Working with Python Date & Time

By Nimesh Kumar Dagur

Working with Date & Time

- A Python program can handle date and time in several ways.
- Converting between date formats is a common task for computers.
- Python's **datetime** and **time** modules help track dates and times.

Date and time: datetime

The datetime module provides us with objects which we can use to store information about dates and times:

- **datetime.date** is used to create dates which are not associated with a time.
- **datetime.time** is used for times which are independent of a date.
- **datetime.datetime** is used for objects which have both a date and a time.
- **datetime.timedelta** objects store *differences* between dates or datetimes – if we subtract one datetime from another, the result will be a timedelta.
- We can query these objects for a particular component (like the year, month, hour or minute), perform arithmetic on them, and extract printable string versions from them if we need to display them.

Date Formats: strftime

Examples are based on datetime.datetime(2013, 9, 3, 9, 6, 5)

Code	Meaning	Example
%a	Weekday as locale's abbreviated name.	Tue
%A	Weekday as locale's full name.	Tuesday
%w	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.	2
%d	Day of the month as a zero-padded decimal number.	03
%b	Month as locale's abbreviated name.	Sep
%B	Month as locale's full name.	September
%m	Month as a zero-padded decimal number.	09
%y	Year without century as a zero-padded decimal number.	13
%Y	Year with century as a decimal number.	2013

Date and time: datetime Example

```
import datetime
# this class method creates a datetime object with the current date and time
now = datetime.datetime.today()
print(now.year)
print(now.hour)
print(now.minute)
print(now.strftime("%a, %d %B %Y"))
long_ago = datetime.datetime(2015, 10, 12, 12, 27, 58)
print(long_ago) # remember that this calls str automatically
print(long_ago < now)
difference = now - long_ago
print(type(difference))
print(difference) # remember that this calls str automatically
```

```
2015
12
48
Tue, 13 October 2015
2015-10-12 12:27:58
True
<class 'datetime.timedelta'>
1 day, 0:20:42.958732
```

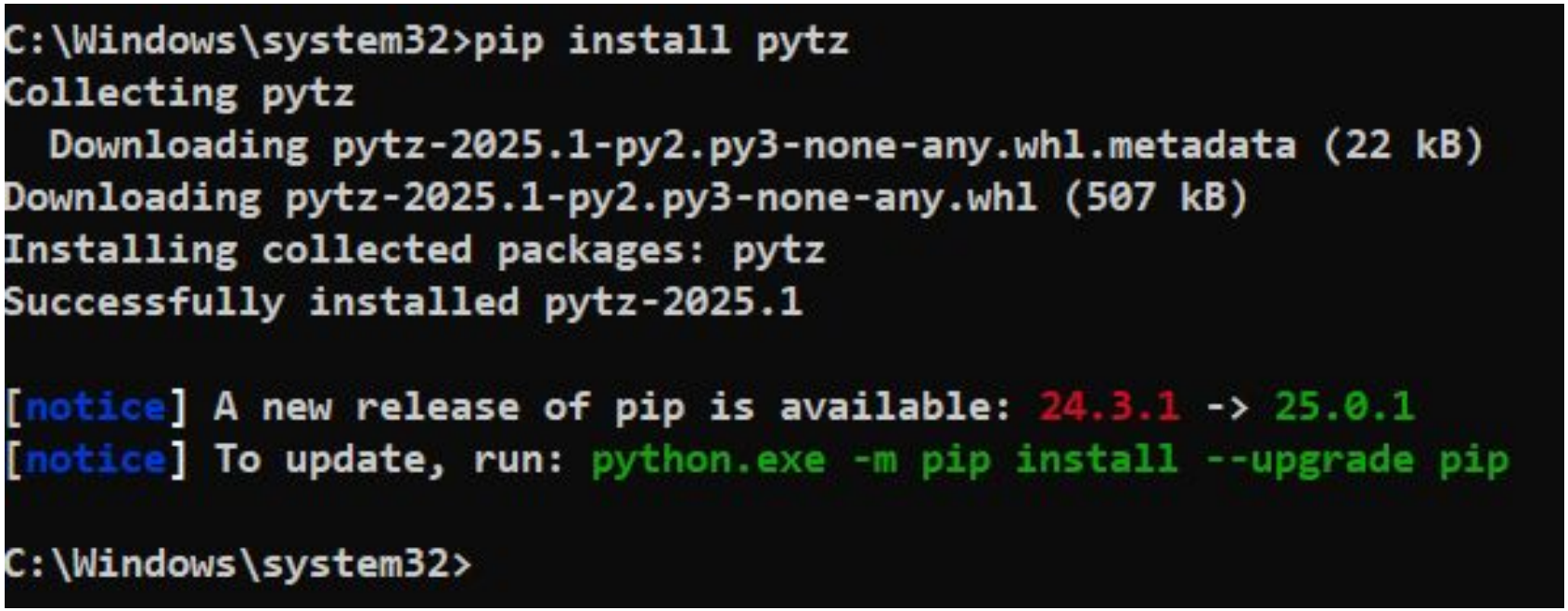
```
from datetime import datetime,timedelta
now = datetime.now()
print "Today: ", now
print "Yesterday: ", now - timedelta(days=1)
print "Day before Yesterday: ", now - timedelta(days=2)

print "Tomorrow: ", now + timedelta(days=1)
print "Day after Tomorrow: ", now + timedelta(days=2)


print "1 week ago: ", now - timedelta(weeks=1)
print "1 week from now: ", now + timedelta(weeks=1)

# Calculating a 15 day trial period
trial_started = datetime.now()
trial_ends = trial_started + timedelta(days=14)
print "Started Trial on ", trial_started
print "Trial Ends  ", trial_ends
print "Trial Expires in  ", trial_ends - now
```

```
Today:  2016-03-31 09:26:23.708000
Yesterday:  2016-03-30 09:26:23.708000
Day before Yesterday:  2016-03-29 09:26:23.708000
Tomorrow:  2016-04-01 09:26:23.708000
Day after Tomorrow:  2016-04-02 09:26:23.708000
1 week ago:  2016-03-24 09:26:23.708000
1 week from now:  2016-04-07 09:26:23.708000
Started Trial on  2016-03-31 09:26:23.755000
Trial Ends   2016-04-14 09:26:23.755000
Trial Expires in   14 days, 0:00:00.047000
```

Date Comparison in python

```
from datetime import datetime
date1 = datetime(2015,03,04)
date2 = datetime(2015,03,03)
date3 = datetime(2015,03,05)
date4 = datetime(2015,03,03)

print "Date 1: ",date1
print "Date 2: ",date2
print "Date 3: ",date3
print "Date 4: ",date4

if date1 < date3:
    print "date1 is greater than date3"

if date1 > date2:
    print "date1 is greater than date2"

if date2 == date4:
    print "date2 and date4 are equal"
```

```
Date 1:  2015-03-04 00:00:00
Date 2:  2015-03-03 00:00:00
Date 3:  2015-03-05 00:00:00
Date 4:  2015-03-03 00:00:00
date1 is greater than date3
date1 is greater than date2
date2 and date4 are equal
```


C:\Windows\system32>pip install pytz
Collecting pytz
Downloading pytz-2025.1-py2.py3-none-any.whl.metadata (22 kB)
Downloading pytz-2025.1-py2.py3-none-any.whl (507 kB)
Installing collected packages: pytz
Successfully installed pytz-2025.1
[notice] A new release of pip is available: 24.3.1 -> 25.0.1
[notice] To update, run: python.exe -m pip install --upgrade pip
C:\Windows\system32>

time module

- There is a **time** module available in Python which provides functions for working with times, and for converting between representations.
- The function *time.time()* returns the current system time in ticks since 12:00am, January 1, 1970.

time module

```
>>> import time
>>> t= time.time()
>>> print t
1459402163.73
>>> currenttime=time.localtime(t)
>>> print currenttime
time.struct_time(tm_year=2016, tm_mon=3, tm_mday=31, tm_
hour=10, tm_min=59, tm_sec=23, tm_wday=3, tm_yday=91, tm
_isdst=0)
>>> print time.strftime("%b %d %Y %H:%M:%S",currenttime)
Mar 31 2016 10:59:23
>>> t=(2016,4,1,10,59,59,4,92,0)
>>> print time.strftime("%b %d %Y %H:%M:%S",t)
Apr 01 2016 10:59:59
```

Python Namespace and Scope

- *If you have ever read 'The Zen of Python' (type "import this" in Python interpreter), the last line states,* ***Namespaces are one honking great idea -- let's do more of those!*** *So what are these mysterious namespaces? Let us first look at what name is.*
- **Name** (also called identifier) is simply a name given to objects.
- Everything in Python is an object.
- Name is a way to access the underlying object.
- **Example:**

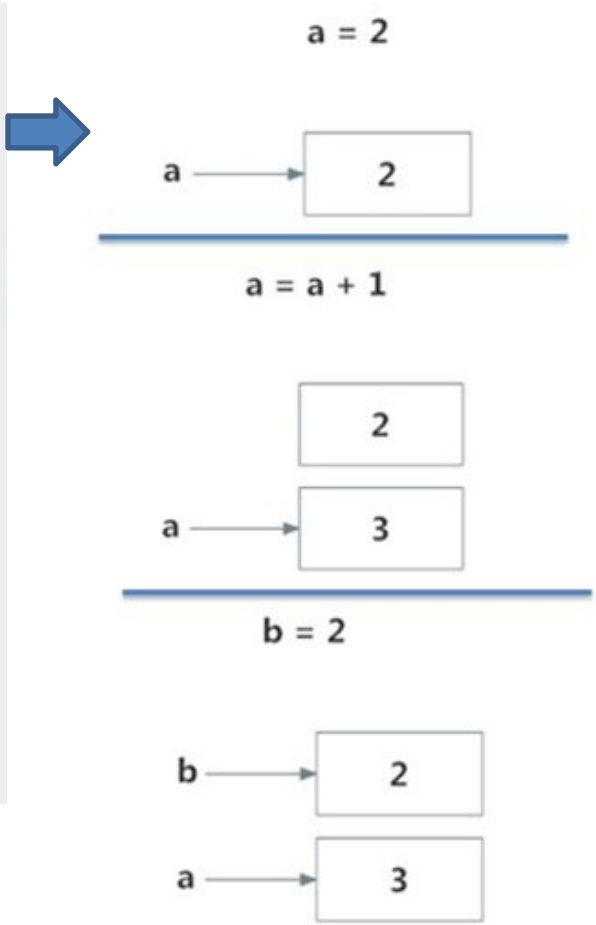
  a = 2,

  here 2 is an object stored in memory and *a* is the name we associate it with. We can get the address (in RAM) of some object through the built-in function, id().

Python Name

```
>>> a = 2
>>> id(2)
507098816
>>> id(a)
507098816
```

```
>>> a = 2
>>> id(a)
507098816

>>> a = a+1
>>> id(a)
507098848
>>> id(3)
507098848

>>> b = 2
>>> id(b)
507098816
```



Initially, an object 2 is created and the name a is associated with it, when we do a = a+1, a new object 3 is created and now a associates with this object. Note that id(a) and id(3) have same values. Furthermore, when we do b = 2, the new name b gets associated with the previous object 2.

- This is efficient as Python doesn't have to create a new duplicate object.
- This dynamic nature of name binding makes Python powerful; a name could refer to any type of object.

```
>>> a = 5
>>> a = 'Hello World!'
>>> a = [1,2,3]
```

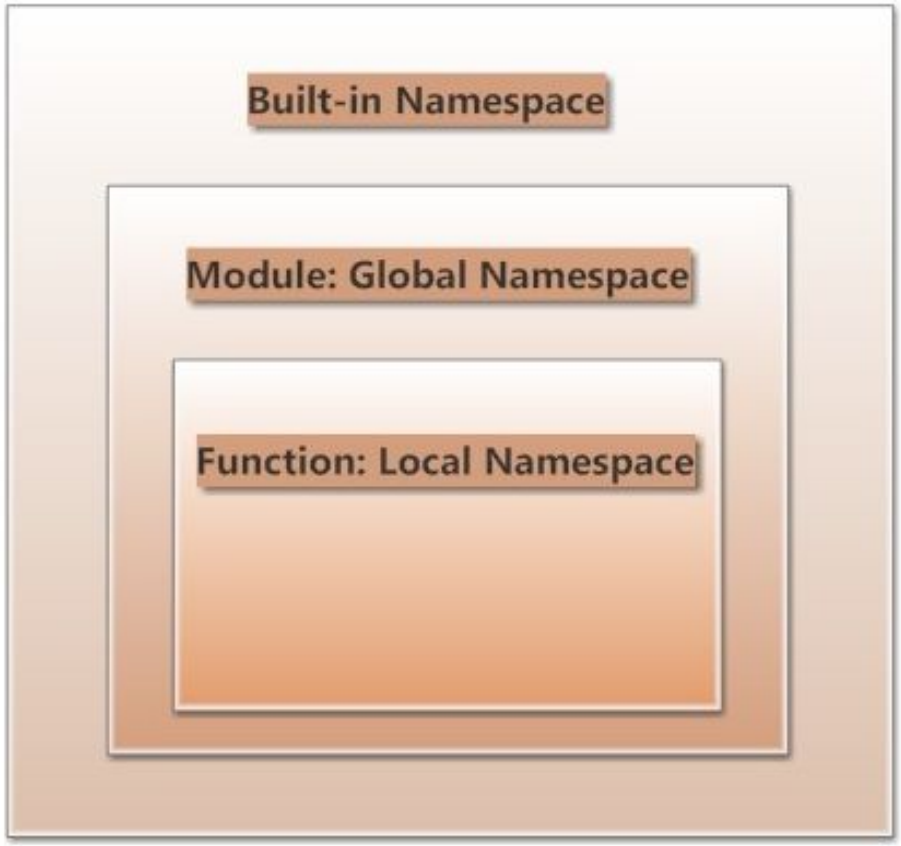
- All these are valid and *a* will refer to three different types of object at different instances.
- Functions are objects too, so a name can refer to them as well.
- Our same name *a* can refer to a function and we can call the function through it, pretty neat.

```
>>> def func():
...         print("Hello")
...
>>> a = func
>>> a()
Hello
```

Namespace

- Namespace is a collection of names.
- In Python, you can imagine a namespace as a mapping of every name, you have defined, to corresponding objects.
- Different namespaces can co-exist at a given time but are completely isolated.
- A namespace containing all the built-in names is created when we start the Python interpreter and exists as long we don't exit.
- This is the reason that built-in functions like id(), print() etc. are always available to us from any part of the program.

- Each module creates its own global namespace.
- Since, these different namespaces are isolated, same name that may exist in different modules do not collide.
- Modules can have various functions and classes.
- A local namespace is created when a functions is called, which has all the names defined in it.
- Similar, is the case with class



Python Scope

- Although there are various unique namespaces defined, we may not be able to access all of them from every part of the program.
- The concept of scope comes into play.
- Scope is the portion of the program from where a namespace can be accessed directly without any prefix.
- At any given moment, there are at least three nested scopes.

1. Scope of the current function which has local names
2. Scope of the module which has global names
3. Outermost scope which has built-in names

- When a reference is made inside a function, the name is searched in the local namespace, then in the global namespace and finally in the built-in namespace.
- If there is a function inside another function, a new scope is nested inside the local scope.

Example of Scope and Namespace in Python

```
def outer_function():
    b = 20
    def inner_func():
        c = 30


a = 10
```

Here, the variable a is in the global namespace. Variable b is in the local namespace of outer_function() and c is in the nested local namespace of inner_function().

When we are in inner_function(), c is local to us, b is nonlocal and a is global. We can read as well as assign new values to c but can only read b and c from inner_function().

If we try to assign as a value to b, a new variable b is created in the local namespace which is different than the nonlocal b. Same thing happens when we assign a value to a.

Example of Scope and Namespace in Python

```
def outer_function():
    a = 20
    def inner_function():
        a = 30
        print('a =',a)

    inner_function()
    print('a =',a)

a = 10
outer_function()
print('a =',a)
```

a=

a=

a=

a=

a=

a=

- If we declare *a* as global, all the reference and assignment go to the global *a*.
- In this program, three different variables *a* are defined in separate namespaces and accessed accordingly.

```
def outer_function():
    global a
    a = 20
    def inner_function():
        global a
        a = 30
        print('a =',a)

    inner_function()
    print('a =',a)

a = 10
outer_function()
print('a =',a)
```

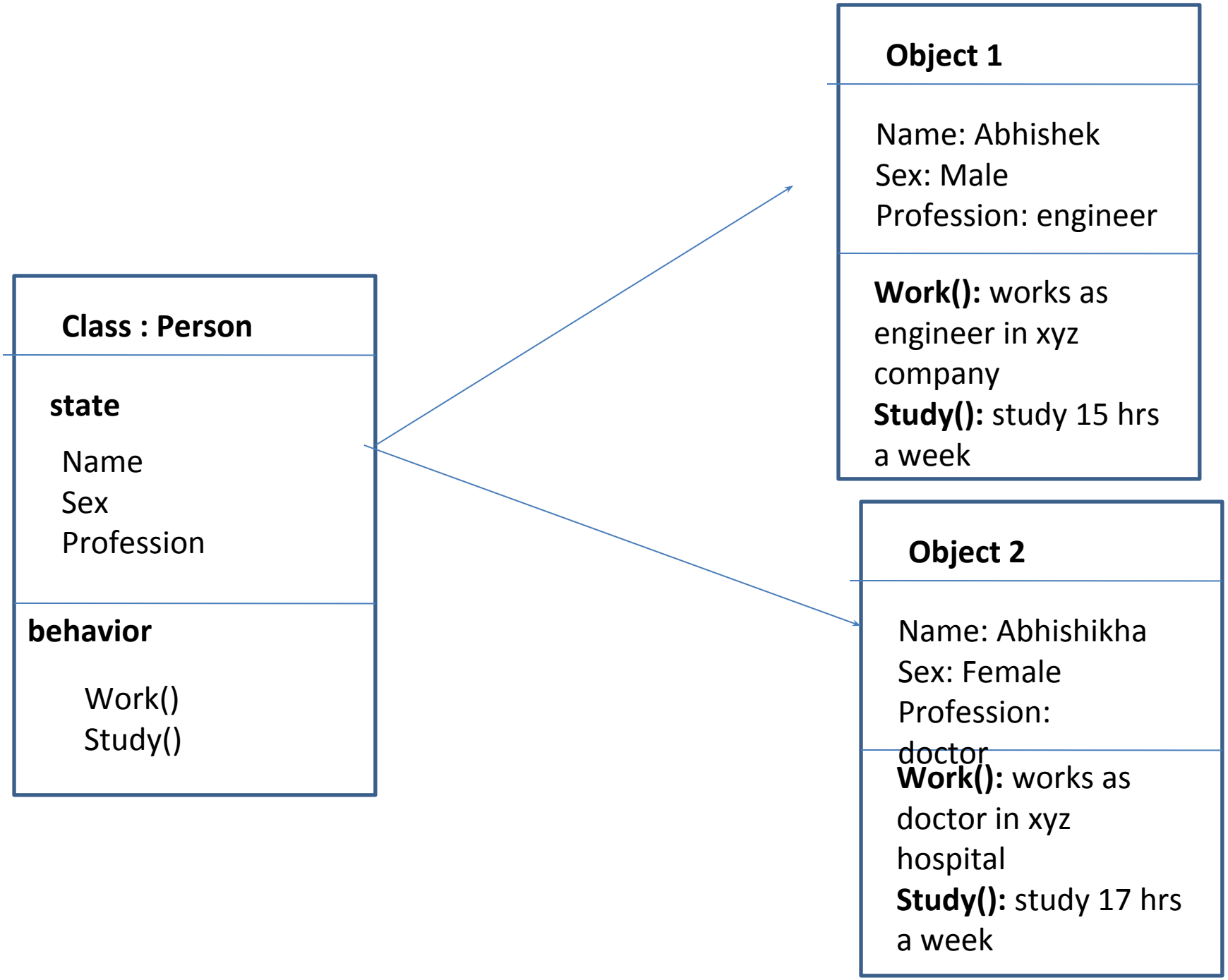
Here, all reference and assignment are to the global a due to the use of keyword global.

Python Objects and Class

- By Nimesh Kumar Dagur,CDAC Noida, India

Python Objects and Class

- Python is an object oriented programming language.
- Unlike procedure oriented programming, in which the main emphasis is on functions, object oriented programming stress on objects.
- Object is simply a collection of data (variables) and methods (functions) that act on those data.
- Class is a blueprint for the object.
- We can think of class like a sketch (prototype) of a house.
- It contains all the details about the floors, doors, windows etc.
- Based on these descriptions we build the house.
- House is the object.
- As, many houses can be made from a description, we can create many objects from a class.
- An object is also called an instance of a class and the process of creating this object is called instantiation


Class : Person
state
Name
Sex
Profession
behavior
Work()
Study()
Object 1
Name: Abhishek
Sex: Male
Profession: engineer
Work(): works as engineer in xyz company
Study(): study 15 hrs a week
Object 2
Name: Abhishikha
Sex: Female
Profession: doctor
Work(): works as doctor in xyz hospital
Study(): study 17 hrs a week

Defining a Class in Python

- Like function definitions begin with the keyword def, in Python, we define a class using the keyword class.
- The first string is called docstring and has a brief description about the class.
- Although not mandatory, this is recommended.
- **Class definition:**

```
class MyNewClass:
    '''This is a docstring. I have created a new class'''
    pass
```

- A class creates a new local namespace where all its attributes are defines.
- Attributes may be data or functions.
- There are also special attributes in it that begins with double underscores (__).
- **For example**, __doc__ gives us the docstring of that class.
- As soon as we define a class, a new class object is created with the same name.
- This class object allows us to access the different attributes as well as to instantiate new objects of that class.

```
>>> class MyClass:
...       "This is my second class"
...       a = 10
...       def func(self):
...            print('Hello')
...
>>> MyClass.a
10

>>> MyClass.func
<function MyClass.func at 0x0000000003079BF8>

>>> MyClass.__doc__
'This is my second class'
```

Creating an Object in Python

- Class object could be used to access different attributes.
- It can also be used to create new object instances (instantiation) of that class.
- The procedure to create an object is similar to a function call.

```
>>> ob = MyClass()
```

- This will create a new instance object named *ob*.
- We can access attributes of objects using the object name prefix.
- Attributes may be data or method.
- Method of an object are corresponding functions of that class.
- Any function object that is a class attribute defines a method for objects of that class.
- This means to say, since MyClass.func is a function, ob.func will be a method object.

```
>>> ob = MyClass()
>>> MyClass.func
<function MyClass.func at 0x0000000000335B0D0>
>>> ob.func
<bound method MyClass.func of <__main__.MyClass object at 0x000000000332DEF0>>

>>> ob.func() ⟹ MyClass.func(ob)
Hello
```

- You may have notices the *self* parameter in function definition inside the class.
- But we called the method simply as ob.func() without any arguments. It still worked.
- This is because, whenever an object calls its method, the object itself is pass as the first argument.
- So, ob.func() translates into MyClass.func(ob).
- In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument.
- For these reasons, the first argument of the function in class must be the object itself.
- This is conventionally called *self*.
- It can be named otherwise but we highly recommend to follow the convention.

```
>>> class className:
        def createName(self,name):
                self.name=name
        def displayName(self):
                return self.name
        def saying(self):
                print("hello %s" % self.name)

>>> second=className()
>>> f1=className()
>>> f1.createName('Nimesh')
>>> second.createName('CDAC')
>>> f1.displayName()
'Nimesh'
>>> f1.saying()
hello Nimesh
>>> second.saying()
hello CDAC
```

Constructors in Python

- Class functions that begins with double underscore (__) are called special functions as they have special meaning.
- Of one particular interest is the **__init__()** function.
- This special function gets called whenever a new object of that class is instantiated.
- This type of function is also called constructors in Object Oriented Programming (OOP).
- We normally use it to initialize all the variables.

Constructors in Python

```
class ComplexNumber:
      def __init__(self,r = 0,i = 0):
            self.real = r
            self.imag = i

      def getData(self):
            print("{0}+{1}j".format(self.real,self.imag))
```

- In the above example, we define a new class to represent complex numbers.
- It has two functions, **__init__()** to initialize the variables (defaults to zero) and **getData()** to display the number properly.

```
>>> c1 = ComplexNumber(2,3)
>>> c1.getData()
2+3j

>>> c2 = ComplexNumber(5)
>>> c2.attr = 10
>>> (c2.real, c2.imag, c2.attr)
(5, 0, 10)
>>> c1.attr
Traceback (most recent call last):
...
AttributeError: 'ComplexNumber' object has no attribute 'attr'
```

- An interesting thing to note in the above step is that attributes of an object can be created on the fly.
- We created a new attribute *attr* for object *c2* and we read it as well.
- But this did not create that attribute for object *c1*

Deleting Attributes and Objects

- Any attribute of an object can be deleted anytime, using the del statement.

```
>>> c1 = ComplexNumber(2,3)
>>> del c1.imag
>>> c1.getData()
Traceback (most recent call last):
...
AttributeError: 'ComplexNumber' object has no attribute 'imag'

>>> del ComplexNumber.getData
>>> c1.getData()
Traceback (most recent call last):
...
AttributeError: 'ComplexNumber' object has no attribute 'getData'
```

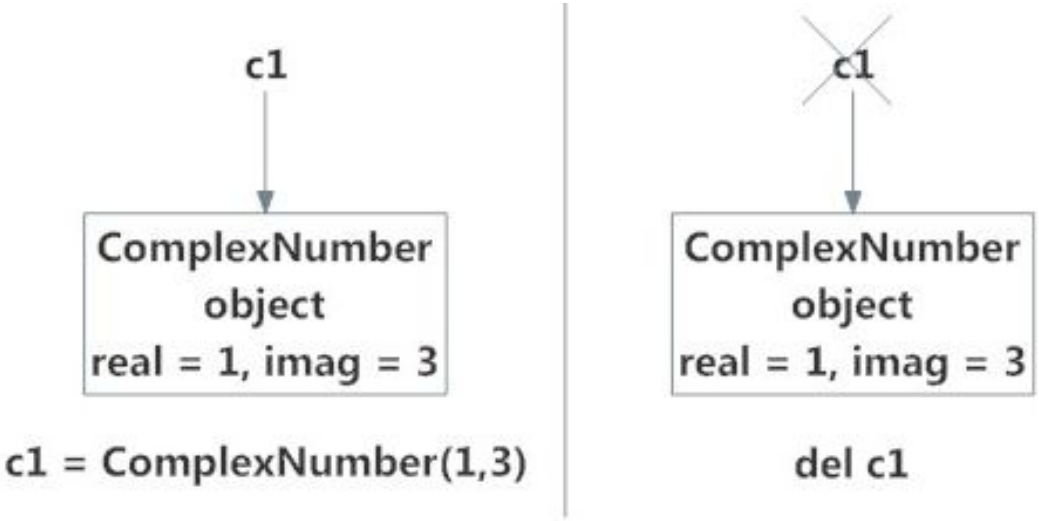
Deleting Attributes and Objects

- We can even delete the object itself, using the del statement

```
>>> c1 = ComplexNumber(1,3)
>>> del c1
>>> c1
Traceback (most recent call last):
...
NameError: name 'c1' is not defined
```

Deleting Attributes and Objects

- When we do c1 = ComplexNumber(1,3), a new instance object is created in memory and the name *c1* binds with it.
- On the command del c1, this binding is removed and the name *c1* is deleted from the corresponding namespace.
- The object however continues to exist in memory and if no other name is bound to it, it is later automatically destroyed.
- This automatic destruction of unreferenced objects in Python is also called **garbage collection**.



Example

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary
```

- The variable ***empCount*** is a class variable whose value is shared among all instances of a this class.
- The first method ***__init__()*** is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
- You declare other class methods like normal functions with the exception that the first argument to each method is *self*.
- Python adds the *self* argument to the list for you; you do not need to include it when you call the methods.

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary

"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

```
Name :  Zara ,Salary:  2000
Name :  Manni ,Salary:  5000
Total Employee 2
```

- You can add, remove, or modify attributes of classes and objects at any time.

```
emp1.age = 7  # Add an 'age' attribute.
emp1.age = 8  # Modify 'age' attribute.
del emp1.age  # Delete 'age' attribute.
```

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary

print "Employee.__doc__:", Employee.__doc__
print "Employee.__name__:", Employee.__name__
print "Employee.__module__:", Employee.__module__
print "Employee.__bases__:", Employee.__bases__
print "Employee.__dict__:", Employee.__dict__
```

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: ()
Employee.__dict__: {'__module__': '__main__', 'displayCount':
<function displayCount at 0xb7c84994>, 'empCount': 2,
'displayEmployee': <function displayEmployee at 0xb7c8441c>,
'__doc__': 'Common base class for all employees',
'__init__': <function __init__ at 0xb7c846bc>}
```