Object-Oriented Programming in Python
(pt. 1)

Jon Courtney

January 23, 2017

Lesson Objectives

By the end of this lesson, you should:

- understand key concepts and terminology used in OO programming,
- know how to define classes and create objects using Python 3 syntax
- recognize and and know how to apply Python magic methods
- begin using OOP in your own code

Benefits of OOP

Object-Oriented Programming was developed to…

- Facilitate large-scale software with many developers
- Promote software reuse:
  - Through “inheritance”, can add new features / behavior to existing code
  - Through “polymorphism”, multiple components use familiar interfaces
- Decouple code, improving maintainability and stability of code
  - Through “encapsulation”, implementation details are hidden
- Improve productivity:
  - Through reuse
  - By promoting “separation of concerns”

Why You Should Know OOP

- Python is a great scripting language even without OOP
- Nonetheless, you should apply yourself to learning OOP…
  - ➡ To converse intelligently with software engineers
  - ➡ To use data science libraries which have OO design
  - ➡ To make your own code more reusable and extensible

Thinking “OO”

OOP requires a shift in thinking. . .

- **from** thinking about *passing* *<u>data</u>* *to* *<u>functions</u>*
- **to** thinking about *affecting* <u>actions</u> (as functions) *on* *<u>objects</u>* (that contain *data*)

For example, in an OO system, a given “object” could be:

- A producer of data
- A set of data (produced by another object)
- A pipeline of operations (performed upon a dataset object)
- A client that consumes and displays the pipeline output

OOP Terminology

Class:

- A template used for creating objects

Object:

- A specific realization of a class

Instance:

- (largely synonymous with "object")

Instantiation:

- What you do when you create an object

Class vs. Object/Instance

A <u>class</u>:

- Defines a *user-defined type*, i.e., an entity with data and actions
  - On par with `float`, `str`, `list`, etc.
  - Testable with `type()` and `isinstance()`
- Consists of *attributes*:
  - Referenced via `self`*.attribute* or *object.attribute*
  - Data attributes (aka class/instance variables)
    - Holds the resulting object “state”
  - Methods (in Python, also technically a class “attribute”)
    - Operations you can perform on an instance of the class

An <u>object</u>:

- Is an *instance* of a class
- Can be one of multiple instances of the same class

Classes and Objects








Python OOP Terminology

`self`:

- A variable to reference an object from within that same object

Attribute:

- A piece of data carried within an object. (Think: “object variable”).
  Usually referenced via `self.attribute` or `object.attribute`

Method:

- A function that is specific to class, and works on the data of a specific object.
  (Think: “object function”.) Usually called via `self.method(…)` or `object.method(…)`

`__init__` :

- “Magic” method called during instantiation to initialize an object

Constructor:

- The mechanism for instantiating an object from a class definition.
  E.g., in Python: `c = Chair()`

Defining a Class, Creating, Using Objects

```
# Define a Library class…
class Library:
    """Library to manage books."""

    def __init__(self, book_list):
        self.books = book_list

    def check_out(self, book):
        # Remove book from self.books
        return self.books

    def check_in(self, book):
        pass

# Create Library object using constructor…
l = Library([‘Anna Karenina’, ‘Moby Dick’, ‘The Great Gatsby’])

# Call check_out() method…
l.check_out(‘Moby Dick’)
```

(Breakout: Class Definition and Object Creation)

Python “Magic” Methods

- Certain operations prompt Python to look for “magic” methods on referenced objects. By convention, these are contained in double underscores (“dunder”), e.g., `__init__`

- The combination of magic methods and duck typing permits powerful polymorphism in Python.

- Other magic methods:

    - **__repr__** : Returns canonical representation of object

    - **__len__** : Returns length of object

    - **__eq__** : Tests for equality with another object

    - **__add__** : Returns “sum” of object with another object

(Breakout: Magic Methods)

Three Hallmarks of OOP

- **Encapsulation**: Data and methods are bundled within class definitions
- **Polymorphism**: Methods/operations of same name act differently (and appropriately) on objects of different types
- **Inheritance**: Organizing complex systems from general to specific, such that more-specific <u>sub-classes</u> *inherit* the more general behavior of their <u>super-class</u>.

Encapsulation Example

```python
class Point:
    """Represent a point in 2-dimensional space using x and y coordinates."""

    def __init__(self, x, y):
        """Initialize a Point with the given x and y coordinate values.

        Parameters
        ----------
        x : float
        y : float
        """
        self.x = x      # Point class encapsulates x and y state
        self.y = y
```

Encapsulation Benefits

- Convenient packaging
  - Data (object “state”) is all in one place
  - Methods are attached to the objects they operate on
- Information Hiding
  - Can change internal state representation without breaking client code
  - Can prevent (or discourage) client code from messing up the internal state of an object

Encapsulation in Python

- Note that Python **does not provide access control** to encapsulated state
  - Outside code can directly access **and change** an object's data
    - E.g., `point.x = 3.0`
    - Is this bad?
- Some languages (e.g., C++, Java) enforce access control via `public` / `protected` / `private` access modifiers
  - In Python "we are all consenting adults"
    - All attributes are essentially public
    - Begin attribute name with an underscore if it is not intended for public use. E.g., `point._x`

Polymorphism Example

```
class Point:
    """Represent a point in 2-dimensional space using x and y coordinates.”""

  . . .

    def __add__(self, other):
        """Return a new Point representing the sum of self and other.

        Parameters
        ----------

        other : Point


        Examples
        --------
        >>> Point(1, 1) + Point(2, 3)   # ‘+’ operator works on both ints and Points
        Point(3, 4)
        >>> Point(1, 1) + Point(-1, -1)
        Point(0, 0)
        “""
        pass
```

Polymorphism Benefits

- As a user of an object, you don’t have to change your client code when you want to interact with an object of a new type
- In sklearn…

```
model = LinearRegression()
…
model = LogisticRegression()
…
model = MLPRegressor()
…
model.fit(X,y)  # Same method calls, different types
model.predict(X)
model.score(X, y)
```

Polymorphism in Python

- Different objects treated the same if they support same interface
- Typically, objects acquire shared interfaces through inheritance
- However, Python employs duck-typing:
  - “If it looks like a duck and quacks like a duck…”
  - Polymorphism just works if object supports the necessary attribute or method
  - If an expected method is missing during execution, Python raises an `AttributeError`

Inheritance: Super-classes and Sub-classes

- Class hierarchies organized from general to specific
- A more-specific <u>sub-class</u> (or “derived” or “child” class) *inherits* the more general behavior of its <u>super-class</u> (or “base”or “parent”) class:
  - Subclasses receive all the functionality of their super-class
  - Subclass methods may override super class methods of the same name
- Multiple inheritance is allowed
- Check if object belongs to class or superclass using `isinstance()`

Inheritance Example

```
class Polygon:

   def __init__(self):
      self.points = list()

   . . .
   def draw(self):
      pass


class Triangle(Polygon):     # isinstance(self, Polygon) == True
  . . .
   def draw(self):           # Overrides Polygon.draw()
      # Draw the triangle


class Octagon(Polygon):
  . . .
   def draw(self):           # Overrides Polygon.draw()
      # Draw the octagon
```

Multiple Inheritance Example

```
class Polygon:

   def __init__(self):
      self.points = list()


class Drawable:

   def draw(self):
      pass


class Triangle(Polygon, Drawable):   # Superclasses searched left to right
  . . .
   def draw(self):          # Overrides Drawable.draw()
      # Draw the triangle


class Octagon(Polygon, Drawable):
  . . .
   def draw(self):          # Overrides Drawable.draw()
      # Draw the octagon
```

Inheritance Benefits

- Inheritance is a common way to experience benefits of polymorphism
- Permits “separation of concerns”
  - No need to clutter up the “Polygon” class with “if/elif” (case) statements to handle behavior of variants — put these in subclasses
  - Makes libraries much easier to understand
    - E.g. “Model” vs. LinearRegression vs. LogisticRegression
- Permits code reuse and rapid development

Inheritance and __init__

If one class inherits from another, the subclass must ensure that the superclass is properly initialized:

- Use `super().__init__()` to call superclass' `__init__` method
- Always initialize superclass before subclass:
  - Call `super().__init__()` from subclass `__init__`

Python 3 Inheritance Syntax

```
class Polygon:              # "new style" class, default in Python 3

  def __init__(self):
    self.points = list()

  . . .
  def draw(self):
    pass


class Triangle(Polygon):

  def __init__(self):
    super().__init__()      # Calls __init__() on the superclass

  def draw(self):
    # Draw the triangle
```

Python 2 Inheritance Syntax

```
class Polygon(object): # “new style” class, must be explicit in Python2

  def __init__(self):
    self.points = list()

  . . .
  def draw(self):
    pass


class Triangle(Polygon):

  def __init__(self):
    super(Polygon, self).__init__()   # Ugly!!!

  def draw(self):
    # Draw the triangle
```

Python OOP Conventions

- Use <u>self</u> to refer to the current instance from within the same object
  - `self.method()   # Calls method on one’s self`
  - `self.attribute  # Access data from one’s self`
- Classes named in *CamelCase*
- Methods named in *snake_case*
- Signal private state and methods with _leading _underscore
- Magic methods use __double_underscores__

(Breakout: Inheritance)

Thinking “OO”

Think: *Nomen Est Omen*

- Latin: “Name is destiny”
- The way you describe your intended system points to its OO structure
  - Nouns ⇒ classes
    - Example: Polygon
    - Python convention: Uppercased, CamelCased
  - Verbs ⇒ methods
    - Example: Polygon.<u>draw</u>()
    - Python convention: lowercased, snake_cased
- And vice-versa: How you name classes & methods will influence how users (even yourself!) think about your code. So choose names thoughtfully!

Think: Relationships

- “is-a” : reflects *inheritance*
  - Subclass “is-a” superclass
  - Duck “is-a” Bird “is-a” Animal
- “has-a”: reflects *composition* or *aggregation*
  - class “has-a” different class
  - Car “has-a” Engine

Think: Interface “Contracts”

“Design by contract” is a helpful thought process when designing methods (and functions) and informs test-driven development.

Three components:

- **Preconditions**: What does the method/function expect when it begins?
- **Postconditions**: What does the method/function guarantee when it exits?
- **Invariants**: What state must the method/function maintain for consistency?

Resources

- *Writing Idiomatic Python* - Jeff Knupp
- *Python 3 Object-Oriented Programming* - Dusty Philips
- *Fluent Python* - Luciano Ramalho
- *Effective Python* - Brett Slatkin
- *Design Patterns: Elements of Reusable Object-Oriented Software* - Gamma, Helm, Johnson, Vlissides