**Python training Notes:**

**Course Name:** **SCRIPT 307: Basic Python**

**Day 5: 15 Sep 2017**

**Expectation Setting : ASL (Assisted Self-Learning) 3Hrs session daily**

**\*\*\*\*\*\*Course outline for : SCRIPT 302: Basic Python**

[**https://persistentuniversity.persistent.co.in/coursedetails.aspx?cid=837**](https://persistentuniversity.persistent.co.in/coursedetails.aspx?cid=837)

**And then do self-study and hands on assignments form below learning course link:**

<https://knowledgecenter.persistent.co.in/ViewCourse/pmoc>

**Micro Learning Online link :** <https://persistentuniversity.persistent.co.in/microlearning/Course/837>

Welcome to Micro Learning!!

You can now complete a course by learning small units at a time, i.e. learning Micro Nuggets. Take out 15 minutes time daily and complete one micro nugget and a short quiz. Click on Complete button to ensure the completion and to get access to next Micro Nugget on next day.

Use the link given above to visit the Micro Learning Site to view the next micro nugget.

Every Micro Nugget is of short period 10-15mins approximately.

***Please visit the following URL to view the collaborative learning group***

<https://persistentuniversity.persistent.co.in/CollaborativeLearningGroup/View.aspx?SkillID=8451>

***My Python recording youtube as part of Smart India Hackathon***

<https://www.youtube.com/watch?v=D2SkCdNZW08>

**Try Below Codes:**

**Execption handling**

**1\_Exception\_Demo1.py**

#try-except statement

def reading():

#f = open('data11111.txt') #default read mode

try:

f = open('data1111.txt') #default read mode

except IOError as a:

print 'could not open file =', a #exception handler code

print "Learning Exception Handling......"

reading() #calling a function

print "END!!!!!"

"""

try:

f = open('data.txt') #default read mode

except IOError as a:

print 'could not open file =', a #exception handler code

"""

**2\_Exception\_Demo2.py**

#try statement with multiple excepts

def safe\_float(object):

try:

retval = float(object)

except ValueError:

retval = 'could not convert non-number to float'

except TypeError:

retval = 'object type cannot be converted to float'

return retval

ret1=safe\_float(3) #call successful

print ret1

print "--------------------------------------------"

ret2=safe\_float('abc') #ValueError

print ret2

print "--------------------------------------------"

ret3 = safe\_float({'a': 'Dict'}) #TypeError

print ret3

print "--------------------------------------------"

**3\_Exception\_finally.py**

try:

testfile = open('data.txt')

try:

txns = testfile.readlines()

print "File data = ",txns

finally:

print "Inner In finally"

except IOError:

print 'unable to access test file\n'

finally:

print "Outer In finally"

#testfile.close()

**4\_User\_Exception1.py**

#creating a sub class

#Exception is pre defined class, create user defined class MYError as subclass of Exception

class MyError(Exception):

pass

try:

raise MyError("Some information about what went wrong")

except MyError as error:

print("Situation:", error)

"""

class AgeError(Exception):

pass

"""

**map\_temp\_ex.py**

def fahrenheit(T): #converting temp into fahrenheit

return ((float(9)/5)\*T + 32)

def celsius(T): #converting temp into celsius

return (float(5)/9)\*(T-32)

#-------------------------------------------------------------------

temp = (36.5, 37, 37.5,39) #temp in celcius

print "Temp in celsius = ", temp

F = map(fahrenheit, temp)

print "Temp in farenite = ", F

C = map(celsius, F)

print "Temp back in celsius = ", C

"""

#logic code

for i in temp:

F=fahrenheit(i)

alternative solution--->

F = map(fahrenheit, temp)

"""

**map\_lambda\_temp\_ex.py**

#In the example above we haven't used lambda. By using lambda,

#we wouldn't have had to define and name the functions fahrenheit() and celsius().

Celsius = [39.2, 36.5, 37.3, 37.8]

print "Original temp in Celsius =", Celsius

Fahrenheit = map(lambda x: (float(9)/5)\*x + 32, Celsius)

print "Fahrenheit temp using map and lambda =", Fahrenheit

C = map(lambda x: (float(5)/9)\*(x-32), Fahrenheit)

print "Celsius temp using map and lambda =", C

"""

>>> f = lambda x, y : x + y

>>> f(1,1)

"""

**Map\_lambda\_multiple\_list.py**

a = [1,2,3,4]

b = [17,12,11,10]

c = [-1,-4,5,9]

print map(lambda x,y:x+y, a,b)

#[18, 14, 14, 14]

print map(lambda x,y,z:x+y+z, a,b,c)

#[17, 10, 19, 23]

print map(lambda x,y,z:x+y-z, a,b,c)

#[19, 18, 9, 5]

**1\_MathDemo.py**

import math

print 'The value of PI is approximately %5.3f.' % math.pi

print "Sqrt of 25 = ", math.sqrt(25)

print math.pow(2,3)

2\_math\_map.py

#map function

from math import \*

r = [0, 1, 2, 3, 4, 5, 6]

target\_list = map(cos, r)

print "Target list =",target\_list

print "---------------------------------------------"

'''

Built-in function that works on a list

'map' takes a function and a list (2 arguments)

The function must take only one argument, and return one value

The function is applied to each value of the list

The resulting values are returned in a list

'''

1\_array\_demo.py

from array import array

a = array('H', [4000, 10, 700, 22222])

print sum(a) #26932

#print a[1:3]

#array('H', [10, 700])

'''

The array module provides an array() object that is like a list that

stores only homogeneous data and stores it more compactly.

'''

Iter\_Demo.py

# iter.py

str = "formidable" #str[0]

for i in str: #for loop traverses all elements in a sequence

print i,

print "------------------------------------------"

it = iter(str) #iter is predenfined function

print it.next()

print it.next()

print it.next()

print "------------------------------------------"

print list(it)

'''

#set

s ={1,2,3,4,5} #not sequence type, unique elements

OR

l1 =[1,2,3,4,5]

s1= set(l1)

it = iter(s) #iterable object

'''

lambda\_sort\_ex.py

from operator import itemgetter

employee\_tuples = [('john','A',25),('jane','B',32),('dave','B',28)]

#print sorted(employee\_tuples, key=lambda student:student[2]) #sort by age

print sorted(employee\_tuples, key=lambda s:s[2]) #sort by age

print "reverse = ", sorted(employee\_tuples, key = itemgetter(2), reverse=True)

**Os\_Demo.py**

import os

#os.system('echo $HOME')

os.system('cls')

print ("Good Bye 2015!!!!")

print "AAA"

**shutil\_demo.py**

import shutil

shutil.copy('hello3.txt', 'target.txt')

'''

shutil.copy2(src, dst)

Similar to shutil.copy(), but metadata is copied as well in fact,

this is just shutil.copy() followed by copystat().

'''

#shutil.copytree('C:/Python\_Jan16','C:/Python\_Jan16\_testing')

'''

Recursively copy an entire directory tree rooted at src.

The destination directory, named by dst, must not already exist; it will be

created as well as missing parent directories. Permissions and times of

directories are copied with copystat(), individual files are copied using

shutil.copy2().

'''

shutil.rmtree('C:/Python\_Jan16\_testing')

'''

Delete an entire directory tree

'''

'''

shutil.move(src, dst)

Recursively move a file or directory (src) to another location (dst).

If the destination is a directory or a symlink to a directory,

**sys\_demo.py**

**#!/usr/bin/env python**

**import sys**

**sys.stdout = open("hello.txt","w")**

**print "This is sum" #instead of printing on console, it will print inside file hello.txt**

**sys.stdout.close()**

**'''**

**stdin**

**stdout**

**stderr**

f1 = open("hello.txt","w")

f1.write("Hello !!!")

'''

sys\_err.py

import sys

temp = sys.stderr

sys.stderr = open("error1.txt", "a")

print nosuchvar #error gets printed in file error.txt

#sys.stderr.close()

sys.stderr = temp #o/p will come back to consle err

print nosuchvar

**debugging**

pdb\_demo.py

import pdb

#python debugging

def test\_deb(test\_val):

print "start test val is ", test\_val

ret\_val = test\_val/10

print "end test\_val is ", test\_val

return ret\_val

#print test\_deb(20) #function call

pdb.run("test\_deb(20)") #invoking function test\_deb(20) by run on pdb

#execute the file

#it goes into pdb prompt

# keep pressing s

# then type\# p test\_val

#p ret\_val

# s shows the line of code to be executed

#b break point

#tbreak temparary break point

**Doctest**

def my\_func(a,b):

'''

>>> my\_func(2,3)

6

'''

return a + b

print my\_func(2,3) #calling a function

#run as

#>> python -m doctest -v doctest\_ex.py

Unitest\_ex.py

import unittest

#def my\_func(a,'ttest\_ex.pyb'):

def my\_func(a,b):

return a \* b

class TestME(unittest.TestCase):

def setup(self):

pass

def testnum(self):

self.assertEqual(my\_func(3,4),12)

def teststr(self):

self.assertEqual(my\_func('a',3),'aaa')

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**Python Unit Testing**

**Testing**, in general programming terms, is the practice of writing code (separate from your actual application code) that invokes the code it tests to help determine if there are any errors. It does not prove that code is correct (which is only possible under very restricted circumstances). It merely reports if the conditions the tester thought of are handled correctly or not.

What kinds of things can be caught in testing? Syntax errors are unintentional misuses of the language, like the extra . in my\_list..append( abc). Logical errors are created when the algorithm (which can be thought of as "the way the problem is solved") is not correct. Perhaps the programmer forgot that Python is "zero-indexed" and tried to print the last character in a string by writing print(my\_string[len(my\_string)]) (which will cause an IndexError to be raised). Larger, more systemic errors can also be checked for. Perhaps the program always crashes when the user inputs a number greater than 100, or hangs if the web site it's retrieving is not available.

All of these errors can be caught through careful testing of the code. Unit testing, specifically tests a single "unit" of code in isolation. A unit could be an entire module, a single class or function, or almost anything in between. What's important, however, is that the code is isolated from other code we're not testing (which itself could have errors and would thus confuse test results)

Using Python's built-in **unittest** framework, any member function whose name begins with test in a class deriving from unittest.TestCase will be run, and its assertions checked, when unittest.main() is called. If we "run the tests" by running python test\_primes.py, we'll see the output of the unittest framework printed on the console:

**Exercise:**  Create a file named **primes.py** and create below function in it. This checks if a number is prime or not?

def is\_prime(number):

"""Return True if \*number\* is prime."""

for element in range(number):

if number % element == 0:

return (False)

return(True)

**Exercise:** Please create a file named **test\_prime.py** and create below function in it. This will conduct unit testing on our function is\_prime .

import unittest # importing the unittest framework

from primes import is\_prime #importing function is\_prime from the primes module

class PrimesTestCase(unittest.TestCase):

"""Tests for `primes.py`."""

def test\_is\_five\_prime(self):

"""Is five successfully determined to be prime?"""

self.assertTrue(is\_prime(5))

def test\_is\_four\_prime(self):

"""Is Four successfully NOT determined to be prime?"""

self.assertFalse(is\_prime(4))

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**Exercise:** go to command prompt->file location where you have saved both the files and run below command. You can also run it from IDLE by opening the file test\_prime.py and click on Run option from drop down menu.

C:\Desktop>python test\_prime.py

EE

======================================================================

ERROR: test\_is\_five\_prime (\_\_main\_\_.PrimesTestCase)

Is five successfully determined to be prime?

----------------------------------------------------------------------

Traceback (most recent call last):

File "test\_prime.py", line 9, in test\_is\_five\_prime

self.assertTrue(is\_prime(5))

File "C:\Users\farukh\_hashmi\Desktop\primes.py", line 4, in is\_prime

if number % element == 0:

ZeroDivisionError: integer division or modulo by zero

======================================================================

ERROR: test\_is\_four\_prime (\_\_main\_\_.PrimesTestCase)

Is Four successfully determined to be prime?

----------------------------------------------------------------------

Traceback (most recent call last):

File "test\_prime.py", line 12, in test\_is\_four\_prime

self.assertTrue(is\_prime(4))

File "C:\Users\farukh\_hashmi\Desktop\primes.py", line 4, in is\_prime

if number % element == 0:

ZeroDivisionError: integer division or modulo by zero

----------------------------------------------------------------------

Ran 2 tests in 0.000s

FAILED (errors=2)

**Python Unit Testing: Rectifying the errors**

This output shows us that our tests resulted in failure.

The issue here is clear: we are preforming the modulo operation over a range of numbers that includes zero, which results in a division by zero being performed. To fix this, we simply **change the range to begin at 2** rather than 0, noting that modulo by 0 would be an error and modulo by 1 will always be True (and a prime number is one wholly divisible only by itself and 1, so we needn't check for 1).

A failing test has resulted in a code change. Once we fix the error (changing the line in is\_prime to for element in range(2, number):), we get the following output:

C:\Desktop>python test\_prime.py

..

----------------------------------------------------------------------

Ran 2 tests in 0.000s

OK

**Python Doc Tests**

The doctest module will search for pieces of text in your code that resemble interactive Python sessions. It will then execute those sessions to verify that they work exactly as written. This means that if you wrote an example in a docstring that showed the output with a trailing space or tab, then the actual output of the function has to have that trailing whitespace too. Most of the time, the docstring is where you will want to put your tests.

Create the below function and save the file as **doctest\_example.py** and run it to observe doc test.

''‘ Module showing how doctests can be included with source code Each '>>>' line is run as if in a python shell, and counts as a test. The next line, if not '>>>' is the expected output of the previous line.

If anything doesn't match exactly (including trailing spaces), the test fails. '''

def multiply(a, b):

"""

>>> multiply(4, 3)

11 # A bad test because the expected result was 12 but here we have given 11

>>> multiply(4, 3)

12 # A good test

>>> multiply('a', 3)

'aaa‘ # A good test

"""

return a \* b

if \_\_name\_\_ == "\_\_main\_\_":

**Python Performance Enhancement: Profiling**

Profiling a Python program is doing a dynamic analysis that measures the execution time of the program and everything compose it. That means measuring the time spent in each of its functions. This will give you data about where your program is spending time, and what area might be worth optimizing.

Since Python 2.5, Python provides a C module called [**cProfile**](https://docs.python.org/2/library/profile.html) which has a reasonable overhead and offers a good enough feature set

**Exercise**: Create a module named **cProfiler\_example.py** where 2 functions are defined .The **fast** function will run at normal speed and the **slow** function will take around 3 seconds to run. The **main** function calls the other three.

Use **cProfiler** to see where the bottle neck is present in excuting the code.

We see the program took 3.5 seconds to run. If you examine the results, you will see that **cProfile** has identified the **slow** function as taking 3 seconds to run. That’s the biggest bottleneck after the **main** function. Normally when you find a bottleneck like this, you would try to find a faster way to execute your code or perhaps decide that the runtime was acceptable. In this example, we know that the best way to speed up the function is to remove the **time.sleep** call or at least reduce the sleep length.

import time

def fast():

print("I run fast!")

def slow():

time.sleep(3)

print("I run slow!")

def main():

fast()

slow()

if \_\_name\_\_ == '\_\_main\_\_':

main()

>>> import cProfiler\_example

>>> cProfile.run('cProfiler\_example.main()')

I run fast!

I run slow!

I run a little slowly...

300 function calls in 3.516 seconds

Ordered by: standard name

ncalls tottime percall cumtime percall filename:lineno(function)

1 0.000 0.000 3.516 3.516 <string>:1(<module>)

6 0.000 0.000 0.017 0.003 PyShell.py:1336(write)

1 0.000 0.000 3.516 3.516 cProfiler\_example.py:21(main)

1 0.000 0.000 0.004 0.004 cProfiler\_example.py:4(fast)

1 0.000 0.000 3.003 3.003 cProfiler\_example.py:9(slow)

42 0.000 0.000 0.000 0.000 rpc.py:150(debug)

**The key differences between Python 2.7.x and Python 3.x**

[**http://sebastianraschka.com/Articles/2014\_python\_2\_3\_key\_diff.html**](http://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html)

[**https://www.codementor.io/python/tutorial/python-2-7-vs-python-3-4**](https://www.codementor.io/python/tutorial/python-2-7-vs-python-3-4)

**Eclipse Settings**

**windows🡪preferences**

**or help->install new software**

**click on ADD**

**window : Name : enter as Python 2**

**search pydev download in google**

**put this location** [**http://pydev.org/updates**](http://pydev.org/updates)

[http://update-production-pydev.s3.amazonaws.com/pydev/updates](http://update-production-pydev.s3.amazonaws.com/pydev/updates/site.xml)

**More Info at:** **http://www.pydev.org/download.html**

**then preferences**

**interpreter**

**selete c:\python\python.exe**

Assignment

Solution for Country example

fh = open ('country.txt', 'r')

"""Data storage in dictionary"""

country\_lang ={} #empty dictionary

for line in fh:

line = line.rstrip() #line \n is removed

lang = line.split(',')[-1] #split function return a list

if lang in country\_lang:

country\_lang[lang] += 1

else:

country\_lang[lang] = 1

for i in country\_lang:

print i , country\_lang[i]

print "-------------------------------------------"

print country\_lang

**Assignment on ----->Country.txt file :**

1. **Language and Country**

Store the Country data only for Language and its list of countries in a dictionary.

Display the o/p as shown below -

>>>

{'Portuguese': ['Brazil'], 'Franch': ['Cameroon', 'Djibouti', 'Equatorial Guinea', 'France'], 'Chinese': ['China'], 'Vietnamese': ['Vietnam'], 'German': ['Germany'], 'English': ['United Kingdom', 'United States', 'Fiji', 'Canada', 'Ireland'], 'Japanese': ['Japan'], 'Greek': ['Greece'], 'Indian': ['India'], 'Spanish': ['Venezuela', 'Argentina', 'Honduras'], 'Arabic': ['Yemen', 'Bahrain'], 'Hungerian': ['Hungary'], 'Italian': ['Italy']}

2.

Let’s say I give you a list saved in a variable: a = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100].

Write one line of Python that takes this **list a** and makes a new **list b** that has only the even elements of this list in it.

List Comprehension

**SLT**

**Course outline for : SCRIPT 302: Basic Python**

[**https://persistentuniversity.persistent.co.in/coursedetails.aspx?cid=837**](https://persistentuniversity.persistent.co.in/coursedetails.aspx?cid=837)

**Subjective Paper pattern**

1. ***(Total: 15 Marks)***

**For Question 1 create file as Q1.py (Any other file name will not be accepted)**

1. ***(Total: 15 Marks)***

**For Question 2 create file as Q2.py (Any other file name will not be accepted)**

1. ***(Total: 5 Marks)***

**For Question 3 create file as Q3.py (Any other file name will not be accepted)**

1. ***(Total: 5 Marks)***

**For Question 4 create file as Q4.py (Any other file name will not be accepted)**

1. ***(Total: 5 Marks)***

**For Question5 – Question6 create files Q5.py – Q6.py, copy the given code and modify. Please mention the changes with appropriate comments. (Any other file name will not be accepted)**

**Problem / Faulty Code: wrong1.py**

1. ***(Total: 5 Marks)***

**For Question 6 create file as Q6.py (Any other file name will not be accepted)**

**Problem / Faulty Code: wrong2.py**