**Lesson Outline**

* **Object-oriented programming syntax**
  + procedural vs object-oriented programming
  + classes, objects, methods and attributes
  + coding a class
  + magic methods
  + inheritance
* **Using object-oriented programming to make a Python package**
  + making a package
  + tour of scikit-learn source code
  + putting your package on PyPi

**Why Object-Oriented Programming?**

Object-oriented programming has a few benefits over procedural programming, which is the programming style you most likely first learned. As you'll see in this lesson,

* object-oriented programming allows you to create large, modular programs that can easily expand over time;
* object-oriented programs hide the implementation from the end-user.

Consider Python packages like **[Scikit-learn](https://github.com/scikit-learn/scikit-learn" \t "_blank)**, [**pandas**](https://pandas.pydata.org/), and **[NumPy](http://www.numpy.org/" \t "_blank)**. These are all Python packages built with object-oriented programming. Scikit-learn, for example, is a relatively large and complex package built with object-oriented programming. This package has expanded over the years with new functionality and new algorithms.

When you train a machine learning algorithm with Scikit-learn, you don't have to know anything about how the algorithms work or how they were coded. You can focus directly on the modeling.

Here's an example taken from the **[Scikit-learn website](http://scikit-learn.org/stable/modules/svm.html" \t "_blank)**:

from sklearn import svm

X = [[0, 0], [1, 1]]

y = [0, 1]

clf = svm.SVC()

clf.fit(X, y)

How does Scikit-learn train the SVM model? You don't need to know because the implementation is hidden with object-oriented programming. If the implementation changes, you as a user of Scikit-learn might not ever find out. Whether or not you SHOULD understand how SVM works is a different question.

In this lesson, you'll practice the fundamentals of object-oriented programming. By the end of the lesson, you'll have built a Python package using object-oriented programming.

**Lesson Files**

This lesson uses classroom workspaces that contain all of the files and functionality you will need. You can also find the files in the [**data scientist nanodegree term 2 GitHub repo**](https://github.com/udacity/DSND_Term2/tree/master/lessons/ObjectOrientedProgramming).

# Object-Oriented Programming (OOP) Vocabulary

* **class** - a blueprint consisting of methods and attributes
* **object** - an instance of a class. It can help to think of objects as something in the real world like a yellow pencil, a small dog, a blue shirt, etc. However, as you'll see later in the lesson, objects can be more abstract.
* **attribute** - a descriptor or characteristic. Examples would be color, length, size, etc. These attributes can take on specific values like blue, 3 inches, large, etc.
* **method** - an action that a class or object could take
* **OOP** - a commonly used abbreviation for object-oriented programming
* **encapsulation** - one of the fundamental ideas behind object-oriented programming is called encapsulation: you can combine functions and data all into a single entity. In object-oriented programming, this single entity is called a class. Encapsulation allows you to hide implementation details much like how the scikit-learn package hides the implementation of machine learning algorithms.

In English, you might hear an attribute described as a property, description, feature, quality, trait, or characteristic. All of these are saying the same thing.

# Function vs Method

In the video above at 1:44, the dialogue mistakenly calls ***init*** a function rather than a method. Why is ***init*** not a function?

A function and a method look very similar. They both use the def keyword. They also have inputs and return outputs. The difference is that a method is inside of a class whereas a function is outside of a class.

# What is self?

If you instantiate two objects, how does Python differentiate between these two objects?

shirt\_one = Shirt('red', 'S', 'short-sleeve', 15)

short\_two = Shirt('yellow', 'M', 'long-sleeve', 20)

That's where self comes into play. If you call the change\_price method on shirt\_one, how does Python know to change the price of shirt\_one and not of shirt\_two?

shirt\_one.change\_price(12)

Behind the scenes, Python is calling the change\_price method:

**def** **change\_price**(self, new\_price):

self.price = new\_price

Self tells Python where to look in the computer's memory for the shirt\_one object. And then Python changes the price of the shirt\_one object. When you call the change\_price method, shirt\_one.change\_price(12), self is implicitly passed in.

The word self is just a convention. You could actually use any other name as long as you are consistent; however, you should always use self rather than some other word or else you might confuse people.

# Set and Get methods

The last part of the video mentioned that accessing attributes in Python can be somewhat different than in other programming languages like Java and C++. This section goes into further detail.

The shirt class has a method to change the price of the shirt: shirt\_one.change\_price(20). In Python, you can also change the values of an attribute with the following syntax:

shirt\_one.price = 10

shirt\_one.price = 20

shirt\_one.color = 'red'

shirt\_one.size = 'M'

shirt\_one.style = 'long\_sleeve'

This code accesses and changes the price, color, size and style attributes directly. Accessing attributes directly would be frowned upon in many other languages **but not in Python**. Instead, the general object-oriented programming convention is to use methods to access attributes or change attribute values. These methods are called set and get methods or setter and getter methods.

A get method is for obtaining an attribute value. A set method is for changing an attribute value. If you were writing a Shirt class, the code could look like this:

**class** **Shirt**:

**def** **\_\_init\_\_**(self, shirt\_color, shirt\_size, shirt\_style, shirt\_price):

self.\_price = shirt\_price

**def** **get\_price**(self):

**return** self.\_price

**def** **set\_price**(self, new\_price):

self.\_price = new\_price

Instantiating and using an object might look like this:

shirt\_one = Shirt('yellow', 'M', 'long-sleeve', 15)

print(shirt\_one.get\_price())

shirt\_one.set\_price(10)

In the class definition, the underscore in front of price is a somewhat controversial Python convention. In other languages like C++ or Java, price could be explicitly labeled as a private variable. This would prohibit an object from accessing the price attribute directly like shirt\_one.\_price = 15. However, Python does not distinguish between private and public variables like other languages. Therefore, there is some controversy about using the underscore convention as well as get and set methods in Python. Why use get and set methods in Python when Python wasn't designed to use them?

At the same time, you'll find that some Python programmers develop object-oriented programs using get and set methods anyway. Following the Python convention, the underscore in front of price is to let a programmer know that price should only be accessed with get and set methods rather than accessing price directly with shirt\_one.\_price. However, a programmer could still access \_price directly because there is nothing in the Python language to prevent the direct access.

To reiterate, a programmer could technically still do something like shirt\_one.\_price = 10, and the code would work. But accessing price directly, in this case, would not be following the intent of how the Shirt class was designed.

One of the benefits of set and get methods is that, as previously mentioned in the course, you can hide the implementation from your user. Maybe originally a variable was coded as a list and later became a dictionary. With set and get methods, you could easily change how that variable gets accessed. Without set and get methods, you'd have to go to every place in the code that accessed the variable directly and change the code.

You can read more about get and set methods in Python on this [**Python Tutorial site**](https://www.python-course.eu/python3_properties.php).

# A Note about Attributes

There are some drawbacks to accessing attributes directly versus writing a method for accessing attributes.

In terms of object-oriented programming, the rules in Python are a bit looser than in other programming languages. As previously mentioned, in some languages, like C++, you can explicitly state whether or not an object should be allowed to change or access an attribute's values directly. Python does not have this option.

Why might it be better to change a value with a method instead of directly? Changing values via a method gives you more flexibility in the long-term. What if the units of measurement change, like the store was originally meant to work in US dollars and now has to handle Euros? Here's an example:

#### Example Dollars versus Euros

If you've changed attribute values directly, you'll have to go through your code and find all the places where US dollars were used, like:

shirt\_one.price = 10 *# US dollars*

and then manually change to Euros

shirt\_one.price = 8 *# Euros*

If you had used a method, then you would only have to change the method to convert from dollars to Euros.

**def** **change\_price**(self, new\_price):

self.price = new\_price \* 0.81 *# convert dollars to Euros*

shirt\_one.change\_price(10)

For the purposes of this introduction to object-oriented programming, you will not need to worry about updating attributes directly versus with a method; however, if you decide to further your studies of object-oriented programming, especially in another language such as C++ or Java, you'll have to take this into consideration.

# Modularized Code

Thus far in the lesson, all of the code has been in Jupyter Notebooks. For example, in the previous exercise, a code cell loaded the Shirt class, which gave you access to the Shirt class throughout the rest of the notebook; however, if you were developing a software program, you would want to modularize this code.

You would put the Shirt class into its own Python script called, say, shirt.py. And then in another Python script, you would import the Shirt class with a line like: from shirt import Shirt.

For now, as you get used to OOP syntax, you'll be completing exercises in Jupyter notebooks. But midway through the lesson, you'll modularize object-oriented code into separate files.

 This section of the course is merely reminding you to use docstrings and to comment your code. It's not just going to help you understand and maintain your code**. It will also make you a better job candidate.**

From this point on, please always comment your code. Use both in-line comments and document level comments as appropriate.

Below is an example of a class with docstrings and a few things to keep in mind:

* Make sure to indent your docstrings correctly or the code will not run. A docstring should be indented one indentation underneath the class or method being described.
* You don't have to define 'self' in your method docstrings. It's understood that any method will have self as the first method input.

**class** **Pants**:

"""The Pants class represents an article of clothing sold in a store

"""

**def** **\_\_init\_\_**(self, color, waist\_size, length, price):

"""Method for initializing a Pants object

Args:

color (str)

waist\_size (int)

length (int)

price (float)

Attributes:

color (str): color of a pants object

waist\_size (str): waist size of a pants object

length (str): length of a pants object

price (float): price of a pants object

"""

self.color = color

self.waist\_size = waist\_size

self.length = length

self.price = price

**def** **change\_price**(self, new\_price):

"""The change\_price method changes the price attribute of a pants object

Args:

new\_price (float): the new price of the pants object

Returns: None

"""

self.price = new\_price

**def** **discount**(self, percentage):

"""The discount method outputs a discounted price of a pants object

Args:

percentage (float): a decimal representing the amount to discount

Returns:

float: the discounted price

"""

**return** self.price \* (1 - percentage)