Unit 6
Classes and Objects

"Everything in Python, from numbers to modules, is an object".
Python hides most of the object machinery by means of special syntax.

eg:
>>>num = 5
This creates an object of type 'integer' with the value 5, and assign an object reference to the name 'num'.

The only time you need to look inside objects is when you want to make your own or modify the behaviour of existing objects.

Some trivia about Objects...
1. An object contains both data (variables, called attributes) and code (functions, called methods).
2. It represents a unique instance of some concrete thing.

eg:
The integer object with value 7 is an object that facilitates methods such as addition and multiplication.
8 is a different object that belongs to class 'integer'.

When you create new objects no one has ever created before, you must create a class that indicates what they contain.

Objects can be thought of as 'nouns' and their methods as 'verbs'. An object represents an individual thing and its methods define how it interacts with other things.

Note:
(1) In Python, OOP is entirely optional! You can get plenty of work done with simpler constructs such as functions, or even simple top-level script code. Because using classes well, requires planning. It is interesting to people who work in "strategical" mode (long term product development) than to people who work in "tactical" mode (where time is in very short supply).

(2) When used well, classes can actually cut development time radically.

(3) They are also employed in popular Python tools like the *tkinter GUI API*

Defining a class:

Classes are Python's main OOP tool.
OOP offers a different and often more effective way of programming, in which we factor code to minimize redundancy and write new programs by customizing existing code instead of changing it in place.

In Python, classes are created with a new statement: class
eg:

```
>>>class Person(): pass
...

# This creates a class called 'Person' with no attributes and methods!!
# Remember 'pass', ... it is used to defer/postpone the definition/code

>>>print(Person)
<class '_main_.Person'>
>>>
# Defining a class named Person creates a class object. Because 'Person' is
defined at the top level, its "full name" is_main_.Person
```

To create an instance (object) of Person class..

Syntax:

```
>>>object_name = Class_name()
```

eg:

```
>>>someone = Person()

>>>type(someone)
<class '_main_.Person'>
>>>
```

Attributes...
The 'dot' operator is used to assign values to an instance/object

Syntax:

```
        object.attribute = value
```

eg:

```
>>>someone.name = 'Alice'
>>>someone.age = 18
```

```
>>>someone.height = 165.5
```

Accessing the attributes:

Using print..

eg(1): Directly..

```
>>>print(someone.age)
18
```

eg(2): Thru a method..

```
>>>def print_P(p):
. . .       print('Name: %s, Age: %d, Height: %g' %(p.name, p.age, p.height))
. . .
>>>
>>>print_P(someone)
Name: Alice, Age: 18, Height: 165.5
>>>
```

Note: It is preferable to define methods/functions inside the class definition...

```
>>>class Person():
. . .       def print_P(self):
. . .                   print('Name: %s, Age: %d, Height: %g' %(self.name, self.age,
self.height))
. . .
>>>
>>>someone = Person()
>>>someone.name = 'Alice'
>>>someone.age = 18
>>>someone.height = 165.5

>>>someone.print_P()
Name: Alice, Age: 18, Height: 165.5
```

Constructors, the initialization method:

A better way of initializing attributes of an object in a class.

Syntax:

```
>>> class ClassName():
. . . def_init_(self, value):
. . .             self.attr = value
. . .
>>>
```

__init_() is the special Python name for a method that initializes an individual object from its class definition
'self' argument specifies that it refers to the individual object itself.

eg:

```
>>>class Some():
. . . def_init_(self, s1):
. . .            self.name = s1
. . .
>>>

>>>s1=Some('Bob')
>>>print(s1.name)
Bob
```

Note:
1) Inside the *'Person'* class definition, you access the name attribute as *'self.name'*.
2) When an actual object is created, say, *'someone'*, you refer to it as *'someone.name'*.
3) Its not necessary to have an *init ()* in every class definition

Exercise:
Define a class called Student with attributes: name, age, marks of 3 courses and average_marks. Write a program that reads the details of a student from the user and prints them after calculating the average_marks in a suitable format.

Consider the following example,

```
>>>class Point():
. . . pass
. . .
>>>
```

```
>>>P1 = Point()
# An object is created for the Point class

>>>P1.x = 3.0
>>>P1.y = 4.0
# Two attributes 'x' and 'y' defined with values 3.0 and 4.0 respectively.
```

Now, let's define another class..

```
>>>class Rectangle():
. . .       ''' has the following attributes:
. . .       width, height and corner '''
. . .
>>>

# Recall, the definition contains only the 'docstring' specifying the attributes.
```

Let's create an object of class Rectangle..

```
>>>R1 = Rectangle()
>>>R1.width = 10.0
>>>R1.height = 20.0
>>>R1.corner = Point()
#This attribute is an object of another class!
>>>R1.corner.x = 5.0
>>>R1.corner.y = 5.0
```

Objects are mutable.

eg(1):

```
>>>R1.width
10.0
>>>R1.width +=5.0
>>>R1.height
20.0
>>>R1.height +=5.0
```

eg(2):

```
>>>def big_Rect(R, w, h):
. . .       R.width += w
. . .       R.height += h
. . .
>>>
```

```
>>>big_Rect(R1, 5.0, 5.0)
>>>print(R1.width)
15.0
>>>print(R1.height)
25.0
```

Note:
No ‘scope-resolution’ operator in Python. But, the method can be defined during class definition or sometime later outside the class. It is usually preferred to write the methods inside the class definition.

Copying
Aliasing makes a program difficult to read as changes in one place might have unexpected effects in another place.

An alternate to aliasing is 'copying' an object
(Also referred to as 'shallow copy'!)

eg(1):
```
>>>p1 = Point()
>>>p1.x = 10.0
>>>p1.y = 15.0

>>>from copy import copy
>>>p2 = copy(p1)
>>>

>>>p2.x
10.0
>>>p2.y
15.0
>>>p1 is p2
False #not aliased, but copied!!

>>>p1.x is p2.x
?
```

eg(2):
```
>>>R1 = Rectangle()
>>>R1.width = 10.0
>>>R1.height = 20.0
>>>R1.corner = Point()
```

```
>>>R1.corner.x = 5.0
>>>R1.corner.y = 5.0
>>>R2 = copy.copy(R1)
>>>R1 is R2
False
>>>R1.corner is R2.corner
True
```

This operation is called a 'shallow copy' because it copies the object and any references it contains, but not the embedded objects (an object as an attribute of another object).

Deep copy
'deepcopy()' is a method in module 'copy' that not only copies the object but also the objects it refers to and the objects they refer to and so on..

eg:

```
>>>from copy import deepcopy
>>>R3 = deepcopy(R2)
>>>R3 is R2
False
>>>R3.corner is R2.corner
False
```

Exercises

I. Simulate a simple calculator. Define a class 'Calci' having two attributes 'x' and 'y'. The class has a constructor for initializing 'x' and 'y'. Let the program perform the operations: add, sub, mul, pow and divmod on the two attributes (x and y) of the Calci class.

II. Write a program that identifies the age of a person. Define a class 'Person' with attributes name and date of birth. Use constructor to initialize the attributes of 'Person' object. Identify the object's 'age' according to the current date and display the object details appropriately.

[Hint: Use module datetime. Functions to be considered are 'datetime.date()' and 'datetime.date.today()']

III. Rewrite the 'Person' class so that the Person's age is calculated for the first time when a new Person object is created, and recalculated (when it is requested) if the date has changed since the last time that it was calculated.

Classes and Functions

Consider the Time class...

```
>>>class Time():
. . .       ''' Represents time of the day
. . .       attributes hour, minute and second '''
. . .
```

We create an object T1 and initialize the attributes..

```
>>>T1 = Time()
>>>T1.hour = 11
>>>T1.minute = 25
>>>T1.second = 15
```

Exercise
Write a function that takes the Time object as input argument and prints the time in the form of hour:minute:second.
[Use '%.2d' format specifier that prints an integer using at least 2 digits, adds leading zeros if necessary.]

Pure Functions and Modifiers

Two types of functions/methods can be defined for a class:
(1) Pure functions and (2) Modifiers

Pure functions are those which do not modify any objects passed as arguments.

eg:
Consider the 'Time' class, let's add 2 objects of this class.

```
def add_time(T1, T2):
    T3 = Time()
    T3.hour = T1.hour + T2.hour
    T3.minute = T1.minute + T2.minute
    T3.second = T1.second + T2.second
    if T3.second >= 60:
        T3.second  -= 60
          T3.minute += 1
     if T3.minute >= 60:
         T3.minute -= 60
        T3.hour += 1
    if T3.hour >= 24:
        T3.hour -= 24
    return T3
```

Modifiers
Functions/methods that modify the objects that are passed as arguments are called modifiers.

eg: To increment the second's value of Time object..

```
def incr(t1):
    t1.second += 1

    if t1.second >= 60:
        t1.second -= 60
        t1.minute += 1
    if t1.minute >= 60:
        t1.minute -= 60
        t1.hour += 1
    if t1.hour >= 24:
        t1.hour -= 24
    return t1
```

Pure functions Vs Modifiers

- Some programming languages allow only pure functions
- Evidences show pure functions are faster to develop and less prone to errors
- Modifiers are convenient at times, typical for functional programming style

Classes and methods

A method is a function that is associated to a particular class.

Defining a class method:

Syntax:
1) Outside class definition

```
class Class_name():
      ''' Documentation about
      this class...'''

def method_name(obj):
          statements
```

2) Inside class definition

```
a) class classname():
       def method_name(self):
           statements
```

OR

```
b) class classname():
       def method_name(self, other):
           statements
```

Calling the method:

Syntax:

For 1 and 2(a) methods with single object as argument:

1. Class_name.method_name(obj)
2. obj.method_name()

eg:

```
>>>Time.incr(t1)
```

OR

```
>>>t1.incr()
```

For 2(b) method with 2 objects..

obj2.method_name(obj1)

eg:

```
>>>t3 = t2.add_time(t1)
```

OR

>>>t3 = t1.add_time(t2)

The __str__ method

It gives the string representation of an object. The return value must be a 'str' object. Usually used with a print statement.

eg:
```
class Point():
     def_init_(self,v1,v2):
          self.x = v1
          self.y = v2
    def_str_(self):
          return '(%2.2g , %2.2g)' % (self.x, self.y)


p1 = Point(2.5, 3.5)
print(p1)
```

Operator Overloading

The methods defining the operator to be overloaded need to be defined 'inside' the class definition only!!

Usage:
operation(self, other)

Python permits the following operators to be overloaded
*+ , -, *, //, %, **, <<, >>, &, ^, |*

```
__add__(self, other)
__sub__(self, other)
_mul_(self, other)
_floordiv_(self, other)
_mod_(self, other)
_divmod_(self, other)
_pow_(self, other)
```

__lshift__(self, other)
__rshift__(self, other)
__and_(self, other)
__xor_(self, other)
__or_(self, other)

and the relational operators:

<, <=, ==, !=, >=, >

__lt__(self, other)
__le__(self, other)
__eq__(self, other)
__ne__(self, other)
__ge__(self, other)
__gt__(self, other)

In-place modifiers (self is modified)
[Also known as Incrementing arithmetic delimiters]

*+= -= *= /= //= %= **= <<= >>= &= |= ^=*
__iadd__(self, other)
__isub__(self, other)
__imul_(self, other)
__idiv_(self, other)
__itruediv_(self, other)
__ifloordiv_(self, other)
__imod__(self, other)
__ipow__(self, other)
__ilshift_(self, other)
__irshift_(self, other)
__iand_(self, other)
__ixor_(self, other)
__ior_(self, other)

egs...

Consider the Sample class having 1 attribute 'x' of int data type.

```
class Sample():
    def_init_(self):
        self.x = 0
    def_init_(self, value):
        self.x = value
    def_str_(self):
        return ('Value of x = %s' %self.x)
    def_add_(self, other):
        return self.x + other.x
```

#you can add as many methods depending on the required number of operators to be overloaded

Let's create some objects to the Sample class,

```
>>>s1 = Sample()
>>>s2 = Sample(10)
>>>s3 = Sample(4)

>>>print(s1)
Value of x = 0
>>>print(s2)
Value of x = 10
>>>print(s3)
Value of x = 4
>>>s2 + s3
14
```

Exercises:

I. Write a program to compare two numbers and find the square root of the larger using operator overloading. Define a class 'Test' having an integer-value attribute. Define two overloading functions to solve the problem.

II. Write a program to find the distance between two points (Euclidean distance). Define a class 'Point' with the coordinates (x, y) as attributes. Define appropriate methods for Point class (initializing, printing, distance calculation).

Destructors

Syntax:

```
def_del_(self):
    # message indicating object demise
```

eg:

```
class Life():
     def_init_(self, name='unkown'):
          print('Hello ' + name)
          self.name = name
     def live(self):
          print(self.name)
     def_del_(self):
          print('Goodbye ' + self.name)


>>>A1 = Life('Alice')
Hello Alice

>>>A1.live()
Alice

>>>A1 = 'Bob'
Goodbye Alice
```

Composition, Inheritance & Polymorphism

Inheritance - the 'is-a' relationship between the subclass and its superclass (eg: A duck 'is-a' bird)

```
class SuperClass():
     #data fields
    # instance methods

class SubClass(SuperClass):
     #data fields
    # instance methods
```

Exercise:
Define classes 'Shape', 'Square' and 'Rectangle'. Define a method to print the area and perimeter of 'Shape'. Define a method to find the area and perimeter for 'Square' and 'Rectangle'. Use the concept of inheritance to display area and perimeter for a given geometric shape.

Polymorphism -
Python has a loose implementation of polymorphism; this means that it applies the same operation to different objects, regardless of their class.

Let's consider 3 classes with the same initializer...

```
class Quote():
    def_init_(self, person, words):
        self.person = person
        self.words = words
    def who(self):
        return self.person
    def says(self):
        return self.words + '.'

class QuestionQuote(Quote):
    def says(self):
        return self.words + '?'

class ExclamationQuote(Quote):
    def says(self):
        return self.words + '!'

q1 = Quote('Alice', 'So, this is the wonderland')
print(q1.who(), 'says: ' , q1.says())

q2 = QuestionQuote('Grumpy', 'Who ate my Supper')
print(q2.who(), 'says: ' , q2.says())

q3 = ExclamationQuote('Wimpy kid', 'ZOO-WEE MAMA')
print(q3.who(), 'says: ' , q3.says())
```

Note:
(1) Since the *init ()* is not overridden in the subclasses, Python automatically calls the *init ()* method of the parent class Quote to store the instance variables person and words.

(2) Three versions of the *says()* method provides different behaviour for the three classes. This is traditional polymorphism in OO languages. Python goes a little further and lets you run the *who()* and *says()* methods of any objects that have them.

Let's define a class called *Blah* that has no relation to *Quote* class:

```
class Blah():
    def who(self):
        return 'Blab'
    def says(self):
        return 'This morning... I woke up, at night'
```

The *who()* and *says()* methods of various objects of completely unrelated classes can be run using...

```
def who_says(obj):
    print(obj.who(), 'says ' , obj.says())

b1 = Blah()

who_says(q1)
who_says(q2)
who_says(q3)
who_says(b1)
```

Exercise:
Create the following classes: ‘Employee’, ‘Faculty’, ‘Technical’ and ‘Administrative’. Find the salary of a given employee using inheritance and polymorphism.
[Salary = Basic + TA + DA + HRA – PF – IT]

Composition & Aggregation - 'has-a' relation

Inheritance is a good technique to use when you want a child class to act like its parent class most of the time. It’s tempting to build elaborate inheritance hierarchies, but sometimes 'composition' or 'aggregation' make more sense.

eg:

```
class Salary():
   def_init_(self,pay):
       self.pay=pay
```

```
    def get_total(self):
        return self.pay*12

class Employee():
    def_init_(self, pay,bonus):
        self.pay = pay
        self.bonus = bonus
        self.salary = Salary(self.pay)

    def annual_salary(self):
        return 'Total " + str(self.salary.get_total()+self.bonus)

E1 = Employee(100,10)
print(E1.annual_salary())
```

Note:
In Composition, one of the classes are composed of one or more instances of other classes. In other words, one class is container and other class is content and if you delete the container object then all of its contents objects are also deleted.
In above eg, class *Employee* is *'container'* and class *Salary* is *'content'*.

Aggregation
(A weak form of Composition!)

eg:

```
class Salary():
    def_init_(self,pay):
        self.pay= pay
    def get_total(self):
        return self.pay*12

class Employee():
    def_init_(self,pay,bonus):
        self.pay = pay
        self.bonus = bonus

    def annual_salary(self):
        return "Total " +str(self.pay.get_total()+self.bonus)
```

```
S1 = Salary(100)
E1 = Employee(S1,10)
print(E1.annual_salary())
```

Note:
If you delete the container object contents, contents objects can live without container object.

eg(2). A duck 'is-a' bird, but it 'has-a' tail.
A tail is not a kind of duck, but part of a duck.

```
class Bill():
     # Bill means Beak
    def init (self, description):
          self.description = description

class Tail():
     def init (self,length):
           self.length = length

class Duck():
     def init (self, bill,tail):
           self.bill = bill
           self.tail = tail
    def about(self):
         print('This duck has a ', bill.description, 'bill
and a ', tail.length, 'tail')

tail = Tail('long')
bill = Bill('wide orange')
duck = Duck(bill, tail)
duck.about()
```

Output:

```
This duck has a wide orange bill and a long tail
```

Defining scope of attributes

Inherently all the attributes have public scope, but can be changed to have private scope by using '_' before the attribute name...

eg: if 'name' is an attribute of Person class,

```
class Person():
    def_init_(self, name):
        self._name = name
```

End of Unit 6!!

Loop else:

Consider this example:

```
l1 = [1, 3, 5, 7]
pos = 0
while pos < len(l1):
    if l1[pos]%2 == 0:
        print('Found an even number! ')
        break
    pos+=1
else:
  print('No even number found! ')
```

Output:

?