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 3: Functions, modules, file I/O and exceptions

Duration: 2 hours

01/07/18

Functions in Python

Functions is essentially a reuable block of code, used to perform a specific or related actions. Python provides a lot of built-in functions as well as allows user-defined functions.

How to define a function in Python:

1. A function block should begin with **def** followed by the function name and parentheses ()
2. All if any input parameters or function aruments should be placed within these parentheses.
3. The first statement of a fucntion can be a comment/documentation statement for the function
4. The colon (:) marks the begining of a function code block. The code block needs to follow the Indentation
5. The return statement exits a function block. It can also return some parameter to the caller.
6. The return statement with no arguement will be same as return None statement

Functions in Python: Syntax and usage

The basic syntax of a python function is as follows:

```
def myfunction( myparams ):
    “function description (this is optional but recommended”
    <code statements>
    return None
```

Following is an example of a simple function and how it is called:

```
>>> def myprintfunc( str ):
...     "This function prints the given string after adding its own prefix"
...     print "LOG: " + str
...     return None
...
>>> myprintfunc("Hello python")
LOG: Hello python
>>>
```


Functions in Python: Passsing parameters

In python the variables are like tags to a memory location. So when a variable is passed to a function, it will actually refer to the same memory location. So if any change is made to the variable within the function then its value in the calling function will also change. In case of non mutable datatypes the value in the calling function does not change. Following is an example;

```
def myfunc(mynum, mylist):
    mynum = mynum + 1
    mylist.append("New item")
    print "In myfunc [{}], [{}]".format(mynum, mylist)
    return None

mynum = 10
mylist = [1, 2, 3, 4, 5]
print "Initial value {}, {}".format(mynum, mylist)
myfunc(mynum, mylist)
print "Final value {}, {}".format(mynum, mylist)
```

Output:

```
Initial value 10, [1, 2, 3, 4, 5]
In myfunc 11, [1, 2, 3, 4, 5, 'New item']
Final value 10, [1, 2, 3, 4, 5, 'New item']
```


Functions in Python: Type of arguments

Python allows multiple ways in which arguments can be passed to a function

1. Required argument: This is a standard way of calling a function. Here it is expected that the arguments are passed in the same order in which the fuction expects it.
   Eg:

```
def myfunc(mystring, myint):
    print “String: {}, Int: {}”.format(mystring, myint)
    return None

myfunc(“Hello”, 10)
```

2. Keyword based argument: When keyword is used along with the argument, it allows the arguments to be position independent.
   Eg:

```
def myfunc(mystring, myint):
     print "String: {}, Int: {}".format(mystring, myint)
     return None

myfunc(myint=10, mystring="Test position")
```


Functions in Python: Type of arguments

Python allows multiple ways in which arguments can be passed to a function

3. Default argument: If the funtion is defined in such a way that it mentions a default argument, then This argument is no longer mandatory. Function call can be made with out sending that argument. Eg:

```
def myfunc(mystring, myint=0):
    print “String: {}, Int: {}”.format(mystring, myint)
    return None

myfunc(“Hello”)
```

4. Variable length argument: This mentod allows arbitary number of arguments to be passed to the function. The variable which holds the values of all the nonkeyword variable agruments begins With (*). Eg:

```
def myfuncvars(mystring, *varis):
    print "Keyword argument is {}".format(mystring)
    print "Number of non keywork vars is {}".format(len(varis))
    for arg in varis:
       print arg

myfuncvars("Hello", 1, 2, 99.6, "Test", ['a','b'])
```


Functions in Python: Return statement

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Example:

```
def sum( arg1, arg2 ):
    total = arg1 + arg2
    return total;

total = sum( 10, 20 );
print "The Sum is: {}".format(total)
```


Functions in Python: Anonymous function or lambda function

These functions are called anonymous since they are not declared with a standard def keyword. Following are some properties of such functions.

1. They begin with a keyword lambda
2. lambda requires on expression and its syntax is “lambda arguments : expression”
3. lambda function can accept any number of arguments
4. It cannot access variables other than those in its parameter list and those which are global
5. Mostly lambda functions are passed as parameters to a function which expects a function objects as parameter like map

Example:

```
dict_a = [{'name': 'python', 'points': 10}, {'name': 'java', 'points': 8}]

map(lambda x : x['name'], dict_a) # Output: ['python', 'java']

map(lambda x : x['points']*10,  dict_a) # Output: [100, 80]

map(lambda x : x['name'] == "python", dict_a) # Output: [True, False]
```


Python Modules:

- The modules allow programmer to logically organize the python code. With this way all related code can be gouped into a single module making it easier to understand and use.

- A module is really just a file containing Python code. It can define functions, classes and variables.

- Usually the python code for a module named 'my_module' will resides in a file named my_module.py.

- Within a module, the module’s name (as a string) is available as the value of the global variable __name__

Example:

A python file contaning add, sub, mul and div operations names as arithmatic.py will be a module

arithmatic.my_add, arithmatic.my_sub, arithmatic.my_mul, arithmatic.my_div
will be the functions in that module

Python Modules: Importing modules

There are several ways of improting the functionality of the modules into the script.

- Either the entire module can be imported
  The syntax to import the entire module is “import module1[, module2[,... moduleN]”
  In this case when the interpreter encounters the import statement, it imports the modules if it is present in its search path.

  A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.

  Eg: `import arithmatic`

- Only specific functions can be imported
  The syntax to import only specific attributes is “from modname import name1[, name2[, ... nameN]]”

  Eg: `from arithmatic import my_add`

  This statement does not import the entire arithmatic module into the current namespace; it just introduces the item my_add from the module arithmatic into the global symbol table of the importing module.

Python Modules: Importing modules

There is another way of importing all the attributes into the current name space. The syntax for this is from modname import *

This provides an easy way to import all the items from a module into the current namespace; however, this statement is not recommended since it could cause conflict with names in the existing script since all the symbols from the module are direcly imported in the same namespace.

Locating Modules:

When importing a module, the Python interpreter searches for the module in the following sequences −

1. The current directory.

2. If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.

3. If all else fails, Python checks the default path.
   On ubuntu, this default path is usually /usr/local/lib/python/.

Python Modules: Namespace and scoping

- A Python statement can access variables in a local namespace and in the global namespace.
  If a local and a global variable have the same name, the local variable shadows the global variable.

- Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local. Hence to assign a value to a global variable within a function, one must use the global statement.

  Eg:

```
index = 1
def increment_index():
    # Uncomment the following line to fix the code:
    # global index
    index = index + 1

print index
increment_index()
print index
```


Python Modules: Built-in functions

There are some built-in functions which can give more information about the modules.

- The dir() function

The dir() built-in function returns a sorted list of strings containing the names defined by a module.

Eg:

```
>>> import arithmatic
>>> arith_methods = dir(arithmatic)
>>> print arith_methods
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'my_add',
'my_div', 'my_mul', 'my_sub']
>>>
```

- The globals() and locals() functions

The globals() and locals() functions can be used to return the names in the global and local namespaces depending on the location from where they are called.

- locals() called from within a function, it will return all the names that can be accessed locally from that function
- globals() called from within a function, it will return all the names that can be accessed globally from that function.

The return type of both these functions is dictionary. Therefore, names can be extracted using the keys() function

Python Modules: reloading a module

There are another built-in functions which can be used to reload the module. The functions is **reload(module_name)**

- When the module is imported into a script, the code in the top-level portion of a module is executed only once
- To reexecute the top-level code in a module, we can use the reload() function.

Packages in Python

A package can be considered simply a collection of modules. It simplifies the importing of attributes from multiple different modules. The package folder needs to contain a file __init__.py
This file contains all the import statements from various python files in that package

Eg:
Following is the folder for arithmatic package and content of the __init__.py is

```
arithmatic
|-- arithmatic.py
|-- arith_max.py
|-- arith_min.py
`-- __init__.py
```

```
from arithmatic import my_add
from arithmatic import my_sub
from arithmatic import my_mul
from arithmatic import my_div
from arith_min import my_min
from arith_max import my_max
```


Python – Opening and closing files

Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a file object.

The open() Method:
The open() function is used to open the file. This function creates a file object, which would be utilized to call other support methods associated with it.

Syntax is: file_object = open(file_name [, access_mode][, buffering])

file_name − The file_name argument is a string value that contains the name of the file

access_mode − The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc.

buffering − If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file.
If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size.
If negative, the buffer size is the system default(default behavior).

The close() Method:
The close() method of a file object flushes any unwritten information and closes the file object, after which no more writing can be done. Python automatically closes a file when the reference object of a file is reassigned to another file.

Python – Reading and Writing Files

Python provides file object which has easy methods for reading and writing the files.

The write() Method:
The write() method writes any string to an open file. It is important to note that Python strings can have binary data and not just text. The write() method does not add a newline character ('\n') to the end of the string.
Eg:

```
fd = open("foo.txt", "wb")
fd.write( "Hello World\n New Line\n");
fd.close()
```

The read() Method:
The read() method reads a string from an open file. It is important to note that Python strings can have binary data apart from text data.
Eg:

```
fd = open("foo.txt", "r+")
str = fd.read(10);
print "Read String is : ", str
fd.close()
```


Python – File access modes

Following are available file access modes.

```
r    Opens a file for reading only.
rb   Opens a file for reading only in binary format.
r+   Opens a file for both reading and writing.
rb+  Opens a file for both reading and writing in binary format.
```

The file pointer is placed at the beginning of the file. This is the default mode.

```
w    Opens a file for writing only.
wb   Opens a file for writing only in binary format.
w+   Opens a file for both writing and reading.
wb+  Opens a file for both writing and reading in binary format.
```

In all cases if the file exists, then overwrites the file. If the file does not exist, creates a new file.

```
a    Opens a file for appending.
ab   Opens a file for appending in binary format.
a+   Opens a file for both appending and reading.
ab+  Opens a file for both appending and reading in binary format.
```

The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.

Python – File navigation

Python provides the following file navigation methods:

1. The tell() method:
   The tell() method tells you the current position within the file; in other words, the next read or write \
   will occur at that many bytes from the beginning of the file.
   Example:

```
fd = open("foo.txt", "r+")
position = fd.tell();
print "Current file position : ", position
```

2. The seek(offset[, from]) method:
   The seek(offset[, from]) method changes the current file position. The offset argument indicates the number of bytes to be moved. The from argument specifies the reference position from where the bytes are to be moved.

   If from is set to 0, it means use the beginning of the file as the reference position and 1 means use \
   the current position as the reference position and if it is set to 2 then the end of the file would be taken as the reference position.
   Example:

```
# Reposition pointer at the beginning once again
position = fd.seek(0, 0);
```


Python – Renaming and removing files

Python provides the **os** module which contains several methods that can help renaming and deleting files.

The rename() Method:
The rename() method takes two arguments, the current filename and the new filename. The syntax is: os.rename(current_file_name, new_file_name)

Eg:

```
os.rename( "test1.txt", "test2.txt" )
```

The remove() Method:
The remove() method is used to delete files by supplying the name of the file to be deleted as the argument. Syntax is: os.remove(file_name)

Eg:

```
os.remove("text2.txt")
```


Python – Directories

Python provides the **os** module which contains several methods that can help create, remove and change directories.

Method mkdir(): Syntax: os.mkdir("newdir")
This function will create a new directory with the name specfied in the argument

Method rmdir(): Syntax: os.rmdir("dirname")
This method deletes the directory, which is passed as an argument in the method.
Eg:

```
import os
os.mkdir("test")
os.rmdir( "/tmp/test"  )
```

Method chdir(): Syntax: os.chdir("newdir")
This method can be used to change the working directory. The argument is the name of the directory that we want the current directory

Method getcwd(): Syntax: os.getcwd()
This methods returns the current working directory.

Python – Exception handling

What is an exception?
In python an exception is an event, which is raised when the execution of a program is disrupted from its normal flow. This would essentially be when the python script encounters a condition that it cannot handle. An exception is a Python object that represents an error.

NOTE:
When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Syntax for handling execeptions:
The syntax for handling python execptions is to use the **try....except...else** block

```
try:
   Statements which might raise an exception on some error
   ......................
except Exception1:
   If there is Exception1, then execute this block.
except Exception2:
   If there is Exception2, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```


Python – Exception handling

Some importants tips regarding the implementation of execption handling blocks

- A single try statement can have multiple except statements.
  This is useful when the try block contains statements that may throw different types of exceptions.

- You can also provide a generic except clause, which handles any exception.

- After the except clause(s), you can include an else-clause.
  The code in the else-block executes if the code in the try: block does not raise an exception.

- The else-block is a good place for code that does not need the try: block's protection.

Eg of exception handling

```
try:
   fh = open("testfile", "w")
   fh.write("Writing a new line to the file")
except IOError:
   print "Error: Fauled to open the file"
else:
   print "File successfully written"
   fh.close()
```


Python – Types of except clauses

The except Clause with No Exceptions:

This kind of a try-except statement catches all the exceptions that occur. This will catch all exceptions but will not help much in identifying which one was it. Hence it is not considered a very good programming practice.

```
try:
   You do your operations here;
except:
   If there is any exception, then execute this block.
else:
   If there is no exception then execute this block.
```

The except Clause with Multiple Exceptions

Same except statement can be used to handle multiple exceptions. Following is the syntax:

```
try:
   You do your operations here;
except(Exception1[, Exception2[,...ExceptionN]]):
   If there is any exception from the given exception list,
   then execute this block.
else:
   If there is no exception then execute this block.
```


Python – The try .... finally clause

The finally block is a place to put any code that must execute, whether the try-block raised an exception or not.

NOTE: else clause cannot be used along with a finally clause.

Following is an example:

```
try:
   fd = open("file.txt", "w")
   try:
      fd.write("Writing a line to the file")
   finally:
      print "Closing the file"
      fd.close()
except IOError:
   print "Error: Failed to write to the file"
```


Python – Argument of an Exception

An exception can have an argument, which can gives additional information about the issue. The contents of the argument may vary depending on the exception. The exception's argument can be captured by supplying a variable in the except clause.

Following is the syntax

```
try:
   You do your operations here;
except ExceptionType, Argument:
   You can print value of Argument here...
```

Following is the example

```
try:
   fd = open("file.txt", "w")
   try:
      fd.write("Writing a line to the file")
   finally:
      print "Closing the file"
      fd.close()
except IOError, Argument:
   print "Error: Failed to write to the file - ",Argument
```


Python – Raising an exception

Programmer can raise an exception using the raise statement. The syntax for raise statement is:
raise [Exception [, args [, traceback]]]

Here, Exception is the type of exception (for example, NameError) and argument is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.

Following is an example of raising exception:

```
def isPositive(num):
   if num < 0:
      raise Exception(num)
   else:
      return "True"
result = isPositive(-1)
print(result)
```

Output:

```
File "raise.py", line 9, in <module>
  result = isPositive(-1)
File "raise.py", line 3, in isPositive
  raise Exception(num)
Exception: -1
```


Thank you

01/07/18