Object-Oriented Programming: fundamentals in Python

Benjamin S. Skrainka

October 14, 2015

Overview

Object-Oriented Programming (OOP) is a fundamental approach to building large-scale software:

- For complex problems, large teams, huge code bases
- Abstracts problems into *nouns* (*classes*) and *methods* (*verbs*)
- Alternatives: bottom up, top down, waterfall, . . .
- Data scientists should understand OOP:
  - To use libraries with an OO design (OOD)
  - To build more reliable software

Objectives

Lecture objectives:

- Use OOP as consumer of libraries
- Define fundamental OOP concepts and terms
- Design and implement simple programs using OOP

Agenda

Today's agenda:

Time	Activity
9:00-9:30	Review Assignment Solutions
9:30-10:30	Lecture
10:30-12:00	Work on exercise
12:00-1:15	Lunch
1:15-2:00	Review morning exercise
2:00-5:00	Work on assignment

References

A couple references:

- Object-Oriented Analysis and Design by Booch, et al
- Large-Scale C++ Software Design by Lakos
- Design Patterns: Elements of Reusable Object-Oriented Software by the ‘gang of four’
- Head First Design Patterns
- A nice blog post on use cases

An example (1/3)

You (may) have already seen OO in many parts of Python:

- Containers: `list`, `dict`, `Counter`, `defaultdict`, `set`
- Numpy: `ndarray`, `matrix`
- scikit-learn: `LinearRegression`, `Ridge`, `Lasso`, `LogisticRegression`

An example (2/3)

Different classes support in **scikit-learn** support an identical interface:

```
lr = LogisticRegression()
lr.fit(X_train, y_train)
lr.score(X_test, y_test)
...
# Different class, same interface!
rr = Ridge()
rr.fit(X_train, y_train)
rr.score(X_test, y_test)
```

An example (3/3)

Polymorphism means clients can reuse a family of classes:

- All classes support same interface
- No need to worry about implementation

```
(X, y) = load_awesome_data()
params = {...}
lr = Lasso()

# Can pass any model to GridSearchCV!
lr_model = grid_search.GridSearchCV(lr, params)
lr_model.fit(X, y)
```

Benefits of OO

Good OO design makes your life easier:

- *Polymorphism*: use identical interface for similar objects
- *Encapsulation*: boundary between interface and implementation
- *Inheritance*: model relationships between classes

Plus:

- Guaranteed initialization via *constructor*
- Easier to write and use libraries
- Reuse (debugged) software components

Basic concepts: object vs. class

A *class* and an *object* are easily confused:

- A *class* describes the a new data type with both data and class-specific functions (*methods*). I.e., a class is:
  - Data
  - Methods
  - Constructor (`__init__`) ensures proper initialization
- An *object* is a specific instance of a class
- **Warning**: the base class for all classes in Python is named `object`...

Example: class vs. object

Can instantiate many objects from a class, each with its own data:

```
In [2]: c1 = Counter(range(4))

In [3]: c1
Out[3]: Counter({0: 1, 1: 1, 2: 1, 3: 1})

In [4]: c2 = Counter(range(4))

In [5]: c2
Out[5]: Counter({0: 1, 1: 1, 2: 1, 3: 1})

In [6]: id(c1)
Out[6]: 4419096112

In [7]: id(c2)
Out[7]: 4424179792
```

Basic OO Design (OOD)

Describe the *use cases* for the problem you are solving:

- Implement *nouns* as *classes*
- Implement *verbs* as *methods*
- Interfaces are a contract – do not violate them!
- Only access objects via interface

In Python, "*We are all consenting adults*":

- Nothing prevents violating encapsulation
- But, your code will become unmaintainable

Identifying a use case

Create use cases to identify requirements for your software:

> *As a [type of user] I want to [perform some task] so that I can [achieve a goal, benefit, outcome]*

Example:

> *As a user of Amazon Web Services, I want to launch an EC2 cluster so that I can run my data science application*

How to define a class

To define a class:

```
class Library(object):  #   Always specify the class 'object'
    """Library to manage books."""

    def __init__(self, book_list):
        pass

    def checkout(self, book):
        pass

    def checkin(self, book):
        pass
```

How to write methods

Methods look like a regular method with an extra argument `self`:

- `self` refers to the object instance the method is operating on
- Otherwise, write like a regular function
- Use `self.mydata` to access class's data member `mydata`
- Python will automatically pass the correct object in the `self` slot when you apply the method to an object

Example (1/2)

Write a method with `def` but inside the class:

```python
class Person(object):
    def __init__(self, name_):
        self.name = name_

    def whoami(self):
        print "I am {0}!".format(self.name)
```

Example (2/2)

Instantiate and use your class:

```
p = Person('Fred')
p.whoami()
```

How to instantiate a class

To instantiate a class, just assign it to a variable:

- Specify arguments to constructor
- Allocates memory for an object of the class's type
- Calls `__init__` to initialize object

```
books = ['Master and Margarita', 'Anna Karenina',
         'Sentimental Education', 'Macbeth']
my_lib = Library(books)
my_book = my_lib.check_out('Macbeth')
```

How to store data

You can store data at several scopes

- Member data
  - Data which is specific to each instance
  - Use `self.mydata` to access `mydata` member of object
- Method data
  - Defined like regular arguments and variables in function
  - Only accessible while method is active
- Class data
  - Data which is specific to all instances of a class
  - Outside scope of class

How to write code: *T*est *D*riven *D*evelopment (TDD)

Write code using TDD:

- Write unit test first
- Write stubs for code
- RED: check that unit test fails
- GREEN: write code that passes unit test
- GREEN: refactor code so that it is faster, better, etc.

TDD is part of the Agile software methodology

Scope

Classes restrict access at several levels:

- A method's variables & arguments are only accessible in the method
- Can store data in an object using `self` in methods:
  - Accessible by all of an object's methods
  - An object can access data of another object of the same class
- By convention, names which start with _ are private
- Respect encapsulation – beware: you are not forced to!

Using a constructor

Write a constructor for every class:

```
class   Library(object):

    def __init__(self, book_list=None):
        ...
```

- Initializes each instance of class
- Derived class must call `super` to initialize base class

Other magic methods

There are many common methods you may choose to support:

- `__len__`: return length of object
- `__repr__`: return representation of object
- `__str__`: return string representation/summary of object
- standard math and relational operations

For more detail, see this post on magic methods

Duck typing

Beware! Python supports *duck typing*:

> *If it looks like a duck and quacks like a duck, it is a duck*

- Classes can support only part of an interface
- No requirement for strict inheritance structure like other languages
- Python is weakly typed unlike strongly typed languages like C++
- Run PEP8 to catch errors

Class relationships

Two main relationships:

- *IsA* ⇒ inheritance
- *HasA* ⇒ aggregation
- Outside scope of lecture

Inheritance & constructors

Must call base class constructor if using inheritance:

```
class    fancy_dict(dict):

    def __init__(self, my_arg):
        super(dict, self).__init__()
        ...
```

Properties & decorators

Properties and decorators are outside the scope of this lecture:

- Start with an `@` followed by a name
- E.g., `@property`, `@staticmethod`, ...
- Google or consult a Python text

Conclusion

You should now know:

- What is the difference between a class and an object?
- When to implement a class or a method?
- How to implement a (basic) class?
- How to implement a method?
- How to instantiate a class?
- How to invoke a method on a class?