  
    '''General model of a Metric'''  
    def score(self, y_true, y_hat):  
        pass  
  
class RMSE(Metric):  
    '''RMSE Metric'''  
    def score(self, y_true, y_hat):  
        pass  
  
class MAPE(Metric):  
    '''MAPE Metric'''  
    def score(self, y_true, y_hat):  
        pass
```

Polymorphism

OO code enables polymorphism:

- ▶ Can treat multiple objects the same as long as they support the same interface
- ▶ Usually, objects must instantiate classes which have 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 as long as the object instantiates a class which defines the necessary attribute or method

More on duck-typing

Many languages – think C++, Java, and FORTRAN – use *strong typing* and require that classes use inheritance to support polymorphism

- ▶ Python is *weakly typed*: types are determined on the fly based on usage
- ▶ A class does not need to inherit the interface to have an *IsA* relationship
 - ▶ Classes only need to support the interface
 - ▶ Inheritance makes it easier to ensure that the interface is supported, e.g., via an ABC
 - ▶ A class may only support part of an interface
- ▶ At run-time, Python will check if an object has the desired method or attribute
 - ▶ If the method is missing, Python will raise an `AttributeError`

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
 - ▶ This is basic 'defensive programming'
 - ▶ Then problems occur from either using the library incorrectly or an error in the library
- ▶ 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

Very basic OOP design

When designing in the OO style, decompose your problem into nouns and verbs:

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

Basic OO design

Typically, 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
 - ▶ Implement code
 - ▶ Refactor code
- ▶ Does interface make sense?

Example of first version of a class

```
class FancyRegression(object):  
    def __init__(self):  
        pass  
  
    def fit(self, X, y):  
        pass
```

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

Core OOP using Python

Getting Started

Define classes to embody concepts:

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

```
class FancyRegression(object):  
    pass
```

How to define a class

```
class FancyRegression(object):  
    '''Profound doc goes here!  
    Remark: Capitalize class name  
    Remark: Always inherit from 'object'  
    '''  
  
    def __init__(self, sharpness=1.0, shininess=0.0):  
        '''Setup necessary resources.'''  
        self.sharpness = sharpness  
        self.shininess = shininess  
  
    def fit(self, X, y):  
        '''Train model using XYZ method.'''  
        pass
```

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

Inheritance

To inherit from a base class, specify the parent classes instead of object when you define the class:

```
class FancyRegression(Regression):  
    ...
```

- ▶ Can call all of parent's methods on child
- ▶ But, child can override methods from parent to specialize behavior
- ▶ Can check if an object is a specific class via `isinstance()`

```
def __init__(self, ...):
```

Define the special method `__init__` to initialize each instance of a class:

- ▶ Handles instance-specific initialization
- ▶ Called whenever an instance of the class is created
- ▶ Use `self` to refer to the instance's member data and functions
- ▶ No need to worry about cleanup because of garbage collection, unlike other languages

If a class inherits from another, the derived class must call the base class's constructor:

- ▶ Use `super(MyClass, self).__init__()` to call base class's `__init__()`
- ▶ Always initialize base class before derived class

Example: `def __init__(self, ...):`

```
class FancyRegression(Regression):  
    def __init__(self, sharpness=1.0, shininess=0.0):  
        '''Setup necessary resources.'''  
        super(FancyRegression, self).__init__()  
        self.sharpness = sharpness  
        self.shininess = shininess
```

Public vs. private

In Python, you cannot enforce that a method is private:

- ▶ Start name with `_` to indicate that a function, method, or class is private
- ▶ But, 'we are all consenting adults' so deviants can still access private resources

Advanced OOP using Python

Key advanced OOP features in Python

Key features:

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

`*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

```
my_list = list(...)
genius_func(*my_list)
```

- ▶ 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

```
my_dict = {'a': 15, 'b': -92}
genius_func(**my_dict)
```

Example

- ▶ Case 1: supply all args via a list

```
def myargs(arg1, arg2, arg3):  
    return arg1 * arg2 + arg3
```

```
>> z = [ 2, 3, 4 ]  
>> myargs(*z)  
10
```

- ▶ Case 2: process variable number of arguments

```
def args2list(*args):  
    return [ ix for ix in args]
```

```
>> args2list(1, 2, 3, 4)  
[1, 2, 3, 4]
```

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):
        super(ObjCounter, self).__init__()
        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

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` or `cls` argument
- ▶ Just access by prepending name with the class's name:

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

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

Magic methods (1/2)

Add support to your class for *magic methods*:

- ▶ To make your class iterable
- ▶ To make your class callable like a function with state (i.e., a functor)
- ▶ To make your class behave like a container, e.g., support `len()`

See: [magic methods](#)

Magic methods (2/2)

Popular magic methods:

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

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:

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

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 FancyRegression(object):  
    def __init__(self, name)  
        self._name = name  
  
    @property  
    def name(self):  
        return self._name  
  
    @name.setter  
    def name(self, new_name):  
        self._name = new_name
```

Example of different types of methods

From [StackOverflow](#)

```
class A(object):  
    def foo(self,x):  
        print "executing foo(%s,%s)"%(self,x)  
  
    @classmethod  
    def class_foo(cls,x):  
        print "executing class_foo(%s,%s)"%(cls,x)  
  
    @staticmethod  
    def static_foo(x):  
        print "executing static_foo(%s)"%x
```

Example: continued

```
In [20]: a = A()
```

```
In [21]: a.foo(1)  
executing foo(<__main__.A object at 0x1083db5d0>,1)
```

```
In [22]: a.class_foo(2)  
executing class_foo(<class '__main__.A'>,2)
```

```
In [23]: a.static_foo(3)  
executing static_foo(3)
```

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__(state):
        self.state = state

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

    def _impl(elem)
        ...
```

ABCs

An *Abstract Base Class* (ABC):

- ▶ Defines a standard interface for derived objects
- ▶ Cannot be instantiated – to ‘access,’ must derive a class from the ABC
- ▶ May contain some implementation for methods

See doc on [abc](#) module for details

Verification, unit tests, and debugging

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
 - ▶ 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

Unit tests and TDD

Unit tests exercise your code so you can test individual functions:

- ▶ Use a unit test framework – `unittest2` (best) or `nose`
- ▶ Unit tests should exercise key cases and verify interfaces
- ▶ A unit test can setup fixtures (i.e., resources) needed for testing
- ▶ *Test Driven Development* is a good approach to development:
 - ▶ *Red*: implement test and check it fails
 - ▶ *Green*: implement code and make sure it passes
 - ▶ *Green*: refactor and optimize implementation
- ▶ ‘Only refactor in the presence of working tests’
- ▶ Save time by verifying interfaces and catching errors early
- ▶ Catch errors if a future change breaks things

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 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

`code.interact()` trick

In some environments (e.g., Cython), PDB may not work:

- ▶ Use `code.interact()` to start a Python interpreter with local context
- ▶ Exit by typing `^D`
- ▶ Better than printing...
- ▶ Need to import any libraries you want to use

```
...  
import code  
code.interact('Ring 5 of Inferno', local=locals())  
...
```


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 in `__init__()`?
- ▶ What are magic methods?
- ▶ What are the benefits of TDD? What does Red/Green/Green mean?