Modern College of Arts, Science and Commerce,

Shivajinagar, Pune-5 (Autonomus)

T. Y. B. C. A. (Science) Semester-V

Python Programming Workbook

Name: _____

College Name: _____

Roll No.: _____ Division: _____

Academic Year: _____

Introduction

1. **About the work book:**
   This workbook is intended to be used by T.Y.B.C.A. (Science) students for the Python Assignments in Semester–V. This workbook is designed by considering all the practical concepts / topics mentioned in syllabus.

2. **The objectives of this workbook are:**
   1) Defining the scope of the course.
   2) To bring the uniformity in the practical conduction and implementation in all colleges affiliated to SPPU.
   3) To have continuous assessment of the course and students.
   4) Providing ready reference for the students during practical implementation.
   5) Provide more options to students so that they can have good practice before facing the examination.
   6) Catering to the demand of slow and fast learners and accordingly providing the practice assignments to them.

3. **How to use this workbook:**
   The Section-II (Python) is divided into 10 assignments. Each assignment has three SETs. It is mandatory for students to complete SET A and SET B in given slot.

2.1 Instructions to the students

Please read the following instructions carefully and follow them.

1) Students are expected to carry this book every time they come to the lab for practical.
2) Students should prepare oneself beforehand for the Assignment by reading the relevant material.
3) Instructor will specify which problems to solve in the lab during the allotted slot and student should complete them and get verified by the instructor. However student should spend additional hours in Lab and at home to cover as many problems as possible given in this work book.
4) Students will be assessed for each exercise on a scale from 0 to 5.

Not Done	0
Incomplete	1
Late Complete	2
Needs Improvement	3
Complete	4
Well Done	5

2.2 Instruction to the Instructors

1) Explain the assignment and related concepts in around ten minutes using white board if required or by demonstrating the software.
2) You should evaluate each assignment carried out by a student on a scale of 5 as specified above by ticking appropriate box.
3) The value should also be entered on assignment completion page of the respective Lab course.

Table of Contents for Section-II

Assignment Completion Sheet

Assignment No	Description	Marks (out of 2)
1	Basic Python	
2	Python Strings	
3	Python Tuple	
4	Python Set	
5	Python Dictionary	
6	Functions in Python	
7	Files	
8	Directories	
9	Python Classes and objects	
10	Exception Handling	

Assignment No.1 : Basic Python

Python is an interpreted high-level programming language for general-purpose programming. Python features a dynamic type system and automatic memory management. It supports multiple paradigms, including object-oriented, imperative, functional and procedural, and has a large and comprehensive standard library.

Starting the Interpreter

After installation, the python interpreter lives in the installed directory. By default it is /usr/local/bin/pythonX.X in Linux/Unix and C:\PythonXX in Windows, where the 'X' denotes the version number. To invoke it from the shell or the command prompt we need to add this location in the search path.
Search path is a list of directories (locations) where the operating system searches for executables. For example, in Windows command prompt, we can type set path=%path%;c:\python33 (python33 means version 3.3, it might be different in your case) to add the location to path for that particular session.

Python Use

Python is used by hundreds of thousands of programmers and is used in many places. Python has many standard library which is made up of many functions that come with Python when it is installed. On the Internet there are many other libraries available that make it possible for the Python language to do more things. These libraries make it a powerful language; it can do many different things.

Some things that Python is often used for are:

- Web development
- Game programming
- Desktop GUIs
- Scientific programming
- Network programming.

First Python Program

This is a small example of a Python program. It shows "Hello World!" on the screen.
Type the following code in any text editor or an IDE and save as *helloWorld.py*

```
print("Hello world!")
```

Now at the command window, go to the location of this file. You can use the cd command to change directory.
To run the script, type **python helloWorld.py** in the command window. We should be able to see the output as follows:
Hello world!
If you are using PyScripter, there is a green arrow button on top. Press that button or press Ctrl+F9 on your keyboard to run the program.
In this program we have used the built-in function **print()**, to print out a string to the screen. String is the value inside the quotation marks, i.e. Hello world!. Now try printing out your name by modifying this program.

1. Immediate/Interactive mode

Typing python in the command line will invoke the interpreter in immediate mode. We can directly type in Python expressions and press enter to get the output.

>>>

is the Python prompt. It tells us that the interpreter is ready for our input. Try typing in 1 + 1 and press enter. We get 2 as the output. This prompt can be used as a calculator. To exit this mode type exit() or quit() and press enter.

Type the following text at the Python prompt and press the Enter –

>>> print "Hello World"

2. Script Mode Programming

Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

Let us write a simple Python program in a script. Python files have extension **.py**. Type the following source code in a *test.py* file −

```
print"Hello World"
```

We assume that you have Python interpreter set in PATH variable. Now, try to run this program as follows −

```
$ python test.py
```

This produces the following result −

```
Hello, Python!
```

3. Integrated Development Environment (IDE)

We can use any text editing software to write a Python script file.

We just need to save it with the .py extension. But using an IDE can make our life a lot easier. IDE is a piece of software that provides useful features like code hinting, syntax highlighting and checking, file explorers etc. to the programmer for application development.

Using an IDE can get rid of redundant tasks and significantly decrease the time required for application development.

IDEL is a graphical user interface (GUI) that can be installed along with the Python programming language and is available from the official website.

We can also use other commercial or free IDE according to our preference, the PyScripter IDE can be used for testing. It is free and open source.

Python Comments

In Python, there are two ways to annotate your code.

Single-line comment – Comments are created simply by beginning a line with the hash (#) character, and they are automatically terminated by the end of line.
For example:

```
#This would be a comment in Python
```

Multi-line comment- Comments that span multiple lines – used to explain things in more detail – are created by adding a delimiter (“”””) on each end of the comment.

```
""" This would be a multiline comment
in Python that spans several lines and
describes your code, your day, or anything you want it to
...
"""
```

Indentation

The enforcement of indentation in Python makes the code look neat and clean.

For example:

```
if True:
    print('Hello')
    a=5
```

Incorrect indentation will result into IndentationError.

Standard Data Types

Python has five standard data types −

- Numbers
- String
- List
- Tuple
- Dictionary

Python Numbers: Integers, floating point numbers and complex numbers falls under Python numbers category. They are defined as int, float and complex class in Python.

Python supports four different numerical types −

- int (signed integers)
- long (long integers, they can also be represented in octal and hexadecimal)
- float (floating point real values)
- complex (complex numbers)

Python Strings : Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes.

Example: str=’Hello all’

Python Lists :

Lists are the most versatile of Python's compound data types. A list contains items can be of different data types separated by commas and enclosed within square brackets ([]).

```
list_obj=['table', 59 ,2.69,“chair”]
```

Python Tuples:

A tuple is another sequence immutable data type that is similar to the list. A tuple consists of a number of values separated by commas and enclosed in parentheses (( )).

```
Example:

tuple_obj=(786,2.23, “college” )
```

Python Dictionary

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs.

Dictionaries are enclosed by curly braces ({ })

```
Example:dict_obj={'roll_no': 15,'name':’xyz’,'per': 69.88}
```

Python Operators :

Python language supports the following types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators

- Identity Operators

Arithmetic, logical, Relational operators supported by Python language are same as other languages like C,C++.

i. **Arithmetic Operators**:

The new arithmetic operators in python are,

a) **** (Exponent)-** Performs exponential (power) calculation on operators

Example: a**b =10 to the power 20

b) **// (Floor Division) -** The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity)

Example: 9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0

ii. **Logical operators :**

Logical operators are the and, or, not operators.

a) and - True if both the operands are true
b) or - True if either of the operands is true
c) not - True if operand is false (complements the operand)

iii. **Relational / Comparison operators :**

== (equal to), != (not equal to), < (less than),<= (Less than or equal to), > (greater than) and >= (Greater than or equal to) are same as other language relational operators. The new relational operator in python is,

<>- If values of two operands are not equal, then condition becomes true.

Example: (a <> b) is true. This is similar to != operator.

iv. **Assignment Operators:** The following are assignment operators in python which are same as in C, C++.

=, +=, -=, *=, /=, %=, **=, //=

v. **Bitwise Operators:**The following are bitwise operators in python which are same as in C,C++.

&(bitwise AND), |(bitwise OR) ,^ (bitwise XOR),~ (bitwise NOT),<<(bitwise left shift), >>(bitwise right shift)

vi. **Membership operators** :

in and **not in** are the membership operators; used to test whether a value or variable is in a sequence.

in - True if value is found in the sequence
not in - True if value is not found in the sequence

vii. **Identity operators :**

is and **is not** are the identity operators both are used to check if two values are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

is - True if the operands are identical

is not - True if the operands are not identical

Decision making Statement

Python programming language provides following types of decision making statements.

i. **If statement**: It is similar to that of other languages

Syntax

```
if expression:
    statement(s)
```

ii. **IF...ELIF...ELSE Statements:**

Syntax

```
if expression:
    statement(s)
else:
    statement(s)
```

iii. **nested IF statements:**

In a nested **if** construct, you can have an **if...elif...else** construct inside another **if...elif...else** construct.

Syntax

```
if expression1:
    statement(s)
if expression2:
    statement(s)
elif expression3:
    statement(s)
else:
    statement(s)
elif expression4:
    statement(s)
else:
    statement(s)
```

Python – Loops

i. while loop:

A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

Syntax-

```
while expression:
    statement(s)
```

Example:

```
count=0
while(count <3):
    print'The count is:', count
    count= count +1
```

ii. for loop:

It has the ability to iterate over the items of any sequence, such as a list or a string.

Syntax

```
for iterating_var  in sequence:
    statements(s)
```

Example:

```
for x in'Hi':

    print x
```

Command Line Arguments

You can get access to the command line parameters using the sys module. len(sys.argv) contains the number of arguments. To print all of the arguments simply execute str(sys.argv)

```
import sys
print('Arguments:', len(sys.argv))
print('List:', str(sys.argv))
```

Storing command line arguments

```
import sys
print('Arguments:', len(sys.argv))
print('List:', str(sys.argv))
if sys.argv< 2:
   print('To few arguments, please specify a filename')
filename = sys.argv[1]
print('Filename:', filename)
```

Set A]

1) Write a program which accepts 5 integer values and prints “DUPLICATES” if any of the values entered are duplicates otherwise it prints “ALL UNIQUE”.
   Example: Let 5 integers are (32, 45, 90, 45, 6) then output “DUPLICATES” to be printed.

2) Write a program which accepts an integer value as command line and print “Ok” if value is between 1 to 50 (both inclusive) otherwise it prints ”Out of range”

3) Write a program which finds sum of digits of a number.
   Example n=125 then output is 8 (1+2+5).

4) Write a program which prints Fibonacci series of a number.

Set B]

1) Write a program which accept an integer value ‘n’ and display all prime numbers till ‘n’.

2) Write a program to display following pattern.
   1 2 3 4
   1 2 3
   1 2
   1

3) Write a program that finds distance between 2 points (x1,y1) and (x2,y2) using the equation
   $\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$

Set C]

1) Write a binary search function which searches an item in a sorted list. The function should return the index of element to be searched in the list.

Assignment Evaluation

0 : Not Done [ ] 1 : Incomplete [ ] 2 : Late Complete [ ]
3 : Needs Improvement [ ] 4 : Complete [ ] 5 : Well Done [ ]

Signature of Instructor

Assignment No.2 : Python String

A string is a sequence of characters. A character is simply a symbol. For example, the English language has 26 characters. Computers do not deal with characters, they deal with numbers (binary). Even though you may see characters on your screen, internally it is stored and manipulated as a combination of 0's and 1's.

This conversion of character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encoding used.

In Python, string is a sequence of Unicode character. Unicode was introduced to include every character in all languages and bring uniformity in encoding.

We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. Creating strings is as simple as assigning a value to a variable. For example –

```
var1 = 'Hello World!'
var2 = "Python Programming"
```

Accessing Values in Strings

Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring. For example −

```
#!/usr/bin/python

var1 = 'Hello World!'
var2 = "Python Programming"

print "var1[0]: ", var1[0]
print "var2[1:5]: ", var2[1:5]
```

Updating Strings

You can "update" an existing string by *re*assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether. For example −

```
#!/usr/bin/python

var1 = 'Hello World!'

print "Updated String :- ", var1[:6] + 'Python'
```

When the above code is executed, it produces the following result −

Updated String :- Hello Python

Escape Characters

Following table is a list of escape or non-printable characters that can be represented with backslash notation.

An escape character gets interpreted; in a single quoted as well as double quoted strings.

Backslash Notation	Description
\a	Bell or alert
\b	Backspace
\cx or \C-x	Control-x
\e	Escape
\f	Formfeed
\M-\C-x	Meta-Control-x
\n	Newline
\nnn	Octal notation, where n is in the range 0.7
\r	Carriage return
\s	Space
\t	Tab
\v	Vertical tab
\x	Character x

String Special Operators

Assume string variable **a** holds 'Hello' and variable **b** holds 'Python', then −

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give -HelloHello
[]	Slice - Gives the character from the given Index	a[1] will give e

[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1
%	Format - Performs String formatting	See at next section

String Formatting Operator

One of Python's coolest features is the string format operator %. This operator is unique to strings and makes up for the pack of having functions from C's printf family. Following is a simple example –

```
#!/usr/bin/python
print "My name is %s and weight is %d kg!" % ('Kamil ', 65)

When the above code is executed, it produces the following result −
My name is Kamil and weight is 65 kg!
```

Here is the list of complete set of symbols which can be used along with % −

Format Symbol	Conversion
%c	Character
%s	string conversion via str prior to formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer *lowercaseletters*
%X	hexadecimal integer *UPPERcaseletters*
%e	exponential notation *withlowercase'e'*
%E	exponential notation *withUPPERcase'E'*
%f	floating point real number
%g	the shorter of %f and %e
%G	the shorter of %f and %E

Other supported symbols and functionality are listed in the following table –

Symbol	Functionality
*	argument specifies width or precision

-	left justification
+	display the sign
<sp>	leave a blank space before a positive number
#	add the octal leading zero ′0′ or hexadecimal leading '0x' or '0X',
	depending on whether 'x' or 'X' were used.
0	pad from left with zeros *insteadofspaces*
%	'%%' leaves you with a single literal '%'
Var	mapping variable *dictionary arguments*
m.n.	m is the minimum total width and n is the number of digits to display after the decimal point *ifappl*.

Triple Quotes

Python's triple quotes comes to the rescue by allowing strings to span multiple lines, including verbatim NEWLINEs, TABs, and any other special characters.

The syntax for triple quotes consists of three consecutive **single or double** quotes.

```
#!/usr/bin/python
para_str = """this is a long string that is made up of several lines and non-printable characters such as
TAB ( \t ) and they will show up that way when displayed. NEWLINEs within the string, whether
explicitly given like this within the brackets [ \n ], or just a NEWLINE within the variable assignment
will also show up. """
print para_str
```

When the above code is executed, it produces the following result. Note how every single special character has been converted to its printed form, right down to the last NEWLINE at the end of the string between the "up." and closing triple quotes. Also note that NEWLINEs occur either with an explicit carriage return at the end of a line or its escape code \n −

```
this is a long string that is made up of
several lines and non-printable characters such as
TAB ( ) and they will show up that way when displayed. NEWLINEs within the string, whether
explicitly given like this within the brackets [
], or just a NEWLINE within
the variable assignment will also show up.
```

Raw strings do not treat the backslash as a special character at all. Every character you put into a raw string stays the way you wrote it −

```
#!/usr/bin/python
print 'C:\\nowhere'
```

When the above code is executed, it produces the following result −

```
C:\nowhere
```

Built-in String Methods

Python includes the following built-in methods to manipulate strings −

Sr. No.	Methods	Description
1	**Capitalize()**	Capitalizes first letter of string
2	**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.
3	**endswith(suffix, beg=0, end=len(string))**	Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.
4	**isalnum()**	Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
5	**isalpha()**	Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
6	**isdigit()**	Returns true if string contains only digits and false otherwise.
7	**islower()**	Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
8	**isnumeric()**	Returns true if a unicode string contains only numeric characters and false otherwise.
9	**isspace()**	Returns true if string contains only whitespace characters and false otherwise.
10	**istitle()**	Returns true if string is properly "titlecased" and false otherwise.
11	**isupper()**	Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
12	**join(seq)**	Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.
13	**len(string)**	Returns the length of the string
14	**ljust(width[, fillchar])**	Returns a space-padded string with the original string left-justified to a total of width columns.
15	**lower()**	Converts all uppercase letters in string to

		lowercase.
16	**lstrip()**	Removes all leading whitespace in string.
17	**maketrans()**	Returns a translation table to be used in translate function.
18	**max(str)**	Returns the max alphabetical character from the string str.
19	**min(str)**	Returns the min alphabetical character from the string str.
20	**replace(old, new [, max])**	Replaces all occurrences of old in string with new or at most max occurrences if max given.
21	**rfind(str, beg=0,end=len(string))**	Same as find(), but search backwards in string.
22	**rjust(width,[, fillchar])**	Returns a space-padded string with the original string right-justified to a total of width columns.
23	**rstrip()**	Removes all trailing whitespace of string.
24	**split(str="", num=string.count(str))**	Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.
25	**splitlines(num=string.count('\n'))**	Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.
26	**swapcase()**	Inverts case for all letters in string.
27	**title()**	Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.
28	**translate(table, deletechars="")**	Translates string according to translation table str(256 chars), removing those in the del string.
29	**upper()**	Converts lowercase letters in string to uppercase.
30	**zfill (width)**	Returns original string leftpadded with zeros to a total of width characters;

		intended for numbers, zfill() retains any sign given (less one zero).
31	**isdecimal()**	Returns true if a unicode string contains only decimal characters and false otherwise.

SET A]

1. Write a Python program to count the number of characters (character frequency) in a string.
Sample String : google.com'
Expected Result : {'o': 3, 'g': 2, '.': 1, 'e': 1, 'l': 1, 'm': 1, 'c': 1}

2. Write a Python program to get a string made of the first 2 and the last 2 chars
from a given a string. If the string length is less than 2, return instead of the empty string.
Sample String : 'General12'
Expected Result : 'Ge12'
Sample String : 'Ka'
Expected Result : 'KaKa'
Sample String : ' K'
Expected Result : Empty String

3. Write a Python program to get a string from a given string where all
occurrences of its first char have been changed to '$', except the first char itself. Sample String : 'restart'
Expected Result : 'resta$t'

4.Write a Python program to get a single string from two given strings, separated by a space and swap the first two characters of each string.
Sample String : 'abc', 'xyz'
Expected Result : 'xycabz'

SET B]

1. Write a Python program to change a given string to a new string where the first and last chars have been exchanged.
2. Write a Python program to remove the characters which have odd index values of a given string.
3. Write a Python program to count the occurrences of each word in a given sentence.
4. Write a python program to count repeated characters in a string.
Sample string: 'thequickbrownfoxjumpsoverthelazydog'
Expected output :

o 4
e 3
u 2
h 2
r 2
t 2

Assignment Evaluation

0: Not Done [ ] 1: Incomplete [ ] 2: Late Complete [ ]

3: Needs Improvement [ ] 4: Complete [ ] 5: Well Done [ ]

Signature of Instructor

Assignment No. 3 : Python Tuple

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also. For example −

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5 );
tup3 = "a", "b", "c", "d";
```

The empty tuple is written as two parentheses containing nothing −

```
tup1 = ();
```

To write a tuple containing a single value you have to include a comma, even though there is only one value −

```
tup1 = (50,);
```

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

Accessing Values in Tuple:

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −

```
#!/usr/bin/python

tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5, 6, 7 );

print "tup1[0]: ", tup1[0]
print "tup2[1:5]: ", tup2[1:5]
```

When the above code is executed, it produces the following result −

```
tup1[0]:      physics
tup2[1:5]:      [2, 3, 4, 5]
```

Updating Tuple

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following example demonstrates –

```
#!/usr/bin/python

tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');

#  Following action is not valid for tuples
#  tup1[0] = 100;

#  So let's create a new tuple as follows
tup3 = tup1 + tup2;
print tup3
```

When the above code is executed, it produces the following result −

```
(12, 34.56, 'abc', 'xyz')
```

Delete Tuple Elements

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the **del** statement. For example:

```
#!/usr/bin/python

tup = ('physics', 'chemistry', 1997, 2000);

print tup
del tup;
print "After deleting tup : "
print tup
```

This produces the following result. Note an exception raised, this is because after **del tup** tuple does not exist any more −

```
('physics', 'chemistry', 1997, 2000)
After deleting tup :
Traceback (most recent call last):
   File "test.py", line 9, in <module>
      print tup;
NameError: name 'tup' is not defined
```

Basic Tuple Operations

Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.

Python Expression	Results	Description
len(1, 2, 3)	3	Length
1, 2, 3 + 4, 5, 6	1, 2, 3, 4, 5, 6	Concatenation
′Hi!′,* 4	′Hi!′,′Hi!′,′Hi!′,′Hi!′	Repetition
3 in 1, 2, 3	True	Membership
for x in 1, 2, 3: print x,	1 2 3	Iteration

Indexing, Slicing, and Matrixes

Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings. Assuming following input −

```
L = ('spam', 'Spam', 'SPAM!')
```

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

No Enclosing Delimiters

Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples, as indicated in these short examples −

```
#!/usr/bin/python

print 'abc', -4.24e93, 18+6.6j, 'xyz'
x, y = 1, 2;
print "Value of x , y : ", x,y
```

When the above code is executed, it produces the following result −

```
abc -4.24e+93 (18+6.6j) xyz
Value of x , y : 1 2
```

Built-in Tuple Functions

Python includes the following tuple functions :

Function	Description
all()	Return True if all elements of the tuple are true (or if the tuple is empty).
any()	Return True if any element of the tuple is true. If the tuple is empty, return False.
enumerate()	Return an enumerate object. It contains the index and value of all the items of tuple as pairs.
len()	Return the length (the number of items) in the tuple.
max()	Return the largest item in the tuple.
min()	Return the smallest item in the tuple
sorted()	Take elements in the tuple and return a new sorted list (does not sort the tuple itself).
sum()	Retrun the sum of all elements in the tuple.
tuple()	Convert an iterable (list, string, set, dictionary) to a tuple.

SET A

1. Write a Python program to create a list of tuples with the first element as the number and second element as the square of the number.
2. Write a Python program to create a tuple with numbers and print one item.
3. Write a Python program to unpack a tuple in several variables.
4. Write a Python program to add an item in a tuple.

SET B

1. Write a Python program to convert a tuple to a string.
2. Write a Python program to get the 4^{th} element from front and 4^{th} element from last of a tuple.
3. Write a Python program to find the repeated items of a tuple.
4. Write a Python program to check whether an element exists within a tuple.

Assignment Evaluation

0: Not Done [ ] 1: Incomplete [ ] 2: Late Complete [ ]

3: Needs Improvement [ ] 4: Complete [ ] 5: Well Done [ ]

Signature of Instructor

Assignment No. 4 : Python Set

A set is an unordered collection of items. Every element is unique (no duplicates) and must be immutable (which cannot be changed).

However, the set itself is mutable. We can add or remove items from it.

Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc.

How to create a set?

A set is created by placing all the items (elements) inside curly braces {}, separated by comma or by using the built-in function set().

It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have a mutable element, like list, set or dictionary, as its element.

Example

```
# set of integers
my_set = {1, 2, 3}
print(my_set)
# set of mixed datatypes
my_set = {1.0, "Hello", (1, 2, 3)}
print(my_set)
```

Creating an empty set is a bit tricky.

Empty curly braces {} will make an empty dictionary in Python. To make a set without any elements we use the set() function without any argument.

```
# initialize a with {}
a = {}
# check data type of a
# Output: <class 'dict'>
print(type(a))
# initialize a with set()
a = set()
# check data type of a
```

```
# Output: <class 'set'>

print(type(a))
```

How to change a set in Python?

Sets are mutable. But since they are unordered, indexing have no meaning.
We cannot access or change an element of set using indexing or slicing. Set does not support it.
We can add single element using the add() method and multiple elements using
the update() method. The update() method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.

```
# initialize my_set
my_set = {1,3}
print(my_set)

# if you uncomment line 9,
# you will get an error
# TypeError: 'set' object does not support indexing

#my_set[0]

# add an element
# Output: {1, 2, 3}
my_set.add(2)
print(my_set)

# add multiple elements
# Output: {1, 2, 3, 4}
my_set.update([2,3,4])
print(my_set)

# add list and set
# Output: {1, 2, 3, 4, 5, 6, 8}
my_set.update([4,5], {1,6,8})
print(my_set)
```

How to remove elements from a set?

A particular item can be removed from set using methods, discard() and remove().

The only difference between the two is that, while using discard() if the item does not exist in the set, it remains unchanged. But remove() will raise an error in such condition.

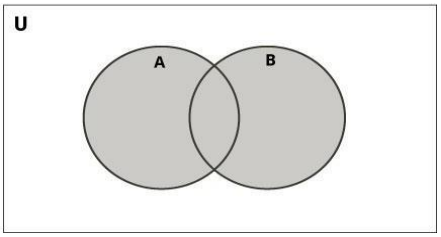
```
my_set = {1, 3, 4, 5, 6}
print(my_set)
# discard an element
# Output: {1, 3, 5, 6}
my_set.discard(4)
print(my_set)
# remove an element
# Output: {1, 3, 5}
my_set.remove(6)
print(my_set)
# discard an element
# not present in my_set
# Output: {1, 3, 5}
my_set.discard(2)
print(my_set)
```

Python Set Operations

Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. We can do this with operators or methods. Let us consider the following two sets for the following operations.

```
>>> A ={1,2,3,4,5}
>>> B ={4,5,6,7,8}
```

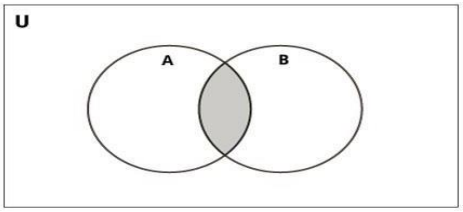
Set Union



Union of *A* and *B* is a set of all elements from both sets.

Union is performed using | operator. Same can be accomplished using the method union()

Set Intersection



Intersection of *A* and *B* is a set of elements that are common in both sets.

Intersection is performed using & operator. Same can be accomplished using the method intersection().

A = {1, 2, 3, 4, 5}

B = {4, 5, 6, 7, 8}

#union

print(A | B)

InterSection

Output: {4, 5}

print(A & B)

Method	Description
add()	Add an element to a set
clear()	Remove all elements form a set
copy()	Return a shallow copy of a set
difference()	Return the difference of two or more sets as a new set
difference_update()	Remove all elements of another set from this set
discard()	Remove an element from set if it is a member. (Do nothing if

	the element is not in set)
intersection()	Return the intersection of two sets as a new set
intersection_update()	Update the set with the intersection of itself and another
isdisjoint()	Return True if two sets have a null intersection
issubset()	Return True if another set contains this set
issuperset()	Return True if this set contains another set
pop()	Remove and return an arbitary set element. Raise KeyErrorif the set is empty
remove()	Remove an element from a set. If the element is not a member, raise a KeyError
symmetric_difference()	Return the symmetric difference of two sets as a new set
symmetric_difference_update()	Update a set with the symmetric difference of itself and another
union()	Return the union of sets in a new set
update()	Update a set with the union of itself and others

Built-in Functions with Set

Built-in functions like all(), any(), enumerate(), len(), max(), min(), sorted(), sum() etc. are commonly used with set to perform different tasks.

Function	**Description**
all()	Return True if all elements of the set are true (or if the set is empty).
any()	Return True if any element of the set is true. If the set is empty, return False.

enumerate()	Return an enumerate object. It contains the index and value of all the items of set as a pair.
len()	Return the length (the number of items) in the set.
max()	Return the largest item in the set.
min()	Return the smallest item in the set.
sorted()	Return a new sorted list from elements in the set(does not sort the set itself).
sum()	Retrun the sum of all elements in the set.

SET A

1. Write a Python program to do iteration over sets.
2. Write a Python program to add and remove operation on set.

SET B

1. Write a Python program to create an intersection of sets.
2. Write a Python program to create a union of sets.
3. Write a Python program to create set difference and a symmetric difference
4. Write a Python program to find the length of a set.
5. Write a Python program to find maximum and the minimum value in a set.

SET C

1. Write a Python program to create a shallow copy of sets.

Note : Shallow copy is a bit-wise copy of an object. A new object is created that has an exact copy of the values in the original object.

Assignment Evaluation

0: Not Done [ ] 1: Incomplete [ ] 2: Late Complete [ ]

3: Needs Improvement [ ] 4: Complete [ ] 5: Well Done [ ]

Signature of Instructor

Assignment No. 5 : Python Dictionary

Python dictionary is an unordered collection of items. While other compound data types have only value as an element, a dictionary has a key: value pair.

Dictionaries are optimized to retrieve values when the key is known.

How to create a dictionary?

Creating a dictionary is as simple as placing items inside curly braces {} separated by comma.

An item has a key and the corresponding value expressed as a pair, key: value.

While values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

```
# empty dictionary
my_dict = {}
# dictionary with integer keys
my_dict = {1: 'apple', 2: 'ball'}
# dictionary with mixed keys
my_dict = {'name': 'John', 1: [2, 4, 3]}
# using dict()
my_dict = dict({1:'apple', 2:'ball'})
# from sequence having each item as a pair
my_dict = dict([(1,'apple'), (2,'ball')])
```

How to access elements from a dictionary?

While indexing is used with other container types to access values, dictionary uses keys. Key can be used either inside square brackets or with the `get()` method.

The difference while using `get()` is that it returns `None` instead of `KeyError`, if the key is not found.

```
my_dict = {'name':'Jack', 'age': 26}
```

```
# Output: Jack
print(my_dict['name'])

# Output: 26
print(my_dict.get('age'))
```

How to change or add elements in a dictionary?

Dictionary are mutable. We can add new items or change the value of existing items using assignment operator.

If the key is already present, value gets updated, else a new key: value pair is added to the dictionary.

```
my_dict = {'name':'Jack', 'age': 26}

# update value
my_dict['age'] = 27

#Output: {'age': 27, 'name': 'Jack'}
print(my_dict)

# add item
my_dict['address'] = 'Downtown'

# Output: {'address': 'Downtown', 'age': 27, 'name': 'Jack'}
print(my_dict)
```

Python Dictionary Methods

Methods that are available with dictionary are tabulated below. Some of them have already been used in the above examples.

Method	**Description**
clear()	Remove all items form the dictionary.
copy()	Return a shallow copy of the dictionary.
fromkeys(seq[, v])	Return a new dictionary with keys from seq and value equal to v(defaults to None).

get(key[,d])	Return the value of key. If key doesnot exit, return d (defaults to None).
items()	Return a new view of the dictionary's items (key, value).
keys()	Return a new view of the dictionary's keys.
pop(key[,d])	Remove the item with key and return its value or d if key is not found. If d is not provided and key is not found, raises KeyError.
popitem()	Remove and return an arbitary item (key, value). Raises KeyError if the dictionary is empty.
setdefault(key[,d])	If key is in the dictionary, return its value. If not, insert key with a value of d and return d (defaults to None).
update([other])	Update the dictionary with the key/value pairs from other, overwriting existing keys.
values()	Return a new view of the dictionary's values

SET A

1. Write a Python script to sort (ascending and descending) a dictionary by value.
2. Write a Python script to add a key to a dictionary.
   Sample Dictionary : {0: 10, 1: 20}
   Expected Result : {0: 10, 1: 20, 2: 30}
3. Write a Python script to concatenate following dictionaries to create a new one.
   Sample Dictionary :
   dic1={1:10, 2:20}
   dic2={3:30, 4:40}
   dic3={5:50,6:60}
   Expected Result : {1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60}
4. Write a Python script to check if a given key already exists in a dictionary.
5. Write a Python program to iterate over dictionaries using for loops.

SET B

1. Write a Python script to generate and print a dictionary that contains a number (between 1 and n) in the form (x, x*x).
   Sample Dictionary (n = 5) :
   Expected Output : {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

2. Write a Python script to print a dictionary where the keys are numbers between 1 and 15 (both included) and the values are square of keys.
   Sample Dictionary
   {1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100, 11: 121, 12: 144, 13: 169, 14: 196, 15: 225}

3. Write a Python program to combine two dictionary adding values for common keys.
   d1 = {'a': 100, 'b': 200, 'c':300}
   d2 = {'a': 300, 'b': 200, 'd':400}
   Sample output: Counter({'a': 400, 'b': 400, 'd': 400, 'c': 300})

4. Write a Python program to create and display all combinations of letters, selecting each letter from a different key in a dictionary.
   Sample data : {'1':['a','b'], '2':['c','d']}
   Expected Output:
   ac
   ad
   bc
   bd

SET C

1. Write a Python program to create a dictionary from two lists without losing duplicate values.
   Sample lists: ['Class-V', 'Class-VI', 'Class-VII', 'Class-VIII'], [1, 2, 2, 3]
   Expected Output: defaultdict(<class 'set'>, {'Class-VII': {2}, 'Class-VI': {2}, 'Class-VIII': {3}, 'Class-V': {1}})
2. Write a Python program to match key values in two dictionaries.
   Sample dictionary: {'key1': 1, 'key2': 3, 'key3': 2}, {'key1': 1, 'key2': 2}
   Expected output: key1: 1 is present in both x and y

Assignment Evaluation

0: Not Done [ ] 1: Incomplete [ ] 2: Late Complete [ ]

3: Needs Improvement [ ] 4: Complete [ ] 5: Well Done [ ]

Signature of Instructor

Assignment No. 6 : Functions in Python

A function is a block of organized, reusable code that is used to perform a single, related action. Functions gives modularity and reusing of code in a program.
There are many built-in functions like in Python but you can also create your own functions. These functions are called user-defined functions.

How to define a function
SYNTAX:

```
Def functionname( parameters ):
  "function_docstring"
function_body
return [expression]
```

The function blocks starts with the keyword ***def*** followed by the function name and parentheses and colon(:).
The input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
The first statement of a function is an optional statement - the documentation string of the function or docstring.
The code block within every function starts with a colon (:) and is indented.
The statement return [expression] exits a function, the return statement with no arguments means return Nothing.

Consider the following function:

```
def revno(n):
        sum=0
        while (n>0):
                rem=n%10
                n=n/10
                sum=(sum*10)+rem
        return sum

print "The reverse of 54321 is :",revno(54321)
```

Calling a function
Once the function is defined, you can execute it by calling it from another function or directly from the Python prompt. In the above code the function is called from print statement.

Function Arguments:
There are the following types of formal arguments:
1. Required arguments
2. Keyword arguments
3. Default arguments
4. Variable-length arguments

Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

```
#!/usr/bin/python
def display( str ):
printstr;
return;
# call the function
display("hello");
```

Keyword Arguments:

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name. In this we can change the order of arguments or may skip it.

```
#!/usr/bin/python
def display( classname, roll_no ):
print "Class: ", classname;
print "Roll_no ", roll_no;
return;
# call the function
display(roll_no=10,  classname="TYBCA"  );
```

Default Arguments:

A default argument is an argument that considers a default value if a value is not provided in the function call for that argument.

```
#!/usr/bin/python
def display( classname, roll_no=11 ):
print "Class: ", classname;
print "Roll_no ", roll_no;
return;
display(classname="TYBCA", roll_no=10 );
display("SYBCA");
```

output is…

```
Class:  TYBCA
Roll_no  10
Class:  SYBCA
Roll_no  11
```

Variable length arguments:

We may require more arguments than specified while defining the function. These arguments are called variable-length arguments and are not named in the function definition.

def functionname([formal_args,] *var_args_tuple):

"function_docstring"

function_body

return [expression]

An asterisk (*) is placed before the variable name that will hold the values of all non keyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call.

```
#!/usr/bin/python
def display( arg1, *varargs ):
print "Output is: "
print arg1
forvar in varargs:
printvar
return;
# call the function
display( 100 );
display( 10, 20, 30 );
Output is:
100
Output is:
10 20 30
```

Return Statement:

The statement return [expression] exits a function, passing back an expression to the caller.

```
defcheckprime(n):
if n>1 :
fori in range(2,n):
if(n%i)==0:
return 0
return 1

a=checkprime(10)
if a==1:
print " no is prime"
else:
print " no is not prime"
```

Functions returning multiple values:

```
def display(x, y):
return x * 3, y * 4
a, b = display(5, 4)
print a
print b
```

Anonymous Functions:

The anonymous functions are not declared in the standard manner by using the def keyword.
The lambda keyword is used to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression.
- They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression.
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

SYNTAX:

lambda [arg1 [,arg2,.....argn]]:expression

```
#!/usr/bin/python
total = lambda arg1, arg2: arg1 + arg2;
print "Value of total : ", total( 100, 20 )
print "Value of total : ", total( 200, 10 )
#output
Value of total :  120
Value of total :  210
```

Scope of Variables:

All variables in a program may not be accessible at all locations in that program. It depends on where you have declared a variable.The scope of a variable is the portion of the program where you can access it. Thereare two basic scopes of variables in Python:

- Global variables
- Local variables

Variables that are defined inside a function body arehaving local scope, and defined outside the function body have a global scope.

Recursive Functions

The function which calls itself is a recursive function. Python allows us to write recursive functions.

Recursive function for factorial of a number:

```
def fact(n):
if(n==1):
return 1
else:
return n* fact(n-1)
print fact(4)
```

Function ducktyping

In the Python programming, the duck type is a type of dynamic style. In this style, the effective semantics of an object are determined by the current set of methods and properties rather than inheriting from a specific class or by implementing a particular interface.

In the duck type, the concern is not the type of object itself, but how it is used.

Consider the following code in which there is fly method for Parrot and Airplane classes but not for Whale class, so error will be generated for whale object

```
class Parrot:
def fly(self):
print("Parrot flying")
class Airplane:
def fly(self):
print("Airplane flying")
class Whale:
def swim(self):
print("Whale swimming")
def action(entity):
entity.fly()
parrot = Parrot()
airplane = Airplane()
whale = Whale()
action(parrot) # prints `Parrot flying`
action(airplane) # prints `Airplane flying`
action(whale) # Throws the error `'Whale' object has no attribute 'fly'`
```

```
class Duck:
def quack(self):
    print "Quaaaaaack!"
class Bird:
def quack(self):
    print "bird imitate duck."
class Doge:
def quack(self):
    print "doge imitate duck."
defin_the_forest(duck):
duck.quack()
duck = Duck()
bird = Bird()
doge = Doge()
for x in [duck, bird, doge]:
in_the_forest(x)
```

List comprehension:

List comprehension is the way to define and create list in Python. These lists have often the qualities of sets, but are not in all cases sets.
List comprehension is a complete substitute for the lambda function as well as the functions map(), filter() and reduce(). Consider the list comprehension to convert Celsius values into Fahrenheit :

```
Celsius = [39.2, 36.5, 37.3, 37.8]
Fahrenheit = [ ((float(9)/5)*x + 32) for x in Celsius ]
print Fahrenheit
######output####
[102.56, 97.7, 99.14, 100.03999999999999]
```

Cross product of two sets:

```
colours = [ "red", "green", "yellow", "blue" ]
things = [ "house", "car", "tree" ]
coloured_things = [ (x,y) for x in colours for y in things ]
printcoloured_things
output
[('red', 'house'), ('red', 'car'), ('red', 'tree'), ('green', 'house'), ('green', 'car'), ('green', 'tree'), ('yellow',
'house'), ('yellow', 'car'), ('yellow', 'tree'), ('blue', 'house'), ('blue', 'car'), ('blue', 'tree')]
```

Unpacking argument list

Python allows us to use variable number of arguments. We can also change the values inside functions using argument unpacking

packing and unpacking allows us to do:

1. validate arguments before passing them
2. set defaults for positional arguments
3. create adaptors for different pieces of code / libraries
4. modify arguments depending on context
5. log calls to methods

Consider the following code:

```
def func1(x, y, z):
print x
print y
   print z
def func2(*args):
   # Convert args tuple to a list so we can modify it
args = list(args)
args[0] = 'Hello'
args[1] = 'Everybody'
func1(*args)
func2('Goodbye', 'Hi', 'welcome')
####output
Hello
Everybody
welcome
```

Generator function

```
def infinite_generator(start=0):
        while True:
                yield start
                start += 1

for num in infinite_generator(4):
        print num
if num> 20:
        break
```

Try this generator function:

```
def vowels():
yield "a"
yield "e"
yield "i"
yield "o"
yield "u"
for i in vowels():
print(i)
```

Consider the following example for iterator and generator

```
def  simplegenerator():
        yield 'aaa'
        yield 'bbb'
        yield 'ccc'

def list_tripler(somelist):
        for item in somelist:
                item *= 3
                yield item

def limit_iterator(somelist, max):
        for item in somelist:
                if item > max:
                        yield item
def test():
        itr = simplegenerator()
        for item in itr:
                print item
```

```
alist = range(5)
it = list_tripler(alist)
for item in it:
        print item
alist = range(8)
        it = limit_iterator(alist, 4)
for item in it:
print item
it = simplegenerator()
try:
printit.next()
      print it.next()
printit.next()
printit.next()
exceptStopIteration, exp:
      print 'reached end of sequence'
if _name___== '_main_':
test()
```

SET A:

1. Write a recursive function which print string in reverse order.
2. Write an anonymous function to find area of circle.
3. Write a function which print a dictionary where the keys are numbers between 1 and 20 (both included) and the values are square of keys

SET B:

1. Write a program to display Fibonacci series using generator and Iterators.
2. Write a program to find gcd of number use recursion
3. Define a function that accept two strings as input and find union and intersection of them.
4. Write a user defined functions hex () which convert given number into hexadecimal,oct() which convert given number into octal and bin() which convert given number into binary.

SET C:

1. Write a program to illustrate function duck typing.

Assignment Evaluation

0: Not Done [ ] 1: Incomplete [ ] 2: Late Complete [ ]

3: Needs Improvement [ ] 4: Complete [ ] 5: Well Done [ ]

Signature of Instructor

Assignment No. 7 : FILE

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

The open Function:
Before you can read or write a file, you have to open it using Python's built-in open() function. This function creates a file object, which is used to call other support methods associated with it.

SYNTAX:
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 that you want toaccess.
Access_mode: The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. This is optional parameter and the default file access mode is read (r).
Buffering: If the buffering value is set to 0, no buffering will take place. If the buffering value is 1, line buffering will be performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action will be performed with the indicated buffer size. If negative, the buffer size is the system default.

Different modes of opening a file:

Mode	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file.
r+	Opens a file for both reading and writing. The file pointer will be at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer will be at the beginning of the file.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Opens a file for both reading and writing. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
wb+	Opens a file for both reading and writing in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
a	Opens a file for appending. 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.
ab	Opens a file for appending 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.

a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
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. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

The fileObject attributes:
After opening a file, you have a file object and you can get information related to the file using the file attributes given below:

Attribute	**Description**
file.closed	Returns true if file is closed, false otherwise.
file.mode	Returns access mode with which file was opened.
file.name	Returns name of the file.
file.softspace	Returns false if space explicitly required with print, true otherwise.

Consider the following sample code to open a file and display the attributes

```
#display the file attributes
fobj = open("sample.txt", "wb")
print "File name: ", fobj.name
print "File state: ", fobj.closed
print "Opening mode: ", fobj.mode
print "Softspace flag: ", fobj.softspace
```

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.

SYNTAX:
fileObject.close();

example: fobj.close()

Reading and Writing to files:
Python provides the methods to read from an opened file and write to an opened file.

Using the write() method:
The write() method writes any string to an open file.Python strings can have text as well as binary data.
Note:The write() method does not add a newline character ('\n') to the end of the string:

SYNTAX:
fileObject.write(string)

string is the content to be written to opened file.
Consider the following code to write to sample.txt file

```
#!/usr/bin/python
fobj = open("sample.txt", "wb")
fobj.write("1: I like python programming\n2: Welcome to python programming\n")
fobj.close()
```

Using writelines() method

The method writelines() writes a sequence of strings to the file. The sequence can be any iterable object
producing strings, generally a list of strings. It does not returnany value.

SYNTAX:

fileObject.writelines(sequence)

where sequence is the sequence of strings.
Consider the following code:

```
#!/usr/bin/python
fobj = open("sample1.txt", "wb")
str=["1: I like python programming\n2: Welcome to python programming\n"]
line =fobj.writelines(str);
# Close opened file
fobj.close()
```

Reading from a file:

Using the read() method:

The read() method reads a string from an open file. The string can be text as well as binary.

SYNTAX:

fileObject.read([count]);

count is the number of bytes to be read from file.
This method starts reading from the beginning of the file and if count is missing, then it tries to read as much as possible, maybe until the end of the file.

Consider the following code to read from file sample.txt

```
#!/usr/bin/python
fobj = open("sample.txt", "r+")
str = fobj.read(25);
print "Read String is : ", str
fobj.close()
#####output#######
Read String is :  1: I like python program
```

Using readline() method

The method readline() reads one entire line from the file. A trailing newline character is kept in the string. If the size argument is present and non-negative, it is a maximum byte count including the trailing newline and an incomplete line may be returned.
An empty string is returned only when EOF is encountered immediately.

SYNTAX

fileObject.readline(size)

where size is the number of bytes to be read from the file.It returns the line read from the file.

```
########## using readline function####
#!/usr/bin/python
fobj = open("sample.txt", "r")
print "Name of the file: ", fobj.name
line = fobj.readline()
print "Read Line: %s" % (line)
line = fobj.readline(5)
print "Read Line: %s" % (line)
fobj.close()
######output###
Name of the file:  sample.txt
Read Line: 1: I like python programming
Read Line: 2: We
```

Using readlines() method:

The method readlines() reads until EOF using readline() and returns a list containing the lines. If the optionalsizehint argument is present, instead of reading up to EOF, whole lines are read

SYNTAX:

fileObject.readlines(sizehint)

It returns a list containing the lines.

```
#!/usr/bin/python
fobj = open("sample.txt", "r")
print "Name of the file: ", fobj.name
line = fobj.readlines()
print "Read Line: %s" % (line)
line = fobj.readlines(2)
print "Read Line: %s" % (line)
fobj.close()
//output/////
Name of the file:  sample.txt
Read Line: ['1: I like python programming\n', '2: Welcome to python programming\n', '3: This is file handling assignment\n', '4: This is sample file\n', '5: Read the file line wise\n']
Read Line: []
```

Using next() method:

The method next() is used when a file is used as an iterator, typically in a loop, the next() method is called repeatedly. This method returns the next input line, or raises StopIteration when EOF is hit.

SYNTAX:
fileObject.next()

It returns the next input line.

```
#!/usr/bin/python
fobj = open("sample.txt", "r")
print "Name of the file: ", fobj.name
#Assuming file has 5 lines
for index in range(5):
line = fobj.next()
print "Line No %d - %s" % (index, line)
fobj.close()
///output
Name of the file:  sample.txt
Line No 0 - 1: I like python programming
Line No 1 - 2: Welcome to python programming
Line No 2 - 3: This is file handling assignment
Line No 3 - 4: This is sample file
Line No 4 - 5: Read the file line wise
```

File Positions :
The **tell()** method tells you the current position within the file.
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.

```
############seek  function#########
#!/usr/bin/python
fobj = open("sample.txt", "r+")
str = fobj.read(25);
print "Read String is : ", str
# Check current position
position = fobj.tell();
print "Current file position : ", position
# Reposition pointer at the beginning once again
position = fobj.seek(0, 0);
str = fobj.read(25);
print "Again read String is : ", str
fobj.close()
##output###
Read String is :  Python is a great languag
Current file position :  25
Again read String is :  Python is a great languag
```

```
##########using tell function###
#!/usr/bin/python
fobj = open("sample.txt", "r")
print "Name of the file: ", fobj.name
line = fobj.readline()
print "Read Line: %s" % (line)
# Get the current position of the file.
pos = fobj.tell()
print "Current Position: %d" % (pos)
fobj.close()
####output
Name of the file:  sample.txt
Read Line: 1: I like python programming
Current Position: 29
```

Renaming and Deleting Files:
Python os module provides methods that help you perform file-processing operations, such as renaming and deleting files.We have to import it first and then use the methods in it.

The rename() Method:
It renames the current file.

SYNTAX :
os.rename(current_file_name, new_file_name)
The remove() Method:
remove() is used to delete the file given as argument to remove().

SYNTAX :
os.remove(file_name)

consider the following code to rename and delete the file

```
######renaming a file#####
#!/usr/bin/python
importos
# Rename a file from sample.txt to sample2.txt
os.rename( "sample.txt", "sample2.txt" )
```

```
##removing a file
#!/usr/bin/python
importos
# Delete file sample2.txt
os.remove("sample2.txt")
```

SET A

1. Create a file named xyz.txt. Accept 5 lines from user and write it to the file.
2. Write a python program to count number of lines in a file.
3. Write a program to copy file a.txt to b.txt
4. Open the file xyz.txt and count the no of characters in it.
5. Accept file name from user, Display the attributes of the file.

SET B

1. Write a program to read a file convert the lowercase character to uppercase and write them to another file, display another file.
2. Write a python program which accept word and count the frequency (occurrences) of given word in a file.
3. Write a Python program to read last n lines of a file.
4. Open the file xyz.txt and display it in reverse order of lines.

SET C

1. Write a program to read numbers from numbers.txt file. Find the odd numbers and write it to *Odd.txt*, find even numbers and write to *even.txt*

Assignment Evaluation

0: Not Done [ ] 1: Incomplete [ ] 2: Late Complete [ ]

3: Needs Improvement [ ] 4: Complete [ ] 5: Well Done [ ]

Signature of Instructor

Assignment no 8 :DIRECTORIES

We can perform directory operations using Python. The os module provides various methods for files and directory manipulations.

1. Make directory:

os.mkdir(path) **#** path is the pathname for directory

2. Change directory:

os.chdir(path)

3. Removing a directory:

os.rmdir(path)

4. Display current directory:

os.getcwd()

Consider the following code to illustrate above functions:

```
#### creating directories####
#!/usr/bin/python
Import os
# Create a directory "test"
os.mkdir("test")
```

```
# Changing directories
#!/usr/bin/python
importos
# Changing a directory to "/test"
os.chdir("/root/Desktop/Python-2.7.14/Python/test")
```

Displaying current working directory

```
#!/usr/bin/python
importos
# This would give location of the current directory
printos.getcwd()
```

Removing a directory

```
#!/usr/bin/python
importos
# This will remove "/Python/test"
os.rmdir( "test" )
```

After executing each module check using ls command

Splitting the path name

split path into dir part, file part

syntax: os.path.dirname("path") → returns the dir part of path.

```
importos.path
fp = "/home/Desktop/Python/fun.py"
printos.path.split(fp)
printos.path.dirname(fp)
printos.path.basename(fp)
```

#Get file extension

SYNTAX:
os.path.splitext("path") → returns a 2-tuple. Split a path into first part and file extension part.

```
importos.path
printos.path.splitext("/home/Desktop/Python/fun.py")
printos.path.splitext("/home/Desktop/Python/fun.py")
```

Traversing Directory Tree:
The method walk() generates the file names in a directory tree by walking the tree either top-down or bottom-up.

SYNTAX:
os.walk(top[, topdown=True[, onerror=None[, followlinks=False]]])

Parameters:
top: Each directory rooted at directory, yields 3-tuples, i.e., (dirpath, dirnames, filenames)
topdown: If optional argument topdown is True or not specified, directories are scanned from top-down. If
topdown is set to False, directories are scanned from bottom-up.
Onerror : This can show error to continue with the walk, or raise the exception to abort the walk.
Followlinks: This visits directories pointed to by symlinks, if set to true.

//The sample code is as below:

```
#!/usr/bin/python
import os
for root, dirs, files in os.walk(".", topdown=False):
        for name in files:
                print(os.path.join(root, name))
        for name in dirs:
                print(os.path.join(root,   name))
```

SET A:

1. Write a program to create a directory rename it then remove it.
2. Write a program to display the file lists in the directory.
3. Write a program to search a given file in the current directory. Display proper message.

SET B:

1. Write a program to accept a directory name and list the files in it extension wise. i.e. .txt, .html, .doc etc.
2. Write a Python program to List all Files along with their path in Directory.
3. Write a program to copy all files from one directory to another directory.

SET C:

1. Write a program to count the types of files in directory. Display the type and count. i.e. .txt 12, .html 10 etc.

Assignment Evaluation

0: Not Done [ ] 1: Incomplete [ ] 2: Late Complete [ ]

3: Needs Improvement [ ] 4: Complete [ ] 5: Well Done [ ]

Signature of Instructor

Assignment No. 9 :Python Classes /Objects

Classes

Classes are Python's main object-oriented programming (OOP) tool. Classes are just a way to define new sorts of stuff, which reflect real objects in program's domain. Classes model the behavior of objects in the "real" world. Methods implement the behaviors of these types of objects. Member variables hold (current) state. Classes enable to implement new data types in Python.

Classes are mostly just a namespace, much like modules. But unlike modules, classes also have support for multiple copies, namespace inheritance, and operator overloading. There are two kinds of objects in Python's OOP model—class objects and instance objects.

Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

Class objects provide default behavior and serve as generators for instance objects. Instance objects are the real objects your programs process; each is a namespace in its own right, but inherits (i.e., has access to) names in the class it was created from. Class objects come from statements and instances from calls; each time you call a class, you get a new instance.

The class: statement is used to define a class. The class: statement creates a class object and binds it to a name.

A simple class

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Class objects support two kinds of operations: attribute references and instantiation. Attribute references use the standard syntax used for all attribute references in Python: obj.name. Valid attribute names are all the names that were in the class's namespace when the class object was created. So, if the class definition looked like this:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

MyClass.i and MyClass.f are valid attribute references, returning an integer and a function object, respectively.

x = MyClass() creates a new *instance* of the class and assigns this object to the local variable x.

The instantiation operation (“calling” a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named __init__(), like this:

```
def __init__(self):
    self.data = []
```

Define a class called FirstClass , using the Python class statement:

```
>>> class FirstClass:
# define a class object
...
def setdata(self, value):
# define class methods
...
self.data = value
# self is the instance
...
def display(self):
...
print self.data
# self.data: per instance
```

Like all compound statements, class starts with a header line that lists the class name, followed by a Body of one or more nested and indented statements. The nested statements are defs ; they define functions that implement the behavior the class means to export. As def is an assignment; it assigns to names in the class statement's scope and so generates attributes of the class. Functions inside a class are usually called method functions; they're normal defs, but the first argument automatically receives an implied instance object when called.

```
class Complex:
    def __init__(self, realpart, imagpart):
       self.r = realpart
        self.i = imagpart

x = Complex(3.0, -4.5)
x.r, x.i
(3.0, -4.5)
```

Class definitions, like function definitions (def statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an if statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods.

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the end), a *class object* is created. This is basically a wrapper around the contents of the namespace created by the class definition; we'll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (ClassName in the example).

Example:

```
class InputOutString(object):
def __init__(self):
self.s = ""

def getString(self):
self.s = raw_input()

def printString(self):
print self.s.upper()

strObj = InputOutString()

strObj.getString()
strObj.printString()
```

Example:

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary
```

- The variable *empCount* is a class variable whose value is shared among all instances of a this class. This can be accessed as *Employee.empCount* from inside the class or outside the class.
- The first method *__init__()* is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
- You declare other class methods like normal functions with the exception that the first argument to each method is *self*. Python adds the *self* argument to the list for you; you do not need to include it when you call the methods.

Example :

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary

"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

Example :

```
import math
class Point:
"""A class to represent a two-dimensional point"""
Def __init__(self):
self.x = 0
self.y = 0
def move(self, dx, dy):
self.x += dx
self.y += dy
def distance(self, p2):
dx = self.x - p2.x
dy = self.y - p2.y
return
math.sqrt(dx *dx+dy*dy)
p1 = Point()
p1.move(2,3)
p2 = Point()
```

```
p2.move(5,7)
print
p1.distance(p2)
```

Objects

An object is an instance of a class. Integers, floating-point numbers, strings, lists, and functions are all objects in Python. With the exception of function objects, these objects are as passive data. Two floating-point numbers can be added and concatenate two strings with the + operator. We can pass objects to functions and functions can return objects. Objects fuse data and functions together. A typical object consists of two parts: Data and methods. An object's data is sometimes called its attributes or fields. Methods are like functions, and they also are known as operations. The data and methods of an object constitutes its members. Using the same terminology as functions, the code that uses an object is called the object's client. Just as a function provides a service to its client, an object provides a service to its client. The services provided by an object can be more elaborate that those provided by simple functions, because objects make it easy to store persistent data.

Instance Objects

```
x.counter = 1
while x.counter < 10:
   x.counter = x.counter * 2
print x.counter
del x.counter
```

The other kind of instance attribute reference is a method. A method is a function that "belongs to" an object. (In Python, the term method is not unique to class instances: other object types can have methods as well. For example, list objects have methods called append, insert, remove, sort, and so on. Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, x.f is a valid method reference, since MyClass.f is a function, but x.i is not, since MyClass.i is not. But x.f is not the same thing as MyClass.f — it is a method object, not a function object.

Method Objects

Usually, a method is called right after it is bound:

```
x.f()
```

In the MyClass example, this will return the string 'hello world'. However, it is not necessary to call a method right away: x.f is a method object, and can be stored away and called at a later time. For example:

```
xf = x.f
while True:
   print xf()
```

will continue to print hello world until the end of time.

What exactly happens when a method is called? x.f() was called without an argument above, even though the function definition for f() specified an argument. Python raises an exception when a function that requires an argument is called without any — even if the argument isn't actually used.

The special thing about methods is that the object is passed as the first argument of the function.

In example, the call x.f() is exactly equivalent to MyClass.f(x). In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument.
When an instance attribute is referenced that isn't a data attribute, its class is searched. If the name denotes a valid class attribute that is a function object, a method object is created by packing (pointers to) the instance object and the function object just found together in an abstract object: this is the method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

Class and Instance Variables
Instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

```
class Dog:

    kind = 'canine'         # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

 d = Dog('Fido')
 e = Dog('Buddy')
 d.kind                  # shared by all dogs
'canine'
e.kind                  # shared by all dogs
'canine'
d.name                  # unique to d
'Fido'
e.name                  # unique to e
'Buddy'
```

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

 d = Dog('Fido')
 e = Dog('Buddy')
 d.add_trick('roll over')
 e.add_trick('play dead')
 d.tricks
['roll over']
e.tricks
['play dead']
```

SET A:

1) Python Program to Create a Class in which One Method Accepts a String from the User and Another method Prints it. Define a class named Country which has a method called printNationality. Define subclass named state from Country which has a method called printState . Write a method to print state, country and nationality.
2) Write a Python class which has two methods get_String and print_String. get_String accept a string from the user and print_String print the string in upper case. Further modify the program to reverse a string word by word and print it in lower case.
3) Define a class named Rectangle which can be constructed by a length and width. The Rectangle class has a method which can compute the area and volume.
4) Define a class named Shape and its subclass (Square/Circle). The subclass has an init function which takes a an argument (length/redious). Both classes have an area and volume function which can print the area and volume of the shape where Shape's area is 0 by default.
5) Write a Python Program to Accept, Delete and Display students details such as Roll.No, Name, Marks in three subject, using Classes. Also display percentage of each student.

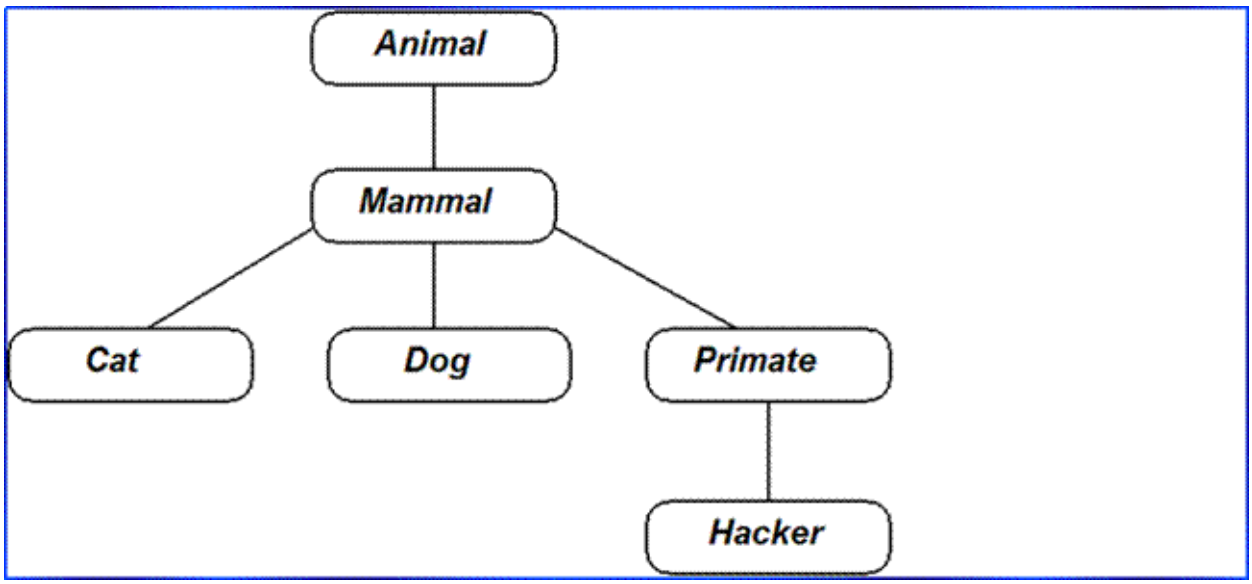
6) Write a Python program that defines a class to find the three elements that sum to zero from a set of n real numbers.
   Input array : [-25, -10, -7, -3, 2, 4, 8, 10]
   Output : [[-10, 2, 8], [-7, -3, 10]]
7) Write a Python class to find a pair of elements (indices of the two numbers) from a given array whose sum equals a specific target number.
   a. Input: numbers= [10,20,10,40,50,60,70], target=50
      Output: 3, 4

SET B

1) Write a Python program that defines a class named circle with attributes radius and center, where center is a point object and radius is number. Accept center and radius from user. Instantiate a circle object that represents a circle with its center and radius as accepted input.
2) Write a function named pt_in_circle that takes a circle and a point and returns true if point lies on the boundry of circle.
3) Write a Python program that takes Time object. Defines function called mul_time that takes time object and a number and returns new time object that is product of the original time object and the number.
4) Write a Python Program to Create a Class Set and Get All Possible Subsets from a Set of Distinct Integers.
5) Python Program to Create a Class which Performs Basic Calculator Operations

SET C

1) Define datetime module that provides time object. Using this module write a program that gets current date and time and print day of the week.
2) Consider the class tree sketched in the figure below. Code a set of 6 class statements to model this taxonomy with Python inheritance. Then, add a speak method to each of your classes which prints a unique message, and a reply method in your top-level Animal superclass which simply calls self.speak to invoke the category-specific message printer in a subclass below (remember, this will kick off an independent inheritance search from self). Finally, remove the speak method from your Hacker class, so that it picks up the default above it. When you're finished, your classes should work this way:



Assignment Evaluation

0: Not Done [ ] 1: Incomplete [ ] 2: Late Complete [ ]

3: Needs Improvement [ ] 4: Complete [ ] 5: Well Done [ ]

Signature of Instructor

Assignment no. 10: Exception Handling

What is an Exception?

An exception is an error that happens during execution of a program. When that error occurs, Python generate an exception that can be handled, which avoids program to crash.

Python has many *built-in exceptions* which forces the program to output an error when something in it goes wrong. When these exceptions occur, it causes the current process to stop and passes it to the calling process until it is handled. If not handled, our program will crash.

For example, if function `A` calls function `B` which in turn calls function `C` and an exception occurs in function `C`. If it is not handled in `C`, the exception passes to `B` and then to `A`. If never handled, an error message is spit out and our program comes to a sudden, unexpected halt.

Why use Exceptions?

Exceptions are convenient in many ways for handling errors and special conditions in a program. When you think that you have a code which can produce an error then you can use exception handling.
In typical Python programs, exceptions may be used for a variety of things:
Error handling
Python raises exceptions when it detects errors in programs at runtime; you can either catch and respond to the errors internally in your programs or ignore the exception. If ignored, Python's default exception-handling behavior kicks in; it kills the program and prints an error message showing where the error occurred.

Event notification
Exceptions can also signal a valid condition, without having to pass result flags around a program or test them explicitly. For instance, a search routine might raise an exception on success, rather than return an integer 1.

Special-case handling
Sometimes a condition may happen so rarely that it's hard to justify convoluting code to handle it. You can often eliminate special-case code by handling unusual cases in exception handlers instead.

Unusual control-flows
And finally, because exceptions are a type of high-level goto, you can use them as the basis for implementing exotic control flows. For instance, although back-tracking is not part of the language itself, it can be implemented in Python with exceptions and a bit of support logic to unwind assignments.
Python exceptions are a high-level control flow device. They may be raised either by Python or by our programs; in both cases, they may be caught by try statements. Python try statements come in two flavors—one that handles exceptions and one that executes finalization code whether exceptions occur or not.

Raising an Exception

You can raise an exception in your program by using the raise exception statement. Raising an exception breaks current code execution and returns the exception back until it is handled.

Exception Errors

Below is some common exceptions errors in Python:

IOError	If the file cannot be opened.
ImportError	If python cannot find the module
ValueError	Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value
KeyboardInterrupt	Raised when the user hits the interrupt key (normally Control-C or Delete)
EOFError	Raised when one of the built-in functions (input() or raw_input()) hits an end-of-file condition (EOF) without reading any data.

Detailed list is provided at end.

Exception Errors Examples

Now, when we know what some of the exception errors means, let's see some examples:

```
except IOError:
   print('An error occurred trying to read the file.')

except ValueError:
   print('Non-numeric data found in the file.')

except ImportError:
   print "NO module found"

except EOFError:
   print('Why did you do an EOF on me?')

except KeyboardInterrupt:
   print('You cancelled the operation.')

except:
   print('An error occurred.')
```

Set up exception handling blocks

The words "try" and "except" are Python keywords and are used to catch exceptions. To use exception handling in Python, you first need to have a catch-all except clause.

The code within the try clause will be executed statement by statement. If an exception occurs, the rest of the try block will be skipped and the except clause will be executed.
Syntax is

try:
some statements here
except:
exception handling

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

Let's see a short example on how to do this:

```
try:
    print 1/0

except ZeroDivisionError:
    print "You can't divide by zero, you're silly."
```

```
var1                                    =                                       '1'
try:
var1 = var1 + 1 # since var1 is a string, it cannot be added to the number 1
except:
print(var1, " is not a number") #so we execute this

print(var1)
```

How does it work?
The error handling is done through the use of exceptions that are caught in try blocks and handled in except blocks. If an error is encountered, a try block code execution is stopped and transferred down to the except block. In addition to using an except block after the try block, you can also use the finally block. The code in the finally block will be executed regardless of whether an exception occurs.
When a try statement is started, Python marks the current program context, so it can come back

if an exception occurs. The statements nested under the try header are run first; what happens next depends on whether exceptions are raised while the try block's statements are running or not.

The try statement works as follows.

- First, the try clause (the statement(s) between the try and except keywords) is executed.
- If no exception occurs, the except clause is skipped and execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.
- If an exception occurs while the try block's statements are running, Python jumps back to the Try and runs the statements under the first except clause that matches the raised exception. Control continues past the entire try statement after the except block runs (unless the Except block raises another exception).
- If an exception happens in the try block and no except clause matches, the exception is propagated up to a try that was entered earlier in the program, or to the top level of the process (which makes Python kill the program and print a default error message).
- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above.
- In other words, except clauses catch exceptions that happen while the try block is running, and the else clause is run only if no exceptions happen while the try block runs. The except clauses are very focused exception handlers; they catch exceptions that occur only within the statements in the associated try block.

A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:

```
. except (RuntimeError, TypeError, NameError):
...     pass
```

Code Example

Let's write some code to see what happens when you not use error handling in your program.

This program will ask the user to input a number between 1 and 10, and then print the number.

```
number = int(raw_input("Enter a number between 1 - 10"))

print "you entered number", number
```

This program works perfectly fun as long as the user enters a number, but what happens if the user puts in something else (like a string)?

```
Enter a number between 1 - 10
Hello
```

You can see that the program throws us an error when we enter a string.

```
Traceback (most recent call last):
  File "enter_number.py", line 1, in
    number = int(raw_input("Enter a number between 1 - 10
"))
ValueError: invalid literal for int() with base 10: 'hello'
```

ValueError is an exception type. Let's see how we can use exception handling to fix the previous program.

```
import sys

print "Lets fix the previous code with exception handling"

try:
    number = int(raw_input("Enter a number between 1 - 10
"))

except ValueError:
    print "Err.. numbers only"
    sys.exit()

print "you entered number ", number
```

If we now run the program, and enter a string (instead of a number), we can see that we get a different output.

```
Lets fix the previous code with exception handling
Enter a number between 1 - 10
hello
Err.. numbers only
```

Example:

```
var1 = '1'
try:
var2 = var1 + 1 # since var1 is a string, it cannot be added to the number 1
except:
var2 = int(var1) + 1
print(var2)
```

A class in an except clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an except clause listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

Note that if the except clauses were reversed (with except B first), it would have printed B, B, B — the first matching except clause is triggered.

The last except clause may omit the exception name(s), to serve as a wildcard. Use this with extreme caution, since it is easy to mask a real programming error in this way! It can also be used to print an error message and then re-raise the exception (allowing a caller to handle the exception as well):

Try ... except ... else clause

The try … except statement has an optional *else clause*, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception. For example:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

The else clause in a try, except statement must follow all except clauses, and is useful for code that must be executed if the try clause does not raise an exception.

The use of the else clause is better than adding additional code to the try clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the try … except statement.

When an exception occurs, it may have an associated value, also known as the exception's *argument*. The presence and type of the argument depend on the exception type.

The except clause may specify a variable after the exception name. The variable is bound to an exception instance with the arguments stored in instance.args. For convenience, the exception instance defines __str__() so the arguments can be printed directly without having to reference .args. One may also instantiate an exception first before raising it and add any attributes to it as desired.

```
try:
    data = something_that_can_go_wrong

except IOError:
    handle_the_exception_error

else:
    doing_different_exception_handling
```

Exceptions in the else clause are not handled by the preceding except clauses.

Make sure that the else clause is run before the finally block.

```
>>> try:
    raise Exception('spam', 'eggs')
 except Exception as inst:
    print(type(inst))     # the exception instance
     print(inst.args)      # arguments stored in .args
    print(inst)           # __str__allows args to be printed directly,
                          # but may be overridden in exception subclasses
     x, y = inst.args      # unpack args
     print('x =', x)
     print('y =', y)

<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

If an exception has arguments, they are printed as the last part ('detail') of the message for unhandled exceptions.

Exception handlers don't just handle exceptions if they occur immediately in the try clause, but

also if they occur inside functions that are called (even indirectly) in the try clause. For example:

```
def this_fails():
     x = 1/0
try:
     this_fails()
except ZeroDivisionError as err:
     print('Handling run-time error:', err)
Handling run-time error: division by zero
```

Try ... finally clause

The finally clause is optional. It is intended to define clean-up actions that must be executed under all circumstances

```
try:
    raise KeyboardInterrupt
finally:
    print 'Goodbye, world!'
...
Goodbye, world!
KeyboardInterrupt
```

A finally clause is always executed before leaving the try statement, whether an exception has occurred or not.
If you don't specify an exception type on the except line, it will catch all exceptions, which is a bad idea , since it means your program will ignore unexpected errors as well as ones which the except block is actually prepared to handle.

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Raising Exceptions

The raise statement allows the programmer to force a specified exception to occur. For example:

```
raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

The sole argument to raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
raise ValueError  # shorthand for 'raise ValueError()'
```

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the raise statement allows you to re-raise the exception:

```
 try:
     raise NameError('HiThere')
except NameError:
     print('An exception flew by!')
     raise
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere
```

```
try:
    a=10
    print a
    raise NameError("Hello")
except NameError as e:
        print "An exception occurred"
        print e
```

User-defined Exceptions

Programs may name their own exceptions by creating a new exception class (see Classes for more about Python classes). Exceptions should typically be derived from the Exception class, either directly or indirectly.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception. When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions:

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass
```

```
class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition is not allowed
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

Most exceptions are defined with names that end in "Error," similar to the naming of the standard exceptions.

```
class ErrorInCode(Exception):
     def __init__(self, data):
   self.data = data
     def __str__(self):
        return repr(self.data)

try:
    raise ErrorInCode(2000)
except ErrorInCode as ae:
    print "Received error:", ae.data
```

Sr.No.	Exception Name	Description
1	**Exception :**	Base class for all exceptions
2	**StopIteration :**	Raised when the next() method of an iterator does not point to any object.
3	**SystemExit :**	Raised by the sys.exit() function.
4	**StandardError:**	Base class for all built-in exceptions except StopIteration and SystemExit.
5	**ArithmeticError:**	Base class for all errors that occur for numeric calculation.
6	**OverflowError:**	Raised when a calculation exceeds maximum limit for a numeric type.

7	**FloatingPointError:** Raised when a floating point calculation fails.
8	**ZeroDivisionError:** Raised when division or modulo by zero takes place for all numeric types.
9	**AssertionError:** Raised in case of failure of the Assert statement.
10	**AttributeError:** Raised in case of failure of attribute reference or assignment.
11	**EOFError:** Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
12	**ImportError:** Raised when an import statement fails.
13	**KeyboardInterrupt:** Raised when the user interrupts program execution, usually by pressing Ctrl+c.
14	**LookupError:**Base class for all lookup errors.
15	**IndexError:** Raised when an index is not found in a sequence.
16	**KeyError:**Raised when the specified key is not found in the dictionary.
17	**NameError:** Raised when an identifier is not found in the local or global namespace.
18	**UnboundLocalError:** Raised when trying to access a local variable in a function or method but no value has been assigned to it.
19	**EnvironmentError:** Base class for all exceptions that occur outside the Python environment.
20	**IOError:**Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
21	**IOError:** Raised for operating system-related errors.
22	**SyntaxError:** Raised when there is an error in Python syntax.
23	**IndentationError:** Raised when indentation is not specified properly.
24	**SystemError:** Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
25	**SystemExit:** Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
26	**TypeError:** Raised when an operation or function is attempted that is invalid for the specified data type.
27	**ValueError:** Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
28	**RuntimeError:** Raised when a generated error does not fall into any category.
29	**NotImplementedError:** Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

SET A

1) Write a Python program to input a positive integer. Display correct message for correct and incorrect input.

2) Define a custom exception class which takes a string message as attribute
3) Write text file named test.txt that contains integers, characters and float numbers. Write a Python program to read the test.txt file. And print appropriate message using exception to print weather line contains integer, character or float value.
4) Write a function called oops that explicitly raises a IndexError exception when called. Then write another function that calls oops inside a try/except statement to catch the error.
5) Change the oops function you just wrote to raise an exception you define yourself, called MyError, and pass an extra data item along with the exception. Then, extend the try statement in the catcher function to catch this exception and its data in addition to IndexError, and print the extra data item.

SET B

1) Write a Python class to find validity of a string of parentheses, '(', ')', '{', '}', '[' and ']. These brackets must be close in the correct order, for example "()" and "()[]{}" are valid but "[)", "({[)]" and "{{{" are invalid using exception.
2) Define a class Date(Day, Month, Year) with functions to accept and display it. Accept date from user. Throw user defined exception “invalidDateException” if the date is invalid.

SET C

1) Change the oops function in question 4 from SET A to raise an exception you define yourself, called MyError, and pass an extra data item along with the exception. You may identify your exception with either a string or a class. Then, extend the try statement in the catcher function to catch this exception and its data in addition to IndexError, and print the extra data item. Finally, if you used a string for your exception, go back and change it be a class instance.

2) Write a function called safe (func, *args) that runs any function using apply, catches any exception raised while the function runs, and prints the exception using the exc_type and exc_value attributes in the sys module. Then, use your safe function to run the oops function you wrote in Exercises 3. Put safe in a module file called tools.py, and pass it the oops function interactively. Finally, expand safe to also print a Python stack trace when an error occurs by calling the built-in print_exc() function in the standard traceback module (see the Python library reference manual or other Python books for details)

Assignment Evaluation

0: Not Done [ ] 1: Incomplete [ ] 2: Late Complete [ ]

3: Needs Improvement [ ] 4: Complete [ ] 5: Well Done [ ]

Signature of Instructor