NOTES 5

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, pandas, and NumPy. 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:

```
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.

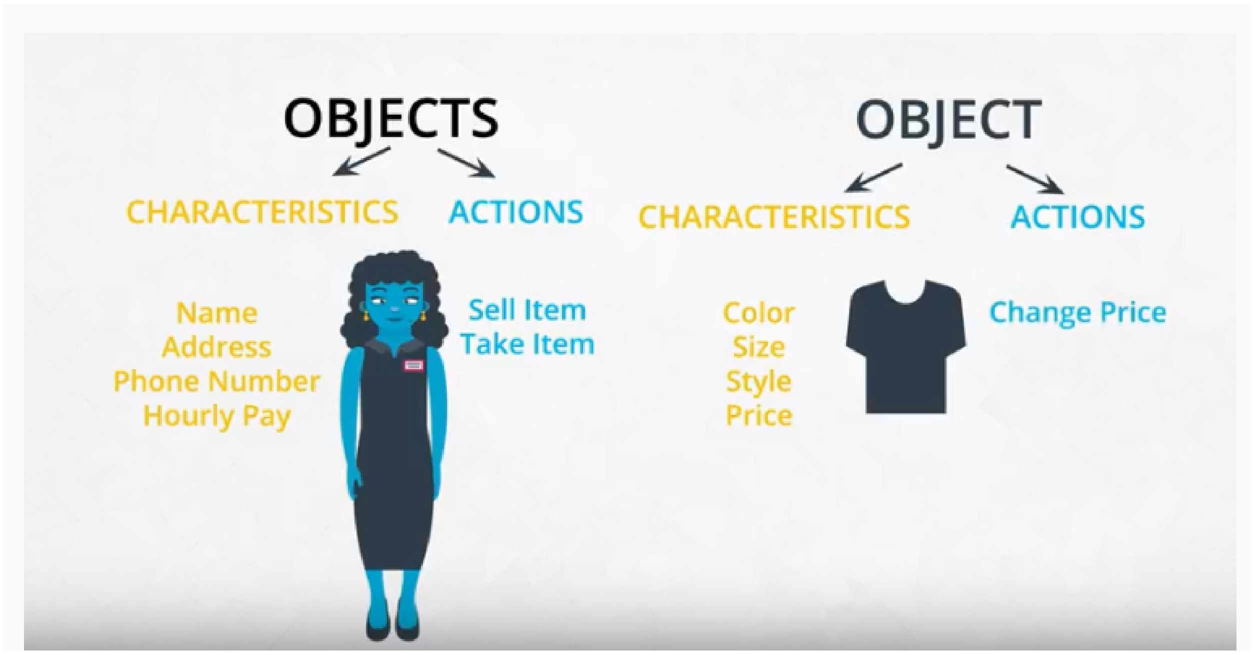
Lesson files

This lesson uses classroom workspaces that contain all of the files and functionality you need. You can also find the files in the data scientist nanodegree term 2 GitHub repo.

Procedural versus object-oriented programming

Objects are defined by characteristics and actions

Here is a reminder of what is a characteristic and what is an action.



Objects are defined by their characteristics and their actions

Characteristics and actions in English grammar

You can also think about characteristics and actions is in terms of English grammar. A characteristic corresponds to a noun and an action corresponds to a verb.

Let's pick something from the real world: a dog. Some characteristics of the dog include the dog's weight, color, breed, and height. These are all nouns. Some actions a dog can take include to bark, to run, to bite, and to eat. These are all verbs.

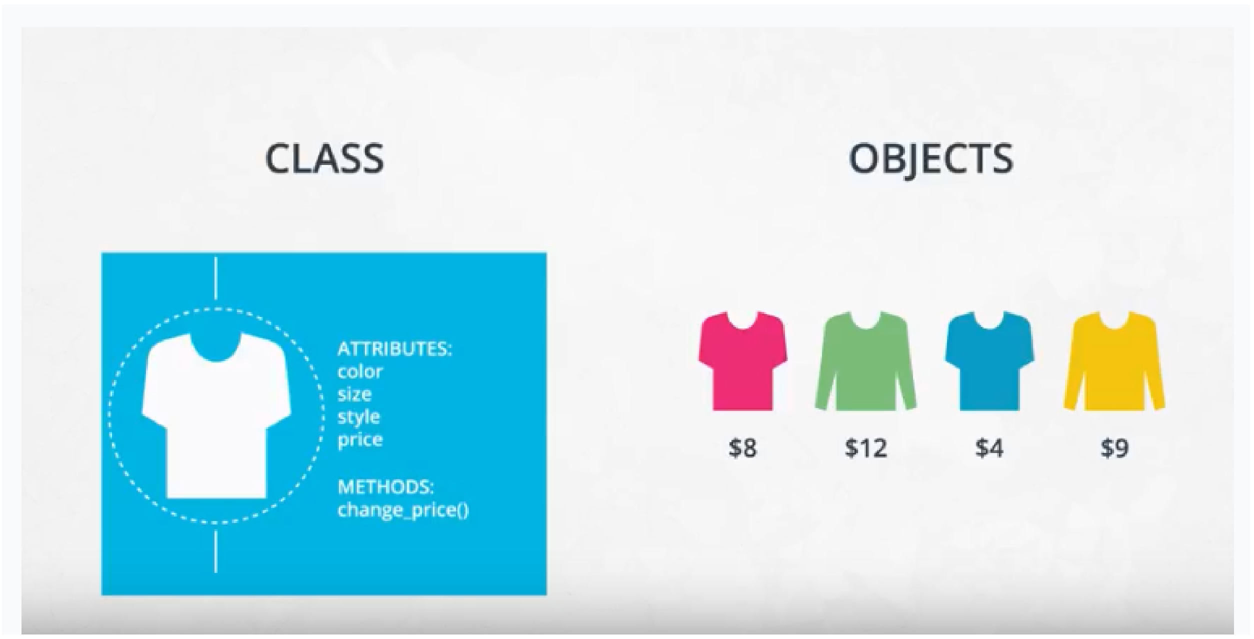
ACTIONS CAN ONLY BE VERBS

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, or a blue shirt. 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.

Here is a reminder of how a class, an object, attributes, and methods relate to each other.



A class is a blueprint consisting of attributes and methods.

Function versus 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)
shirt_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. 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 consisten, but you should use `self` to avoid confusing 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, you could use the following code:

```
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 the following code:

```
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`. Unlike other
languages, Python does not distinguish between private and public
variables. 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. Perhaps, 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.

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 if 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, such as in the following:

```
shirt_one.price = 10 # US dollars
```

Then, you'll have to manually change them 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 don't need to worry about updating attributes directly versus with a method; however, if you decide to further your study 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.

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, which you might call `shirt.py`. 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. Midway through the lesson, you'll modularize object-oriented code into separate files.

Commenting object-oriented code

Did you notice anything special about the answer key in the previous exercise? The `Pants` class and the `SalesPerson`
 class contained docstrings! A docstring is a type of comment that describes how a Python module, function, class, or method works. Docstrings are not unique to object-oriented programming.

For this section of the course, you just need to remember to use docstrings and to comment your code. It will help you understand and maintain your code and even make you a better job candidate.

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

To learn more about docstrings, see Example Google Style Python Docstrings.

Docstrings and object-oriented code

The following example shows a class with docstrings. Here are 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)
```

Resources for review

The example in the next part of the lesson assumes you are familiar with Gaussian and binomial distributions.

Here are a few formulas that might be helpful:

Gaussian distribution formulas

probability density function

f(x ∣ μ,σ2)=12πσ2e−(x−μ)22σ2
f(x \space | \space \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}}e^-\frac{(x-\mu)^2}{2\sigma^2}
f(x ∣ μ,σ2)=2πσ2

1e−2σ2(x−μ)2

where: μ is the mean σ is the standard deviation σ2 is the variance
\begin{aligned}
\ \text{where:}
\ \mu& \space \text{is the mean}
\ \sigma& \space \text{ is the standard deviation}
\ \sigma&^2 \space \text{is the variance}
\end{aligned}
where: μ is the mean σ is the standard deviation σ2 is the variance

Binomial distribution formulas

mean

μ=n∗p\mu = n * pμ=n∗p

In other words, a fair coin has a probability of a positive outcome (heads) p = 0.5. If you flip a coin 20 times, the mean would be 20 * 0.5 = 10; you'd expect to get 10 heads.

variance

σ2=np(1−p)\sigma^2 = n p (1 - p)σ2=np(1−p)

Continuing with the coin example, n would be the number of coin tosses and p would be the probability of getting heads.

standard deviation

σ=np(1−p)\sigma = \sqrt{n p (1 - p)}σ=np(1−p)

In other words, the standard deviation is the square root of the variance.

probability density function

f(k,n,p)=n!k!(n−k)!pk(1−p)(n−k)
f(k, n, p) = \frac{n\footnotesize{!}}{k!(n - k)!}p^k(1-p)^{(n-k)}
f(k,n,p)=k!(n−k)!n!pk(1−p)(n−k)

Further resources

If you would like to review the Gaussian (normal) distribution and binomial distribution, here are a few resources:

This free Udacity course, Intro to Statistics, has a lesson on Gaussian distributions as well as the binomial distribution.

This free course, Intro to Descriptive Statistics, also has a Gaussian distributions lesson.

There are also relevant Wikipedia articles:

- Gaussian Distributions Wikipedia
- Binomial Distributions Wikipedia

https://onlinestatbook.com/2/calculators/normal_dist.html