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 1: Introduction to Python basics

Duration: 2 hours

01/07/18

Introduction

Python is a general purpose high level programming language with the following key features

Interpreted language

- Unlike compiled languages like C/C++, python is an interpreted language.

- At runtime commands are interpreted and processed by the python interpreter.

Interactive language

- Python command intrepetor provides a interactive prompt

- Individual commands and the tried/executed on the prompt directly

Portable

- Python language is portable across different platforms

Easy to use

- Python language is intutive and has a clear syntax.

- This makes it readable and fairly easy to code.

- Supports both procedural as well as object oriented programming pattern,

Uses

Since Python is generic and fairly easy to learn it is used for several applications. There is a huge number of libraries available that support python.

They include areas like

- Math and scientific (SciPy, NumPy, SymPy)
- Graph plotting (mathplotlib, Plotly, pyQtGraph)
- GUI toolkits (pyQT, pyGTK, wxPython)
- Audio handling (pyAudio, Nsound, pySonic)
- Computer Vision and machine learning (openCV, bindings with TensorFlow)
- Several test automation frameworks like Robot Framework, Selenium, Cypress, etc

This is just a small list, there are several libaries for, networking, multi-threading, web applications, database management.

Installing Python

Python is available for a wide variety of platforms including Linux, Windows and MacOS. We will be convering the steps to install it on Linux and Windows

Installing on Linux

- Standard Ubuntu desktop installation will come with some version of Python 2.7 preinstalled.
- If its not already installed then it can be installed via either apt-get command or synaptic package manager
- If anyone wants to build it from sources, following are the steps
    - Download source from https://www.python.org/downloads/release/python-2715/
    - Change directory to Python unzip folder and run command `./configure`
    - Run `make` to compile and `make install` to install

Installing on Windows

- Download the Python installer from https://www.python.org/downloads/release/python-2715/
- Run the downloaded installer (*.msi) file and following the installation wizard.

Setting up Python environment

This will be required only if python was manually installed. If its pre-installed or if installed by the installation wizard then these steps may not be required.

Setting Python PATH on Linux

- For Bash or Sh: Use command `export PATH="$PATH:/usr/local/bin/python"`
- For csh: Use command `setenv PATH "$PATH:/usr/local/bin/python"`

Considering python install location is /usr/local/bin/python. In most cases the default installation path is /usr/bin/python

Setting Python PATH on Windows

- Run the command `path %path%;C:\Python`
- It can also be set using the Windows UI
  - Right Click on “My Computer” then;
  - “Properties” -> “Advanced System Settings” -> “Environment Variables” -> “New”

NOTE: Default install location is C:\Python

Python package installer

There are several useful python packages available and most of them can be installed using the Python package installer (pip)

- In Python version 2.7.9 and above or 3.4 and above, PIP is pre-installed.
- On older versions of python it can be installed using get-pip.py script
  - Download it from the location [https://bootstrap.pypa.io/get-pip.py]
  - Install it with the following command [`python get-pip.py`]
  - Installing it behind a proxy

    [`python get-pip.py –proxy="http://[user:passwd@]proxy.server:port"`]
  - If PIP needs to be upgraded
    - On Linux: `pip install -U pip`
    - On Windows: `python -m pip install -U pip`
- Installing some package with PIP

```
$ pip install SomePackage
[...]
Successfully installed SomePackage
```

Running Python interpreter

Being an interactive language it is possible to launch Python interpreter from the command line.

- On Linux it can be launched by command **`$python`**

- On windows it can be launched by command **`C:> python`**

Sample command being run in the interpreter

```
~$ python
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> print "Hello World ... from Python"
Hello World ... from Python
>>>
```

Writing the first Python Program

Writing the first Python program

- Open any text editor any create a new file, hello.py

- Enter the following text in the file

```
#!/usr/bin/python
print "Hello World ... from Python"
```

Note the first line #!/usr/bin/python. It points to the python interpreter.

Running the program

```
$ python hello.py
```

Or

```
$ chmod +x hello.py
$ ./hello.py
Hello World from Python
$
```

Identifiers in Python

Identifiers in Python are names used for variables, functions, classes, objects or modules.

Following are some restrictions on naming the identifiers

- Punctuation characters such as @, $, and % are not allowed in the identifiers
- Identifiers are case sensitive **Buffer** and **buffer** are different variables
- Identifiers can start with an letter from a to z or A to Z or an underscore _
- This can be followed by letters, digits and underscores.

Reserved words

and	exec	not	assert	finally	or
break	for	pass	class	from	print
continue	global	raise	def	if	return
del	import	try	elif	in	while
else	is	with	except	lambda	yield

Convention for Python identifiers

- Class names start with an uppercase letter.
- All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.
- Starting an identifier with two leading underscores indicates a strongly private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special Name. These are mostly used for built-in identifiers.

Lines identation in Python

Identation

- Unlike languages like C/C++, Python does not provide braces to indicate code blocks
- Instead code blocks are identified by line identations, which is stricly enforced
- The number of spaces used for identation is variable but the same (or mutliples of the same) needs to be followed for the entire code block

Example

```
        try:
            for line in open(self.version_log,'r').readlines():
                try:
                    if "BUILD_NUMBER" in line:
                        self.version = line.split(' ', 2)[1]
                        self.version = self.version.strip('\n')
                except IndexError:
                    print "Ignoring line [{}]".format(line.strip('\n'))
                    pass
        except IOError:
            print "Unable to open [{}] file".format(self.version_log)
            self.version = "NA"
```

Multi-line statements, Quotations and comments in Python

Mulit-line statements

- Python statements end with a new line, however it allow the use of the line continuation character (\) to denote that the line should continue

- Statements contained within the [], {}, or () do not need to use the line continuation character

Quotations

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

Comments

- The hash (#) which is not inside any string literal marks the begnining of a comment

Example

```
#!/usr/bin/python
Line = 'hello' + \
       'world'
# Printing hello world
print Line
```

Python variables declaration

Variable declaration

- Python does not need explicit declaration of the variable for the interpreter to to reserve memory

- Declaration happens automatically when the variable is assigned a value

Assigning values to variable

```
#!/usr/bin/python
index   = 100         # Integer assignment
percent = 97.5        # Floating point assignment
name    = "Hello"     # String assignment
print index
print percent
Print name
```

Multiple assignments

- Assigning same value to multiple variables

```
i = j = k = 100
```

- Assigning multiple variables with values of different types

```
i,j,k = 1, 99.5, “percent”
```

Python standard data types

Following are standard data types supported by Python

- **Numbers:** Python supports four numerical types namely; int, long, float, complex
- **String:** Strings are identified as a contiguous set of characters represented in the quotation marks.

  Python allows either single or double quotation marks.
- **List:** List are comma separated list of items, they are similar to arrays in C

  However a key difference is one list can contains items of different data types

  Eg: `list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]`

  Note: Elements call be accessed by index. First list[0], last list[-1], third list [2]
- **Tuple:** Tuples are similar to list but they cannot be updated. Like a readonly list.

  Unlike list they are represented by (()) brackets

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
```

- **Dictionary:**

  Dictionaries are hash table types, with associated key value pairs

```
dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"
```

Python advanced types

Additionally some are some python standard data types and objects. Some of them are listed below

- **Iterator Types:** Python supports a concept of built in iterator.

  Eg:

```
print("List Iteration")
l = ["How", "are", "you"]
for i in l:
   Print(i)
```

In the above example built in iterator is used.

- **Binary Types:** These include bytes, bytearray, memoryview. These are typically used for manipulating binary data

Python type of operators

Python supports the following types of operators

- Arithmetic: Add(+), Subtract(-), Multiply(*), Exponent(**), Divide(/), Floor division(//), Modulus(%)
  NOTE: In python 2.7 (/) and (//) both are same and perform floor or integer division.

- Comparison (Relational):
  Evaluates true, if left and right operands are, equal(==), not equal(!= or <>), less than(<), greater than (>), less than equal(<=) or greater than equal(>=)

- Assignment: Assign (=), add and assign (+=), subtract and assign (-=), multiply and assign (*=), divide and assign (/=), Modulus and assign (%=), exponent and assign (**=)

- Bitwise: Binary AND(&), Binary OR(|), Binary XOR(^), One's complement (~), left (<<) and right(>>)

- Membership: This operator tests if the variable in the left operand is part of the specified sequence
  For 'in' operator, true if the variable is part of the sequence and for 'not in' operator if its not

- Identity: This operator tests if both the operands are points to the same memory location.
  The 'is' operator evaluates true if they point to the same object and 'is not' evaluates true if they don't

Python built-in modules

Python provides a lot of built-in modules, following is a non-exhaustive list of those

Built-in Functions: abs(), all(), any(), hash(), zip(), bin(), chr(), int(), isinstance(), len(), map(), vars() and may more

Built-in Types: Already covered in previous slides

Text Processsing: string, re, textwrap, unicodedata, etc

Numeric and Mathematics modules: numbers, math, decimal, random, statistics etc

File and Directory access: pathlib, os.path, stat, glob, shutils etc

Data compression: zlib, gzip, bz2, lzma zipfile, tarfile

Cryptography services: hashlib, hmac, secrets, random numbers

IPC and networking: signal, mmap, socket, ssl, asyncio etc

Internet data handling and Markup processing: mimetypes, email, json, html, xml

..... and many more

Python scan user inputs

Following is an example to scan user inputs in the program

```
#!/bin/python

first_name = raw_input("Input your First Name:")
last_name = raw_input("Input your Last Name:")
print ("Hello  " + first_name + " " + last_name)
```

Note:
In python 2.7
raw_input() takes exactly what the user typed and passes it back as a string.

In python 3.0
The same behaviour of raw_input() is implemented in input()

Note:
Since the raw_input returns a string, so if numbers are expected, it needs to be converted to numeric types
Eg: `num = “3”`
`int_num = int(num)`
There are several such conversions possible: int(), long(), float(), str(), etc

Handling command line arguments

In most program it is required to accept some inputs from the user at the time of launching.

Python provides a built-in module to do that. This module is **'sys'**

This modules provides access to python interpreter variables and function with interact with it.

Following is a simple example of a Python script using command line arguments.

```
#!/usr/bin/python

import sys

print 'Number of arguments:', len(sys.argv), 'arguments.'
print 'Argument List:', str(sys.argv)
```

When launched with 3 arguments (arg1, arg2 and arg3), it will print the following output

```
$python main.py arg1 arg2 arg3
Number of arguments: 4 arguments.
Argument List: ['main.py', 'arg1', 'arg2', 'arg3']
```

Handling command line arguments (Continued)

There are some Python standard library modules which greatly simply handling command line arguments

Getopt:
This module provides a C stype getopt() like API's to parse command line arguments (https://docs.python.org/2/library/getopt.html)

Eg:

```
import getopt, sys

try:
    opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "output="])
except getopt.GetoptError as err:
    print str(err)  # will print something like "option -a not recognized"
    sys.exit(2)

output = None
verbose = False
for o, a in opts:
   .....
```

Handling command line arguments (Continued)

Argparse:
This is another module for command line argument parsing. It requires writing much lesser code to extract the argumets
(https://docs.python.org/2/library/argparse.html)

Eg:

```
import argparse

parser = argparse.ArgumentParser(description='Description of your program')
parser.add_argument('-f','--foo', help='Description for foo argument', required=True)
parser.add_argument('-b','--bar', help='Description for bar argument', required=True)
args = vars(parser.parse_args())

print args['foo']
print args['bar']
```

Exercise 1 (suggested time between 2 to 3 hours)

- Installing Python and setting up the environment

Use Window or Linux desktop system to try the python installation and environment setup

- Executing commands in Python interpreter

Run simple command like print or assigning values to variables and then printing them.

- Arithmatic operations

```
>>> x=3
>>> y=2
>>> z=7
>>> x*y*z
42
>>> a=x*y*z
>>> print a
42
```

- Arithmatic operations

```
>>> (2+4+6)*3-12/3
```

- String Concatnation

```
>>> "spam"
'spam'
>>> "spam," + " " + "eggs and spam"
'spam, eggs and spam'
>>>
```

Exercise 1 continued (suggested time between 2 to 3 hours)

Writing some simple python programs

1. Write a Python program to read user input from console and print the string
2. Write a Python program to calculate area of a circle
   - Read radius from the console (Hint: conversion to different data type needed)
   - Calculate the area (Hint: use of exponent operator)
3. Write a Python program to add two positive integers without using the '+' operator.

   Hint: Use binary operators
4. Write a Python program to display the first and last colors from the following list.

   color_list = ["Red","Green","White" ,"Black"]
5. Write a Python program to check whether a specified value is contained in a group of values.

   Test Data: 3 -> [1, 5, 8, 3] : True

   -1 -> [1, 5, 8, 3] : False
6. Accept command line arguments of different types, extract them from the list and print.

Thank you

01/07/18

Session 2 (1): Decision making and Loops
Session 2(2): Nums, String, List, Tuples, Dicts

Duration: 2 hours

01/07/18

Python decision making statements

Python assumes any non-zero or non-null value as TRUE while zero or null value as false. It provides following types of decision making statements.

- if statement:

This statement consiste of an expression which if evalued to TRUE, then the following statements are executed

- if else statement:

If the 'if' statement is followed by an optional else statement, then if the expression evaluates to FALSE then the statement following else are executed

- if elif else statement:

In this case elif stands for 'else if'. This is used for evaluating different expressions sequentially. This is similar to using switch/case statements.

- Nested if statements:

These are combinations of if, elif and else statement without another if/else block

Python decision making statements examples

Following are an examples of if, elif and else statements

```
if x < 0:
    print “x is less than zero”
elif x == 0:
    print “x is zero”
elif x == 1:
    print “x is one”
else:
    print “x is greater than one”
```

Python loop statements (for)

Python programming language provides following types of loops to handle iterative requirements.

for loop:

- The for loop on python is different than those in C or C++.
- In C and C++ the for loop only accepts arithmatic progressions for iterating.
- The for loop is python accepts any items in a sequence, in the order in which they appear in the sequence.
- The items could be part of a list or a string.

Python loop statements

It it is required to iterate through a sequence of number then Python provides a built-in function called **range()**

This function automatically generates an arithmatic progression list.

For example:

```
>>> range(5)
[0, 1, 2, 3, 4]
>>>
```

In this case a range(5) function automatically generated a list containing numbers from 0 to 4.

Using this in for loop iteration will be similar to the for loops in C or C++

Python loop statements (for)

Following are some example of using for loop

Using any list object

```
>>> words = ['test', 'list', 'containing', 'arbitary', 'words']
>>> for w in words:
...     print w
...
test
list
containing
arbitary
words
>>>
```

Here we can see the for loop printed one word from the list in each iteration

Python loop statements (for)

Following are some example of using for loop

Using range() function

```
>>> words = ['test', 'list', 'containing', 'arbitary', 'words']
>>> for indx in range(len(words)):
...     print 'Word at indx {} is {}'.format(indx, words[indx])
...
Word at indx 0 is test
Word at indx 1 is list
Word at indx 2 is containing
Word at indx 3 is arbitary
Word at indx 4 is words
```

Here we can see the for loop interated on an arithmatic index from 0 to 4. This aritmatic index is obtained using the range() funcion on the length on the list.

Python loop statements (while)

Python also supports traditional while loops for iterative operations. It will continue executing the statements as long as the expression provided in while evaluated to true.

Eg:

```
>>> count = 0
>>> while count < 5:
...     print count
...     count = count+1
...
0
1
2
3
4
>>>
```

Python break, continue and pass statements

Break statement:

In python the behaviour of the break statement is similar to that in C. It will cause the execution to break out of the innermost loop. The loop terminates and execution transfers to the statement immediately following the loop.

Continue statement:

The continue statement causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

Pass statement:

The pass statement actually does nothing. It is used when some statement is requird syntactically but no actual action is expected.

Python break, continue and pass statement examples

Example of continue statement

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print "Number {} is
even".format(num)
...         continue
...     print "Number is {}".format(num)
...
Number 2 is even
Number is 3
Number 4 is even
Number is 5
Number 6 is even
Number is 7
Number 8 is even
Number is 9
>>>
```

Example of break statement

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print "Number {} is
even".format(num)
...         break
...     print "Number is {}".format(num)
...
Number 2 is even
>>>
```

Example of pass statement

```
>>> while 1:
...     pass
...

^CTraceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
>>>
```

Python else clauses for loops

Python loop statements may also have an else clause. It is executed when the loop terminates through exhaustion of the list (with for) or when the condition becomes false (with while). But it is not executed when the loop is terminated by a break statement. Following is an example

With else clause

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         print(n, 'is a prime number')
...
(2, 'is a prime number')
(3, 'is a prime number')
(4, 'equals', 2, '*', 2)
(5, 'is a prime number')
(6, 'equals', 2, '*', 3)
(7, 'is a prime number')
(8, 'equals', 2, '*', 4)
(9, 'equals', 3, '*', 3)
>>>
```

Without else clause

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     print(n, 'is a prime number')
...
(2, 'is a prime number')
(3, 'is a prime number')
(4, 'equals', 2, '*', 2)
(4, 'is a prime number')
(5, 'is a prime number')
(6, 'equals', 2, '*', 3)
(6, 'is a prime number')
(7, 'is a prime number')
(8, 'equals', 2, '*', 4)
(8, 'is a prime number')
(9, 'equals', 3, '*', 3)
(9, 'is a prime number')
```

Python number manipulation

Earlier we have seen the numeric data types. These are used to store numbers. They are immutable data types, means that changing the value of a number data type results in a newly allocated object.

Following is an example:

```
>>> myvar = 1
>>> id(myvar)
163801264
>>> myvar = 2
>>> id(myvar)
163801252
>>>
```

Where we can see, the memory address refered by myvar has changed after it was assigned a different value. (id() function returns the memory address corresponding to the variable)

The older value no longer refers to any variable and will be automaticallycleared by python garbage collection

Python number manipulation with built-in functions

Type conversion built-in function:

- int(x) to convert x to a plain integer.
- long(x) to convert x to a long integer.
- float(x) to convert x to a floating-point number.
- complex(x) to convert x to a complex number with real part x and imaginary part zero.
- complex(x, y) to convert x and y to a complex number with real part x and imaginary part y.

  x and y are numeric expressions

Following are two built-in numeric constants:

Pi

```
>>> from math import pi
>>> print pi
3.14159265359
>>>
```

exp

```
>>> from math import e
>>> print e
2.71828182846
>>>
```

Python number manipulation with built-in functions

Commonly used Mathematical built-in function:

- abs(x) - The absolute value of x
- cmp(x, y) - Returns -1 if x < y, 0 if x == y, or 1 if x > y
- max(x1, x2,...) The largest of its arguments: the value closest to positive infinity
- min(x1, x2,...) The smallest of its arguments: the value closest to negative infinity
- pow(x, y) The value of x**y.
- sqrt(x) The square root of x for x > 0
- modf(x) The fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float.

Other than these there are some more function like:

exp(x), ceil(x), floor(x), round(x [,n]), log(x), log10(x)

Trigonometric built-in Functions:

sin(x), cos(x), tan(x), degrees(x), radians(x), asin(x), acos(x), atan(x), etc

Python String manipulation

Strings are by far the most commonly used data types in Python. Simple way to assign a string to a variable is by enclosing the text in single of double quitoes. Triple quotes “”” can be used for initializing multiline verbatim strings.

Eg:

```
>>> str = "Hello"
>>> str
'Hello'
>>>

>>> str1 = 'How are you'
>>> str1
'How are you'
>>>

>>> str2 = """This is a comment
...       text, with tabs, spaces and
...       new line characters"""
>>> str2
'This is a comment\n     text, with tabs, spaces and\n     new line characters'
```

Python String manipulation using special operators

Following are some of the most commonly used string operators:

- % operator:

This is used mainly as a format specifier and is used to concatenate the string with the corresponding variable values. This works exactly like the format specifiers in C (printf)

Eg:

```
>>> print "My name is %s and my employee ID is %d" % ('Sachin', 123456)
My name is Sachin and my employee ID is 123456
>>>
```

Note:

In later versions of python there is a built-in function available for string formatting. It is str.format() and has a syntax {} .format(value1). The same example mentioned above using the format function is:

Eg:

```
>>> print "My name is {} and my employee ID is {}".format('Sachin', 123456)
My name is Sachin and my employee ID is 123456
>>>
```

Python String manipulation using special operators

Other than %, there are several other operators which are used with strings

- **Concatenation:** +, Eg: `"Hello" + "World"` will return `HelloWorld`
- **Repetation:** *, Eg: `"Hello" * 2` will return `HelloHello`
- **Slice:** [], this returns the character at the given index. Eg: str = “Hello”, then str[1] will give 'e'
- **Range slice:** [:], This will return characters from the given range. Eg: str[1:4] will give 'ell'
- **Membership: in:** This will return true if the character or string is present in the given string.

  Eg: `'l' in "Hello"` or `'ll' in "Hello"` will retrun true
- **Membership not in:** This will return true if the character of string is **not** present in the given string.

  Eg: `'q' not in "Hello"` or `'abc' not in "Hello"` will retrun true
- **Raw string: r/R:** This operator is used to suppress the actual meaning of escape characters

  Eg:

```
>>> print r'\n'
\n
>>>
```

Python String manipulation using built-in methods

Following is a list of some commonly useful built-in string functions:

isalnum( )	Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
isalpha( )	Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
isdigit( )	Returns true if string contains only digits and false otherwise.
isnumeric( )	Returns true if a unicode string contains only numeric characters and false otherwise.
isspace( )	Returns true if string contains only whitespace characters and false otherwise.
isupper( )	Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise
islower( )	Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
join(seq)	Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.
len( )	Returns the length of the string
lower( )/upper( )	Converts all uppercase letters in string to lowercase or all lower case to uppercase
min()/max()	Returns the min/max alphabetical character from the string str.
replace(old, new [, max])	Replaces all occurrences of old in string with new or at most max occurrences if max given.

Python String manipulation using built-in methods

Following is a list of some more built-in string functions:

find(str, beg=0 end=len(string))	Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.
count(str, beg= 0,end=len(string))	Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.
lstrip()/rstrip()/strip	Remove leading/trailing white spaces or strip both leading and trailing spaces
ljust/rjust(width[, fillchar])	Returns a space-padded string with the original string left/right-justified to a total of width columns.
decode(encoding='UTF-8',errors='strict')	Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.
encode(encoding='UTF-8',errors='strict')	Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.

There are some more string function which are uncommon so not included in the list. Python document can be referred to get more information on those.

Python Lists

- The list is a most versatile datatype available in Python which can be written as a list of comma separated values (items) between square brackets.
- Important thing about a list is that items in a list need not be of the same type.
- Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

Example of List

```
list1 = ['python', 'list example', 2, 20];

list2 = [1, 2, 3, 4, 5 ];

list3 = ["a", "b", "c", "d"]
```

Common operations on Lists

```
>>> list1 = ['python', 'list example', 2, 20];
>>> print "list1[1]: ", list1[1]
list1[1]:  list example
>>> list2 = [1, 2, 3, 4, 5, 6, 7 ];
>>> print "list2[1:5]: ", list2[1:5]
list2[1:5]:  [2, 3, 4, 5]
```

Python Lists

Following is an example for updating a list

```
>>> list1 = ['python', 'list example', 2, 20];
>>> print list1
['python', 'list example', 2, 20]
>>> list1[1] = 2
>>> list1[2] = 10
>>> list1[3] = "New string"
>>> print list1
['python', 2, 10, 'New string']
>>>
```

Here we can see that the individual items in the list could be updated. Also the type of the updated items was also different that the original item types.

Python Lists

Following is an example of appending and deleting an item in the list

```
>>> list1 = ['python', 'list example', 2, 20];
>>> print list1
['python', 'list example', 2, 20]
>>> list1 = list1 + ["New string"]
>>> print list1
['python', 'list example', 2, 20, 'New string']
```

Following is an example for deleting an item from a list (considering the same list as above)

```
>>> del(list1[1])
>>> print list1
['python', 2, 20, 'New string']
>>>
```

We can see the 1st index corresponding to item 'list example' is now deleted

Python Lists operators

Following are basic list operators:

Concatenation (+): Concatenate two lists

Repetation (*): Repeat the same item in the list

Membership (in): Returns true if the item is in the list

Iteration: This is an internal operator used on for loops.

Indexing: The items of the list can be accessed using its positional index. Negative index would count the list items from the end.

Slicing: [:] operator can be used to slice the list, same as with strings

Following are some built-in functions which operate on the lists:

cmp(list1, list2) - Compares elements of both lists

max/min(list) - Returns item from the list with max/min value.

len(list) - Return the length of the string

Python Lists methods

Following are the methods available with list objects:

`list.append(obj)`	This will append an object obj to list. Similar to '+'
`list.count(obj)`	Returns count of how many times obj occurs in list. Similar to len()
`list.extend(seq)`	This will appends the contents of seq to list
`list.index(obj)`	Returns the lowest index in list that obj appears
`list.insert(index, obj)`	Inserts object obj into list at offset index
`list.pop(obj=list[-1])`	Removes and returns last object or obj from list
`list.remove(obj)`	Removes object obj from list
`list.reverse()`	Reverses objects of list in place
`list.sort([func])`	Sorts objects of list, use compare func if given

Python Tuples

A tuple is a sequence of immutable Python objects. They are similar to lists but they cannot be changed. Unlike square brackets in case of lists, the tuples are represented with parentheses.

Eg:

```
>>> tup1 = ('Hello', 'python', 2.7, 3.0);
>>> tup2 = (1, 2, 3, 4, 5 );
>>> tup3 = "a", "b", "c"
>>> print tup3
('a', 'b', 'c')
```

NOTE: A comma separated set of values can also be used to initialize a tuple. The parentheses can be dropped.

Python Tuples – Updating and deleting

Tuples are immutable, which means the actual values of the tuple elements cannot be change.

However portions of existing tuples can be extracted to create new tuples.

Eg:

```
>>> tup1 = ("a", "b", "c", "d", "e");
>>> tup2 = (1, 2, 3, 4, 5 );
>>> tup3 = tup1 + tup2
>>> print tup3
('a', 'b', 'c', 'd', 'e', 1, 2, 3, 4, 5)
>>>
```

Deleting an individual tuple element is not possible, however entire tuple can be deleted.

Eg: Deleting the previously created tup3

```
>>> del tup3
>>> print tup3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'tup3' is not defined
>>>
```

Python Tuples – Operators

The supported operators same as that of python Lists

Concatenation (+): Concatenate two lists

Repetation (*): Repeat the same item in the list

Membership (in): Returns true if the item is in the list

Iteration: This is an internal operator used on for loops.

Indexing: The items of the list can be accessed using its positional index. Negative index would count the list items from the end.

Slicing: [:] operator can be used to slice the list, same as with strings

Following are some built-in functions which operate on the lists:

cmp(list1, list2) - Compares elements of both lists

max/min(list) - Returns item from the list with max/min value.

len(list) - Return the length of the string

Python Dictionaries

Dictionary in python is essentially a list of key value pairs. Unlike lists and tuples, dicts are enclosed in curly brackets.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Eg:

```
>>> dict = {'Name': 'Sachin', 'Surname': 'Deodhar', 'Id': 123456}
>>> dict
{'Surname': 'Deodhar', 'Name': 'Sachin', 'Id': 123456}
>>> dict['Name']
'Sachin'
>>> dict['Id']
123456
>>>
```

Python Dictionaries – Updating the dict entry

An empty dictionary can be created using the following command

```
>>> mydict = {}
>>> print mydict
{}
>>>
```

Key value pairs can be added to this dictionary using the following commands.

```
>>> mydict['Name'] = "Sachin"
>>> mydict['Surname'] = "Deodhar"
>>> mydict['Id'] = 123456
>>> mydict
{'Surname': 'Deodhar', 'Name': 'Sachin', 'Id': 123456}
>>>
```

Similary existing entry can be modified using

```
>>> mydict['Id'] = 987654
>>> mydict
{'Surname': 'Deodhar', 'Name': 'Sachin', 'Id': 987654}

>>>
```

Python Dictionaries – Deleting dictionary elements

It is possible to delete individual entries from the dictionary

```
>>> mydict
{'Surname': 'Deodhar', 'Name': 'Sachin', 'Id': 987654}
>>> del mydict['Id']
>>> mydict
{'Surname': 'Deodhar', 'Name': 'Sachin'}
>>>
```

It is also possible to clear all entries from the dictionary

```
>>> mydict
{'Surname': 'Deodhar', 'Name': 'Sachin', 'Id': 987654}
>>> mydict.clear()
>>> mydict
{}
>>>
```

Entire dictionary itself can also be deleted

```
>>> del(mydict)
>>> mydict
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'mydict' is not defined
>>>
```

Python Dictionaries – Keys and built-in functions

- The dictonary keys need to be unique.

- When duplicate keys encountered during assignment, the last assignment wins

- Keys must be immutable. Which means you can use strings, numbers or tuples as dictionary keys

Built-in dictionary functions

`cmp(dict1, dict2)`	Compares elements of both dict.
`len(dict)`	Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.
`str(dict)`	Produces a printable string representation of a dictionary
`type(variable)`	Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.

Python Dictionaries – methods

Python includes following dictionary methods

`dict.clear()`	Removes all elements of dictionary dict
`dict.copy()`	Returns a shallow copy of dictionary dict
`dict.fromkeys()`	Create a new dictionary with keys from seq and values set to value.
`dict.get(key, default=None)`	For key key, returns value or default if key not in dictionary
`dict.has_key(key)`	Returns true if key in dictionary dict, false otherwise
`dict.items()`	Returns a list of dict's (key, value) tuple pairs
`dict.keys()`	Returns list of dictionary dict's keys
`dict.setdefault(key, default=None)`	Similar to get(), but will set dict[key]=default if key is not already in dict
`dict.update(dict2)`	Adds dictionary dict2's key-values pairs to dict
`dict.values()`	Returns list of dictionary dict's values

Exercise 2: (suggested time 3 hours)

1. Write a Python script to calculate length of a given string

2. Write a Python script to count the number of occurances of a given character in the string.
   Accept the character with -c command line argument and the input string with -i command line argument
   Eg:
      Command: myscript.py -i "My string is this" -c "s"
      Output: The occurance of [s] in the given string is [3] times

3. Write a Python script to swap two words in the give string
   Eg:
      Input String: "Hello World"
      Output String: "World Hello"

4. Write a Python program to count occurrences of a substring in a string
   Eg:
      Input String: "This is a sample text. Use this text for testing your sample script"
      Count the occurences of words: sample, text and string.

Exercise 2 (Continued): (suggested time 3 hours)

1. Write a program to iterate through a list and print every item in the list.

2. Write a Python program to find the index of an item in a specified list.
   Eg: mylist = ['red', 'green', 'blue', 'yellow', 'black']
   Output should return the index for item 'blue'

3. Write a Python program to check a list is empty or not.
   Eg: If mylist = []
   the it should be true, else it should be false

4. Write a Python program to remove duplicates from a list.

5. Write a program to replace the last item of the list with another list
   Eg: Input is mylist1 = [1,2,3,4,5], mylist2 = [5,6,7,8,9]
   The output should be [1,2,3,4,5,6,7,8,9]

Exercise 2 (Continued): (suggested time 3 hours)

1. Write a Python script to add new key,value pairs to an empty dictionary.
   Eg: Start with myDict = {}
   Then add key, value pairs like (1,red), (2, green), (3, blue)

2. Write a Python script to concatenate following dictionaries to create a new one.
   Eg: myDict1 = {1:red, 2:green, 3:blue}
   myDict2 = {4:yello, 5: green}
   Expected output = {1:red, 2:green, 3:blue, 4:yello, 5: green}

3. Write a program to add multiple values to a key.
   Eg: a = {}
   key = "somekey"
   a.setdefault(key, [])
   a[key].append(1)

4. Write a Python program to convert a list to a tuple

Thank you

01/07/18

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
```

Exercise 3: (Suggested time 3 hours)

1. Write a Python function to find the Max from a given list of numbers.

2. Write a Python function to calculate the factorial of a number (a non-negative integer).
   The function accepts the number as an argument. (Hint: Recursive function)

3. Write a Python function that takes a list and returns a new list with unique elements of the first list.
   Sample List : [1,2,3,3,3,3,4,5]
   Unique List : [1, 2, 3, 4, 5]

4. Write a program to print the documentation content mentioned at the begining of the function
   Eg:

```
def my_add(arg1, arg2):
    "This is the document for my_add"
    result = arg1 + arg2
    print "arithmatic: add = {}".format(result)
    return result
```

   The program should print the line “This is the document for my_add”
   Hint: Use the function's internal variable __doc__.
   Eg: my_add.__doc__

Exercise 3: (Continued, Suggested time 3 hours)

Create two sample module named 'arithmatic', 'minmax' and 'sq_sqrt'
- Arithmatic module must contains your implementations for add, sub, mul and div
- Minmax module must contain your implementations for min and max
- sq_sqrt module must contain your implementations for square and sqrt functions

1. Try importing these modules into you code
2. Import either the entire module or only required functions

Create a sample package named 'my_math' using these modules

1. Create the __init__.py file for the package
2. Use the package in your program

Exercise 3: (Continued, Suggested time 3 hours)

The following functions are not robust, and does not handle any exceptions. Modify them to handle the required exceptions.

```
def example1():
    for i in range( 3 ):
        x = int( input( "enter a number: " ) )
        y = int( input( "enter another number: " ) )
        print( x, '/', y, '=', x/y )
```

Hint test conditions: What will happen if non numeric type is given as input or one of the input is 0

```
def printUpperFile( fileName ):
   file = open( fileName, "r" )
   for line in file:
       print( line.upper() )
   file.close()
```

Hint test conditions: What happens if the file is not present or if the file is present but the user does not have read permission. What happens if the file is empty.

Thank you

01/07/18

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
```

Classes methods in Python

Following are the type of class methods:

Instance Method:
This is a regular class method which we use most of the times. It takes first parameter as self, which points to the instance of the class.

Class Method:
This is a special method, which is bound to a class rather than an object. It does not accept self instance, instead its first argument points to the class and not the object. This method cannot modify object instance state.
This method is declared with decorator flag @classmethod

Static Method:
This is another special method which is also bound to a class rather than the object. Howver it does not require the class or instance in the argument. It knows nothing about the class as such and only deals with the parameters.
This method is declared with decorator flag @staticmethod

Note:
Other than the mandatory first argument (for instance method and class method), all these methods accept any number of following arguments.

Classes methods in Python

classmethod:

Class methods are typically used as factory methods, to create specific objects of the said class.
Following is an example:

```
class Furniture:
    def __init__(self, name):
        self.name = name
    def getFurnitureName(self):
        print "Furniture Name is {}".format(self.name)
    @classmethod
    def chair(cls):
        return cls("chair")
    @classmethod
    def table(cls):
        return cls("table")

>>> c = Furniture.chair()
>>> c.getFurnitureName()
Furniture Name is chair
>>> t = Furniture.table()
>>> t.getFurnitureName()
Furniture Name is table
```

Classes methods in Python

staticmethod:
Static methods are typically used for group the implementation of helper or utility functions. They do not have access to the class or its object. They only operate on the parameters passed to the function. Following is an example:

```
class Dates:
    def __init__(self, date):
        self.date = date
    def getDate(self):
        return self.date
    @staticmethod
    def toDashDate(date):
        return date.replace("/", "-")

class DatesWithSlashes(Dates):
    def getDate(self):
        return Dates.toDashDate(self.date)

date = Dates("15-12-2016")
dateFromDB = DatesWithSlashes("15/12/2016")
if(date.getDate() == dateFromDB.getDate()):
    print("Equal")
```

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.

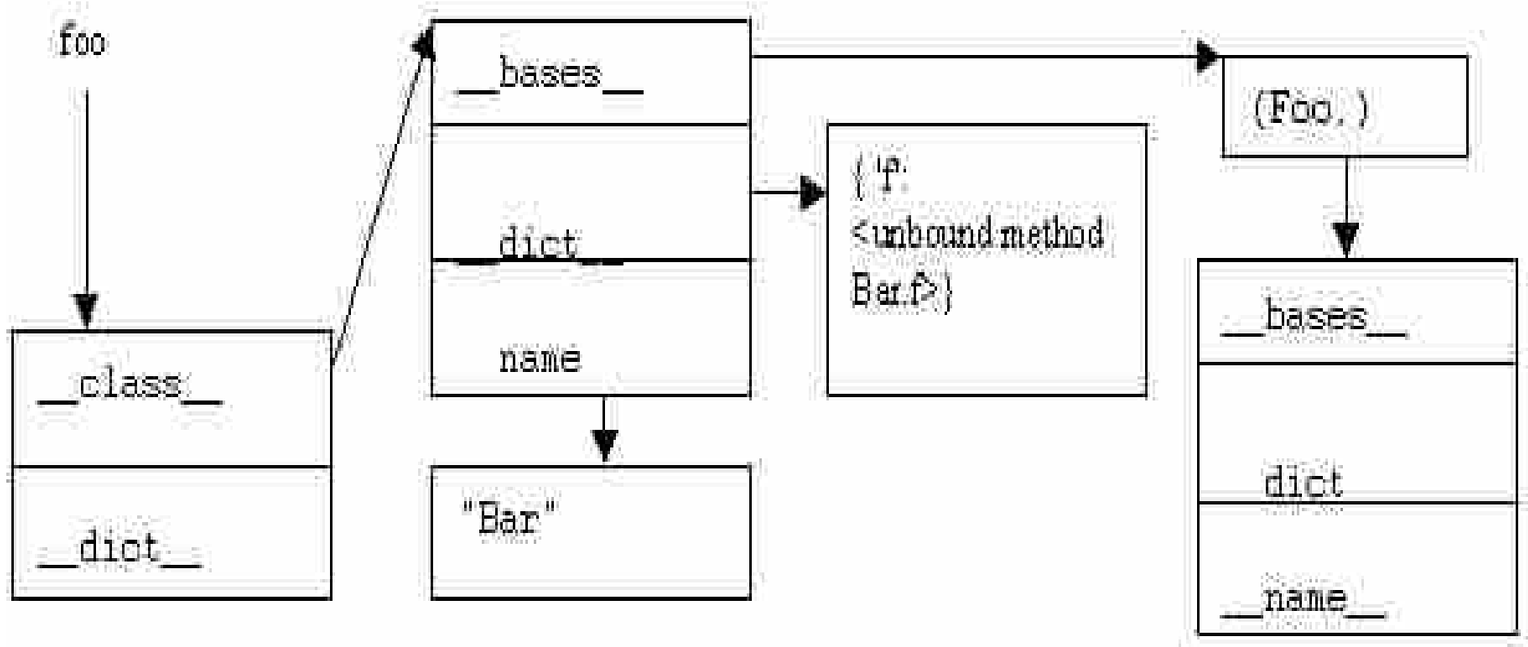
Polymorphism in Python

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

Exercise 4: (Suggested time 3 hours)

This exercise involves creating a sample remote logging server and logging client applications.

Requirement

1. On the server side we need to wait for the client to connect.
2. Client should read data from kivi.log file line by line and send it to the server.
3. Server should write the data received from the client line by line to the log file.
4. The name of the file should identify which client the data was received from.

Implementation suggestions:

1. Create a Server class for network handling on server side
2. Create a Client class for network handling on client side
3. Create a Logging class to read and write the log file
4. Use the read method of the Logging class to read file in client
5. Use the write method of the Logging class to write file in server

Exercise 4: (Suggested time 3 hours)

You can create your client and server classes by extending the standard socket.socket class provided by Python
Eg:

```
class MySocketServer(socket.socket):
    def __init__(self):
        print "Init Server"

class MySocketClient(socket.socket):
    def __init__(self):
        print "Init Client"
```

Note:
This excerise will be a prerequisite for exercise 5 for the next session. In that exercise we will cover handling of multiple connections and processing of the log data on the server side.

Thank you

01/07/18

Session 5: Threading, Regex handling

Duration: 1.5 hours

01/07/18

Multi-Threading in Python

A Thread or a Thread of Execution is defined as the smallest unit that can be scheduled in an operating system. More than one thread can exist within the same process. These threads share the memory and the state of the process.

Advantages of using multiple threads:

1. Multithreaded programs can run faster on computer systems with multiple CPUs, because theses threads can be executed truly concurrent.
2. A program can remain responsive to input. This is true both on single and on multiple CPU
3. Threads of a process can share the memory of global variables. So it is easier to share data among different threads.

In Python there are two modules which can be used to implement multi-threading.

1. Thread module:
This modules implements threading as a function. So a thread corresponds to a function.

2. Threading module:
In this module the threading is implemented via an object. So each thread corresponds to an object.

Multi-Threading in Python

The 'thread' module:

It's possible to execute functions in a separate thread with the module Thread. To do this, we can use the function thread.start_new_thread.

Eg:

```
thread.start_new_thread(function, args[, kwargs])
```

- This method starts a new thread and return its identifier.

- The thread executes the function "function" (function is a reference to a function) with the argument list args which must be a list or a tuple.

- The optional kwargs argument specifies a dictionary or keyword arguments.

- When the function returns, the thread silently exits.

- If the function terminates with an unhandled exception, a stack trace is printed and then the thread exits

Multi-Threading in Python

Example using the 'thread' module:

```
import thread
import time

def print_line( threadName, delay):
   count = 0
   while count < 5:
      count += 1
      print "{}: {}".format(threadName, count)
      time.sleep(delay)

try:
   thread.start_new_thread( print_line, ("Thread 1", 2, ) )
   thread.start_new_thread( print_line, ("Thread 2", 4, ) )
except:
   print "Error: unable to start thread"

while 1:
   pass
```

Multi-Threading in Python

Following are the methods that are provided by the thread class.

Methods:

thread.start_new_thread(function, args[, kwargs]) – Start a new thread, already covered earlier.

thread.interrupt_main() - Raises a keyboardInterrupt exception in main thread. A sub thread can use this function to interrupt the main thread.

thread.exit() - Raises the SystemExit exception. When not caught will cause the thread to exit silently

thread.allocate_lock() - Returns a new lock object

thread.get_ident() - Returns the identifier of the current thread.

thread.stack_size([size]) – Returns the stack size used when creating new threads. The size argument specifies the size. Size 0 uses the platform default.

Multi-Threading in Python

Following are the methods that are provided by the thread.lock object. thread.allocate_lock() will returns a new lock object

Methods:

lock.acquire([waitflag]) -

Without the optional argument, this method acquires the lock unconditionally. If the integer waitflag argument is present, the action depends on its value: if it is zero, the lock is only acquired if it can be acquired immediately without waiting, while if it is nonzero, the lock is acquired unconditionally as before. The return value is True if the lock is acquired successfully, False if not.

lock.release() - Releases the lock. The lock must have been acquired earlier, but not necessarily by the same thread.

lock.locked() - Return the status of the lock: True if it has been acquired by some thread, False if not.

Multi-Threading in Python

Synchronization between threads:
If multiple threads are sharing resources, then it is essential to provide some synchronization mechanism while accessing such resources.

Python's thread module provides this mechanism via the lock implementaion.

New lock object can be created using the following command:

```
from thread import allocate_lock
lock = allocate_lock()
```

To commands used to aquire and release the lock are:

```
lock.acquire()
#some critical section code which we want to guard.
lock.release()
```

NOTE: Older versions of Python included a separate **mutex** module. However it has been deprecated since version 2.6 and has been entirely removed since verion 3 of python.

Multi-Threading in Python

The 'threading' module:

Unlike the thread module where the thread is just represented by a function, in case of threading module, it corresponds to an object of threading class.

Thread can be created using the following function:

```
threading.Thread(group=None, target=None, name=None, args=(), kwargs={})
```

Where:

- Group: should be None, its reserved for future use
- Target: is the callable object to be invoked by the internal run() menthod.
- Name: is the thread name, if not specified a default unique name is contructed
- Args: is the argument tuple which is passed to the callable object
- Kwargs: is a dictionary of keyword arguments passed to the callable object

NOTE:
In most cases instead of directly creating the threads, it is a practice to inherit from the threading.Thread class. This method allows the programmer to overide the default run() methods.

Multi-Threading in Python

Following is an example of creating threads using the threading module.

```
import threading

def print_one(num):
    print("Num {}".format(num))

def print_two(num1 = 0, num2 = 0):
    print("Num1 {}, Num2 {}".format(num1,num2))

t1 = threading.Thread(target=print_one, args=(10,))
t2 = threading.Thread(target=print_two, kwargs={'num1': 1,'num2':2})

t1.start()
t2.start()

t1.join()
t2.join()

print("Done!")
```

Multi-Threading in Python

Following are the methods available with the threading.Thread object

start() - Starts the thread's activity. It must be called at most once per thread object. Calling it multiple times would raise a RuntimeError exception

run() - This methods represents the actual thread activity. It invokes the callable object passed to the target argument of the thread. The subclass can override this method

joint(timeout) – Waits until the thread terminates or exits or untill the timeout occurs.

getName/setName – These APIs are used to get or set the thread name.

isAlive – This method returns true if the thread is alive.

isDaemon/setDaemon – These APIs are used to check if the thread is a daemon thread or to set it. The setDaemon must be called before start()

Multi-Threading in Python

Following are some of the methods available with the threading module

threading.activeCount() - Return the number of thread objects alive at the given time

threading.Condition() - Returns new condition variable object

threading.currentThread() - Returns the curretn Thread object

threading.enumerate() - Returns list of all Thread objects currently alive

threading.Event() - Returns a new Event object

threading.Lock() - Returns a new Lock object

threading.Rlock() - Returns a new reentrant lock object

threading.Semaphore([value]) – Returns a new semaphore object

threading.BoundedSemaphore([value]) - Returns a new bounded semaphore object

threading.settrace(func) - Set a trace function for all threads started from the threading module

threading.setprofile(func) - Set a profile function for all threads started from the threading module

threading.stack_size([size]) - Return the thread stack size used when creating new threads

Multi-Threading in Python

The Lock Objects

A primitive lock is in one of two states, “locked” or “unlocked”. It is created in the unlocked state. It has two basic methods, acquire() and release().

Lock.acquire([blocking]) – This can be called to acquire the lock in blocking or non-blocking way

When invoked with blocking =True, the caller will block in acquire untill the lock is acquired.

When unvoked with blocking = False, the caller will return False immediately if it cannot acquire.

Lock.release() - Release a lock. When invoked on an unlocked lock, a ThreadError is raised.

The Rlock Objects

A reentrant lock is a synchronization primitive that may be acquired multiple times by the same thread. Internally, it uses the concepts of “owning thread” and “recursion level” in addition to the locked/unlocked state used by primitive locks. In the locked state, some thread owns the lock; in the unlocked state, no thread owns it.

It also provides the acquire() and release calls

Multi-Threading in Python

The Condition Objects
- A condition variable is always associated with some kind of lock.
- A condition variable has acquire() and release() methods that call the corresponding methods of the associated lock.
- It also has a wait() method, notify() and notifyAll() methods. These three must only be called when the calling thread has acquired the lock, otherwise a RuntimeError is raised.

Eg:

```
# Consume one item
cv.acquire()
while not an_item_is_available():
    cv.wait()
get_an_available_item()
cv.release()

# Produce one item
cv.acquire()
make_an_item_available()
cv.notify()
cv.release()
```

Multi-Threading in Python

User the Semaphore object:

A semaphore manages an internal counter which is decremented by each acquire() call and incremented by each release() call. The counter can never go below zero; when acquire() finds that it is zero, it blocks, waiting until some other thread calls release().

Methods:

class threading.Semaphore([value]) – Create a semaphore object. If value is not specified, default is 1

acquire([blocking]) – This methods decrements the internal counter by 1. If it is already zero then it blocks waiting for it to become greater than zero.

release() - This methods increments the internal counter. In case of bounded semaphore it prevents it from increasing above the specified initial value.

Multi-Threading in Python

Using the Event objects

Event:
This is the simplest mechanisms for communication between threads. One thread signals an event and other threads wait for it. An event object manages an internal flag that can be set to true with the set() method and reset to false with the clear() method. The wait() method blocks until the flag is true.

Methods:

isSet() – Returns true if the internal flag is true

set() - Sets the internal flag to true

clear() - Clears the internal flag to false

wait([timeout]) – Blocks untill the internal flag is true or until the timeout occurs.

Multi-Threading in Python

Using the Timer objects

Timer:
This class represents an action that should be run only after a certain amount of time has passed. Timer is a subclass of Thread and as such also functions as an example of creating custom threads.

Timers are started, as with threads, by calling their start() method. The timer can be stopped (before its action has begun) by calling the cancel() method.

Methods:

threading.Timer(interval, function, args=[], kwargs={}) – Create a time that will run the function.

Cancel() - Stops the timer.

Example:

```
from threading import Timer
def hello():
    print "hello, world"

t = Timer(30.0, hello)
t.start()  # after 30 seconds, "hello, world" will be printed
```

Handling regular expressions in Python

A regular expression in Python is a sequence of characters, that defines a search pattern. The seach pattern can then be used to run a string-searching algorithm to “find” or “find and replace” on strings.

Metacharacters:

Python defines certain meta characters to have special meaning. Following is a list are some them.

```
^    - Matches the start of sting
.    - Matches a single character except newline. inside [.], means just dot.
[]   - Matches a single charater provided in the list, [abc] will match 'a', 'b' and 'c'
[^]  - Matches single character except from those in the list
()   - Parenthesis defines a subexpression or a block
\t, \n, \r, \f  - Tab, newline, return, form feed
*    - Matches preceeding character zero or more times
{m,n} - Matches the preceding character minimum m times, and maximum n times
{m}  - Matches the preceding character exactly m times
?    - Matches preceeding character zero or one time
+    - Matches preceeding character one or more times
|    - The choice operator matches either the expression before it, or one after
\w   - Matches a word character and \W matches non word characters
\s   - Matches a single whitespace character and \S non-whitespace characters
\    - Escape character
$    - Matches the end of string
```

Handling regular expressions in Python

Regex module in python can be used by importing re (`import re`)

Following are the set of rules used for matching the strings.

1. The search scans the string start to end.
2. The whole pattern must match, but not necessarily the whole string.
3. The search stops at the first match.

Following are the functions available to use with regex

re.match(pattern, string, flags=0) - This takes two arguments- a pattern and a string. If they match, it returns the match object else, it returns None

re.search(pattern, string, flags=0) - This function searches for first occurrence of RE pattern in string.

re.sub(pattern, repl, string, max=0) - This method replaces all occurrences of the RE pattern in string with repl, substituting all occurrences unless max provided. This returns modified string.

Handling regular expressions in Python

Example: re.match

```
>>> print(re.match('center','centre'))
None
>>> m = re.match('center','center')
>>> print m.group(0)
center
```

Example: re.search

```
>>> match=re.search(r'[\w.-]+@[\w-]+.[\w]+','Email me @ john.doe@mail.com')
>>> match.group(0)
'john.doe@mail.com'
>>>
```

Example: re.sub

```
phone = "2004-959-559 # This is Phone Number"
num1 = re.sub(r'#.*$', "", phone)
num2 = re.sub(r'\D', "", phone)
print "Phone Num : ", num1
print "Phone Num : ", num2
Output:
Phone Num :  2004-959-559
Phone Num :  2004959559
```

Thank you

01/07/18