What is Production Environment?

Production environment is a term used mostly by developers to describe the setting where software and other products are actually put into operation for their intended uses by end users.

Concepts that will be useful in writing production grade code:

- Object Oriented Programming

1. Object Oriented Programming

Object Oriented Programming (OOP) tends to be one of the most interesting concepts to learn in Python.

For this lesson we will construct our knowledge of OOP in Python by building on the following topics:

- Objects
- Using the *class* keyword
- Creating class attributes
- Creating methods in a class
- Learning about Polymorphism

Lets start the lesson by remembering about the Basic Python Objects. For example:

Remember how we could call methods on a list?

What we will basically be doing in this lecture is exploring how we could create an Object type like a list. We've already learned about how to create functions. So let's explore Objects in general:

Objects

In Python, *everything is an object*. Remember from previous lectures we can use type() to check the type of object something is:

```
the_sixth_sense = 6
print(type(the_sixth_sense))
```

```
<class 'int'>
```

```
two_and_a_half_men = 2.5
print(type(two_and_a_half_men))
```

```
<class 'float'>
```

```
schindlers_list = list()
print(type(schindlers_list))
```

```
<class 'list'>
```

```
another_list = list()
```

```
tharoor = {'farrago':'a confused mixture'}
print(type(tharoor))
```

```
<class 'dict'>
```

So we know all these things are objects, so how can we create our own Object types? That is where the `class` keyword comes in.

class

User defined objects are created using the `class` keyword. The class is a blueprint that defines the nature of a future object. From classes we can construct instances. An instance is a specific object created from a particular class. For example, above we created the object `another_empty_list` which was an instance of a list object.

Let see how we can use `class`:

```
# Create a new object type called FirstClass
class FirstClass:
    pass
```

```
# Instance of FirstClass
x = FirstClass()
print(type(x))
```

```
<class '__main__.FirstClass'>
```

```
y = FirstClass()
print(type(y))
```

```
<class '__main__.FirstClass'>
```

By convention we give classes a name that starts with a capital letter. Note how `x` is now the reference to our new instance of a FirstClass class. In other words, we **instantiate** the FirstClass class.

Inside of the class we currently just have pass. But we can define class attributes and methods.

An **attribute** is a characteristic of an object. A **method** is an operation we can perform with the object.

For example, we can create a class called Dog. An attribute of a dog may be its breed or its name, while a method of a dog may be defined by a .bark() method which returns a sound.

Let's get a better understanding of attributes through an example.

Attributes

The syntax for creating an attribute is:

```
self.attribute = something
```

There is a special method called:

```
__init__()
```

This method is used to initialize the attributes of an object. For example:

```
class Dog:

    def __init__(self, breed, name):
        self.breed_attribute = breed
        self.name_attribute = name


sam_object = Dog(breed='Lab', name = 'Sam')
frank_object = Dog(breed='Huskie', name = 'Frank')


sam_object.breed_attribute
```

```
'Lab'
```

Lets break down what we have above.The special method

```
__init__()
```

is called automatically right after the object has been created:

```
 def __init__(self, breed):
```

Each attribute in a class definition begins with a reference to the instance object. It is by convention named self. The breed is the argument. The value is passed during the class instantiation.

```
  self.breed = breed
```

Now we have created two instances of the Dog class. With two breed types, we can then access these attributes like this:

```
sam_object.name_attribute
```

```
'Sam'
```

```
frank_object.name_attribute
```

```
'Frank'
```

Note how we don't have any parentheses after breed; this is because it is an attribute and doesn't take any arguments.

Methods

Methods are functions defined inside the body of a class. They are used to perform operations with the attributes of our objects. Methods are a key concept of the OOP paradigm. They are essential to dividing responsibilities in programming, especially in large applications.

You can basically think of methods as functions acting on an Object that take the Object itself into account through its *self* argument.

Let's go through an example of creating a Circle class:

```
class Circle:

  def __init__(self, radius=1):
    self.radius = radius
    self.area = 3.14 * radius * radius

  def setRadius(self, new_radius):
```

```
    self.radius = new_radius
    self.area = 3.14 * new_radius * new_radius

  def getCircumference(self):
    return 2 * 3.14 * self.radius


c = Circle()


c.area
```

```
3.14
```

```
c.getCircumference()
```

```
6.28
```

```
print('Radius is: ', c.radius)
print('Area is: ', c.area)
print('Circumference is: ', c.getCircumference())
```

```
Radius is:  1
Area is:  3.14
Circumference is:  6.28
```

Now let's change the radius and see how that affects our Circle object:

```
c.setRadius(4)

print('Radius is: ', c.radius)
print('Area is: ', c.area)
print('Circumference is: ', c.getCircumference())
```

```
Radius is:  4
Area is:  50.24
Circumference is:  25.12
```

Great! Notice how we used self. notation to reference attributes of the class within the method calls. Review how the code above works and try creating your own method.

Polymorphism

We've learned that while functions can take in different arguments, methods belong to the objects they act on. In Python, *polymorphism* refers to the way in which different object classes can share

the same method name, and those methods can be called from the same place even though a variety of different objects might be passed in. The best way to explain this is by example:

```
class HouseStark:
    def __init__(self, sigil):
        self.sigil = sigil

    def motto(self):
        return "House Stark with sigil " + self.sigil + " has the motto 'Winter is coming'"


class HouseLannister:
    def __init__(self, sigil):
        self.sigil = sigil

    def motto(self):
        return "House Lannister with sigil " + self.sigil + " has the motto 'Hear me roar'"


arya = HouseStark('direwolf')
tyrion = HouseLannister('golden lion')


print(arya.motto())
```

```
House Stark with sigil direwolf has the motto 'Winter is coming'
```

```
print(tyrion.motto())
```

```
House Lannister with sigil golden lion has the motto 'Hear me roar'
```

Here we have a HouseStark class and a HouseLannister class, and each has a `.motto()` method. When called, each object's `.motto()` method returns a result unique to the object.

There a few different ways to demonstrate polymorphism. First, with a for loop:

```
for warrior in [arya, tyrion]:
    print(warrior.motto())
```

```
House Stark with sigil direwolf has the motto 'Winter is coming'
House Lannister with sigil golden lion has the motto 'Hear me roar'
```

Another is with functions:

```
def get_motto(warrior):
    print(warrior.motto())
```

```
get_motto(arya)
get_motto(tyrion)
```

```
House Stark with sigil direwolf has the motto 'Winter is coming'
House Lannister with sigil golden lion has the motto 'Hear me roar'
```

In both cases we were able to pass in different object types, and we obtained object-specific results from the same mechanism.

Great! By now you should have a basic understanding of how to create your own objects with class in Python.

Real World Use Case

Uber's simplified pricing model

When you request an Uber, you enter your pick-up location and the destination. Based on the distance, peak hours, willingness to pay and many other factors, Uber uses a machine learning algorithm to compute what prices will be shown to you.

Let's consider a simplistic version of the Uber Pricing Model where you compute the price based on the distance between the pick-up and the drop location and the time of the booking.

User Inputs:

- Pick-up location (pick_up_latitude, pick_up_longitude)
- Drop location (drop_latitude, drop_longitude)
- Time of booking

Output:

- Final Price

Development Code

Production Grade Code

```
# Calculate the distance between the pick-up location and the drop location

import geopy.distance
import math
```

```
class Maps:

    def __init__(self):
        pass


class SurgePricing:

    def __init__(self):
        pass
```