Object-Oriented Programming Defined

In object-oriented programming, concepts are modeled as classes and objects. An idea is defined using a class, and an instance of this class is called an object. Almost everything in Python is an object, including strings, lists, dictionaries, and numbers. When we create a list in Python, we're creating an object which is an instance of the list class, which represents the concept of a list. Classes also have attributes and methods associated with them. Attributes are the characteristics of the class, while methods are functions that are part of the class.

Classes and Objects in Detail

We can use the **type()** function to figure out what class a variable or value belongs to. For example, **type(" ")** tells us that this is a string class. The only attribute in this case is the string value, but there are a bunch of methods associated with the class. We've seen the **upper()** method, which returns the string in all uppercase, as well as **isnumeric()** which returns a boolean telling us whether or not the string is a number. You can use the **dir()** function to print all the attributes and methods of an object. Each string is an instance of the string class, having the same methods of the parent class. Since the content of the string is different, the methods will return different values. You can also use the **help()** function on an object, which will return the documentation for the corresponding class. This will show all the methods for the class, along with parameters the methods receive, types of return values, and a description of the methods.

Defining Classes (Optional)

We can create and define our classes in Python similar to how we define functions. We start with the **class** keyword, followed by the name of our class and a colon. Python style guidelines recommend class names to start with a capital letter. After the class definition line is the class body, indented to the right. Inside the class body, we can define attributes for the class.

Let's take our Apple class example:

```
>>> class Apple:
...     color = ""
...     flavor = ""
...
```

We can create a new instance of our new class by assigning it to a variable. This is done by calling the class name as if it were a function. We can set the attributes of our class instance by accessing them using dot notation. Dot notation can be used to set or retrieve object attributes, as well as call methods associated with the class.

```
>>> jonagold = Apple()
>>> jonagold.color = "red"
>>> jonagold.flavor = "sweet"
```

We created an Apple instance called jonagold, and set the color and flavor attributes for this Apple object. We can create another instance of an Apple and set different attributes to differentiate between two different varieties of apples.

```
>>> golden = Apple()
>>> golden.color = "Yellow"
>>> golden.flavor = "Soft"
```

We now have another Apple object called golden that also has color and flavor attributes. But these attributes have different values.

What Is a Method?

Calling methods on objects executes functions that operate on attributes of a specific instance of the class. This means that calling a method on a list, for example, only modifies that instance of a list, and not all lists globally.

We can define methods within a class by creating functions inside the class definition. These instance methods can take a parameter called **self** which represents the instance the method is being executed on. This will allow you to access attributes of the instance using dot notation, like **self.name**, which will access the name attribute of that specific instance of the class object. When you have variables that contain different values for different instances, these are called instance variables.

Special Methods

Instead of creating classes with empty or default values, we can set these values when we create the instance. This ensures that we don't miss an important value and avoids a lot of unnecessary lines of code. To do this, we use a special method called a **constructor**. Below is an example of an Apple class with a constructor method defined.

```
...         self.flavor = flavor
>>> class Apple:
...     def __init__(self, color, flavor):
...         self.color = color
```

When you call the name of a class, the constructor of that class is called. This constructor method is always named **__init__**. You might remember that special methods start and end with two underscore characters. In our example above, the constructor method takes the self variable, which represents the instance, as well as color and flavor parameters. These parameters are then used by the constructor method to set the values for the current instance. So we can now create a new instance of the Apple class and set the color and flavor values all in go:

```
>>> jonagold = Apple("red", "sweet")
>>> print(jonagold.color)
Red
```

In addition to the **__init__** constructor special method, there is also the **__str__** special method. This method allows us to define how an instance of an object will be printed when it's passed to the print() function. If an object doesn't have this special method defined, it will wind up using the default representation, which will print the position of the object in memory. Not super useful. Here is our Apple class, with the **__str__** method added:

```
...     def __init__(self, color, flavor):
...         self.color = color
...         self.flavor = flavor
...     def __str__(self):
...         return "This apple is {} and its flavor is {}".format(self.color, self.flavor)
...
```

Now, when we pass an Apple object to the print function, we get a nice formatted string:

```
This apple is red and its flavor is sweet
>>> jonagold = Apple("red", "sweet")
>>> print(jonagold)
```

This apple is red and its flavor is sweet

It's good practice to think about how your class might be used and to define a __str__ method when creating objects that you may want to print later.

Documenting with Docstrings

The Python **help** function can be super helpful for easily pulling up documentation for classes and methods. We can call the **help** function on one of our classes, which will return some basic info about the methods defined in our class:

```
>>> class Apple:
```

```
...     def __init__(self, color, flavor):
...         self.color = color
...         self.flavor = flavor
...     def __str__(self):
...         return "This apple is {} and its flavor is {}".format(self.color, self.flavor)
...
>>> help(Apple)
Help on class Apple in module __main__:

class Apple(builtins.object)
 |  Methods defined here:
 |
 |  __init__(self, color, flavor)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __str__(self)
 |      Return str(self).
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

We can add documentation to our own classes, methods, and functions using **docstrings**. A docstring is a short text explanation of what something does. You can add a docstring to a method, function, or class by first defining it, then adding a description inside triple quotes. Let's take the example of this function:

```
>>> def to_seconds(hours, minutes, seconds):
...     """Returns the amount of seconds in the given hours, minutes and seconds."""
...     return hours*3600+minutes*60+seconds
...
```

We have our function called *to_seconds* on the first line, followed by the docstring which is indented to the right and wrapped in triple quotes. Last up is the function body. Now, when we call the help function on our to_seconds function, we get a handy description of what the function does:

```
>>> help(to_seconds)
Help on function to_seconds in module __main__:

to_seconds(hours, minutes, seconds)
    Returns the amount of seconds in the given hours, minutes and seconds.
```

Docstrings are super useful for documenting our custom classes, methods, and functions, but also when working with new libraries or functions. You'll be extremely grateful for docstrings when you have to work with code that someone else wrote!

Classes and Methods Cheat Sheet (Optional)

Classes and Methods Cheat Sheet

In the past few videos, we've seen how to define classes and methods in Python. Here, you'll find a run-down of everything we've covered, so you can refer to it whenever you need a refresher.

Defining classes and methods

```
class ClassName:
    def method_name(self, other_parameters):
        body_of_method
```

Classes and Instances

- Classes define the behavior of all instances of a specific class.
- Each variable of a specific class is an instance or object.
- Objects can have attributes, which store information about the object.
- You can make objects do work by calling their methods.
- The first parameter of the methods (self) represents the current instance.
- Methods are just like functions, but they can only be used through a class.

Special methods

- Special methods start and end with __.
- Special methods have specific names, like __init__ for the constructor or __str__ for the conversion to string.

Documenting classes, methods and functions

- You can add documentation to classes, methods, and functions by using docstrings right after the definition. Like this:

```
class ClassName:
    """Documentation for the class."""
    def method_name(self, other_parameters):
        """Documentation for the method."""
        body_of_method

def function_name(parameters):
    """Documentation for the function."""
    body_of_function
```

Help with Jupyter Notebooks (Optional)

Help with Jupyter Notebooks

We've aimed to make our Jupyter notebooks easy to use. But, if you get stuck, you can find more information here.

If you still need help, the discussion forums are a great place to find it! Use the forums to ask questions and source answers from your fellow learners.

If you want to learn more about Jupyter Notebooks as a technology, check out these resources:

- Jupyter Notebook Tutorial, by datacamp.com
- How to use Jupyter Notebooks, by codeacademy.com
- Teaching and Learning with Jupyter, by university professors using Jupyter

---

Object Inheritance

In object-oriented programming, the concept of inheritance allows you to build relationships between objects, grouping together similar concepts and reducing code duplication. Let's create a custom Fruit class with color and flavor attributes:

```
...
>>> class Fruit:
...     def __init__(self, color, flavor):
...         self.color = color
...         self.flavor = flavor
```

We defined a Fruit class with a constructor for color and flavor attributes. Next, we'll define an Apple class along with a new Grape class, both of which we want to inherit properties and behaviors from the Fruit class:

```
>>> class Apple(Fruit):
...     pass
...
>>> class Grape(Fruit):
...     pass
...
```

In Python, we use parentheses in the class declaration to have the class inherit from the Fruit class. So in this example, we're instructing our computer that both the Apple class and Grape class inherit from the Fruit class. This means that they both have the same constructor method which sets the color and flavor attributes. We can now create instances of our Apple and Grape classes:

```
>>> granny_smith = Apple("green", "tart")
>>> carnelian = Grape("purple", "sweet")
>>> print(granny_smith.flavor)
tart
>>> print(carnelian.color)
purple
```

Inheritance allows us to define attributes or methods that are shared by all types of fruit without having to define them in each fruit class individually. We can then also define specific attributes or methods that are only relevant for a specific type of fruit. Let's look at another example, this time with animals:

```
>>> class Animal:
...     sound = ""
...     def __init__(self, name):
...         self.name = name
...     def speak(self):
...         print("{sound} I'm {name}! {sound}".format(
...             name=self.name, sound=self.sound))
...
>>> class Piglet(Animal):
...     sound = "Oink!"
...
>>> class Cow(Animal):
...     sound = "Moooo"
...
```

We defined a parent class, Animal, with two animal types inheriting from that class: Piglet and Cow. The parent Animal class has an attribute to store the sound the animal makes, and the constructor class takes the name that will be assigned to the instance when it's created. There is also the speak method, which will print the name of the animal along with the sound it makes. We defined the Piglet and Cow classes, which inherit from the Animal class,

and we set the sound attributes for each animal type. Now, we can create instances of our Piglet and Cow classes and have them speak:

```
>>> hamlet = Piglet("Hamlet")
>>> hamlet.speak()
Oink! I'm Hamlet! Oink!
...
>>> class Cow(Animal):
...     sound = "Moooo"
...
>>> milky = Cow("Milky White")
>>> milky.speak()
Moooo I'm Milky White! Moooo
```

We create instances of both the Piglet and Cow class, and set the names for our instances. Then we call the speak method of each instance, which results in the formatted string being printed; it includes the sound the animal type makes, along with the instance name we assigned.

Object Composition

You can have a situation where two different classes are related, but there is no inheritance going on. This is referred to as **composition** -- where one class makes use of code contained in another class. For example, imagine we have a **Package** class which represents a software package. It contains attributes about the software package, like name, version, and size. We also have a **Repository** class which represents all the packages available for installation. While there's no inheritance relationship between the two classes, they are related. The Repository class will contain a dictionary or list of Packages that are contained in the repository. Let's take a look at an example Repository class definition:

```
>>> class Repository:
...     def __init__(self):
...         self.packages = {}
...     def add_package(self, package):
...         self.packages[package.name] = package
...     def total_size(self):
...         result = 0
...         for package in self.packages.values():
...             result += package.size
...         return result
```

In the constructor method, we initialize the packages dictionary, which will contain the package objects available in this repository instance. We initialize the dictionary in the constructor to ensure that every instance of the Repository class has its own dictionary.

We then define the add_package method, which takes a Package object as a parameter, and then adds it to our dictionary, using the package name attribute as the key.

Finally, we define a total_size method which computes the total size of all packages contained in our repository. This method iterates through the values in our repository dictionary and adds together the size attributes from each package object contained in the dictionary, returning the total at the end. In this example, we're making use of Package attributes within our Repository class. We're also calling the values() method on our packages dictionary instance. Composition allows us to use objects as attributes, as well as access all their attributes and methods.

Augmenting Python with Modules

Python modules are separate files that contain classes, functions, and other data that allow us to import and make use of these methods and classes in our own code. Python comes with a lot of modules out of the box. These

modules are referred to as the Python Standard Library. You can make use of these modules by using the **import** keyword, followed by the module name. For example, we'll import the **random** module, and then call the **randint** function within this module:

```
>>> import random
>>> random.randint(1,10)
8
>>> random.randint(1,10)
7
>>> random.randint(1,10)
1
```

This function takes two integer parameters and returns a random integer between the values we pass it; in this case, 1 and 10. You might notice that calling functions in a module is very similar to calling methods in a class. We use dot notation here too, with a period between the module and function names.

Let's take a look at another module: **datetime**. This module is super helpful when working with dates and times.

```
>>> import datetime
>>> now = datetime.datetime.now()
>>> type(now)
<class 'datetime.datetime'>
>>> print(now)
2019-04-24 16:54:55.155199
```

First, we import the module. Next, we call the **now()** method which belongs to the **datetime** class contained within the **datetime** module. This method generates an instance of the datetime class for the current date and time. This instance has some methods which we can call:

```
>>> print(now)
2019-04-24 16:54:55.155199
>>> now.year
2019
>>> print(now + datetime.timedelta(days=28))
2019-05-22 16:54:55.155199
```

When we call the print function with an instance of the datetime class, we get the date and time printed in a specific format. This is because the datetime class has a **__str__** method defined which generates the formatted string we see here. We can also directly call attributes and methods of the class, as with **now.year** which returns the year attribute of the instance.

Lastly, we can access other classes contained in the datetime module, like the **timedelta** class. In this example, we're creating an instance of the timedelta class with the parameter of 28 days. We're then adding this object to our instance of the datetime class from earlier and printing the result. This has the effect of adding 28 days to our original datetime object.

Supplemental Reading for Code Reuse (Optional)

Supplemental Reading for Code Reuse

The official Python documentation lists all the modules included in the standard library. It even has a turtle in it...

Pypi is the Python repository and index of an impressive number of modules developed by Python programmers around the world.