Smarter education for smarter planet

IBM Innovation Centre for Education

Welcome to:

Introduction to Python Versions of Python Installation

Unit objectives

After completing this unit, you should be able to know:

- How to create isolated environments,
- How to use the environments without conflicting with other Python environments called virtual environments.
- How to execute external code within Python and how to run them in parallel.
- Introduction to File open / close, Opening binary files,
- How to call functions, raise exceptions,
- OOP concepts, Base classes, Derived classes, Inherit classes

File & Directory manipulation

The most common file operation are

- open ()
- close ()
- append ()
- readline ()
- input () and raw_input () for python 2.x users

File functions

File Open : Syntax

- Different modes during opening of the file :
- r - Opens a file for reading only
- w -Opens a file for writing only
- a -Opens a file for appending
- ab+ - Opens a file for both appending and reading in binary format

file_name – string value

Syntax

access_mode –Read,Write,Append

buffer_size_format – flags the file buffering

file object = open(file_name [, access_mode][, buffer_size_format])

File object attributes

Some of the file Object attributes that are supported in the file object are :

- file.closed - This attribute of the file object returns true if file is closed, false otherwise.
- file.mode - This attribute of the file object returns the access mode with which file was opened.
- file.name - This attribute of the file object returns name of the file.

close ( ) method:

- The close() method of a file object closes the file object and no further operations on the file can be made.

- It also flushes any unwritten information before closing the file object, after which no more writing can be done.

- If by default the program closes before that, then Python closes the file automatically and the reference to the file will not be there anymore.

  By tradition It is a good practice to use the close() function at the end of your process or the program.   fileobject.close ()

Sample

Sample example to open() an existing file and close ()

```
#!/usr/bin/python3

# Opens an existing file called helloworld.py, in the
current folder.
fh = open(“helloworld.py”, "r")
print "Name of the file: ", fh.name

# Close the file at the end of it.
fh.close()
```

Example (1of 2)

Example to open() a new file to write and close ()

```
#!/usr/bin/python3

# Opens an existing file called helloworld.py, in the current
folder.
fh = open(“helloworld.py”, "w")
print "Name of the file: ", fh.name

# Close the file at the end of it.
fh.close()
```

Example (2 of 2)

Example to open() a new file in read-write mode and close ()

```
#!/usr/bin/python3

# Opens an existing file called helloworld.py, in the current folder.
# If the file is not present - then this errors out !!!!!

fh = open(“helloworld.py”, "r+")
print "Name of the file: ", fh.name

# Close the file at the end of it.
fh.close()
```

Opening a binary file (1 of 2)

Program to open() a binary file in read and write mode and close ()

```
#!/usr/bin/python3

# Open file
fh = open(“a.out”, "rb+")
print "Name of the file: ", fh.name

# Close the file at the end of it.
fh.close()
```

Opening a binary file (2 of 2)

Program to open() a file to write contents to it. Overwrites the file, and creates a new one if file is existing !

```
#!/usr/bin/python3

# Openfile
fh = open(“helloworld.py”, "w")
print "Name of the file: ", fh.name

# Close the file at the end of it.
fh.close()
```

Programming (1 of 2)

Program to open() a file to read and write contents to it. Overwrites the file, and creates a new one if file is existing

```
#!/usr/bin/python3

# Openfile

fh = open(“helloworld.py”, "w+")
print "Name of the file: ", fh.name

# Close the file at the end of it.
fh.close()
```

Programming (2 of 2)

Program to append to a file to read and write contents to it

```
#!/usr/bin/python3

# Openfile

fh = open(“helloworld.py”, "a")
print "Name of the file: ", fh.name

# Close the file at the end of it.
fh.close()
```

File Attributes

Checking file attributes

- These indicate the various mode the file was opened, name of the file and whether it was closed or not
- The mode is associated with the file handle and can be referenced using the fh.attribute. e.g: fh.name, fh.closed, and fh.mode.
- The attributes are :
- Name -file object returns the file name being operated upon
- Closed -file object returns a Boolean state indicating it was Open or Closed
- Mode- what access mode it was opened in

Example (1 of 2)

Example program to show the different attributes

```
# Open a file to read and write.
fh = open(“learningpython.txt", "wb")
print “File Name: ", fh.name
print “File is Open or Closed status : ", fh.closed
print “File open access mode : ", fh.mode
close fh
```

Example (2 of 2)

write ( ) & read( )

Example : write( )

```
fh =
open("learningpython.txt",
"w+")
fh.write ( "hello world\n")
fh.write ( "hello world\n")
fh.write ( "hello world\n")
fh.close()
```

Example: read ( )

```
# Open a file to read line
by line.
fh = open("foo.txt", "r+")
str = fh.read()
print (“ String from file is :
", str)
#  End
fh.close()
```

read(read_fixed_size), readline( ) , tell()

- read(read_fixed_size)
- fh.read (read_fixed_size) # this lets the number of bytes to be read in each read.
- The line could be a long line- but only the specified bytes in the argument will be read

- readline ( )
- This function reads a line in the file.
- To read line by line - this has to be in a while loop.

- tell ()
- This function tells at what position is the pointer in the file at.

Two Input modes to read data from Keyboard

- Interactive User data comes in the form of keyboard, where user can enter the data to the program and the python program can process it.
- The two input Python libraries are input() and raw_input ().
- This function got replaced with “input()” in Python 3.x

```
#!/usr/bin/python3
x = input ( "Please enter a value
between 1 and 10\n")
print ( x ).    # Revert , so that we know
what was entered.
print ( type (x)).   # This should say it is
of class “str”

if ( x <= 1 or x > 10 ) :
            print ("User entered a value
less than 1 or greater than 10 ")
```

```
Resulting Output :

:$python3 userinput.py
 Please enter a value
between 1 and 10
-10
-10
<class 'str'>
User entered a value
less than 1 or greater
than 10
```

Checkpoint

1. How do you close the file ()

- A.fh = open(“filename.txt”, “r”) ; close ()
- B.fh = open(“filename.txt”, “r”) ; fh.close ().
- C.fh = open(“filename.txt”, “r”) ; close(fh)

- 2. If a certain file does NOT exist in the folder, can user use the “r+ to read and write ?
- True
- False.

- 3. Can user open a binary file using Python’s open() function ?
- True.
- False

- 4. What is the access mode to open an existing file, read-write-binary
- A.rwb
- B.wrb
- C.r+
- D.rb+

Checkpoint

5. User has to read and write the file, Assume that the file does not exist in the first place. what access mode would be used to open and close.

- A.rw
- B.w+.
- C.wr
- D.r+

- 6. User wants to append to an existing file. what access mode flag would the user use ?
- A.arw
- B.awr
- C.a+

- 7. User wants to write “hello world” to an existing external file. Identify the correct format.
- A.fh =open (“learningpython.txt”, r+);   fh.write (“hello world”); fh.close()
- B.fh =open (“learningpython.txt”, w+);   fh.write (hello world); fh.close()
- C.fh =open (“learningpython.txt”, rw);   fh.write (“hello world”); fh.close()

Checkpoint

8. User wants to read one line by line from an external file. Which function would be used ?

- A.readline.
- B.read()
- C.lineread()

9. What does read(10) mean ?

- A.Read from the 10th line in the file
- B.Read 10th character onwards in the file.
- C.Read 10 characters at a time.

10. User wants to know the current location of the file pointer, Which function would he use ?

- A.seek ()
- B.tell ().
- C.write ()

Creating Virtual Environment

- Virtual environment is a private space that developers can create to have their separate space for development.
- The Virtual Environment creates a separate folder with all required executables needed by the Python development environment.
- User can install this using the command: **pip install virtualenv**
- (The reverse of install if 'uninstall' - pip uninstall virtualenv)
- To check if virtualenv package is installed - Enter this on a command prompt or a terminal-**virtualenv — version**

Installation on Mac OS

```
$ sudo pip3 install virtualenv
Collecting virtualenv
  Downloading virtualenv-15.2.0-py2.py3-none-any.whl (2.6MB)
    100% |████████████████████████████████| 2.6MB 429kB/s
Installing collected packages: virtualenv
Successfully installed virtualenv-15.2.0
```

On successful installation - run this command to see the version it was able to download and install.

```
$virtualenv --version
15.2.0
```

To create a virtual environment

Create a separate folder (mkdir pythonprogramcourse)

cd to that folder. (cd pythonprogramcourse)

```
virtualenv my_project
```

This creates a folder called 'my_project'

To view the contents - cd to my_project and view the files/directories
It will contain all the python executables needed for the private my_project

```
$ ls
bin                include              lib                   pip-selfcheck.json
```

cd to bin folder and you will see a separate version of python in it.

If you are not interested in this version of python- you can create a separate virtual environment with the required version of python using the below command.

Virtual environment

```
pip list
```

This will show just the packages in the new environment. From now on, user can install the packages in the environment for use.
This 'pip list' lists the different packages that are available in the current environment.
If you see the packages and not sure if they belong to the current new environment, then check if you have the activate command correctly !

Install new package using the pip command.

pip install <package name>
e.g: pip install cloudant

Install the NoSQL Cloudant Data base.

```
$ pip install cloudant
```

Another means of checking is to see if the VIRTUAL_ENV variable is set.

```
set | grep VIRTUAL
```

will show that the environment is set to the new folder created.

```
VIRTUAL_ENV=/Users/p3/my_project
```

Install Flask in virtual env :

```
pip install flask
```

```
$ pip3 install flask
```

Parallel Processing, os.fork() and os.exec ()

- All programs when executed becomes a process i.e a child process of a parent process
- New processes are created initially by the Operating system using a fork() system call
- The child process will have a different unique id and the child's parent process ID is the parent's id
- The new process will be independent of the parent's attributes like memory
- In Python the function call is called os.fork() and it returns a new ID i.e of the child's process id.
- Following the fork () call is the exec() family of calls, this call loads the new program into the current child process and replaces all the existing attributes to the newly loaded program

Functions

- Functions are building blocks of Programming Language. Makes the code modular, reusable and easier to maintain. Reduces code sizes, and is more elegant.

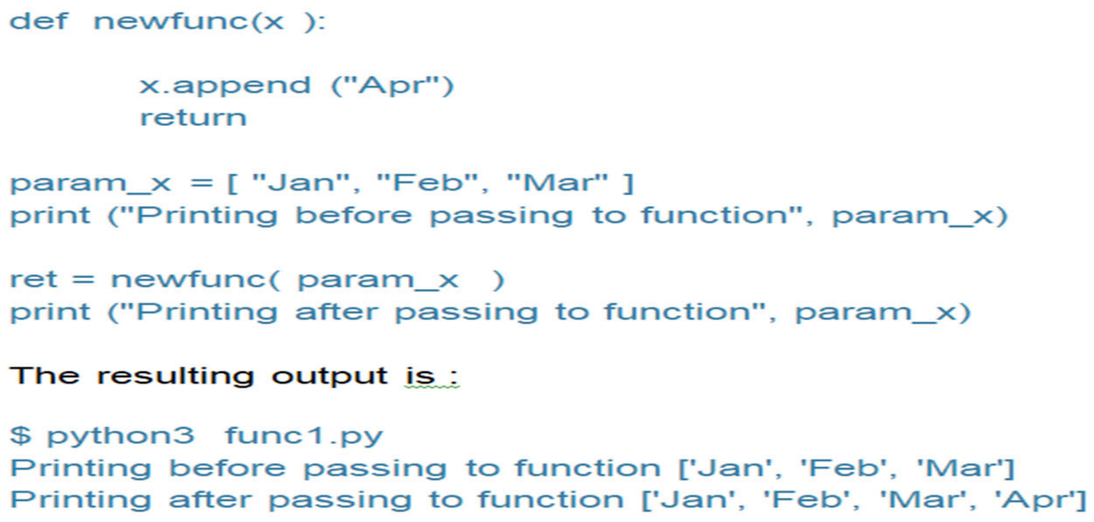
- Functions are defined by the keyword ‘def’ and a function name and parentheses after that. The arguments (if any) can be passed in the parentheses.

- To return from the function, user would insert the “return” statement (if either at the end) or midway in the function depending on the logic, and it can take an optional argument to be returned.

- Functions can return any value or “None”.

Function arguments, pass by value and reference

- The function can pass by value and pass-by-reference
- Let’s look at some examples :
  This is a list and the values are changed in the function and it reflects outside the function as well. (this is pass-by-reference)



```
def  newfunc(x ):

        x.append ("Apr")
        return

param_x  = [ "Jan", "Feb", "Mar" ]
print ("Printing before passing to function", param_x)

ret = newfunc( param_x   )
print ("Printing after passing to function", param_x)
```

The resulting output is :

```
$ python3  func1.py
Printing before passing to function ['Jan', 'Feb', 'Mar']
Printing after passing to function ['Jan', 'Feb', 'Mar', 'Apr']
```

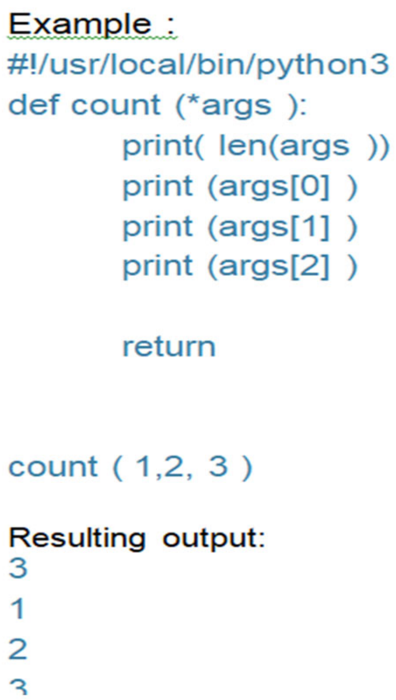
Default arguments

- It is possible to pass in default arguments to function in case the user does not supply the arguments.
- If the user passes the argument in the call, in the right order, the default arguments will be overridden
- Example : No arguments are passed to the function, takes in defaults

Number of arguments to function

- Python offers another feature where the user has capability to pass in variable number of arguments with just 1 argument in the function definition

- *args in the example below defines the incoming arguments as variable (meaning any number)



Example :

```
#!/usr/local/bin/python3
def count (*args ):
        print( len(args ))
        print (args[0] )
        print (args[1] )
        print (args[2] )

        return



count ( 1,2, 3 )
```

Resulting output:

```
3
1
2
3
```

Exception Handling

- Typically when errors happen in the program, it aborts. It might not be a good idea to abort and stop the processing, but rather catch that error and continue with the processing and report the error.
- These are exceptions. and Python (like any other programming language) offers exception handling with some ‘try’, ‘catch’ statements.
- Multiple exceptions can be used to handle a single try statement.

Try Except Code

1. try-except<Exceptionname> - except

```
try :
            code here
Except  ExceptionName1
            code here
            ….
Except  ExceptionName2
            …..
except :
            This is the last except
statement that can be caught -
without mentioning the Exception
type.
```

2. try finally

```
try :
    code here
    …..
    An exception is raised due to
some issue , jumps to finally
section of the code

finally
            code here
```

Types of Exceptions

- There are User Defined Exceptions, as well as System Exceptions.
- User Defined Exceptions can be customised exceptions that the user will develop and raise it when an anomaly arises in the code.

- System Exceptions are ones that are readymade and provided by the Python Programming environment based on different conditions.

- The base exception will always be “Exception”. This is generic in case user does not specifically know the nature of the Exception.

- This does not provide a lot of information to the user, as to the nature of the exception.

- One needs to drill down to different types of Exception to know more about the cause of the exception.

- Reference : https://www.tutorialspoint.com/python/python_exceptions.htm
- Reference : https://www.tutorialspoint.com/python/python_exceptions.htm

Some of the Exception Types

- **EOFError**
- Raised when one of the built-in functions (input() or raw_input()) hits an end-of-file condition (EOF) without reading any data

except

print ("User interrupted error !!")

- **ImportError**
- If python cannot find the module

except ImportError:

print " Your module was not found. Please check and try again! "

Type Error

IBM ICE (Innovation Centre for Education)

Example code showing the TypeError when a float is multiplied by a. string !

TypeError :

This error happens when user tries to call a function, with a different data type that is not declared as the parameter !

```
import sys
dict = {"Sam": "Anderson", "Olivia": "Newton"}

try :
        print (dict["Sam"] )
        print (dict["Olivia"] )
        r = 3.14 * "Hello"  #This raises the TypeError - We can't multiply the float with a string
!

except KeyError:
        print ("KeyError !!  Value d

except SystemExit:
        print (" Sys.exit() function
        print (" Trapped by the exc
```

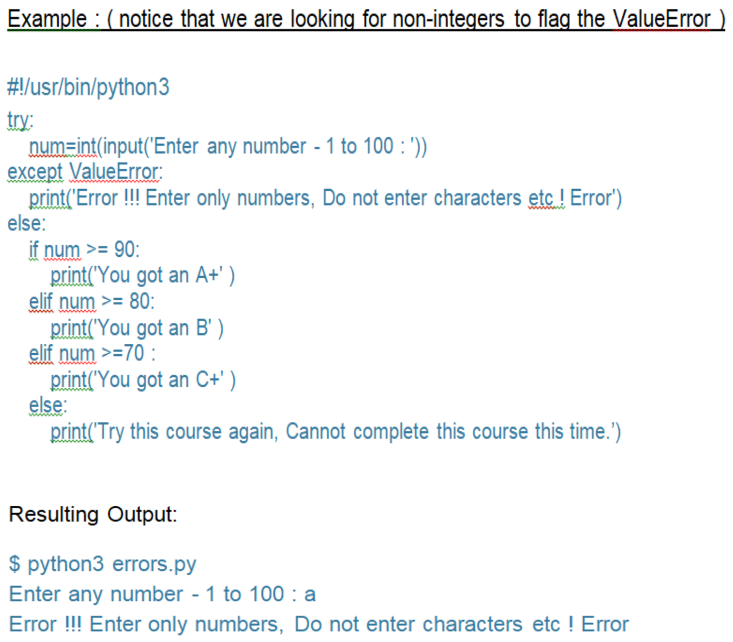
```
except IndentationError:
        print (" Indentation Error encountered ! ");

except TypeError :
        print (" Error Raised due incorrect multiplication attempt ")

else :
        print ( "No error so far !")
```

Value Error

- **ValueError**
- Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value

except

print ("Error during processing ! Non-nu



Example : (notice that we are looking for non-integers to flag the ValueError)

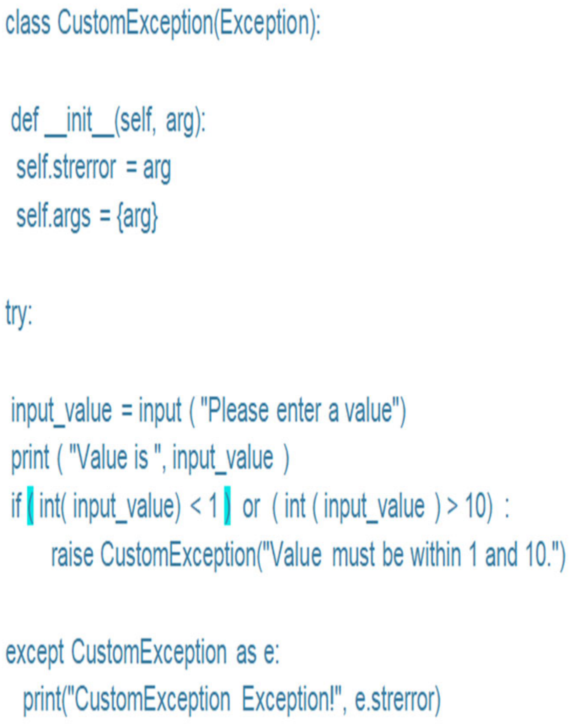
```
#!/usr/bin/python3
try:
   num=int(input('Enter any number - 1 to 100 : '))
except ValueError:
   print('Error !!! Enter only numbers, Do not enter characters etc ! Error')
else:
   if num >= 90:
      print('You got an A+' )
   elif num >= 80:
      print('You got an B' )
   elif num >=70 :
      print('You got an C+' )
   else:
      print('Try this course again, Cannot complete this course this time.')
```

Resulting Output:

```
$ python3 errors.py
Enter any number - 1 to 100 : a
Error !!! Enter only numbers, Do not enter characters etc ! Error
```

Creating custom or User Defined Exceptions

- So far we looked at System exceptions, we can create new custom exceptions depending on the nature of the programming environment and the project.
- To raise an exception, a keyword “raise” must be used.
- The exception must be a separate Class by itself, and must be declared before the exception is called.
- This class has to be inherited from the base “Exception” class.
- Class UserDefinedException (Exception)
- Example of a class that inherits Exception Class, and depending on the input value an exception is raised to the custom exception

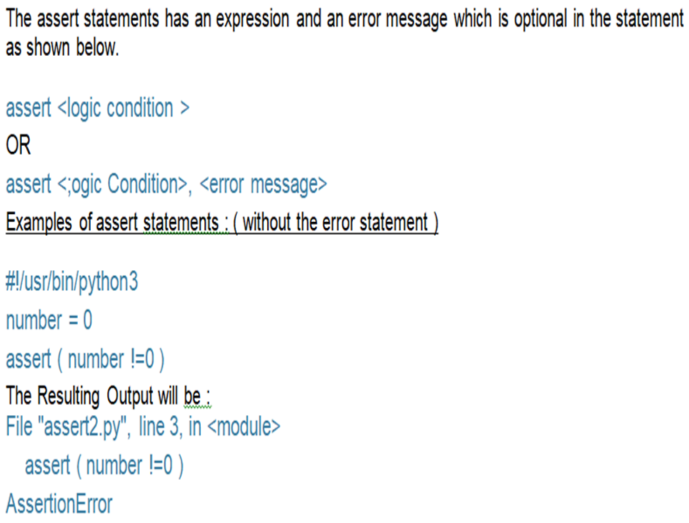


```
class CustomException(Exception):

 def __init__(self, arg):
  self.strerror = arg
  self.args = {arg}

try:

 input_value = input ( "Please enter a value")
 print ( "Value is ", input_value )
 if ( int( input_value) < 1 ) or ( int ( input_value ) > 10) :
     raise CustomException("Value  must be within 1 and 10.")

except CustomException as e:
   print("CustomException  Exception!", e.strerror)
```

Assert statements

- These statements are used to state a fact whether they are true or false.
- They can be used as for debugging and validate certain logic statements. If the developer is confident that the value has to be in a certain datatype, or in a certain range then these statements can be used.

- These are boolean expressions or condition that check if the condition is True or False.
- If the condition is false, then an error or an exception is raised. These exceptions can be interpreted by User Defined Exceptions that was discussed in the earlier chapter.
- Python offers 'assert' keyword or a reserved keyword that can be used to check if the condition is True or False.



The assert statements has an expression and an error message which is optional in the statement as shown below.

```
assert <logic condition >
```

OR

```
assert <;ogic Condition>, <error message>
```

<u>Examples of assert statements : (without the error statement)</u>

```
#!/usr/bin/python3
number = 0
assert ( number !=0 )
```

The Resulting Output will be :

```
File "assert2.py", line 3, in <module>
   assert ( number !=0 )
AssertionError
```

Object oriented Programming

- Class : This is an object that defines the data members like class and instance variables, and methods/functions.

- Method : is the function that is defined in the Class.

- Instance variable : This variable is only visible within the method, and belongs to this class only.

- Inheritance : Classes can be inherited from another class, Here the characteristics of the base class are inherited by the child class.

Object oriented Programming Contd

- The reference to __init__ in a Python class means that it is Class constructor and will be the first one to be called when a Class instance is created.
- Class definition in Python can appear anywhere in the code, but prefer it to be at the top of the code.
- Classes need a mandatory __init__ method in it- this is the constructor in the class.
- The __init__ is the initialiser method and gets called anytime the class is initialized.

Class in Python can be created by

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

Example Python Class for a Library Book

```
#!/usr/bin/python
class LibraryBook:
  'Common base class for all books'
  BooksCount = 0

  def __init__(self, name, pages):
    self.name = name
    self.pages = pages
    LibraryBook.BooksCount += 1

  def displayBookName(self):
    print ("Name : ", self.name,  ", Pages: ", self.pages )
```

This prints the results as :

```
Name : MobyDick , Pages: 300
Name : The Adventure of Tom Sawyer , Pages: 500
Total Books 2
```

```
"This would create first object of LibraryBook class"
book1 = LibraryBook("MobyDick", 300)
"This would create second object of Library Book Class"
book2 = LibraryBook("The Adventure of Tom Sawyer", 500)

book1.displayBookName()
book2.displayBookName()
print ("Total Books %d" % LibraryBook.BooksCount)
```

Example 1

The Python class some built in attributes that can used without declaring it.
We can use the print statement on these to check what they are storing.
e.g:
print ("Classname.__dict__", Classname.__dict__)

__dict__ – This stores the Dictionary and it reports all the functions and namespaces of the Class

For the Class that we defined above the dictionary flag returns,

```
LibraryBook.__dict__:
{'__module__': '__main__', '__doc__': 'Common base class for all books', 'BooksCount': 2,
'__init__': <function LibraryBook.__init__ at 0x104591ae8>, 'displayBookName': <function
LibraryBook.displayBookName at 0x104591b70>, '__dict__': <attribute '__dict__' of 'LibraryBook'
objects>, '__weakref__': <attribute '__weakref__' of 'LibraryBook' objects>}
```

__doc__ – Reports the Class documentation string.

This flag returns - the documentation string - and if there is none , then nothing gets reported. In the above example - the string "Common base class for all books" was set in the class

```
LibraryBook.__doc__:
 Common base class for all books
```

Example 2

IBM ICE (Innovation Centre for Education)

```
Common base class for all books
```

Change the doc string (at the beginning of the class) to something else and print it- to see that it reflects the change. The doc string is the one with the single quote within the class.

__name__ – This returns the name of the Class. In the case of the example above it will be “LibraryBook”

```
LibraryBook.__name__:
 LibraryBook
```

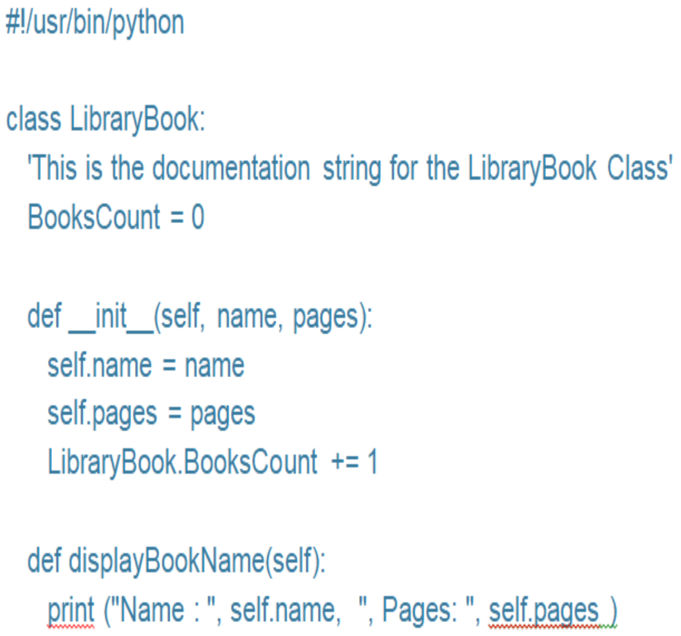
__module__ – Reports the module name in which this class is defined. In the example above since there is no other module defined - it will return the __main__ stating that it in the main module.

```
LibraryBook.__module__:
 __main__
```

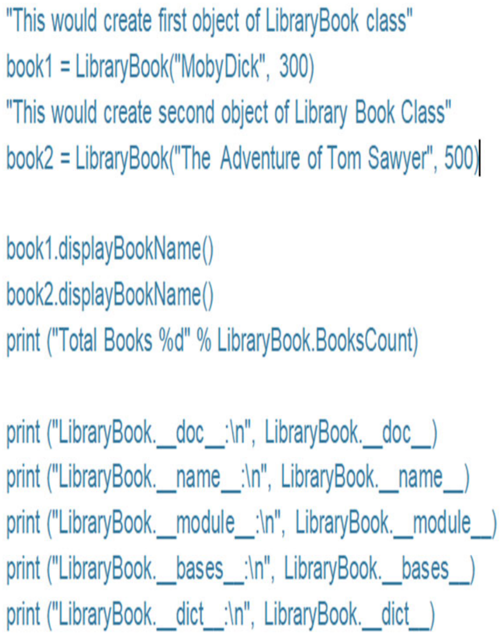
Sample class Example 1

IBM ICE (Innovation Centre for Education)

- Modify the class to print the attributes, as shown in the sample class.



```
#!/usr/bin/python

class LibraryBook:
  'This is the documentation string for the LibraryBook Class'
  BooksCount = 0

  def __init__(self, name, pages):
    self.name = name
    self.pages = pages
    LibraryBook.BooksCount += 1

  def displayBookName(self):
    print ("Name : ", self.name, ", Pages: ", self.pages )
```



```
"This would create first object of LibraryBook class"
book1 = LibraryBook("MobyDick", 300)
"This would create second object of Library Book Class"
book2 = LibraryBook("The Adventure of Tom Sawyer", 500)

book1.displayBookName()
book2.displayBookName()
print ("Total Books %d" % LibraryBook.BooksCount)

print ("LibraryBook.__doc__:\n", LibraryBook.__doc__)
print ("LibraryBook.__name__:\n", LibraryBook.__name__)
print ("LibraryBook.__module__:\n", LibraryBook.__module__)
print ("LibraryBook.__bases__:\n", LibraryBook.__bases__)
print ("LibraryBook.__dict__:\n", LibraryBook.__dict__)
```

Sample class Example 2

```
$ python3 LibraryBook.py
Name :  MobyDick , Pages:  300
Name :  The Adventure of Tom Sawyer , Pages:  500
Total Books 2
LibraryBook.__doc__:
 This is the documentation  string for the LibraryBook  Class
LibraryBook.__name__:
 LibraryBook
LibraryBook.__module__:
 __main__
```

```
LibraryBook.__bases__:
 (<class 'object'>,)
LibraryBook.__dict__:
 {'__module__': '__main__', '__doc__': 'This is the documentation  string for the
LibraryBook Class', 'BooksCount': 2, '__init__': <function LibraryBook.__init__ at
0x103891ae8>, 'displayBookName': <function LibraryBook.displayBookName at
0x103891b70>, '__dict__': <attribute '__dict__' of 'LibraryBook' objects>, '__weakref__':
<attribute '__weakref__' of 'LibraryBook' objects>}
```

Instantiating & Inheriting classes

- Instantiating classes :
- This construct below - instantiates the LibraryBook class and returns a reference.The reference can be used for accessing methods, data etc.
- book1 = LibraryBook("MobyDick", 300)
- **Inheriting classes :**
- User can either develop a new class , or inherit it from another existing class.
- By inheriting the class, user can make use of the base class environment, like methods, instance variables etc.
- New features and functionality can be added into the inherited class.
- The format for the derived class is
- **Class DerivedClassName (BaseClass Name) :**

Example Program 1

Example program of a Derived class (BookSubject) from a Base Class (BookBase)

```
#!/usr/bin/python3
class BookBase:

    def __init__(self, name, pages):
        self.bookname = name
        self.bookpages = pages

    def Name(self):
        return self.bookname + " " + str( self.bookpages)

    def Pages(self):
        return str( self.bookpages)


   # End of Base Class
```

```
# Start of Derived Class from Base Class.
# This derived class has another attribute called “Subject”
class BookSubject(BookBase):

    def __init__(self, name, pages, subject):
        BookBase.__init__(self,name, pages)
        self.subject = subject

    def GetBookName(self):
        return self.Name() + ", " + self.subject

    def GetSubject(self):
          return self.subject


# End of Derived Class
```

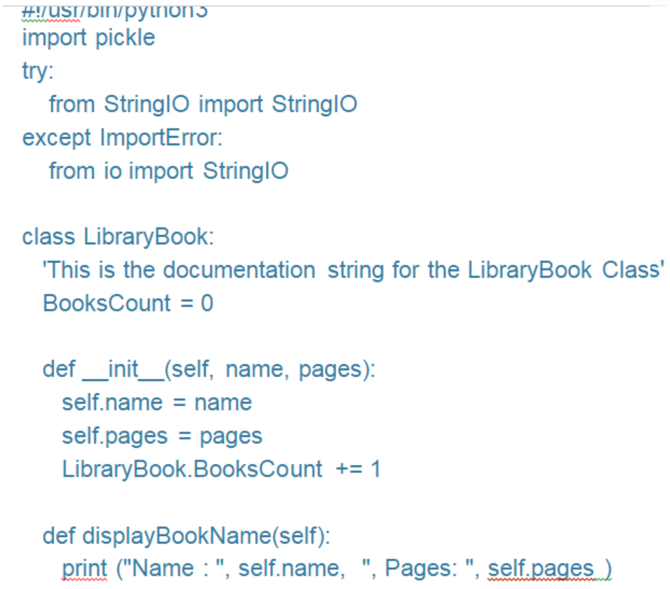
Example Program 2

```
#Back to main program
# 1. Instantiate  a base class with 2 arguments.

x = BookBase("Moby  Dick", 300)

# 2. Instantiate  a Derived  Class with additional  information

y = BookSubject("Adventures  of Tom Sawyer", "700", "English Literature")

# Print the  Base Class's method.
print(x.Name())

# Access the Derived's class methods
print(y.GetBookName())
print(y.GetSubject())
```

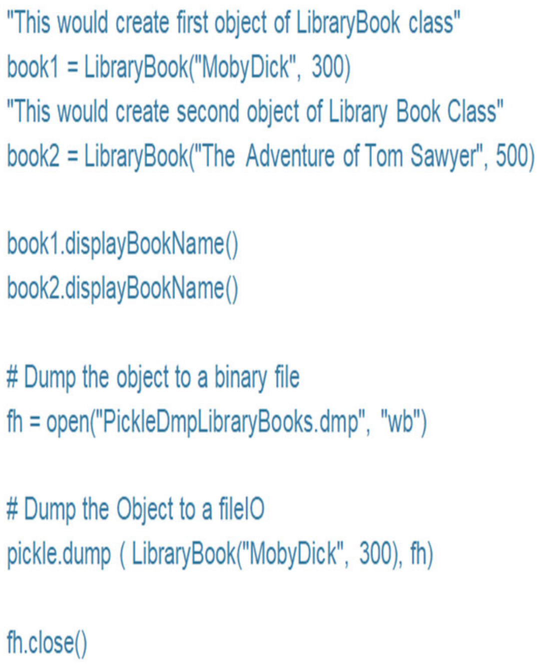
Serializing Python Objects

- Often times it is required to save the structured state of the data in some format so that it can be retrieved later, or minimize the data storage area, or send it across to another point or to someone. Then data can be reused later and picks from where it was last left off.
- For storing (or Serializing) data , a certain module has been developed. This module is called Pickle and has the ability to save, store and later retrieve it the same state and take it from there.
- The process of serializing involves storing the data in byte stream and Unserializing data uses just the reverse of it and converts back to the object form
- import pickle needs to be mentioned in the beginning of the python program in order to use the pickle commands
- The most common functions are “dumps()” and “loads()”.
- The “dumps()” serializes the object (or it’s hierarchy)
- The “loads()”. un-serializes the object back.
- Format :pickle.dump(obj, file, protocol = None, *, fix_imports = True)

Example Program 1



```
#!/usr/bin/python3
import pickle
try:
    from StringIO import StringIO
except ImportError:
    from io import StringIO

class LibraryBook:
  'This is the documentation  string for the LibraryBook  Class'
  BooksCount = 0

  def __init__(self,  name,  pages):
    self.name  = name
    self.pages  = pages
    LibraryBook.BooksCount  += 1

  def displayBookName(self):
    print ("Name : ", self.name,   ", Pages: ", self.pages  )
```



```
"This would create first object of LibraryBook class"
book1 = LibraryBook("MobyDick",  300)
"This would create second object of Library Book Class"
book2 = LibraryBook("The  Adventure of Tom Sawyer", 500)

book1.displayBookName()
book2.displayBookName()

# Dump the object to a binary file
fh = open("PickleDmpLibraryBooks.dmp",  "wb")

# Dump the Object to a fileIO
pickle.dump ( LibraryBook("MobyDick",  300), fh)

fh.close()
```

Example Program 2

```
$ python3.    LibraryBooks.py

Name :  MobyDick  , Pages:  300
Name :  The Adventure of Tom Sawyer , Pages:  500
```

Check if an external file got created after running the script.

```
ls -l  ( or dir ) PickleDmpLibraryBooks.dmp
```

Checkpoint

1. When a file to be opened is not there, what is the Exception Error that is raised ?

a) ZeroDivisionError
b) ImportError
c) IOError

2. When a User hits the Ctrl-C when the program is waiting for key input, what Exception is raised ?

a) StandardError
b) FloatingPointError
c) KeyboardInterrupt

3. A Try(), Catch () statement defines the Python Exception Error catch statements ?

a) True
b) False

Checkpoint

IBM ICE (Innovation Centre for Education)

4. The expect() command is the last statement to be executed in a try, except construct ?

a) True
b) False

5. The try: expect: statements can be used after importing the import trycatch library ?

a) True
b) False

6. Is this a valid try except statement construct ? State True or False

(Answer False, try is not a function)

```
try() :
     Some code
 except ExceptionA():
     some code
 except ExceptionB ()
    some code
```

Checkpoint

7. Is this a valid except statement ?

```
try  :
     Some code

except ExceptionA():
     some code

except ExceptionB ():
    some code

else :
    print ( “ This is  the last statement in the try- except construct”)
```

Checkpoint

8. Identify correct method to instantiate a class LibraryBook

a) book = LibraryBook :
b) book = LibraryBook(Bookname, pages_in_book). # Correct Answer
c) book = LibraryBook (book name)

9. User wants to know how many instances are made so far? Which statement would be used ?
a) print (book.displayBookName ())
b) print (book.displayBooksCount())
c) print (book.displayBooksCount). # Answer

10. Identify the correct statement to load contents of an external file containing the serialised object to program
a) pickle.load ()
b) pickle.dump ()

Unit summary

Having completed this unit, you should be able to know:

- How to create isolated environments,
- How to use the environments without conflicting with other Python environments called virtual environments.
- How to execute external code within Python and how to run them in parallel.
- Introduction to File open / close, Opening binary files,
- How to call functions, raise exceptions,
- OOP concepts, Base classes, Derived classes, Inherit classes