Introduction to Python scripting language

Sachin Deodhar

01/07/18

Content confidentiality and copyrights

- The training material and contents of this couse are for internal KPIT training purpose
- Do not share this document with anyone outside of KPIT
- No part of this document can be copied without explicit permission from KPIT

Course structure

The course is structured is such that it contains sessions as well as practice excersises

Session 1

- Introduction and Python basics

Session 2

- Python Constructs like loops, decision making and data types like numbers and string
- Advanced objects like lists, tuples and dictonaries

Session 3

- Functions, modules, File IO and exceptions

Session 4

- Advanced topics: Classes/Objects
- Advanced topics: Networking

Session 5

- Advanced topics: Multi-threading
- Advanced topics: Reg-expressions

Excersise

- Each session will have some practice excercises which attendee can try later.

Session 4: Classes/Objects, Networking

Duration: 1.5 hours

01/07/18

Classes in Python

A class is a user defined prototype for an object. It defines a set of attributes that characterizes the object and provides some methods or function to access those attributes.

The class statement creates a new class definition. The name of the class immediately follows the keyword class followed by a colon.

Eg:

```
class ClassName:
   'Optional class documentation string'
   class_suite
```

If the class has the optional documentation string, then it is available via __doc__ variable and can be accesses by ClassName.__doc__

The class_suite consists of all the component statements defining class members, data attributes and functions.

Classes in Python

Creation of Object:
The creation of object is essentially divided into two parts

1. Object Creation
- Object creation is controlled by a static class method with the name __new__.
- To create an object of the class Example, the __new__ method of this class is called.
- Python defines this function for every class by default, although user can do that explicitly too.
  NOTE: In most cases there is no need to implement the __new__ method.

2. Object Initialization
- Object initialisation is done by __init__() method.
- It provides a way for the user to initialize the attributes of the object.

Classes in Python

Python Constructors:

1. In a python class, the first method __init__() is a special method. This methods corresponds to the constructor. Python calls this method internally when a new instance of this class is created.

2. When a class is created without a constructor, python will automatically create a default constuctor. This will be an empty definition so will do nothing.

3. It is recommended that every class should have the implementation of the __init__() method so the class members can be initialized.

Python Destructor:

1. In a python class, the destructor mentod is defined as a special method __del__()

2. Python has an internal garbage collection, so freeing up memory resources is not a big concern. However this method can be used to cleanly release other resources, such as closing network connections, or closing files etc.

3. If not defined, the python will internally define it, but it will not do anything.

Classes in Python

Following is an example of a simple class in Python.

Eg:

```
class Example:
   def __init__(self):
      print "Constructor called"

   # destructor
   def __del__(self):
       print "Destructor called"

# creating an object
myObj = Example()
# to delete the object explicitly
del myObj
```

Output:

```
Constructor called
Destructor called
```


Classes in Python

Following is an example of a class definition in Python.

Eg:

```
class Employee:
   'This is a common prototype to contain employee details.'
   id = 0
   name = “”
   salary = 0

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

   def getEmployeeId(self):
      return self.id

   def setEmployeeSalary(self, salary):
      Self.salary = salary
```


Inheritance in Python

Inheritance is an important aspect of object oriented programming. Python supports inheritence. Python classes can inherit from other classes. A class can inherit attributes and behaviour methods from another class.

The syntax for inheritence is: class childClass(parentClass)
Here, the childClass inherts the parentClass.

Eg:

```
class ParentClass:
    # body of ParentClass
    # method1
    # method2

class ChildClass(ParentClass):
    # body of ChildClass
    # method 1
    # method 2
```


Inheritance in Python

Example of Inheritence:

```
class Person:
    def __init__(self, first, last):
        self.firstname = first
        self.lastname = last

    def Name(self):
        return self.firstname + " " + self.lastname

class Employee(Person):
    def __init__(self, first, last, staffnum):
        Person.__init__(self,first, last)
        self.staffnumber = staffnum

    def GetEmployee(self):
        return self.Name() + ", " +  self.staffnumber

x = Person("Marge", "Simpson")
y = Employee("Homer", "Simpson", "1007")
print(x.Name())
print(y.GetEmployee())
```


Multiple Level Inheritance in Python

Python allows us to create a multi level class inheritance hierarchy. Following is an example

Eg:

```
class ParentClass_A:
    # body of ParentClass_A
   def parent():
      print “Parent class”

class ChildClass_A(ParentClass_A):
    # body of ChildClass_A
   def childA():
      print “Child class A”

class ChildClass_B(ChildClass_A):
    # body of ParentClass_C
   def childB():
      print “Child class B”

B = ChildClass_B
B.childA()
B.parent()
```


Multiple Inheritance in Python

Python allows us to derive a class from several classes at once, this is known as Multiple Inheritance. Its general format is:

Eg:

```
Class ParentClass_A:
    # body of ParentClass_A

Class ParentClass_B:
    # body of ParentClass_B

Class ParentClass_C:
    # body of ParentClass_C

Class ChildClass(ParentClass_A, ParentClass_B, ParentClass_C):
    # body of ChildClass
```


Multiple Inheritance in Python

Diamond inheritance problem if both inherited class have the method defined.

Eg:

```
class A:
    def m(self):
        print("m of A called")

class B(A):
    def m(self):
        print("m of B called")

class C(A):
    def m(self):
        print("m of C called")

class D(B,C):
    pass
```

Output:

```
m of B called
```


Multiple Inheritance in Python

Diamond inheritance problem if super and one base class has the method defined.

Eg:

```
class A:
    def m(self):
        print("m of A called")

class B(A):
    pass

class C(A):
    def m(self):
        print("m of C called")

class D(B,C):
    pass
```

Output python 2.7:

```
m of B called
```

Output python 2.7:

```
m of A called
```


Multiple Inheritance in Python

To ensure the super class method or the required inherited class method is called, we can explicitly call the method of the required class

Eg:

```
class A:
    def m(self):
        print("m of A called")
class B(A):
    def m(self):
        print("m of B called")
class C(A):
    def m(self):
        print("m of C called")
class D(B,C):
    def m(self):
        print("m of D called")

>>> x = D()
>>> B.m(x)
m of B called
>>> A.m(x)
m of A called
```


Polymorphism in Python

Python does **not** support the method of performing late binding using virtual tables. It has a different method of performing the attribute or method lookup.

Given an object, following are the steps which are followed to perform the attribute lookup:

1. Given the name of the attribute, and a reference to the instance, it looks into the instance dictionary. If the attribute is found, then it ends the lookup.

2. Otherwise, it goes to the class, and look for the attribute name in the class. If the attribute is found, and it is an instance method, then it is called.

3. Otherwise, it searches search in the base classes, repeating steps 2 and 3 until the attribute is found.

4. If it is not found, raise the AttributeError exception.

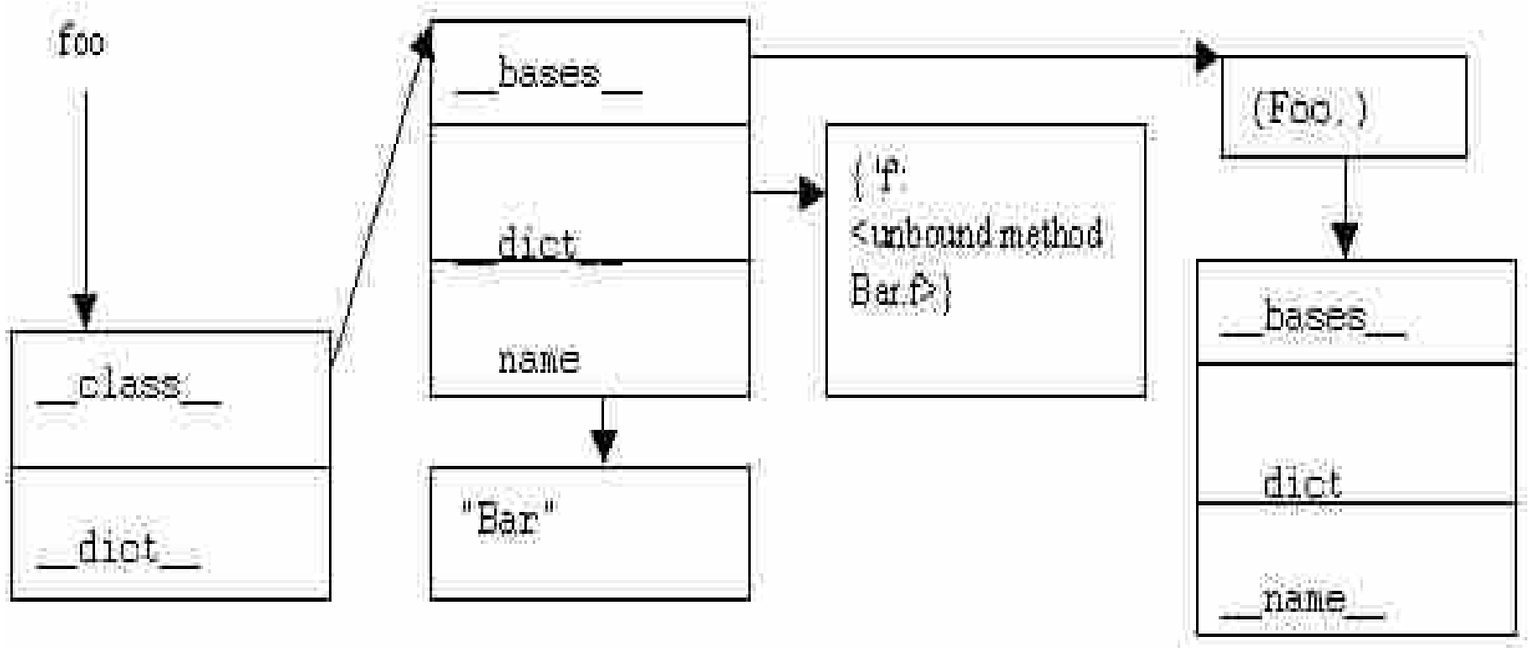
Polymorphism in Python

Following diagram explains the method or attribute lookup in python.

Example:

```
class Foo:
  def f(self):
    print 42
  def g(self):
    print -42

class Bar(Foo):
  def f(self):
    print "Hello, World!"
foo = Bar()
foo.f()
```



Networking in Python

Python provides two levels of access to network services. At a low level, you can access the basic socket support in the underlying operating system, which allows you to implement clients and servers for both connection-oriented and connectionless protocols.

Python also has libraries that provide higher-level access to specific application-level network protocols, such as FTP, HTTP, and so on.

Sockets in Python
Python includes a socket module, which can be used to create socket endpoints for network as well as unix sockets.

Following is the syntax for using the socket module:

```
s = socket.socket (socket_family, socket_type, protocol=0)
```

Where:

- `socket_family` – This is either AF_UNIX or AF_INET, as explained earlier.
- `socket_type` – This is either SOCK_STREAM or SOCK_DGRAM.
- `protocol` – This is usually left out, defaulting to 0.

Networking in Python

Server socket methods:

`s.bind()` - This method binds address (hostname, port number pair) to socket.
`s.listen()` - This method sets up and start TCP listener.
`s.accept()` - This passively accept TCP client connection, waiting until connection arrives (blocking).

Client socket methods:

`s.connect()` - This method actively initiates TCP server connection.

General socket methods:

`s.recv()` - This method receives TCP message
`s.send()` - This method transmits TCP message
`s.recvfrom()` - This method receives UDP message
`s.sendto()` - This method transmits UDP message
`s.close()` - This method closes socket
`s.gethostname()` - Returns the hostname.

Networking in Python

Example of a simple socket server in python:
A socket object is created using socket() method, bind() method is used to bind it with a specific port. The accept() method will wait for a connection from client. Once a connection is accepted, the send() method is used to send a data packet following which the connection is closed.

Eg:

```
import socket                   # Import socket module

s = socket.socket()             # Create a socket object
host = socket.gethostname() # Get local machine name
port = 12345                    # Reserve a port for your service.
s.bind((host, port))            # Bind to the port

s.listen(5)                     # Now wait for client connection.
while True:
   c, addr = s.accept()     # Establish connection with client.
   print 'Got connection from', addr
   c.send('Sending data packet to client')
   c.close()                # Close the connection
```


Networking in Python

Example of a simple socket client in python:
A socket object is created using socket() method, the connect() method is used to open a connection with the server. Once connection is established, the clinet calls the recv() method to receive data packet from the server. The connection is closed by calling the close() method.

Eg:

```
import socket                   # Import socket module

s = socket.socket()             # Create a socket object
host = socket.gethostname() # Get local machine name
port = 12345                    # Reserve a port for your service.

s.connect((host, port))
print s.recv(1024)
s.close                         # Close the socket when done
```


Networking in Python

Networking modules:
Python includes several modules which can be used in the scripts to access several different networking protocols.

Following is a list of some of these modules and the protocols they support.

Protocol	**Common function**	**Port No**	**Python module**
HTTP	Web pages	80	httplib, urllib, xmlrpclib
NNTP	Usenet news	119	nntplib
FTP	File transfers	20	ftplib, urllib
SMTP	Sending email	25	smtplib
POP3	Fetching email	110	poplib
IMAP4	Fetching email	143	imaplib
Telnet	Command lines	23	telnetlib

Thank you

01/07/18