Objects and Classes

Python is an object oriented programming language. Almost everything in Python is an object, with its properties and methods. A Class is like an object constructor, or a "blueprint" for creating objects.

to create a class we use the keyword

```
class <class_name>
```

example

```
class Person:

    def __init__(self,name):
        self.name = name

    def say_hi(self):
        print("hello " + self.name)

p = Person('hi')
p.say_hi()
```

OUTPUT

```
Person.py
```

each time we do Person(), python creates a new object as it can be seen with this

```
p = Person('hi')
q = Person('hi');

print(p)
print(q)
```

OUTPUT

```
<__main__.Person object at 0x7f0f074729b0>
<__main__.Person object at 0x7f0f0747cb38>
```

When printed, the two objects tell us what class they are and what memory address they live at.

We can set arbitrary attributes on an instantiated object using the dot notation:
ex :

```
class Point:
    pass

p1 = Point()
p2 = Point()

p1.x = 5
p1.y = 4

p2.x = 3
```

```
p2.y = 6
print(p1.x, p1.y)
print(p2.x, p2.y)
```

OUTPUT

```
5 4
3 6
```

This code creates an empty Point class with no data or behaviors. Then it creates two instances of that class and assigns each of those instances x and y coordinates to identify a point in two dimensions. All we need to do to assign a value to an attribute on an object is use the syntax <object>.<attribute> = <value>. This is sometimes referred to as **dot notation**. The value can be anything:

```
class Point:
    def reset(self):
        self.x = 0
        self.y = 0

p = Point()
p.reset()
print(p.x, p.y)
```

OUTPUT

```
0 0
```

The self argument to a method is simply a reference to the object that the method is being invoked on. We can access attributes and methods of that object as if it were any other object.

If we dont add **self** we get an error about the arguments

```
class assignVar():

    def reset():
        self.x = 0
        self.y = 0

    def printValues(self):
        print("the value of the object is " + str(self.x) + " " + str(self.y))

p = assignVar()
p.x = 5
p.y = 6

p.printValues()
p.reset();
```

OUTPUT

```
the value of the object is 5 6
Traceback (most recent call last):
```

```
  File "Person.py", line 15, in <module>
    p.reset();
TypeError: reset() takes 0 positional arguments but 1 was given
```

CLASS VS INSTANCE VARIABLES

- instance variables are variables whose value is assigned inside a constructor or method with self.
- Class variables are variables whose value is assigned in class.

```
class CSStudent:

    # Class Variable
    stream = 'cse'

    # The init method or constructor
    def __init__(self, roll):

        # Instance Variable
        self.roll = roll
```

In this case stream is a class variable while roll is an instance variable .So whenver we create a new object the stream will have the same value while the value of roll will be different

INSTANCE ATTRIBUTES

ACCESSING Outside the class

```
<variable>.<instance_attribute>
```

ex :

```
myhouse.price
```

ACCESSING Inside the class

```
self.<instance_attribute>
```

ex :

```
self.price
```

MODIFY outside the class

```
<variable>.<instance_attribute> = new_value
```

ex :

```
myhouse.price = 500
```

MODIFY inside the class

```
self.<instance_attribute> = new_value
```

ex :

```
self.price = 500
```

CLASS ATTRIBUTES

- They belong to the class
- Are shared by all instances
- Same value for all the instances

They defined before the init method in the class

```
<class atribute> = value
```

ACCESSING Outside the class

```
<class_name>.<class_attribute>
```

ACCESSING Inside the class

```
<class_name>.<class_attribute>
```

ex :

```
class Car:

    seats = 4

    def __init__(self, model):
        self.model = model;

    def printModel(self):
        print(str(Car.seats) +" " + self.model) // accessing inside the class

car1 = Car("tesla")
car1.printModel()
print(Car.seats) //accessing outside the class
print(car1.seats) //can also use objects to access it
```

OUTPUT

```
4 tesla
4
4
```

MODIFY outside the class

```
<class_name>.<class_attribute> = value
```

MODIFY inside the class

```
<class_name>.<class_attribute> = value
```

```
class Car:

    seats = 4

    def __init__(self, model):
        self.model = model;

    def printModel(self):
        print(str(Car.seats) +" " + self.model)

car1 = Car("tesla")
car1.printModel()
print(Car.seats)
print(car1.seats)

#modifing the class atribute
Car.seats = 10
print(car1.seats)
```

OUTPUT

```
4 tesla
4
4
10
```

changing the class attribute will change it for all the instances

example
:

```
class Car:

    seats = 4

    def __init__(self, model):
        self.model = model;

    def printModel(self):
        print("the class atribute inside is " + str(Car.seats))
        print("the instance atribute inside is " + str(self.seats))

car1 = Car("tesla")

print(car1.seats)
print(Car.seats)
#modifing the class atribute
Car.seats = 10
car1.seats = 15
```

```
car1.seats = 15

print("acccesing class attribute outisde the class " + str(Car.seats))
car1.printModel()
```

OUTPUT

```
4
4
acccesing class attribute outisde the class 10
the class atribute inside is 10
the instance atribute inside is 15
```

Hence changing the class atribute changes it everywhere. We can also change the attribute specifically for an instance

What would be the output?

```
class A:

    attr = 5

    def __init__(self):
        A.attr +=1

a1 = A()
a2 = A()
print(A.attr)
A.attr = 26
a3 = A()
print(A.attr)
```

OUTPUT

```
7
27
```

CONSTRUCTORS IN PYTHON

- Python is a little different; it has a constructor and an initializer.
- The Python initialization method is the same as any other method, except it has a special name: __init__.
- The leading and trailing double underscores mean, "this is a special method that the Python interpreter will treat as a special case".
- Never name a function of your own with leading and trailing double underscores.
- The constructor function is called __new__ as opposed to __init__, and accepts exactly one argument, the class that is being constructed It also has to return the newly created object.However it is rarely used

__init__()

- It is knows as the reserved method
- It is executed when an instance of the object is created.

__init__(): Common Mistakes

Common Mistakes with __init__()

- **Omitting the def keyword:**

```
__init__(self, width, height): # The def keyword is missing!
        self.width = width
        self.height = height
```

- **Using only one underscore (You must use two):**

```
def _init_(self, width, height): # There is only ONE underscore instead of two!
        self.width = width
        self.height = height
```

- **Omitting self as the first parameter:**

```
def __init__(width, height): # Self is missing. It must be the FIRST parameter!
        self.width = width
        self.height = height
```

- **Not using self.<attribute> to assign instance attributes:**

```
def __init__(self, width, height):
        width = width        # You should use self.width to assign them!
        height = height
```

Formal Parameters: PEP 8 Style Guide

- Parameters should be separated by a comma and a space after the comma.

Correct:

```
def __init__(self, name, age):
```

Incorrect:

```
def __init__(self,name,age):
```

- If the name of the parameter has more than one word, you should separate the words with an

underscore (snake_case format).

For example:

```
def __init__(hair_color, eye_color, num_children):
```

- If the name of a parameter clashes with the name of a keyword, you should add a trailing underscore to the name.

For example:

```
def __init__(description, type_):
```

What is None?

None is a Python keyword used to define a **null variable or object**.

We typically use it to indicate that a variable has no value or object assigned to it (yet).

It is an object itself, as you can see right here when we check if it is an instance of object:

```
>>> isinstance(None, object)
True
```

Its also a data type of its own (NoneType):

```
>>> type(None)
<class 'NoneType'>
```

The None value can be used in comparisons and operations that involve == and is.

The None value can be used in comparisons and operations that involve == and is.

💡 **Tip:** you will learn more about the is operator in the section "Objects in Memory."

You can check if the value of a variable is None with this syntax:

```
if <var> is None:
    # do something if the value is None
```

Alternatively:

```
if <var> is not None:
   # do something if the value is not None
```

You can also use the == (equal to) and != (not equal to) operators, like this:

```
if <var> == None:
    # do something if the value is None
```

```
if <var> != None:
   # do something if the value is not None
```

Comparing None to anything will always return False except when we compare it to None itself.

None is used to assign default arguments to the parameters of __init__().

It can also be used to define optional arguments for functions and methods.

DEFAULT ARGUMENTS

This is useful when we want to omit the arguments when we create an instance

ex :

```
class Patient:

    def __init__(self, name, age, alergies=None, num_children = 0):
        self.name = name
        self.age = age
        self.alergies = alergies
        self.num_children = num_children

    def printObject(self):
        print(self.name,self.age,self.alergies,self.num_children)
person1 = Patient("ram",12)
person2 = Patient("ram",12,["peanuts"])
person3 = Patient("ram",12,["peanuts"],5)
person4 = Patient("ram",12,num_children=5) #to omit an argument we have to specify which paramter

person1.printObject();
person2.printObject();
person3.printObject();
person4.printObject();
```

OUTPUT

```
ram 12 None 0
ram 12 ['peanuts'] 0
ram 12 ['peanuts'] 5
ram 12 None 5
```

Default arguments should have no spaces around the equal sign.

Yes:

```
def __init__(self, name, age=10):
```

No:

```
def __init__(self, name, age = 10):
```

To omit an argument we have to specify the paramter while creating an object.

```
person4 = Patient("ram",12,num_children=5)
```