Object-Oriented Programming

Please note that while reading through this article, you will see technical terms **bolded** and *italicized* followed by a brief explanation of that term in parentheses. As a data scientist, you will more frequently use the terms that have been made bold and italicized.

'Object-oriented' programming' is a mouthful, but it's a lot simpler than it sounds!
Object-oriented programming is basically just programming with objects. In this article, we'll take a look at how to define the class of an object and create functions.

In his philosophy book *Object-Oriented Ontology,* philosopher Graham Harman writes that there are two ways in which we can answer the question 'what is X?':

- We say what X is made of

Or:

- We say what X *does*

(or both)[1]!

For example, if we're answering the question 'what is a plant cell?' we might give an answer that states what plant cells are made of and what they do; or if we're answering 'what is parliamentary democracy?' we might give an answer that states just what it *does*.

Harman's ideas bear some resemblance to Object-Oriented Programming. The key thought is that any object is an instance of a class, and has certain ***methods*** **(**abilities) and ***intrinsic variables*** (qualities) that are specified when it's made. We can understand any Python object, then, by just looking at:

- its ***intrinsic variables*** (it's features or qualities)

And:

- its ***methods*** (it's abilities to do something).

[1] Harman, Graham, 'Object-Oriented Ontology,' Pelican 2018

Defining a Class

An ***object*** is a thing whose general features and abilities have been defined with code. Any object belongs to a certain ***class*** (or type), and that class determines the **intrinsic variables** (features) and **methods** (abilities) common to all objects of that class. Methods are just like functions, but functions are only performable by instances of certain classes. Think of a class as almost like a blueprint for the features and abilities common to all instances of it.

For example, one such class might be *Cat*:

- Its **intrinsic variables** might be things like hair color, eye color, and gender
- Its **methods** might be things like jump(), meow(), eat() and sleep()

Each particular cat is an instance of the class *Cat*, and each instance has the same intrinsic variables and functions as the next one. We can name particular instances of the *Cat* class anything we want, for example 'Timothy', 'Gerald', or 'Jeeves'.

Notice that only the methods — jump(), meow(), eat() and sleep() — have brackets at the end: this is in case we want to give inputs to those methods. It's in those brackets (if we were a bossy cat-owner) that we'd instruct our cat to be more specific about its behaviors.

For example, the instruction Timothy.munch(apples, 5) might tell the *Cat* instance Timothy to eat 5 apples, and the instruction Gerald.nap(3) tells the cat Gerald to nap for 3 hours.

We can make such a class for cats with the following real Python code:

```
class Cat:
    def __init__(self, name, hairColour, eyeColour, gender):
        self.name = name,
        self.hairColour = hairColour
        self.eyeColour = eyeColour
        self.gender = gender

    def jump(self):
        print('Jumping joyously!')

    def meow(self):
        print('Meowing cutely.')

    def eat(self):
        print('Eating abundantly!')

    def sleep(self):
        print('Sleeping deeply...')

```

Intrinsic Variables of a Class

The above code might look intimidating and weird at first, but let's break it down. With line 1, we declare a class that we chose to call 'Cat'. The word 'class' goes pink, because once we type it, Python thinks: "Ah! They're trying to make a class. Ok. I'll lookout for the other things they have to do in that case." In other words, it's a Python ***keyword***, and calls upon one of Python's built-in abilities, namely, to make classes. Python's *full* of built-in abilities and features, and the process of learning to program is largely just familiarising yourself with these features and playing by Python's rules (such as spelling things correctly and using the right punctuation.)

Then we see something pretty strange on line 2: a ***constructor***: it allows us to make instances (remember Gerald, Timothy, and Jeeves) of the *Cat* class where we can plug in our particular, distinguishing features for those little guys. Any instance of the *Cat* class will thus have a name, a particular hair colour, an eye colour, and a gender, when it's made (plus all the methods, or abilities, specified by the class).

- Another word for giving an instance to those features is ***initializing*** it. This is why the built-in '**__init__**' function - which we have to involve when we make a class in Python - has the name it does. The variable called 'self' always has to be there, too – it just serves to keep things specific to the particular cat we're making when we make an instance.

Methods of a Class

Then we see those cat abilities we talked about! Jumping, meowing, eating and sleeping are now methods that any *Cat* instance has. So let's make a couple of *Cat* instances, i.e., cute unique guys with unique features:

```
21  Timothy = Cat('Timothy', 'Amber', 'Blue', 'Male')
22
23  Timpthy.sleep()
24
25  Gerald = Cat('Gerald', 'White', 'Brown', 'Male')
26
27  Gerald.jump()
```

On line 21, we make a cat called Timothy and initialize him with that name, and various other particular values for his intrinsic variables. On line 23, we tell him to execute the sleep() method. On line 25, we make a different cat called Gerald, with different qualities, and then tell Gerald to jump. You may have noticed that when we use the sleep() and jump() method, we're using them *via* Timothy and Gerald, and we don't have to put anything in the brackets of sleep() and jump(); this is because it's clear to Python who the 'self' is each time, i.e., who we want to do the sleeping and the jumping. The code, when run, will output the following:

```
Sleeping deeply…

Jumping joyously!
```

As you can see, we access the intrinsic variables and methods of objects when programming by using the **full stop symbol '.'**. Bear in mind, too, that all Python objects have been thought up and defined by somebody, somewhere. As a result, any convention of writing, say, Timothy.munch(apples,5) rather than Timothy.munch(5, apples) when we're trying to get Timothy to munch 5 apples is always in part arbitrary (we, of course, don't have such a munch() method in the Class example we've been working through, but suppose it was there). The order of the things inside the brackets to a method - otherwise known as the function's ***parameters*** or ***inputs*** - was decided by the people who defined the method in question, often years ago, for reasons even they often forget!

Notice, too, that we'd already been sneakily introduced to object-oriented programming when we looked at stuff like integers, strings, and lists. Any particular string – such as 'confetti flamingo' or 'Filibuster skullcap 3000' – has certain intrinsic variables and functions due to its being an instance of the class *String*. That class was written by the makers of Python over twenty years ago. The same goes for integers, lists, and Python's other data types.

Creating Functions

We've seen how classes are made, now let's see how functions are created. Here's a function, that we've chosen to call 'fourTimes', being defined:

```
1  def fourTimes(word):
2      return(word * 4)
```

This function takes an input (that we've chosen to call 'word', but we could have called anything) and prints out that word repeated four times in a row, without spaces. The *** operator** serves as mathematical multiplication when the operands are both numbers, but when one of the operators is a string and the other a number, the string is repeated as many times as the number. (For other operations that work on strings, check out this resource).

The ***return*** keyword signifies: here's what this function *outputs* (or produces). It outputs the input * 4.

- If the input is a string, the output will be a new string: the original string repeated 4 times.
- If the input is a number, the output will be the number multiplied by 4.

If we now execute this function with the code:

```
4  print(fourTimes('mole'))
```

Then we will get the output:

```
molemolemolemole
```

Our execution is printing the result of inputting the string 'mole' into the function fourTimes(). Python is unable to 'see' that the value of a variable called 'word' should probably be a string.

Defining vs. Executing

Here's an important thing to note: there's a key difference between ***defining*** (or creating) a function and ***executing*** (or running) a function. Likewise, there's an important difference between ***defining*** **a class** (playing God, by defining a type of thing, and saying what will be common to its instances) and ***instantiating*** **a class** (also playing God, but by making an instance of the class). When we're defining a function, we're stating what that function does to any inputs it might have, and what it outputs. To recap a little, some functions are nestled within classes, and these are called methods (like the jump() method within the *Cat* class, above), and can only be used if we first instantiate the class. Functions, strictly speaking, can be used without instantiating any class.

Printing vs. Returning

There's a crucial difference between a function ***printing*** something, and a function ***returning*** (outputting) something. Printing just pushes things out to the console for us to read, and is useful for ***debugging***: we can print() to identify how far the Python interpreter got through our linear sequence of instructions before it hit a problem. Returning, by contrast, specifies what the function officially outputs or produces. For example:

```
1  def no_Output(number):
2      print(number * 4)
3  y = no_Output(3)
4  print(y)
```

Lines 1 and 2 inclusive define a new function called no_Output. Everything the function does is indented, which is only one thing, on line 2: it prints number * 4, where 'number' is the name of our input. (Remember: the value of the number variable could actually be a string, despite what we've chosen to call it). But the no_Output function doesn't actually return (or output) anything.

So, in line 3, when we try to make the value of the variable y the result of plugging 3 into no_Output, nothing actually gets stored in y. The output we get from the print(y) instruction on line 4 is simply:

```
None
```

Which is what Python outputs if we're printing a variable with no value. The moral of the story, which can confuse even intermediate programmers: functions *returning* stuff is different from functions *printing* stuff.

Global vs. Local Variables

Imagine we're running a zoo in a dystopian society where people *love* groups of giraffes whose size is exactly 5, but *hate* groups of giraffes of different sizes. We currently have 3 giraffes. *Gulp*. If we manage to get exactly 5 giraffes (no more, no less) we'll make $100. Otherwise we'll make nothing. Also, in our desperate, struggling zoo, we concoct the idea of a cash tree that makes exactly $50,000 (but obviously, we never manage to make one). Check out the following code:

```
1  giraffes = 3
2
3  if (giraffes == 5):
4      income = 100
5
6  def cashTree():
7      money = 50000
8      return money
9
10 print(giraffes)
11 print(income)
12 print(money)
```

So on line 1 we can see that we actually have 3 giraffes, not 5. The value of the Boolean condition within the if-statement on line 3 is False, so tragically, the variable called *income* is never created! Also, while we ***define*** the cashTree() function, we never actually *execute* it; so the variable called *money* is never created either! As a result, our output from the print statements on lines 10-12 is:

```
3
Traceback (most recent call last):
  File "main.py", line 11, in <module>
    print(income)
NameError: name 'income' is not
defined
```

The value of our *giraffes* variable successfully printed out (something we requested on line 10). But line 11 threw a NameError: there's no *income* variable, because it would only have been created had our *giraffes* variable had exactly the value 5!

What we're looking at here is the difference between ***global*** and ***local*** variables. The variable *giraffes* is ***global***: it's declared outside of any control-flow structure (like an if-statement or while loop), function, or class. By contrast, the variable *income* is local to the if-statement on lines 3 and 4, and the variable *money* is local to the cashTree() function. This means that *unless* the Boolean condition of the if-statement on line 3 has the value True, the variable *income* will not be created, and unless we execute the cashTree() function, the variable *money* will not be created either! As a result, attempts to print those variables without their already having been created will throw an error.

By contrast, look at this miraculous code:

```
1  giraffes = 5
2
3  if (giraffes == 5):
4      income = 100
5
6  def cashTree():
7      money = 50000
8      return money
9
10 print(giraffes)
11 print(income)
12 profit = cashTree()
13 print(profit)
```

Our *giraffes* variable has that magic number as its value: 5. As a result, the *income* variable is created, and successfully printed on line 11 (hooray)! Also, on like 12, we make a variable called *profit*, and assign to it the result of executing the cashTree() function: aka, we assign to it the

value of the local variable *money*, aka, a cool 50000. We then on line 13 print the value of *profit*. So we get the output:

```
5

100

50000
```

Oh the sweet smell of success. We owe whichever shamen gave us the power to execute our moneyTree() business plan a drink or two.

That was a brief foray into object-oriented programming! This is a topic often left for the latter stages of courses, but we think it's useful to be introduced to from the beginning so that you understand they **why** behind the Python code you're learning.