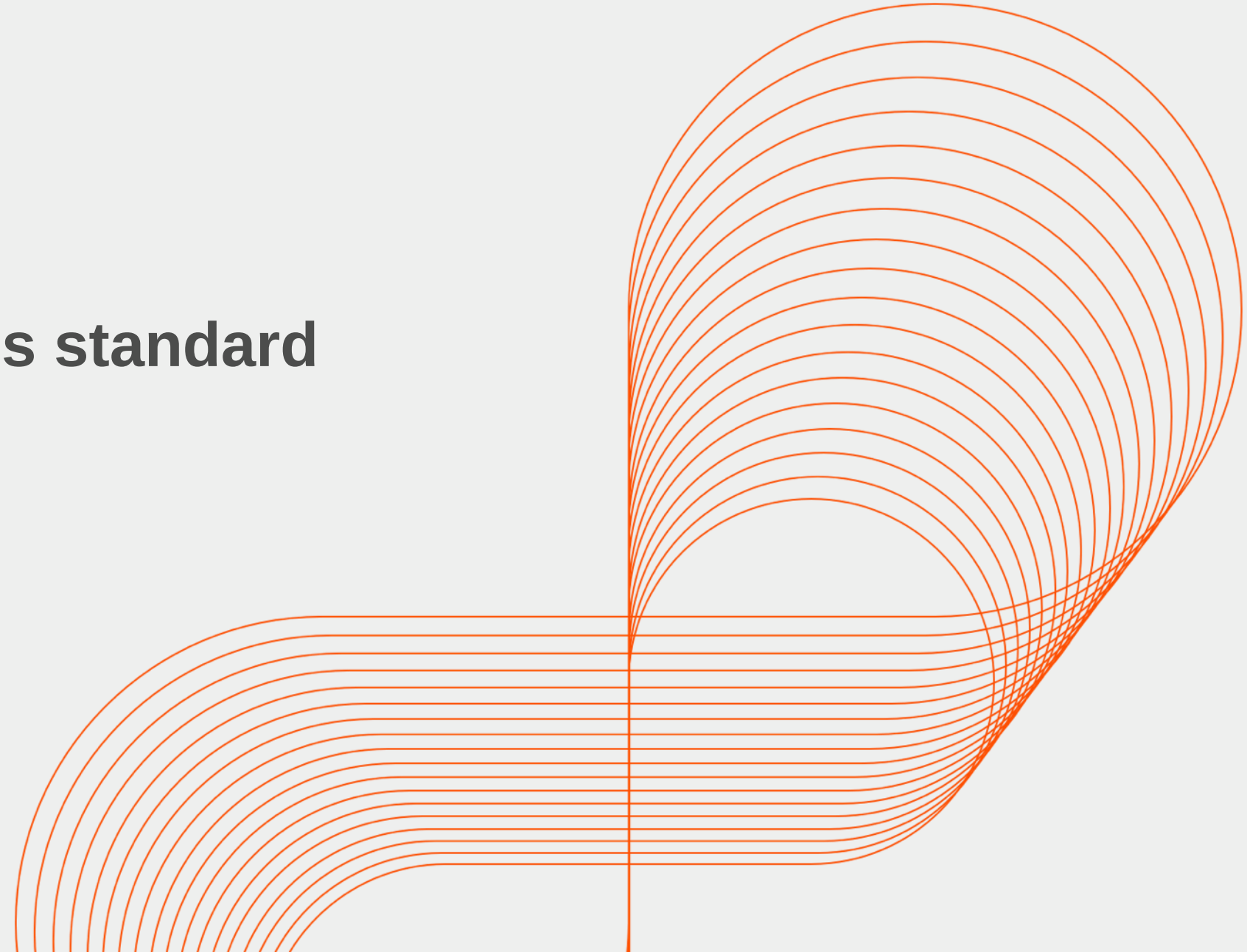
Persistent

Python Various standard libraries



Objectives

At the end of this second session, you will be able to understand:

- Various Python standard libraries

Persistent

Various standard libraries

Module	Description
os	The os module provides dozens of functions for interacting with the operating system
shutil	For daily file and directory management tasks, the shutil module provides a higher-level interface that is easier to use
glob	The glob module provides a function for making file lists from directory wildcard searches
sys	Command line arguments-sys.argv,sys.stdin, sys.stderr, sys.stdout
re	The re module provides regular expression tools for advanced string processing
random	The random module provides tools for making random selections
urllib	Internet access
datetime	The datetime module supplies classes for manipulating dates and times in both simple and complex ways

os, sys, shutil, glob, math, random, pickle, urllib, datetime, timeit, cProfile, pstats modules

sys module

- **Command Line Arguments**

- Common utility scripts often need to process command line arguments. These arguments are stored in the sys module’s argv attribute as a list. For instance the following output results from running -

  python demo.py one two three at the command line:

```
>>> import sys
>>> print sys.argv
['demo.py', 'one', 'two', 'three']
```

Error Output Redirection and Program Termination

- The sys module also has attributes for stdin, stdout, and stderr. The latter is useful for emitting warnings and error messages to make them visible even when stdout has been redirect.
- >>> sys.stderr.write('Warning, log file not found starting a new one\n') Warning, log file not found starting a new one.
- The most direct way to terminate a script is to use sys.exit()

os module

- The **os** module provides dozens of functions for interacting with the operating system:

```
>>> import os
>>> os.getcwd()                    # Return the current working directory 'C:\\Python27'
>>> os.chdir('/server/accesslogs’)  # Change current working directory
>>> os.system('mkdir today')       # Run the command mkdir in the system shell
```

```
>>> dir(os)
<returns a list of all module functions>
```

```
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

shutil module

- For daily file and directory management tasks, the shutil module provides a higher level interface that is easier to use:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
>>>shutil.copytree('c:/Demo1','C:/Demo2')
>>>shutil.rmtree('C:/Demo2)
>>>shutil.move(src, dst)
```

glob module

- The glob module provides a function for making file lists from directory wildcard searches:

```
>>> import glob
>>> glob.glob('*.py') ['primes.py', 'random.py', 'quote.py']
```

math module

- The math module gives access to the underlying C library functions for floating point math:

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

array module

- This module defines an object type which can compactly represent an array of basic values: characters, integers, floating point numbers.
- Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained.

Example

```
from array import array

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

print (sum(a) )       #26932

print (dir(array))

array.reverse(a)

print (a)
```

random module

- The random module provides tools for making random selections:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'

>>> random.random() # random float
0.1797098769370618 6

>>> random.randrange(6) # random integer chosen from range(6)
4
```


Persistent

data persistence: Pickle module

The pickle module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure. “Pickling” is the process whereby a Python object hierarchy is converted into a byte stream, and “unpickling” is the inverse operation, whereby a byte stream is converted back into an object hierarchy.

```
import pickle:
d = {'foo':'bar','cat':'tar’}        #dictionary
FH =open('foo.txt','w')
pickle.dump(d,FH)  #save dictionary object d (in pickle string format )in file #foo.txt writing
L=['hi','world’]                     #list
pickle.dump(L,FH)
s='bye world’                        #string
pickle.dump(s,FH)
FH.close
```

Data persistence: pickle module (contd.)

```
FH=open('foo.txt','r')
d1 = pickle.load(FH)            #unpickling by load function, gives me dictionary
                                #back by reading it
print d1                        #{'foo': 'bar', 'cat': 'tar'}
print "foo value =", d1['cat']  #foo value = tar

d2 = pickle.load(FH)            #['hi', 'world']
print d2

d2 = pickle.load(FH)            #bye world
print d2
```


Persistent

Internet access

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are urllib2 for retrieving data from URLs and smtplib for sending mail:

- **urllib**

```
import urllib.request

with urllib.request.urlopen('http://python.org/') as response:
   html = response.read()

print(html)
```

Internet access (contd.)

- **smtplib**

```
>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Note that the second example needs a mailserver running on localhost.)

Dates and times

The **datetime** module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are timezone aware.

```
>>> # dates are easily constructed and formatted

>>> from datetime import date

>>> now = date.today()

>>> now

datetime.date(2003, 12, 2)
```

Performance measurement: timeit

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The timeit module quickly demonstrates a modest performance advantage:

```
>>>from timeit import Timer

>>>Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()

0.5753528626024577

>>>Timer('a,b = b,a', 'a=1; b=2').timeit()

0.5496253708577079 1
```

In contrast to timeit‘s fine level of granularity, the profile and pstats modules provide tools for identifying time critical sections in larger blocks of code.

Python profilers

cProfile and **profile** provide deterministic profiling of Python programs. A profile is a set of statistics that describes how often and for how long various parts of the program executed. These statistics can be formatted into reports via the **pstats** module.

cProfile is recommended for most users; it’s a C extension with reasonable overhead that makes it suitable for profiling long-running programs.

Example:

```
import cProfile

import re

cProfile.run('re.compile("foo|bar")')
```

cProfile module

- **Python** includes a built in **module** called **cProfile** which is used to measure the execution time of a program. cProfiler **module** provides all information about how long the program is executing and how many times the function get called in a program.
- This makes us know where the program is spending too much time and what to do inorder to optimize it. It is better to optimize the code inorder to increase the efficiency of a program. So, perform some standard tests to ensure optimization and we can improve the program inorder to increase the efficiency.

- **Example:**

```
import cProfile

cProfile.run("10 + 10")
```

cProfile module

- **Output:**

3 function calls in 0.000 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.000	0.000	:1()
1	0.000	0.000	0.000	0.000	{built-in method builtins.exec}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

pstats

- Analysis of the profiler data is done using the Stats class.
- **class pstats.Stats(*filenames or profile, stream=sys.stdout)**
- This class constructor creates an instance of a “statistics object” from a filename (or list of filenames) or from a Profile instance. Output will be printed to the stream specified by stream.
- The file selected by the above constructor must have been created by the corresponding version of profile or cProfile.

```
import pstats

p = pstats.Stats('restats')

p.strip_dirs().sort_stats(-1).print_stats()
```

doctest module

- **doctest** is a **module** included in the **Python** programming language's standard library that allows the easy generation of tests based on output from the standard **Python** interpreter shell, cut and pasted into docstrings.

- **Docstrings** in Python are used not only for the description of a class or a function to provide a better understanding of the code and use but, also used for Testing purposes.

- The input and expected output are included in the docstring, then the doctest module uses this docstring for testing the processed output.

- After parsing through the docstring, the parsed text is executed as python shell commands and the result is compared with the expected outcome fetched from the docstring.

doctest module : Example

- Here’s a simple example:

1. import testmod from doctest to test the function.
2. Define our test function.
3. Provide a suitable docstring containing desired output on certain inputs.
4. Define the logic.
5. Call the testmod function with the name of the function to test and set verbose True as arguments.

doctest module : Example

```
from doctest import testmod
def factorial(n):
        '''
        This function calculates recursively and
        returns the factorial of a positive number.
        Define input and expected output:
        >>> factorial(3)
        6
        >>> factorial(5)
        120
        '''
        if n <= 1:
                return 1
        return n * factorial(n - 1)
if __name__ == "__main__":
        testmod(name ='factorial', verbose = True)
```

doctest module : Example Output

```
Trying:
    factorial(3)
Expecting:
    6
ok
Trying:
    factorial(5)
Expecting:
    120
ok
1 items had no tests:
    factorial
1 items passed all tests:
   2 tests in factorial.factorial
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

Unit Testing in Python – Unittest

- **What is Unit Testing?**
  Unit Testing is the first level of software testing where the smallest testable parts of a software are tested. This is used to validate that each unit of the software performs as designed.

- **Method:**
  White Box Testing method is used for Unit testing.

- **OOP concepts supported by unittest framework:**
- **test fixture:**
  A test fixture is used as a baseline for running tests to ensure that there is a fixed environment in which tests are run so that results are repeatable.
- **test case:**
  A test case is a set of conditions which is used to determine whether a system under test works correctly.

Unit Testing in Python – Unittest

- **test suite:**
  Test suite is a collection of testcases that are used to test a software program to show that it has some specified set of behaviors by executing the aggregated tests together.
- **test runner:**
  A test runner is a component which set up the execution of tests and provides the outcome to the user.
- **Example:**

```
import unittest

class SimpleTest(unittest.TestCase):

    # Returns True or False.

    def test(self):

        self.assertTrue(True)

if __name__ == '__main__':

    unittest.main()
```

Unittest example

- This is the basic test code using unittest framework, which is having a single test. This test() method will fail if TRUE is ever FALSE.
- **unittest.TestCase** is used to create test cases by subclassing it. The last block of the code at the bottom allows us to run all the tests just by running the file.
- **Output:**

.

Ran 1 test in 0.000s

OK

- Here, in the output the “.” on the first line of output means that a test passed.

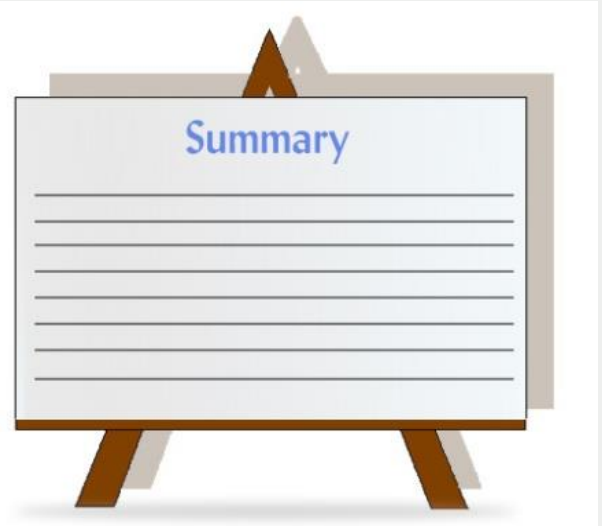
Summary

With this we have come to the end of our session, where we discussed about:

- Various Python standard libraries

In the next session we will discuss about:

- XML Processing in Python



Reference material

- http://www.tutorialspoint.com/python
- http://www.learnpython.org/
- http://docs.python.org/2/tutorial/
- https://docs.python.org/2/tutorial/stdlib.html
- https://docs.python.org/2/tutorial/stdlib2.html
- https://packaging.python.org/installing/

Questions

Key contacts

Sakshi Jamgaonkar

sakshi_jamgaonkar@persistent.com

Asif Immanad

asif_immanad@persistent.co.in

Persistent

Persistent

Thank you!