Data Scientist
Codecademy
Python Fundamentals
Classes

Types

Python equips us with many different ways to store data. A float is a different kind of number from an int, and we stored different data in a list than we do in a dict. These are known as different types. We can check the type of a Python variable using the type() function. A variable's type defines the operations that can be performed on them.

```
print(type(variable_name))
```

Class

A class is a template for a data type. It describes the kind of information that class will hold and how the programmer will interact with the data. A class is defined using the keyword 'class'. It is recommended to capitalize the class names for easier distinction.

pass keyword is used to indicate that the body of the class, function or loop was intentionally left blank so that Python doesn't raises IndentationError.

```
class ClassName:
  pass
```

Instantiation

A class must be instantiated, i.e., we must create an instance of the class, in order to breathe life into the schematic or use it.

```
class_instace = ClassName()
```

Object-Oriented Programming

A class instance is also called an object. The pattern of defining classes and creating objects to represent the responsibilities of a program is known as *Object Oriented Programming* or OOP.

Instantiation takes a class and turns it into an object. The type() function does opposite, it returns the class that the object is an instance of.

When we call type() function with a class instance of a class we defined, it returns something like <class '__main__.ClassName'>

Here __main__ means "this current file that we're running" telling that the class was defined in the currently running script.

Class Variables

When we want the same data to be available to every instance of a class we use a *class variable*. A class variable is a variable that's the same for every instance of the class.

A class variable can be accessed by

```
class_object.variable_name
```

Methods

Methods are functions that are defined as part of a class. The first argument in a method is always the object that is calling the method. Convention recommends that we name this first argument self. Methods always have at least this one argument.

Methods are defined similarly as functions, just indented to be part of the class.

```
class ClassName:
  def method_name(self):
    pass


class_object = ClassName()
class_object.method_name()
```

The argument self refers to the object calling the function.

Methods with Arguments

Methods can have more arguments similarly like functions, just remember that the first argument shall be self which will refer to the calling object.

When accessing class variables inside a method use syntax

```
ClassName.variable_name
```

Constructors

There are several methods that we can define in a Python class that have special behavior. These methods are called magic, or *dunder methods*.

__init__() : used to initialize a newly created object. It is called every time the class is instantiated.

Methods that are used to prepare an object being instantiated are called constructors.

Constructors can be parameterized or not (self is compulsory).

```
class Shouter:
  def __init__(self):
    print(“Hello”)


shout1 = Shouter()
# prints “Hello”


class Shout:
  def __init__(seld, phrase):
    print(phrase.upper())


shout = Shout(“Hello”)
# prints “HELLO”
```

Instance Variables

The data held by an object is referred to as an *instance variable*. Instance variables aren't shared by all instances of a class - they are specific to the object they are attached to.

Attribute Functions

Instance variables and class variables are both accessed similarly in Python. They both are attributes of an object. If we try to access an attribute which is neither of the two it will throw an AttributeError.

hasattr(object, "attribute")

- object the object we are testing to see if it has a certain attribute
- attribute name of attribute we want to see if it exists

getattr(object, "attribute", default)

- object the object whose attribute we want to evaluate
- attribute name of attribute we want to evaluate
- default the value that is returned if the attribute does not exit optional parameter

Self

Instance variables are more powerful when we can guarantee a rigidity to the data the object is holding.

This convenience is most apparent when the constructor creates the instance variables, using the arguments passed in to it.

```
class SearchEngineEntry:

  secure_prefix = "https://"

  def __init__(self, url):

    self.url = url


  def secure(self):

    return "{prefix}{site}".format(prefix=self.secure_prefix, site=self.url)
```

```
codecademy = SearchEngineEntry("www.codecademy.com")
wikipedia = SearchEngineEntry("www.wikipedia.org")

print(codecademy.secure())
# prints "https://www.codecademy.com"

print(wikipedia.secure())
# prints "https://www.wikipedia.org"
```

Everything is an Object

Attributes can be added to user-defined objects after instantiation, so it's possible for an object to have some attributes that are not explicitly defined in an object's constructor. We can use the dir() function to investigate an object's attributes at runtime.

Python's native data types are objects in Python.

String Representation

__repr__(): tells python what we want the string representation of the class to be

must return a string

can have only self parameter