OOP

Objects:

In Python, everything is an object. Remember from previous lectures we can use type()

Eg:
```
print(type(1))
print(type([]))
print(type(()))
print(type({}))
```

O/P:
```
<class 'int'>
<class 'list'>
<class 'tuple'>
<class 'dict'>
```

Class

User defined objects are created using the class keyword. The class is a blueprint that defines the nature of future object. From classes we construct instances. An instance is a specific object created from a particular class.

Eg:
```
# Create a new object type called Sample
class Sample:
    pass
```

Instance of Sample

```
x = Sample()
print(type(x))
```

O/p: `<class '__main__.Sample'>`

By convention we give classes a name & name that starts with a capital letter. x is reference to new instance. In other words we instantiate the Sample class.

An attribute is a characteristics of object.

Attributes

Syntax:

```
self.attribute = something
```

Special method:

`__init__()`

is called automatically right after the object is created.

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

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

```
self.breed = breed
```

Eg:
```
class Dog:
    def __init__(self, breed):
        self.breed = breed

sam = Dog('Lab')
frank = Dog('Huskee')
```

Note: You don't need parentheses ~~with~~ after breed, because it's an attribute & doesn't take any arguments

```
Qn: sam.breed
O/P: 'Lab'

Qn: frank.breed
O/P: 'Huskee'
```

Object Attributes:

Object attr. are same for any instance of class. If we create attr. species for Dog class. Dogs, regardless of their breed

name or ~~their~~ other attributes, will be mammals.

In: class Dog:

    # Class Object Attribute
    species = 'mammal'

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

In: ~~sam = Dog('lab'~~

In: sam = Dog('Lab', 'Sam')

In: sam.name

O/p: 'Sam'

Note: Class Object Attr. are defined outside of any methods. Also by convention we place them before __init__.

In: sam.species
O/p: 'mammal'

Methods:

Methods are functions defined inside body of class. They act on Object

that take object into itself through self argument.

qn:
```
class Circle:
    pi = 3.14

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


    # For resetting radius

    def setRadius(self, new_radius):
        self.radius = new_radius
        self.area = new_radius * new_radius * self.pi


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


c = Circle()

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

o/p:
```
Radius is: 1
Area is: 3.14
Circumference is: 6.28
```

9n: c.setRadius (2)

print("Area: ", c.area)
print("Radius: ", c.radius)
print("Circumference: ", c.circumference())

O/p: Area: 12.56
Radius: 2
Circumference: 12.56

Inheritance

It is a way to form new classes using classes that have already been defined. The newly formed classes are called derived classes, the classes that we derive from are called base classes.

9n:
```
class Animal:
    def __init__(self):
        print("Animal created")

    def whoAmI(self):
        print("Animal")

    def eat(self):
        print("Eating")
```

```
class Dog(Animal):
    def __init__(self):
        Animal.__init__(self)
        print('Dog created')

    def whoAmI(self):
        print('Dog')

    def bark(self):
        print('woof!')
```

In: d = Dog()

O/P: Animal created
Dog created

In: d.whoAmI()

O/P: Dog

In: d.eat()

O/P: Eating

In: d.bark()

O/P: woof!

POLYMORPHISM

It refers to the way in which diff. object classes can share the same method name, & those methods can be called from the same place even though a variety of diff. objects might be passed in.

In:
```
class Dog:

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

    def speak(self):
        return self.name + ' says Woof!'

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

    def speak(self):
        return self.name + ' says Meow!'

niko = Dog('Niko')
felix = Cat('Felix')

print(niko.speak())
print(felix.speak())
```

o/p: Mike says Woof!
Felix says Meow!

E.g.-8

Implementation with a for loop:

```
In: for pet in [mike, felix]:
        print(pet.speak())
```

o/p: Mike says Woof!
Felix says Meow!

Implementation with func^n:

```
In: def pet_speak(pet):
        print(pet.speak())

pet_speak(mike)
pet_speak(felix)
```

o/p: Mike says Woof!
Felix says Meow!

Abstract class

An abstract class is one that never expects to be instantiated.

Date
Page

9n:
```
class Animals:
    def __init__(self, name): #Constr.
        self.name=name

    def speak(self):
        raise NotImplementedError('
Subclass must implement abs. method')

class Dog(Animal):

    def speak(self):
        return self.name+' says
                    Woof!'

class Cat(Animal):

    def speak(self):
        return self.name+' says
                    Meow!'

print(fido.speak())
print(isis.speak())
```

O/p: Fido says Woof!
Isis says Meow!

~~Re~~ Real life e.g.s

→ Opening diff. file types - diff. tools are needed to display Word, pdf & Excel files

→ Adding diff objects - the + operator performs arithmetic & concatenation

Special Method:

These methods are not actually called directly but Python specific lang: syntax.

Eg:
```
class Book:
    def __init__(self, title, author, pages):
        print("A book is created")
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        return "Title: %s, author: %s, pages: %s" %(self.title, self.author,
                                                    self.pages)

    def __len__(self):
        return self.pages

    def __del__(self):
        print("A book is destroyed")
```

Qn: book = Book('Python Rocks!', 'Jose Portilla',
159)

print(book)
print(len(book))
del book

o/p: A book is created
Title: Python Rocks!, author: Jose Portilla,
pages: 159
159
A book is destroyed.