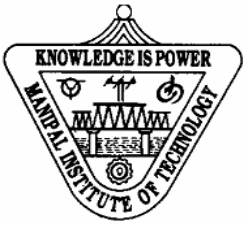
MANIPAL INSTITUTE OF TECHNOLOGY
Manipal – 576 104

DEPARTMENT OF INFORMATION & COMMUNICATION TECHNOLOGY



CERTIFICATE

This is to certify that Ms./Mr. ……...…… Reg.No. …..…… Section: …… Roll No: ……... has satisfactorily completed the lab exercises prescribed for **Advanced Programming Lab** [**ICT 3166**] of Third Year B. Tech. CCE Degree at MIT, Manipal, in the academic year 20_ _- 20_ _.

Date: …….....

Signature of the faculty

CONTENTS

Course Objectives

- Understand the basics of Python scripting language
- Demonstrate database connectivity through front end.
- Design and develop static and dynamic web pages.
- Apply the skills learnt to develop full-fledged applications.

Course Outcomes

At the end of this course, students will be able to

- Explain the basics of Python scripting
- Describe web application and database connectivity
- Develop dynamic web pages with good aesthetics, latest technical know-how's.
- Apply the skills learnt to work in one or more application domain.

Evaluation plan

Split up of 60 marks for Regular Lab Evaluation
Six regular evaluations will be carried out in alternate weeks. Each evaluation is for 10 marks and following is the split up: Record : 4 Marks Evaluation: 4 Marks Execution: 2 Marks Total = 10 Marks Total Internal Marks: 6 * 10 =60 Marks
End Semester Lab evaluation: 20 marks (Duration 2 hrs)
Program write up: 08Marks Program execution: 12 Marks Total: 8 + 12 = 20 Marks
Project evaluation: 20 marks

INSTRUCTIONS TO THE STUDENTS

Pre- Lab Session Instructions

1. Students should carry the Lab Manual Book and the required stationery to every lab session
2. Be in time and follow the institution dress code
3. Must sign in the log register provided
4. Make sure to occupy the allotted seat and answer the attendance
5. Adhere to the rules and maintain the decorum

In- Lab Session Instructions

- Follow the instructions on the allotted exercises
- Show the program and results to the instructors on completion of experiments
- On receiving approval from the instructor, copy the program and results in the lab record
- Prescribed textbooks and class notes can be kept ready for reference if required

General Instructions for the exercises in Lab

- Implement the given exercise individually and not in a group.
- The programs should meet the following criteria:
    - Programs should be interactive with appropriate prompt messages, error messages if any, and descriptive messages for outputs.
    - Comments should be used to give the statement of the problem.
    - Statements within the program should be properly indented.
- Plagiarism (copying from others) is strictly prohibited and would invite severe penalty in evaluation.
- In case a student misses a lab, he/ she must ensure that the experiment is completed before the next evaluation with the permission of the faculty concerned.
- Students missing out lab on genuine reasons like conference, sports or activities assigned by the Department or Institute will have to take **prior permission** from the HOD to attend **additional lab**(with other batch) and complete it **before** the student goes on leave. The student could be awarded marks for the write up for that day provided he submits it during the **immediate** next lab.

- Students who fall sick should get permission from the HOD for evaluating the lab records. However attendance will not be given for that lab.
- Students will be evaluated only by the faculty with whom they are registered even though they carry out additional experiments in other batch.
- Presence of the student during the lab end semester exams is mandatory even if the student assumes he has scored enough to pass the examination
- Minimum attendance of 75% is mandatory to write the final exam.
- If the student loses his book, he/she will have to rewrite all the lab details in the lab record.
- Questions for lab tests and examination are not necessarily limited to the questions in the manual, but may involve some variations and / or combinations of the questions.

THE STUDENTS SHOULD NOT

- Bring mobile phones or any other electronic gadgets to the lab.
- Go out of the lab without permission.

1. Basics of Python Programming - I

Objectives

- To understand the basic data types of Python programming
- Implement programs using lists, tuples and Strings

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991.

1.1 Basics

Python uses whitespace and indentation to determine block structure an example is shown below:

```
# This is Python code.
n = 1
r = 1
while n < 10:
        r = r * n
        n = n + 1
        print(n)
print(r)
```

1.1.1 Comments

Anything following a # symbol in a Python file is a comment. Multiline comments are represented as "'(triple single quotes) or """ (triple double quotes)

1.2 Numbers

Python offers four kinds of numbers: integers, floats, complex numbers, and boolean.

Integers

```
>>> 5 / 2
2.5
>>> 5 // 2
2
>>>5 % 2
1
>>> 2 ** 8
256
```

Float Point Data

```
>>> x = 4.3 ** 2.4
>>> x
33.137847377716483
>>> 3.5e30 * 2.77e45
9.695000000000002e+75
>>> 1000000001.0 ** 3
1.000000003e+27
```

Complex numbers

Operators that could be used on complex numbers are + - * / and % cannot be used.

```
>>> (3+2j) ** (2+3j)
(0.68176651908903363-2.1207457766159625j)
>>> x = (3+2j) * (4+9j)                                    >>x
(-6+35j)
```

```
>>> x.real
-6.0
>>> x.imag
35.0
```

User Input

Function input is used to get the user input and print() function is used to print the data. consider the following example:

```
>>> name = input("College?")
College? MIT
>>> print(name)
MIT
```

1.3 Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One of the differences between them is that all the items belonging to a list can be of different data type. i.e. A list element can be any Python object.

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator. A list automatically grows or shrinks in size as needed.

The statement shown below creates the list and assigns it to a variable L1.

```
L1=[1,2,"hi",[7,8],10.5]
```

Slicing operation in Python can extract or assign to an entire sublist at once. The syntax of slicing is as follows:

```
list[index1:index2]
```

Above statement extract all items including index1 and up to (but not including) index2 into a new list. Consider the following examples on list operations:

```
>>> x = ["first", "second", "third", "fourth"]
>>> x[0]
'first'
>>> x[-1]
'fourth'
>>> x[-2]
'third'
>>> x[1:-1]
['second', 'third']
>>> x[0:3]
['first', 'second', 'third']
>>> x[-2:-1]
['third']
>>> x[:3]
['first', 'second', 'third']
>>> x[-2:]
['third', 'fourth']
```

Consider the following examples for add, remove, and replace elements in a list or to obtain an element or a new list that is a slice from it.

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[1] = "two"
>>> x[8:9] = []
>>> x
```

```
[1, 'two', 3, 4, 5, 6, 7, 8]
>>> x[5:7] = [6.0, 6.5, 7.0]
>>> x
[1, 'two', 3, 4, 5, 6.0, 6.5, 7.0, 8]
>>> x[5:]
[6.0, 6.5, 7.0, 8]
```

List operators and methods are described in the Table 1.1.

Examples on list operators:

Table 1.1: List Operators and Methods

Operator / Method	Description
in (membership)	Returns True if object obj is in list. Otherwise, returns False
+ (List concatenation)	returns a new list and does not modify parameters
*	Returns list repeated by n times. Does not modify list1
list.append(object)	Appends a single object
list.count(value)	Counts the number of occurrences of value in list1
list.reverse()	Reverses the list
list.insert(index,object)	Inserts an object at index position
list.index(object)	Returns index of an object
list.remove(value)	Deletes the first occurrence of the value from the list
list.pop()	Removes the last item from the list
list.sort()	Sort the list having similar kind of objects
list.clear()	Makes the list empty
len(list)	Returns the length of a list
del(list[index])	Removes a list element or slice
min(list)	Returns the smallest element in a list
max(list)	Returns the largest element in a list
and max(list)	

```
>>> 4 in [1, 2, 3, 4, 5]
True
>>> 5 not in [1, 2, 3, 4, 5]
False
>>> 3 in ["one", "two", "three"]
False
>>> z = [1, 2, 3] + [4, 5]
```

```
>>> z
[1, 2, 3, 4, 5]
>>> z = [None] * 4
>>> z
[None, None, None, None]
```

1.4 Tuples

Tuples are data structures that are very similar to lists, but they can't be modified. They can only be created. Tuple is used as keys of dictionaries.

1.4.1 Basics - Tuple

Tuple is created as follows:

```
>>> x = ('a', 'b', 'c')
```

Functions of tuples is listed below:

- count(obj)
- index(ele)
- min(object)
- max(object)
- list(tuple-object)

Special opertors supported by tuple:

- in
- +
- *

1.4.2 Packing and unpacking tuples

The values on the right-hand side are packed into a tuple and then unpacked into the variables on the left-hand side. Packing and unpacking can be performed using list delimiters as well.

```
(one, two, three, four) = (1, 2, 3, 4)
                    Or
one, two, three, four = 1, 2, 3, 4
>>> one
1
>>> two
2
>>> list("hello world")
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> tuple("hello world")
('h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd')
```

An element marked with * Absorbs any number of elements which does not match the other elements . Consider the following examples:

```
>>> x = (1, 2, 3, 4)
>>> a, b, *c = x
>>> a, b, c
(1, 2, [3, 4])
>>> a, *b, c = x
>>> a, b, c
(1, [2, 3], 4)
```

1.5 Strings

Python treats single quotes the same as double quotes. Creating strings is as simple as assigning a value to a variable. Example is shown below

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

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows either pair of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 to the end. The plus (+) sign is the string concatenation operator and the asterisk is the repetition operator. Syntax for accesing strings using slicing:

```
s[x:y:step]
x is included
y th index is excluded
Step is positive or negative number.
```

Example to reverse the string using slicing:

```
s="Hello"
s=s[::-1]
```

String methods of Python is shown in Table 1.2.

String Methods Example of String Methods

```
n = 'one, two, three, four'
print(n.split(', ', 2))
print(n.split(', ', 1))
print(n.split(', ', 5))
print(n.split(', ', 0))
```

Table 1.2: String Methods

Method	Description
str.split([separator [, maxsplit]])	Returns a list of substrings in the string
str.join(sequence)	Combines a list of strings as a single string
int([n=0, base=10])	Convert strings into integer
float([x])	Convert strings into floating point numbers
strip([chars])	Removes any whitespace at the beginning or end of the string
find(substring,[start=0,[end]])	Returns the position of the first character of the first instance of substring in the string object, or –1 if substring doesn't occur in the string.
x.replace(arg1, arg2)	Replace occurrences of arg1 with arg2.
count(substring[,start[,end]]	Returns the number of times the given substring occurs the given string.
startswith(str,[beg,[end]])	Returns true or false
endswith(suffix[, start[, end]])	Returns true or false

Output of the above code is shown below:

```
['one', 'two', 'three, four']
['one', 'two, three, four']
['one', 'two', 'three', 'four']
['one, two, three, four']
```

```
s = "-";
seq = ("a", "b", "c");
print s.join( seq ) # prints a-b-c
>>> "::".join(["Separated", "with", "colons"])
'Separated::with::colons'

>>> "".join(["Separated", "by", "nothing"])
'Separatedbynothing'
```

```
>>> x = "Mississippi"
>>> x.split("ss")
['Mi', 'i', 'ippi']
```

```
>>> int('10000', 8)  #Octal eqivalent
4096
```

```
>>> int('101', 2)
5
>>> int('ff', 16)
255
>>> int('123456', 6)
Traceback (innermost last):
File "<stdin>", line 1, in ?
ValueError: invalid literal for int()
with base 6: '123456'
>>>x='www.python.org'
>>> x.strip("w") #Strips off all ws
'.python.org'
>>> x.strip("gor") #Strips off all g's, o's, and r's
'www.python.'
>>> x.strip(".gorw")
'python'
>>> x = "Mississippi"
>>> x.replace("ss", "+++")
'Mi+++i+++ippi'
```

String constants supported by python is shown in Table 1.3.

1.6 Control Flow Statement

Python has a full range of structures to control code execution and program flow, including common branching and looping structures.

The syntax of the if...else statement, while and for statements are shown below:

Table 1.3: String Constants

Constant	Description
string.whitespace	characters space, tab, linefeed, return, formfeed, and vertical tab.
string.hexdigits	'0123456789abcdefABCDEF'
string.octdigits	'01234567'
string.ascii_lowercase	'abcdefghijklmnopqrstuvwxyz'
string.ascii_uppercase	'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
string.ascii_letters	all upper and lowercase letters

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

```
while condition:
        statements
post-code
```

```
for item in sequence:
        statements
else:
        statements
post-code
```

Range Function

The built-in function range() is the right function to iterate over a sequence of numbers. It generates an iterator of arithmetic progressions. It generates an iterator to progress integers starting with 0 upto n-1.

Examples are shown below:

```
>>> range(5)
range(0, 5)
>>> list(range(5))
```

```
[0, 1, 2, 3, 4]

>>> for var in list(range(5)):
        print (var)
0
1
2
3
4
```

Enumerate function

This is similar to range but has the advantage that the code is clearer and easier to understand. The enumerate function returns tuples such as (index, item).

Syntax of enumearate function is whown below:

```
enumerate(iterable, start=0)
```

Examples on enumerate() are listed below:

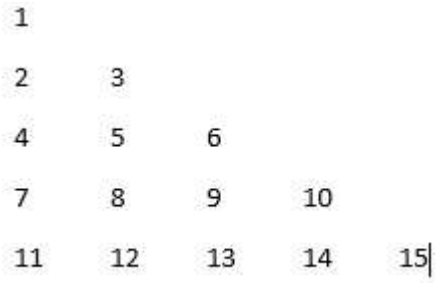
```
a=[5,8,9,5]
for i in enumerate(a):
print(i)
Output
(0, 5)
(1, 8)
(2, 9)
(3, 5)

x = [1, 3, -7, 4, 9, -5, 4]
for i, n in enumerate(x):
```

```
12 if n < 0:
13 print("Found a negative number at index ", i)
14
15 Output
16 Found a negative number at index  2
17 Found a negative number at index  5
```

Lab Exercises

1. Given two list of numbers, create a new list such that it contains odd numbers from the first list and even numbers from the second list.

2. Write a Python program to display the following pattern.



3. Write a Python program to do the following.

   (a) Read 'n' strings.

   (b) Count the number of strings with same first and last characters and having the string length of 2 or more.

   (c) Display the strings having odd length.

Additional Exercises

1. Write a Python progrm to generate prime numbers between 'n' and 'm'.

2. Write a program to check whether the given number is Armstrong number or not.

3. Read a string from the user and check if it is a hexa decimal number or not.

2. Basics of Python Programming - II

Objectives

- To understand sets and dictionaries of Python programming
- Implement the programs using sets and dictionaries

2.1 Sets

A set in Python is an unordered collection of objects. Membership and uniqueness are the main properties of the set. The items in a set are immutable and hashable. This means that int's, floats, strings, and tuples can be members of a set, but lists, dictionaries, and sets themselves can't be the members of the set. + and * does not work on sets.
Methods supported by sets are as follows:

- max
- min
- add(object)
- remove(object)

Operators supported by sets are listed below:

- in
- len
- | to get the union, or combination, of two sets
- & to get their intersection
- ∧ to find their symmetric difference—that is, elements that are in one set or the other but not both.

Examples on set operations:

```
>>> set([1,2,3])|set([3,4,5]) union
{1, 2, 3, 4, 5}

>>> set([1,2,3])&set([3,4,5]) intersection
{3}

>>> set([1,2,3])^set([3,4,5]) symmetric difference
{1, 2, 4, 5}
```

2.1.1 Frozensets

Sets aren't immutable and hashable, they can't belong to other sets. Hence they cannot be the members of other sets. Frozenset, which is just like a set but can't be changed after creation. Frozensets are immutable and hashable, they can be members of other sets.

Following code demonstrates the use of frozenset:

```
>>> x=set([2,3,4])
>>> x
set([2, 3, 4])
>>> y=set([4,7,9])
>>> z=frozenset(x)
>>> z
frozenset([2, 3, 4])
>>> x.add(y)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
>>> x.add(z)
```

```
>>> x
set([2, 3, 4, frozenset([2, 3, 4])])
>>>
```

2.2 Dictionaries

Dictionary is also called as associative array or hash table. Python dictionaries store the values interms of key, value pair. Integers, strings, or other immutable Python objects are used as keys. Both lists and dictionaries provide indexed access to arbitrary values. Creation and accessing elements of dictionary is shown below:

```
y={}                # empty dictionary
y[0]="hello"  # new key value pair is added
y[1]="apt"
y holds {0:"hello",1:"apt"}
```

General form for adding new key, value pair and accessing the value from the dictionary is shown below:

```
dict_name[key]=value
dict_name[key]
```

Lab Exercises

1. Write a Python program to count the number of words in a given sentence using dictionaries.

2. Write a Python program to do the following:

   - Read the order of two matrices.
   - Read the values and store nonzero values in to the dictionary.

- Add the matrices represented in the dictionary form and display the result in two dimensional matrix form.

3. Write a Python program to do the following:

   (a) Read the dictionary from user. Key should be a random number generated by the program [Note: import random random.randrange(0,100) generates random number between 0 to 99.]

   (b) Find the average, if the values of the dictionary are numbers and display the result.

   (c) If the values are strings, then concatenate them and display the result.

   (d) Search the dictionary based on value which is a string.

   (e) Display the dictionary values having strings with only special characters.

Additional Exercises

1. With an example demonstrate the cashing with dictionaries.

2. Write a Python program to find the union, intersection and difference of two lists without using any of the type casting functions, built-in functions and built-in methods. Input lists contain only integer values.

3. Write a python program to do the following:

   (a) Create a text file with minimum eight lines.

   (b) Read the data from the file.

   (c) Store the data in dictionary form considering line numbers as keys, string and total length of the string as values. Value is of type list. Print the resultant dictionary.

(d) Create a dictionary from the file, considering letters represented as keys and frequencies of letters represented as values. Print the dictionaty.

3. Functions, Parameter Options, Local, Nonlocal, Global Variables

Objectives

- To implement programs using concepts of functions
- To understand the use of local and global variables with respect to functions

3.1 Basic function definitions

The basic syntax for a Python function definition is as follows:

```
1 def name(parameter1, parameter2, . . .):
2         Statements \\
```

Python uses indentation to delimit the body of the function definition and parameters are separated by comma. Consider the following simple function which computes the factorial of a number.

```
1 >>>def fact(n):
2 ...       """Return the factorial of the given number"""
3 ...       r = 1    1
4 ...       while n > 0:
5 ...                 r = r * n
6 ...                 n = n - 1
7 ...       return r
```

The second line in the above code is an optional documentation string or docstring. The command functionname.doc prints the documentaion string of

a function. The intention of docstrings is to describe the external behavior of a function and the parameters it takes, whereas comments should document internal information about how the code works. Docstrings are strings that immediately follow the first line of a function definition and are usually triple quoted to allow for multiline descriptions. Browsing tools are available that extract the first line of document strings.

Procedure vs Functions

In some languages, a function that doesn't return a value is called a procedure. Although it is possible to write functions that don't have a return statement, they aren't really procedures. All Python procedures are functions; if no explicit return is executed in the procedure body, then the special Python value None is returned and if return arg is executed, then the value arg is immediately returned. So all Python functions return values. Nothing else in the function body is executed once a return has been executed.

```
1 >>>fact(4)
2 24
3 >>>x=fact(4)
4 >>>x
5 24
6 >>>
```

The return value isn't associated with a variable 1. The fact function's value is printed in the interpreter only. The return value is associated with the variable x is 24.

3.2 Function parameter options

Most functions need parameters, and each language has its own specifications for how function parameters are defined. Python is flexible and provides three

options for defining function parameters. These are outlined in this section.

3.2.1 Positional parameters

The simplest way to pass parameters to a function in Python is by position. In the first line of the function, you specify definition variable names for each parameter; when the function is called, the parameters used in the calling code are matched to the function's parameter variables based on their order. The following function computes x to the power of y:

```
>>> def power(x, y):
...     r = 1
...     while y > 0:
...         r = r * x
...         y = y - 1
...     return r
...
>>> power(3, 3)
27
```

The power function requires that, the number of parameters used by the calling code exactly match the number of parameters in the function definition otherwise a TypeError exception will be raised as shown below:

```
>>> power(3)

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

3.2.2 Default Values

Function parameters can have default values, which you declare by assigning a default value in the first line of the function definition as follows:

def fun(arg1, arg2=default2, arg3=default3, . . .)

Any number of parameters can be given default values. Parameters with default values must be defined as the last parameters in the parameter list. This is because Python, like most languages, pairs arguments with parameters on a positional basis. There must be enough arguments to a function that the last parameter in that function's parameter list that doesn't have a default value gets an argument.

The following function also computes x to the power of y. But if y isn't given in a call to the function, the default value of 2 is used, and the function is just the square function:

```
>>> def power(x, y=2):
...     r = 1
...     while y > 0:
...         r = r * x
...         y = y - 1
...     return r
```

Following interactive session shows the effect of the default arguments:

```
>>> power(3, 3)
27
>>> power(3)
9
```

3.2.3 Passing arguments by parameter name

It is also possible to pass arguments into a function using the name of the corresponding function parameter, rather than its position. Continuing with the previous interactive example, consider the following:

```
1 >>> power(2, 3)
2 8
3 >>> power(3, 2)
4 9
5 >>> power(y=2, x=3)
6 9
```

The arguments to power in the final invocation(line No.5) of it are named. Hence their order is irrelevant. The arguments are associated with the parameters of the same name in the definition of power. So power(3,2) is computed. This type of argument passing is called keyword passing.

Keyword passing, in combination with the default argument capability of Python functions, can be highly useful when you're defining functions with large numbers of possible arguments, most of which have common defaults. For example, consider a function that's intended to produce a list with information about files in the current directory and that uses Boolean arguments to indicate whether that list should include information such as file size, last modified date, and so forth, for each file. The function with all these parameters is defined as follows:

```
1 def fl_info(size=False, create_dt=False, mod_dt=False, ...):
2 #...get file names...
3 if size:
4 # code to get file sizes goes here
5 if create_date:
6 # code to get create dates goes here
```

```
    ....
    return fileinfostructure
```

Call the function list _file _info from by specifying required parameters (in this example, the file size and modification date but not the creation date):

```
fileinfo = list_file_info(size=True, mod_date=True)
```

This type of argument handling is particularly suited for functions with very complex behavior, and one place such functions occur is in graphical user interfaces i.e in Tkinter package to build GUIs in Python. Here, you'll find that the use of optional, keyword named arguments like this is invaluable.

3.2.4 Variable numbers of arguments

Python functions can also be defined to handle variable numbers of arguments. There are two ways to do this. One way handles the relatively familiar case where you wish to collect an unknown number of arguments at the end of the argument list into a list. The other method can collect an arbitrary number of keyword-passed arguments, which have no correspondingly named parameter in the function parameter list, into a dictionary. These two mechanisms are discussed next section.

Variable numbers of arguments – non keywords Prefixing the final parameter name of the function with a * causes all excess non-keyword arguments in a call of a function (that is, those positional arguments not assigned to another parameter) to be collected together and assigned as a tuple to the given parameter. Consider the function maximum which takes the variable number of arguments and returns the maximum.

```
>>> maximum(3, 2, 8)
8
>>> maximum(1, 5, 9, -2, 2)
```

```
4 9
```

A function to find the maximum in a list which takes variable number of arguments as parameter is shown below:

```
1 >>> def maximum(*numbers):
2 ...     if len(numbers) == 0:
3 ...         return None
4 ...     else:
5 ...         maxnum = numbers[0]
6 ...         for n in numbers[1:]:
7 ...             if n > maxnum:
8 ...                 maxnum = n
9 ...         return maxnum
```

Variable numbers of arguments – keywords

An arbitrary number of keyword arguments can also be handled. If the final parameter in the parameter list is prefixed with **, it will collect all excess keyword-passed arguments into a dictionary. The index for each entry in the dictionary will be the keyword (parameter name) for the excess argument. The value of that entry is the argument itself. An argument passed by keyword is excess in this context if the keyword by which it was passed doesn't match one of the parameter names of the function. Consider the following example:

```
1 >>> def example_fun(x, y, **other):
2 ...     print("x:{0},y:{1},keys in 'other': {2}"
3 ...     .format(x,y,list(other.keys())))
4 ...     other_total = 0
5 ...     for k in other.keys():
6 ...     other_total = other_total + other[k]
7 ...     print("The total of values in 'other' is {0}".
```

```
8 ...format(other_total))
```

Trying out this function in an interactive session reveals that it can handle arguments passed in under the keywords foo and bar, even though these aren't parameter names in the function definition:

```
1 >>> example_fun(2, y="1", val1=3, val2=4)
```

x: 2, y: 1, keys in 'other': ['val1', 'val2'] The total of values in 'other' is 7

3.2.5 Mixing argument-passing techniques

It's possible to use all of the argument-passing features of Python functions at the same time, although it can be confusing if not done with care. Rules govern what you can do. See the documentation for the details. Mutable objects as arguments Arguments are passed in by object reference. The parameter becomes a new reference to the object. For immutable objects (such as tuples, strings, and numbers), what is done with a parameter has no effect outside the function. But if you pass in a mutable object (for example, a list, dictionary, or class instance), any change made to the object will change what the argument is referencing outside the function. Reassigning the parameter doesn't affect the argument, as shown in figures 3.1 and 3.2:

```
1 >>> def f(n, list1, list2):
2 ...       list1.append(3)
3 ...       list2 = [4, 5, 6]
4 ...       n = n + 1
5 ...
6 >>> x = 5
7 >>> y = [1, 2]
8 >>> z = [4, 5]
9 >>> f(x, y, z)
```

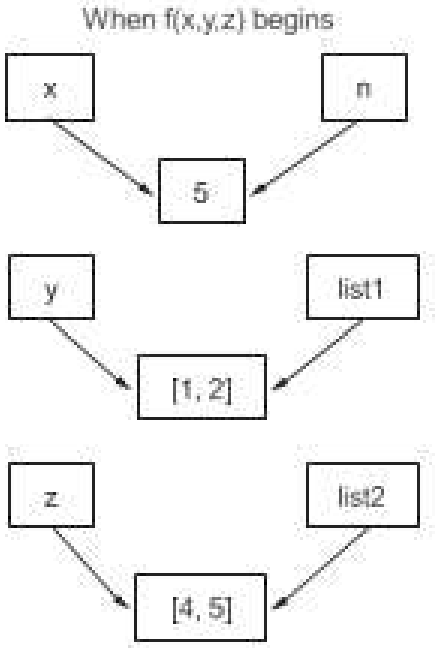


Figure 3.1: At the beginning of function f(), both the initial variables and the function parameters refer to the same objects.

```
10 >>> x, y, z
11 (5, [1, 2, 3], [4, 5])
```

The variable x isn't changed because it's immutable. Instead, the function parameter n is set to refer to the new value of 6. Likewise, variable z is unchanged because inside function f, its corresponding parameter list2 was set to refer to a new object, [4, 5, 6]. Only y sees a change because the actual list it points to was changed.

3.2.6 Local, nonlocal, and global variables

Consider the following example. where the variables r and n are local to any particular call of the factorial function; changes to them made when the function is executing have no effect on any variables outside the function. Any variables in the parameter list of a function, and any variables created within a function by an assignment (like r = 1 in fact), are local to the function.

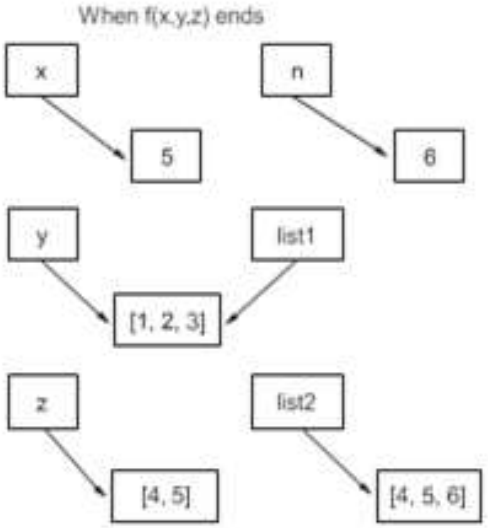


Figure 3.2: At the end of function f(), y (list1 inside the function) has been changed internally, whereas n and list2 refer to different objects.

```
1 def fact(n):
2 """Return the factorial of the given number."""
3 r = 1
4 while n > 0:
5 r = r * n
6 n = n - 1
7 return r
```

You can explicitly make a variable global by declaring it so before the variable is used, using the global statement. Global variables can be accessed and changed by the function. They exist outside the function and can also be accessed and changed by other functions that declare them global or by code that's not within a function. Following example demonstartes the difference between local and global variables:

```
1 >>> def fun():
2 ...     global a
3 ...     a = 1
```

```
4 ...         b = 2
```

This defines a function that treats a as a global variable and b as a local variable and attempts to modify both a and b.

```
1 >>> a = "one"
2 >>> b = "two"
3 >>> fun()
4 >>> a
5 1
6 >>> b
7 'two'
```

The assignment to a within fun is an assignment to the global variable a also existing outside of fun. Because a is designated global in fun, the assignment modifies that global variable to hold the value 1 instead of the value "one". The same isn't true for b the local variable called b inside fun starts out referring to the same value as the variable b outside of fun, but the assignment causes b to point to a new value that's local to the function fun.

Similar to the global statement is the non local statement, which causes an identifier to refer to a previously bound variable in the closest enclosing scope. Global is used for a top-level variable, whereas non local can refer to any variable in an enclosing scope. Consider the following functions with and without nonlocal variables.

```
1 def main_function():
2   """ Without using non local"""
3           msg = "main-function!"
4 def nested_function():
5                   msg = "Nested Function!"
6                   print(msg)
```

```
        nested_function()
        print(msg)
main_function() # function call
```

```
def main_function():
 """ Using non local"""
        msg = "main-function!"
        def nested_function():
                nonlocal msg #nonlocal variable
                msg = "Nested Function!"
                print(msg)
        nested_function()
        print(msg)

main_function() ##Function Call
```

Assiging functions to variables

A variable that refers to a function can be used in exactly the same way as the function. Follwing example demonstrates this.

```
def add(a,b):
        return a+b
def mul(a,b):
        return a*b
```

A variable that refers to a function can be used in exactly the same way as the function as shown below:

```
var1=add;print(var1(2,4))  #prints 6
var1=mul;print(var1(2,4))  #prints 8
d={'var1':add,'var2':mul}
d['var1'](2,4)
```

```
d['var2'](2,4)
```

3.3 Lambda Expressions

Lambda expressions are anonymous little functions that can be defined as inline functions

```
f=lambda a,b,c:a+b+c
print(f(2,3,4))    #prints 9
```

Another example for lambda function which prints length of each item in the list.

```
l=[1234,23,1,234]
l.sort(key=lambda item:len(str(item)))
print(l) #prints [1, 23, 234, 1234]
```

3.4 Generator functions

A generator function is a special kind of function that can be used to define used defined iterators. After defining a generator function, each iteration's value is returned using the yield keyword. When there are no more iterations, an empty return statement or flowing off the end of the function ends the iterations.

```
def four( ):
        """ generator function """
        x = 0
        while x < 4 :
                print ("in-generator,x=",x)
                yield x
```

```
7                       x += 1
```

Following code is used to test the working of generator function

```
1 for i in four():
2         print(i)
```

Output

```
1 ('in-generator,x=', 0)
2 0
3 ('in-generator,x=', 1)
4 1
5 ('in-generator,x=', 2)
6 2
7 ('in-generator,x=', 3)
8 3
```

Lab exercises

1. Write a Python function to take list as parameter and multiply all the numbers in a list. Sample List : (8, 2, 3, -1, 7) Expected Output : -336

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

3. Write a Python program to detect the number of local variables declared in a function.

Additional Exercises

1. Write a Python function that accepts a string and calculates the number of upper case and lower case letters. Sample String : 'The quick Brow Fox' Expected Output : No. of Upper case characters : 3 No. of Lower case Characters : 12

2. Write a Python function that prints out the first n rows of Pascal's triangle.

3. Write a Python program to access a function inside a function.

4. Modules

Objectives

- Define and use the modules
- Understand different ways of importing modules
- Modify module search path using libraries

Modules are used to organize larger Python projects. The Python standard library is split into modules to make it more manageable. You don't need to organize your own code into modules, but if you're writing any programs that are more than a few pages long, or any code that you want to reuse, you should probably do so.

4.1 Module

A module is a file containing code. A module defines a group of Python functions or other objects, the name of the module is same as the name of the file. Modules most often contain Python source code, but they can also be compiled C or C++ object files. Compiled modules and Python source modules are used the same way. As well as grouping related Python objects, modules help avoid name clash problems. For example, you might write a module for your program called mymodule, which defines a function called reverse. In the same program, you might also wish to use somebody else's module called othermodule, which also defines a function called reverse, but which does something different from your reverse function. In a language without modules, it would be impossible to use two different functions named reverse. In Python, it's trivial—you refer to them in your main program as mymodule.reverse and othermodule.reverse. This is because Python uses

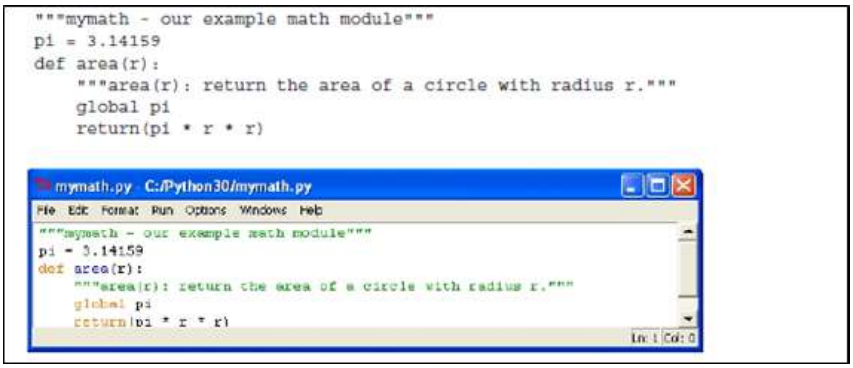


Figure 4.1: An IDLE edit window provides the same editing functionality as the shell window, including automatic indentation and colorization.

namespaces. A namespace is essentially a dictionary of the identifiers available to a block, function, class, module, and so on. Be aware that each module has its own namespace, and this helps avoid naming conflicts. Modules are also used to make Python itself more manageable. Most standard Python functions aren't built into the core of the language but instead are provided via specific modules, which you can load as needed.

Following example shows how to create and use the module. Create a text file called mymath.py, and in that text file enter the Python code shown in Figure 4.1. (If you're using IDLE, select New Window from the File menu and start typing, as shown in Figure 4.1.)

Save this for now in the directory where your Python executable is. This code merely assigns pi a value and defines a function. The .py filename suffix is strongly suggested for all Python code files. It identifies that file to the Python interpreter as consisting of Python source code. As with functions, you have the option of putting in a document string as the first line of your module. Now, start up the Python Shell and type the following:

```
>>> pi
Traceback (innermost last):
File "<stdin>", line 1, in ?
NameError: name 'pi' is not defined
>>> area(2)
```

```
6 Traceback (innermost last):
7 File "<stdin>", line 1, in ?
8 NameError: name 'area' is not defined
```

In other words, Python doesn't have the constant pi or the function *area* as built in. Now, type

```
1 >>> import mymath
2 >>> pi
3 Traceback (innermost last):
4 File "<stdin>", line 1, in ?
5 NameError: name 'pi' is not defined
6 >>> mymath.pi
7 3.1415899999999999
8 >>> mymath.area(2)
9 12.56636
10 >>> mymath.__doc__
11 """mymath - our example math module"""
12 >>> mymath.area.__doc__
13 area(r): return the area of a circle with radius r.
```

pi and *area* are accessed by prepending them with the name of the module that contains them. This guarantees name safety.

If you want to, you can also specifically ask for names from a module to be imported in such a manner that you don't have to prepend it with the module name. Definitions within a module can access other definitions within that module, without prepending the module name.

4.2 The import statement

The import statement takes three different forms.

- import modulename
- from modulename import name1, name2, name3, . .
- from modulename import *

import modulename

Searches for a Python module of the given name, parses its contents, and makes it available. The importing code can use the contents of the module, but any references by that code to names within the module must still be prepended with the module name. If the named module isn't found, an error will be generated.

from modulename import name1, name2, name3, . .

Each of name1, name2, and so forth from within modulename is made available to the importing code. The code after the import statement can use any of name1, name2, name3, and so on without prepending the module name.

from modulename import ∗

∗ stands for all the exported names in module name. Using this import statement it is possible to import all public names from module name those not begins with underscore. It is also possible to import list strings returned by the statement __all__exists

4.2.1 The module search path

Exactly where Python looks for modules is defined in a variable called path, which you can access through a module called sys. Following command displays lists of directories in the search path

```
1 >>> import sys
2 >>> sys.path
```

The sys.path variable is initialized from the value of the environment variable PYTHONPATH, if it exists, or from a default value that's dependent on the

installation.

4.2.2 Location for own modules

User defined modules are place in any one of the following locations:

- One of the directories that Python normally searches for modules.
- Same directory as the program.
- Create a directory (or directories) that will hold userdefined modules, and modify the sys.path variable so that it includes this new directory.

Example - Create a module Simple Python Script (save below as ¡name¿.py file)

```
1 #!/usr/bin/python3
2 print ("Hello, Python!")
```

Execution Steps in Command Prompt:

1. Open command prompt
2. Type the location of python .exe file and the python file (use .py extension) which you want to run.

Using Idle GUI

- Run IDLE.
- Click File, New Window.
- Enter your script in the "Untitled" window.
- In the "Untitled" window, select Run, Run Module (or press F5) to run your script.
- A dialog "Source Must Be Saved.

- Give a name with .py extension and then run the program.

Note: You can pass as many argument(s) to your python scripts (if required) from command line after the filename separated by spaces.

4.2.3 Private names in modules

The statement **from module import *** imports almost all names from a module except the names in the module beginning with an underscore. These names represents the private variables. Consider the following example to understand accessibility of these variables.

```
"""modtest: our test module"""
def f(x):
return x
def _g(x):
return x
a = 4
_b = 2
```

Now, start up an interactive session, and enter the following:

```
>>> from modtest import *
>>> f(3)
3
>>> _g(3)
Traceback (innermost last): File "<stdin>", line 1, in ?
NameError: name '_g' is not defined
>>> a
4
>>> _b
Traceback (innermost last):
```

```
File "<stdin>", line 1, in ?
NameError: name '_b' is not defined
```

The function *f* and variable *a* are imported, but the names _g and _b remain hidden outside of modtest. This behavior occurs only with the statement :

from ... import *

Following statements are used to access _g or _b:

```
>>> import modtest
>>> modtest._b
2
>>> from modtest import _g
>>> _g(5)
5
```

Sample module creation is shown below:

Simple Python Script (save below as ¡name¿.py file)

```
#!/usr/bin/python3
print ("Hello, Python!")
```

Execution Steps - Via Command Prompt

1. Open command prompt
2. Type the location of python .exe file and the python file (use .py extension) which you want to run.

Using Idle GUI

1. Run IDLE.

2. Click File, New Window.

3. Enter your script in the "Untitled" window.

4. In the "Untitled" window, select Run, Run Module (or press F5) to run your script.

5. A dialog "Source Must Be Saved.

6. Give a name with .py extension and then run the program.

Note: You can pass as many argument(s) to your python scripts (if required) from command line after the filename separated by spaces.

Lab exercises

1. Write a Python program to accept a number from the user and print the sine value, square root and log of entered number using built-in modules.

2. Repeat the Question 1 by considering complex number as user input.

3. Write a Python program to list all the environment variables using built-in modules.

4. Write a Python program to accept a User Name and display a greeting message to the user based on the system time using built-in modules.

5. Write a Python program to accept a sentence from the user and capitalize the first letter of each word in the sentence using built-in modules.

Additional Exercises

1. Write a custom Python module to reverse a string. Make use of the same module to check if a string is a palindrome or not.

2. Write a Python program to convert a JSON object to a string and vice versa.

3. Install third party libraries like pyperclip and emoji and demonstrate the same in Python programs. [hint: Use install command in command line before using third party libraries]

5. Classes and Inheritance

Objectives

- To understand the object oriented features of Python programming
- To implement the object oriented concepts in Python

5.1 Classes

A class in Python is a data type. All the data types built into Python are classes. A class is defined as follows:

```
class MyClass:
    Statements
```

Class name followed by Python statements, typically variable assignments and function definitions. After the class definition a new object of the class type (an instance of the class) is created as follows:

```
instance = MyClass()
```

A class and methods of a class is defined as follows:

```
class Person:
        def __init__(self, name, age):
                self.name = name
                self.age = age

        def display(self):
                print("Hello my name is " + self.name)

p1 = Person("Anu", 10)
```

```
10 p1.display()
```

__init__method, is called every time a new instance of a class is created. The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

5.1.1 Class variables

A class variable is a variable associated with a class, not an instance of a class, and is accessed by all instances of the class, in order to keep track of some class-level information, such as how many instances of the class have been created at any point in time. A class variable is created by an assignment in the class body, not in the __init __function, it can be seen by all instances of the class. In the following example *count* is a class variable:

```
1  class Person:
2          count=0
3          def __init__(self, name, age):
4                  self.name = name
5                  self.age = age
6                  Person.count += 1
7
8          def display(self):
9                  print("Hello my name is " + self.name)
10
11 p1 = Person("Anu", 10)
12 p2 = Person("Vishrut",15)
13 p1.display()
14 p2.display()
15 print("No. of Persons:", Person.count)
```

Output of the above code is as follows:

```
Hello my name is Anu
Hello my name is Vishrut
('No. of Persons:', 2)
```

To change the value of a class variable, access it through the class name, not through the instance variable self.

5.1.2 Static methods and Class methods

Static methods are created using the decorator **@staticmethod** and can be invoked using a class instance. Class methods are similar to static methods in that they can be invoked before an object of the class has been instantiated or by using an instance of the class. But class methods are implicitly passed the class they belong to as their first parameter, so you can code them more simply, as in listing 15.2.

```
from datetime import date
class Person:
        def __init__(self, name, age):
                self.name = name
                self.age = age

  # a class method
  @classmethod
        def fromBirthYear(cls, name, year):
                return cls(name, date.today().year-year)

  # a static method
  @staticmethod
        def isAdult(age):
```

```
15                  return age > 18
16
17 p1 = Person('Vishrut', 19)
18 p2 = Person.fromBirthYear('Anu', 1996)
19 print (p1.age)
20 print (p2.age)
21
22 print Person.isAdult(22) #Static method with class name
```

5.2 Inheritance

Inheritance is a powerful feature in object oriented programming.

It refers to defining a new class with little or no modification to an existing class. The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class. Derived class inherits features from the base class where new features can be added to it. This results in re-usability of code. Syntax for inheritance:

```
1 class BaseClass:
2         base class statements
3 class DerivedClass(BaseClass):
4         derived class statements
```

Consider the following example:

```
1 class Shape:
2         def __init__(self, x, y):
3                 self.x = x
4                 self.y = y
5 #Square is derived class of Shape class
```

```
class Square(Shape):
        def __init__(self, side=1, x=0, y=0):
                super().__init__(x, y)
                self.side = side
#Circle is derived class of Shape class
class Circle(Shape):
        def __init__(self, r=1, x=0, y=0):
                super().__init__(x, y)
                self.radius = r
```

5.2.1 Inheritance with class and instance variables

Inheritance allows an instance to inherit attributes of the class. Instance variables are associated with object instances, and only one instance variable of a given name exists for a given instance. Consider the following example:

```
class P:
        z = "Hello"
        def set_p(self):
                self.x = "Class P"
        def print_p(self):
                print(self.x)
class C(P):
        def set_c(self):
                self.x = "Class C"
        def print_c(self):
                print(self.x)
```

Consider the following statements to understand the inheritance with class and instance variables:

```
1
2 >>> c = C()
3 >>> c.set_p()
4 >>> c.print_p()
5 Class P
6 >>> c.print_c()
7 Class P
8 >>> c.set_c()
9 >>> c.print_c()
10 Class C
11 >>> c.print_p()
12 Class C
```

The object c in this example is an instance of class C. C inherits from P, but c doesn't inherit from some invisible instance of class P. It inherits methods and class variables directly from P. There is only one instance (c). Hence any reference to the instance variable x in a method invocation on c must refer to c.x. This is true regardless of which class defines the method being invoked on c.

Lab Exercises

1. Write a Python program to do the following:

   (a) Read details of 'n' employees from the user. Store each employee details as a tuple : (id, name, salary, department).

   (b) Store each tuple in a list.

   (c) Search the record based on employee id entered by the user and display the result.

2. Create a class Vehicle, which stores the name of the vehicle owner, vehicle identification number, name of the manufacturer. Derive a class

Passenger from the class Vehicle, with additional property, number of passengers. Write the methods 1) To take input from the user 2) To display the same.

3. Write a Python complete program by creating a class to get all possible unique subsets from a set of distinct integers. Sample Input : [4, 5, 6] Sample Output : [[], [6], [5], [5, 6], [4], [4, 6], [4, 5], [4, 5, 6]]

Additional Exercises

1. Write a Python program to define a class named Employee with the instance variables id, name, salary and department. Each employee's id should be auto generated in the order of creation starting from 2912. Define constructor, destructor, display method and a class method to compute total salary of all the employees. Read the details of 'n' employees from the user and store created 'n' objects in a list. Implement a search function which displays the details of an employee after reading his id from user.

2. Create a class called Car with the following attributes. i) Engine Number ii) Registration Number iii) Date of Registration iv) Colour of the car v)Owner of the car vi) Model of the car. In the above attributes, Owner attribute is an object of the Owner class which has the following attributes. i) Name of the Owner ii) Address iii)Profession iv) Driving license Number. Define the Owner class and create an array Own[] of owner objects. Write a method displayOwner() to display the details of the owner. In the Car class create an array Own[] of five objects. Also create an array of five Car objects. Define a method for displaying the details of the car. This method must make use of the displayOwner() method of the Owner class.

6. Database Connectivity and Web Designing

Objectives

- To understand the basics of Firebase
- To Design and develop dynamic web applications using Firebase

Firebase grants access to a real-time database. It is a cohesive, cross-platform product that uses Google Cloud to synchronize data across all clients sharing the same database.

6.1 Web application development using firebase

Firebase is a backend platform for building Web, Android and IOS applications. It offers real time database, different APIs, multiple authentication types and hosting platform. Advantages

- It is simple and user friendly. No need for complicated configuration.
- The data is real-time, which means that every change will automatically update connected clients.
- Firebase offers simple control dashboard.
- There are a number of useful services to choose.

Steps to use firebase

- Create firebase account.

Figure 6.1: Firebase App

- Create firebase app: You can create new app from the dashboard page. The following image 6.1 shows the app we created. We can click the Manage App button to enter the app.

- Create html/js App: You just need to create a folder where your app will be placed. Inside that folder, we will need index.html and index.js files. We will add Firebase to the header of our app. **Index.html**

```
<html>
<head>
<script src ="https://cdn.firebase.com/js/client/
2.4.2/firebase.js"></script>
<script type ="text/javascript" src ="index.js">
</script>
</head>
<body>
</body>
</html>
```

- Use NPM or browser to install firebase: If you want to use your existing app, you can use Firebase NPM or Bowers packages. Run one of the following command from your apps root folder.

```
npm install firebase --save
```

```
bower install firebase
```

The Firebase data is representing JSON objects. If you open your app from Firebase dashboard, you can add data manually by clicking on the + sign. We will create a simple data structure. Figure 6.2 shows thecreation of data structure. The command used is :

Figure 6.2: Data Structure Creation

```
var playersRef = firebase.database().ref("players/");
```

We can log Firebase to the console:

```
console.log(firebase)
```

We can create a reference to our player's collection using following code:

```
var ref = firebase.database().ref('players');
console.log(ref);
```

We could create this data by sending the following JSON tree to the player's collection.

```
['john', 'amanda']
```

This is because Firebase does not support Arrays directly, but it creates a list of objects with integers as key names. The reason for not using arrays is because Firebase acts as a real time database and if a couple of users were to manipulate arrays at the same time, the result could be problematic since array indexes are constantly changing. The way Firebase handles it, the keys (indexes) will always stay the same. We could delete john and amanda would still have the key (index) 1.

The set method will write or replace data on a specified path. Let us create a reference to the player's collection and set two players.

```
playersRef.set ({
        John: {
                number: 1,
                age: 30
    },

Amanda: {
number: 2,
age: 20
}
});
```

We can update the Firebase data in a similar fashion. Note that the path specified as **players/john**.

```
var johnRef = firebase.database().ref("players/John");
johnRef.update ({
"number": 10
});
```

When we refresh our app, we can see that the Firebase data is updating.

The push() method will create a unique *id* when the data is pushed. If we

want to create our players from the previous chapters with a unique *id*, we could use the code snippet given below.

```
var ref = new Firebase('https://tutorialsfirebase.
firebaseio.com');
var playersRef = ref.child("players");
playersRef.push ({
name: "John",
number: 1,
age: 30
});
playersRef.push ({
name: "Amanda",
number: 2,
age: 20
});
```

We can get any key from Firebase by using the key() method. For example, if we want to get our collection name, we could use the following snippet.

```
var ref = new Firebase('https://tutorialsfirebase.
firebaseio.com');
var playersRef=ref.child("players");
var playersKey=playersRef.key();
console.log(playersKey);
```

The console will log our collection name (players). Transactional data is used when you need to return some data from the database then make some calculation with it and store it back.

Let us say we have one player inside our player list as shown in the following Figure

We want to retrieve property, add one year of age and return it back to

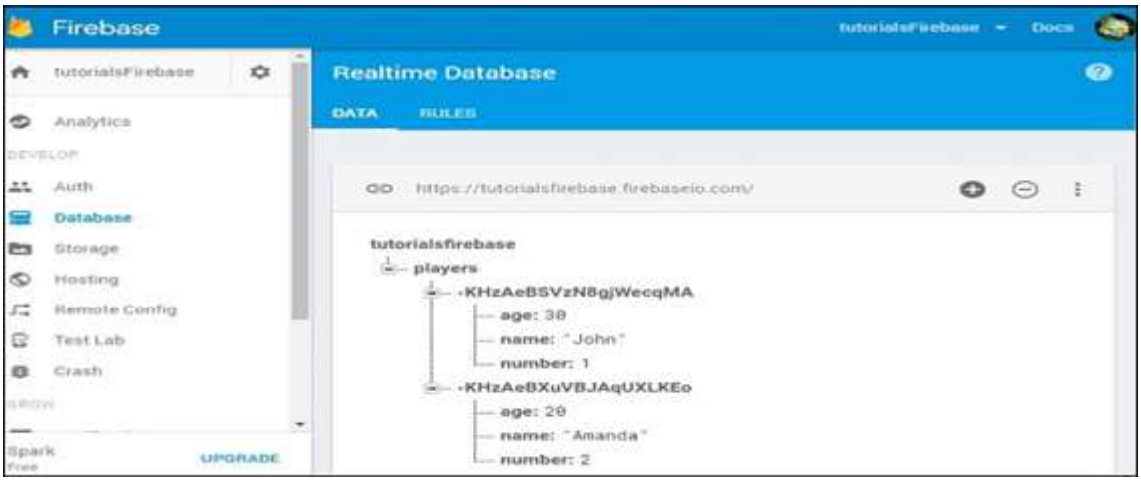
Firebase.

The amandaRef is retrieving the age from the collection and then we can use the transaction method. We will get the current age, add one year and update the collection. as shown in the following code

```
var ref = new Firebase('https://tutorialsfirebase.
firebaseio.com');
var amandaAgeRef = ref.child("players").child
("KGb1LgWbAMMnZC").child('age');
amandaAgeRef.transaction(function(currentAge)
{
        return currentAge + 1;
});
```

If we run this code, we can see that the age value is updated to 21. The following image shows the data we want to read: We can use the on() method to retrieve data. This method is taking the event type as "value" and then retrieves the snapshot of the data. When we add val() method to the snapshot, we will get the JavaScript representation of the data.

Let us consider the following example.




```
var ref = firebase.database().ref();
ref.on("value", function(snapshot) {
        console.log(snapshot.val());
}, function (error) {
        console.log("Error: " + error.code);
});
```

Firebase offers several different event types for reading data. Some of the most commonly used ones are described below. **value** The first event type is value.This event type will be triggered every time the data changes and it will retrieve all the data including children.

child_added

This event type will be triggered once for every player and every time a new player is added to our data. It is useful for reading list data because we get access of the added player and previous player from the list. Let us consider the following example.

```
var playersRef = firebase.database().ref("players/");
playersRef.on("child_added", function(data, prevChildKey)
{
        var newPlayer = data.val();
```

```
5           console.log("name: " + newPlayer.name);
6           console.log("age: " + newPlayer.age);
7           console.log("number: " + newPlayer.number);
8           console.log("Previous Player: " + prevChildKey);
9 });
```

We will get the following result.

child_changed

This event type is triggered when the data has changed.

Let us consider the following example.

```
1 var playersRef = firebase.database().ref("players/");
2 playersRef.on("child_changed",function(data)
3 {
4  var player = data.val();
5  console.log("The updated player name"+player.name);
6 });
```

We can change Bob to Maria in Firebase to get the update.

child_removed

If we want to get access of deleted data, we can use child_removed event type.

Consider the follwing example:

```
1 var playersRef = firebase.database().ref("players/");
2 playersRef.on("child_removed", function(data) {
3     var deletedPlayer = data.val();
```

```
4     console.log(deletedPlayer.name +"has been deleted");
5 });
```

Now, we can delete Maria from Firebase to get notifications.

Detach Callback for Event Type

Let us say we want to detach a callback for a function with value event type. Consider the following example:

```
1 var playersRef = firebase.database().ref("players/");
2 ref.on("value", function(data) {
3         console.log(data.val());
4 }, function (error) {
5         console.log("Error: " + error.code);
6 });
```

We need to use off() method. This will remove all callbacks with value event type.

```
1 playersRef.off("value");
```

Detach All Callbacks

When we want to detach all callbacks, we can use the following command:

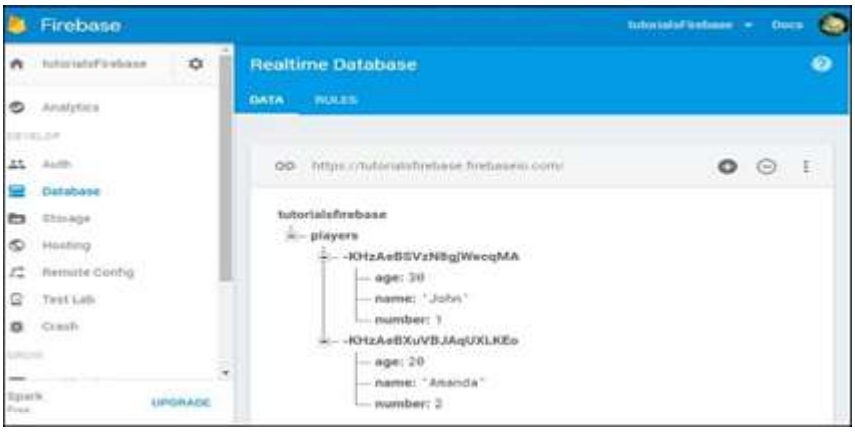
```
1 playersRef.off();
```

6.1.1 Firebase – Queries

Queries related to Firebase are discussed in this subsection.

Order by Child

To order data by name, we can use the following code. Consider the following example.




```
var playersRef = firebase.database().ref("players/");
playersRef.orderByChild("name").
on("child_added",function(data)
{
        console.log(data.val().name);
});
```

Order by Key

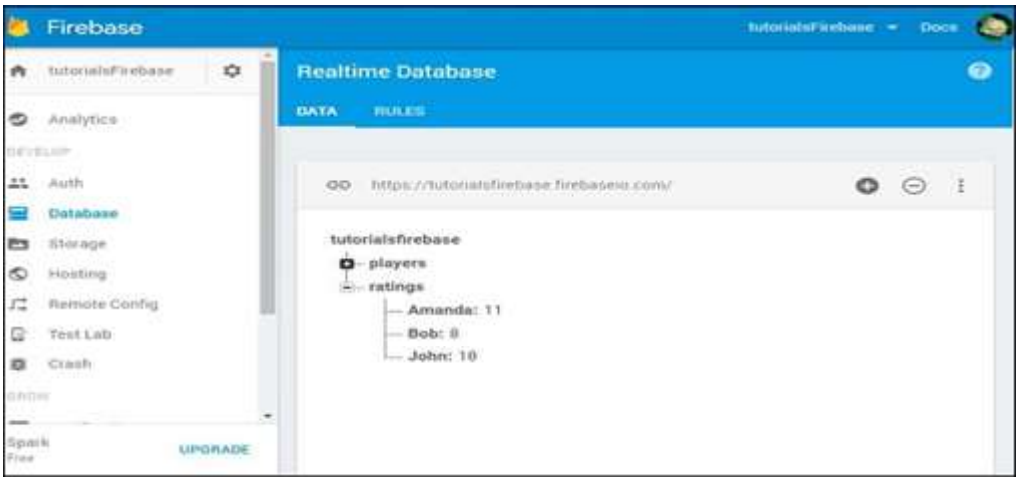
We can order data by key in a similar fashion. Let us consider the following example.

```
var playersRef = firebase.database().ref("players/");
playersRef.orderByKey().on("child_added", function(data)
{
        console.log(data.key);
});
```

Order by Value

We can also order data by value. Let us add the ratings collection in Firebase. The following figure shows the collection in Firebase. Consider the following example.

```
var ratingRef = firebase.database().ref("ratings/");
```




```
ratingRef.orderByValue().on("value", function(data) {
  data.forEach(function(data) {
    console.log("The"+data.key+"rating is"+data.val());
        });
});
```

Firebase offers several ways to filter data as listed below:

- limitToFirst method returns the specified number of items beginning from the first one.
- limitToLast method returns a specified number of items beginning from the last one.

Our example is showing how this works. Since we only have two players in database, we will limit queries to one player.

Let us consider the following example.

```
var P1=firebase.database().ref("players/").limitToFirst(1);
var P2=firebase.database().ref('players/').limitToLast(1);
P1.on("value", function(data) {
        console.log(data.val());
}, function (error) {
        console.log("Error: " + error.code);
```

```
});
P2.on("value", function(data) {
        console.log(data.val());
}, function (error) {
        console.log("Error: " + error.code);
});
```

We can also use other Firebase filtering methods. The startAt(), endAt() and the equalTo() can be combined with ordering methods. In our example, we will combine it with the orderByChild() method.

Let us consider the following example.

```
var playersRef = firebase.database().ref("players/");
playersRef.orderByChild("name").startAt("Amanda")
.on("child_added", function(data) {
console.log("Start at filter: " + data.val().name);
});
playersRef.orderByChild("name").endAt("Amanda").
on("child_added", function(data) {
console.log("End at filter: " + data.val().name);
});
playersRef.orderByChild("name").equalTo("John").
on("child_added", function(data) {
console.log("Equal to filter: " + data.val().name);
});
playersRef.orderByChild("age").startAt(20).
on("child_added", function(data) {
console.log("Age filter: " + data.val().name);
});
```

The first query will order elements by name and filter from the player with the name Amanda. The console will log both players. The second query will log "Amanda" since we are ending query with this name. The third one will log "John" since we are searching for a player with that name. The fourth example is showing how we can combine filters with "age" value. Instead of string, we are passing the number inside the startAt() method since age is represented by a number value.

Lab exercises

1. Develop a web based student management system using Firebase.

Additional Exercises

1. Develop a web based clinical appointment management system using Firebase.

7. File System and Regular Expressions

Objectives

- Describe the basics of file system operations using Python
- Implement data handling using files
- Understand various regular expression features of Python

7.1 Files

File uses a location within a disk to store data / information in computer memory and are accessed using a "path" and "unique name". In Python, file manipulations are done using file objects.

The sequence of actions on file are

- Open a file
- Perform operation (Read / Write / Append)
- Close the file

7.1.1 Opening and Closing Files

Python provides the built-in function open( ) to open a file, which returns a file object. The close() method closes an open file. In some cases, due to buffering, changes are not updated if a file is not closed. **Syntax**

```
file-object = open(filename, [access-mode])
```

- file-name (required) – name of the file to be opened

Mode	Description
r	Opens a file for reading mode only.
w	Opens a file for writing only.
.	Overwrites the file if the file exists. otherwise creates a new file for writing.
r+	Opens a file for reading and writing mode.
.	The file pointer is positioned at the beginning of the file.
w+	Opens a file for writing and reading mode.
.	The file is created if it does not exist, otherwise it is truncated.
a	Opens a file for appending.
.	Creates a new file for writing if the file does not exist.
a+	Opens a file for both appending and reading.
.	The file pointer is placed at the EOF if the file exists and opens in append mode.
.	Otherwise it creates a new file for reading and writing.

- access-mode (optional) – mode in which the file is to be opened. It can be read, write etc. By default, it opens the file in reading mode (r) if not specified explicitly.

Here is the list of different modes in which a file can be opened.

Consider the following example for opening a file and displaying the details:

```
1 f = open("test.txt","w+")
2 print("File name:",f.name)
3 print("File mode:",f.mode)
4 print("File closed:",f.closed)
5 f.close()
6 print("File closed:",f.closed)
```

Output

```
1 File name:test.txt
2 File mode:w+
3 File closed:False
4 File closed:True
```

7.1.2 Writing to Files

In order to write into a file in Python, open the file in write , append or exclusive creation mode. Writing a string or sequence of bytes (for binary files) is done using the write() method. This method returns the number of characters written to the file.

```
with open("test.txt",'w',encoding = 'utf-8') as f:
f.write("My first file\n")
f.write("This file\n\n")
f.write("contains 3 lines\n")
```

This program will create a new file named test.txt in the current directory if it does not exist. If it does exist, it is overwritten. The newline characters are included to distinguish between the different lines.

7.1.3 Reading from Files

To read a file in Python, open the file in reading mode. There are various methods available for this purpose. To read the specific size of data use the read(size) method. If the 'size' parameter is not specified, it reads and returns up to the end of the file. Following code is used for reading the file:

```
>>> f = open("test.txt",'r',encoding = 'utf-8')
>>> f.read(4)    # read the first 4 data
'This'
>>> f.read(4)    # read the next 4 data
' is '
>>> f.read()     # read in the rest till end of file
'My first file\nThis file\ncontains 3 lines\n'
>>> f.read()  # further reading returns empty string
''
```

The read() method returns a newline as \n. Once the end of the file is reached,an empty string is returned on further reading. Current file cursor (position) can be changed using the seek() method. Similarly, the tell() method returns our current position of the cursor (in number of bytes).

```
>>> f.tell()    # get the current file position
56
>>> f.seek(0)   # bring file cursor to initial position
0
>>> print(f.read())  # read the entire file
My first file
This file
contains 3 lines
```

Following code is used read a file line-by-line using a for loop. This is both efficient and fast.

```
>>> for line in f:
...     print(line, end = '')
...
My first file
This file
contains 3 lines
```

In the above program, the lines in the file itself include a newline character \n. So, the end parameter of the print() function is used to avoid two newlines when printing. The readline() method also can be used to read individual lines of a file. This method reads a file till the newline, including the newline character.

```
>>> f.readline()
'My first file\n'

```

```
>>> f.readline()
'This file\n'

>>> f.readline()
'contains 3 lines\n'

>>> f.readline()
''
```

Lastly, the readlines() method returns a list of remaining lines of the entire file. All these reading methods return empty values when the end of file (EOF) is reached.

```
>>> f.readlines()
['My first file\n','This file\n','contains 3 lines\n']
```

7.1.4 Python File Methods

There are various methods available with the file object. Some of them have been used in the above examples.

Table 7.1 describes the list of methods for file handling:

7.2 Directories

Files are collectively placed within different directories and are organized using the built-in functions in os module. Following are the commonly used functions:

- mkdir("namedir"): To make a directory inside the current directory
- getcwd(): Returns the complete path of the current working directory

Table 7.1: File Handling Methods

Method	Description
close()	Closes an opened file.
fileno()	Returns file descriptor(integer) of the file.
flush()	Flushes the write buffer of the file stream.
isatty()	Returns True if the file stream is interactive.
read(n)	Reads at most n characters from the file.
.	Reads till end of file if it is negative or None.
readable()	Returns True if the file stream can be read from.
readline(n=-1)	Reads and returns one line from the file.
.	Reads in at most n bytes if specified.
readlines(n=-1)	Reads and returns a list of lines from the file.
.	Reads in at most n bytes/characters if specified.
seekable()	Returns True if the file stream supports random access.
tell()	Returns the current file location.
writable()	Returns True if the file stream can be written to.
write(s)	Writes the string s to the file and
.	returns the number of characters written.
writelines(lines)	Writes a list of lines to the file.

- chdir("name dir"): To change the current working directory
- rmdir("name"): To delete any directory

Consider the following examples:

```
import os
# help(os)
c = os.getcwd()
print("Current Working Directory : ", c)
```

```
os.mkdir("newdir")
l = os.listdir()
if "newdir" in l:
        print("Created new directory")
else:
        print("Directory not present")
print()
```

```
print("After deleting Directory")
os.rmdir("newdir")
l = os.listdir()
if "newdir" in l:
        print("Created new directory")
else:
        print("Directory not present")
```

Output

```
Created new directory
After deleting Directory
Directory not present
```

```
#list all the files present in the given directory
import os
p = "//home//lifna//Desktop//OST_Lab"
l = os.listdir(p)
for i in l:
        print(i)
```

Output

```
        Perl pgms
        exception.py
        Python ppts
        Perl ppts
        Python pgms
        Notebooks
        Rhistory
        raise.py
```

7.3 Regular Expressions

A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. Regular expressions are widely used in UNIX world.

The module *re* provides full support for Perl-like regular expressions in Python. The *re* module raises the exception re.error if an error occurs while compiling or using a regular expression. There are various characters, which would have special meaning when they are used in regular expression.

7.3.1 The match Function

This function attempts to match RE pattern to string with optional flags. Syntax for this function:

```
1 re.match(pattern, string, flags=0)
```

The re.match function returns a match object on success, None on failure. The description of the parameters:

- pattern: This is the regular expression to be matched.
- string: The string, which would be searched to match the pattern at the beginning of string.
- flags: Different flags can be specified using bitwise-or operator | . These are modifiers, which are listed in the table below. Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. Multiple modifiers also can be provided using exclusive-or operator $\wedge$ operator, as shown previously and may be represented by one of these. Table 7.2 summarizes the available flags.

Table 7.2: Regular Expression Flags

Modifier	Description
re.I	Performs case-insensitive matching.
re.L	Interprets words according to the current locale. This interpretation affects the alphabetic group \w and \W, as well as word boundary behavior \b and \B.
re.M	Makes $match the end of a line (not just the end of the string) and makes ^match the start of any line (not just the start of the string).
re.S	Makes a period dot match any character, including a newline.
re.U	Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.

```
#!/usr/bin/python
import re
line = "Cats are smarter than dogs"
matchObj=re.match( r'(.* )are(.* ?) .* ',line, re.M|re.I)
if matchObj:
        print "matchObj.group() : ", matchObj.group()
        print "matchObj.group(1) : ", matchObj.group(1)
        print "matchObj.group(2) : ", matchObj.group(2)
else:
        print "No match!!"
```

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

```
matchObj.group() : Cats are smarter than dogs
matchObj.group(1) : Cats
matchObj.group(2) : smarter
```

7.3.2 The search Function

This function searches for first occurrence of RE pattern within string with optional flags. The syntax for this function:

```
re.search(pattern, string, flags=0)
```

The parameters are same as that of match function.

7.3.3 Matching Versus Searching

Python offers two different primitive operations based on regular expressions: match checks for a match only at the beginning of the string, while search checks for a match anywhere in the string.

```
#!/usr/bin/python
import re
line = "Cats are smarter than dogs";
matchObj=re.match(r'dogs',line,re.M|re.I)
if matchObj:
    print("match->matchObj.group():",matchObj.group())
else:
        print "No match!!"
        sObj=re.search(r'dogs',line,re.M|re.I)
if searchObj:
        print "search->sObj.group():",sObj.group()
else:
        print "Nothing found!!"
```

Output

```
No match!!
search --> searchObj.group() :  dogs
```

7.3.4 Search and Replace

One of the most important re methods that use regular expressions is sub.

Syntax

```
re.sub(pattern, repl, string, max=0)
```

This method replaces all occurrences of the RE pattern in string with 'repl', substituting all occurrences unless max provided. This method returns modified string.

```
#!/usr/bin/python
import re
phone = "2004-959-559 # This is Phone Number"

# Delete Python-style comments
num = re.sub(r'#.*$', "", phone)
print "Phone Num : ", num

# Remove anything other than digits
num = re.sub(r'\D', "", phone)
print "Phone Num : ", num
```

Output

```
Phone Num :  2004-959-559
Phone Num :  2004959559
```

7.3.5 Regular Expression Patterns

Except for control characters : (+, ?, ·, $, ∗, ∧, (,), [ ], {, }, |, \)
all characters match themselves. You can escape a control character by preceding it with a backslash.

Regular expression syntax of Python programming is listed in the Table 7.3

Table 7.3: File Opening Modes

Pattern	Description
∧	Matches beginning of line.
$	Matches end of line.
·	Matches any single character except newline.
	Using m option allows it to match newline as well.
· · ·	Matches any single character in brackets.
re∗	Matches 0 or more occurrences of preceding expression.
re+	Matches 1 or more occurrence of preceding expression.
re?	Matches 0 or 1 occurrence of preceding expression.
re n	Matches exactly n number of occurrences of preceding expression.
re n,	Matches n or more occurrences of preceding expression.
re n, m	Matches at least n and at
	most m occurrences of preceding expression.
a|b	Matches either a or b.
re	Groups regular expressions and remembers matched text.
?imx	Temporarily toggles on i, m, or x options within a regular expression.
	If in parentheses, only that area is affected.
?-imx	Temporarily toggles off i, m, or x options within a regular expression.
	If in parentheses, only that area is affected.
?:re	Groups regular expressions without remembering matched text.
?imx: re	Temporarily toggles on i, m, or x options within parentheses.
? -imx:re	Temporarily toggles off i, m, or x options within parentheses.
?#· · ·	Comment.
? = re	Specifies position using a pattern. Doesn't have a range.
? !re	Specifies position using pattern negation. Doesn't have a range.
? >re	Matches independent pattern without backtracking.
\w	Matches word characters.
\W	Matches nonword characters.
\s	Matches whitespace. Equivalent to [\t \n \r \f]
\S	Matches nonwhitespace.
\d	Matches digits. Equivalent to [0-9].
\D	Matches nondigits.
\A	Matches beginning of string.
\Z	Matches end of string. If a newline exists, it matches just before newline.
\z	Matches end of string.
\G	Matches point where last match finished.
\b	Matches word boundaries when outside brackets.
	Matches backspace 0x08 when inside brackets.
\B	Matches nonword boundaries.
\n	\t, etc.
	Matches newlines, carriage returns, tabs, etc.
\1...\9	Matches nth grouped subexpression.

Lab Exercises

1. Write a Python program to find the total number of characters, words and lines in a file

2. Write a Python program to find number of occurrences of each word in a file. [Note: use dictionary]

3. Write a Python program reverse.py to print lines of a file in reverse order.

   Smaple Input : input.txt

   Command to Execute : python reverse.py input.txt Smaple

```
She sells seashells on the seashore;
The shells that she sells are seashells I'm sure.
So if she sells seashells on the seashore,
I'm sure that the shells are seashore shells
```

Output

```
I'm sure that the shells are seashore shells.
So if she sells seashells on the seashore,
The shells that she sells are seashells I'm sure.
She sells seashells on the seashore;
```

4. Create a input file, named input.txt, store a list of email IDs. Copy all the valid email IDs to file valid.txt and invalid email IDs to invalid.txt. [Use regular expressions to validate email IDs]

5. Check whether a string starts and ends with the same character or not using regular expressions.

Additional Exercises

1. Write a Python program to print each line of a file in reverse order.

2. Write a Python function *extsort* to sort a list of files based on extension.

Sample Input

```
extsort(['a.c','a.py','b.py','foo.txt','x.c'])
```

Sample Output

```
['a.c', 'x.c', 'a.py', 'b.py','foo.txt']
```

3. Write a Python program wrap.py which takes filename and width as command line arguments and wraps the lines having the length longer than width mentioned.

Execution statement: python wrap.py input.txt 30

Sample Input Sample Output

```
She sells seashells on the seashore;
The shells that she sells are seashells I'm sure.
So if she sells seashells on the seashore,
I'm sure that the shells are seashore shells
```

```
I'm sure that the shells are s
eashore shells.
So if she sells seashells on t
he seashore,
The shells that she sells are
seashells I'm sure.
She sells seashells on the sea
shore;
```

8. Packages, GUI and Networking

Objectives

- To understand the usage of packages
- To design graphical user interface using Python
- To develop a dynamic web application

Package is a directory containing code and possibly further subdirectories. A package contains a group of usually related code files (modules). The name of the package is derived from the name of the main package directory. Packages are a natural extension of the module concept and are designed to handle very large projects. Just as modules group related functions, classes, and variables, packages group related modules. To see how this might work in practice, let's sketch a design layout for a type of project that by nature is very large—a generalized mathematics package, along the lines of Mathematica, Maple, or MATLAB. Maple, for example, consists of thousands of files, and some sort of hierarchical structure is vital to keeping such a project ordered. We'll call our project as a whole mathproj. We can organize such a project in many ways, but a reasonable design splits the project into two parts: ui, consisting of the user interface elements, and comp, the computational elements. Within comp, it may make sense to further segment the computational aspect into symbolic (real and complex symbolic computation, such as high school algebra) and numeric (real and complex numerical computation, such as numerical integration). It may then make sense to have a constants.py file in both the symbolic and numeric parts of the project. The constants.py file in the numeric part of the project defines pi as

```
1 pi = 3.141592
```

whereas the constants.py file in the symbolic part of the project defines pi as

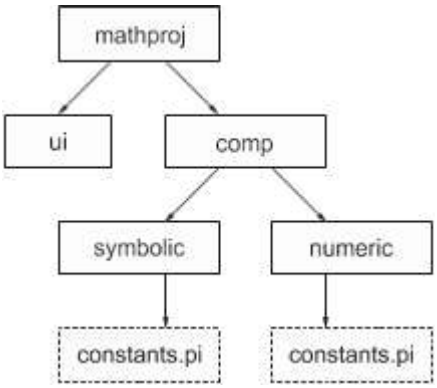


Figure 8.1: Organizing a math package

```
1 class PiClass:
2         def str (self): return "PI"
3         pi = PiClass()
```

This means that a name like pi can be used in (and imported from) two different files named constants.py, as shown in Figure **??** The symbolic constants.py file defines pi as an abstract Python object, the sole instance of the PiClass class. As the system is developed, various operations can be implemented in this class, which return symbolic rather than numeric results. There is a natural mapping from this design structure to a directory structure. The top-level directory of the project, called math- proj, contains subdirectories ui and comp; comp in turn contains subdirectories symbolic and numeric; and each of symbolic and numeric contains its own constants.py file. Given this directory structure, and assuming that the root mathproj directory is installed somewhere in the Python search path, Python code both inside and outside the mathproj package can access the two variants of pi as mathproj.symbolic.constants.pi and mathproj.numeric.constants.pi. In other words, the Python name for an item in the package is a reflection of the directory pathname to the file containing that item. Inner workings of the package mechanism is shown in Figure 8.2. Filenames and paths are shown in

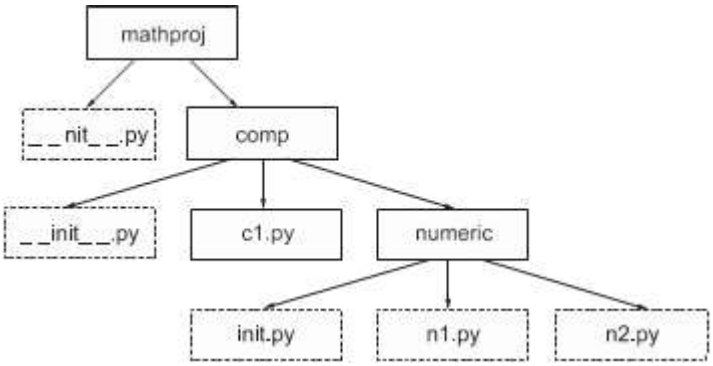


Figure 8.2: Working of a Package

plain text, to avoid confusion as to whether we're talking about a file/directory or the module/package defined by that file/directory. The files used in this example package are shown in listings 8.1 through 8.6. **Listing 8.1 File mathproj/_init _.py**

```
print("Hello from mathproj init")
all = ['comp'] version = 1.03
```

Listing 8.2 File mathproj/_init __.py

```
    all = ['c1']
print("Hello from mathproj.comp init")
```

Listing 8.3 File mathproj/comp/c1.py

```
x=1.00
```

Listing 8.4 File mathproj/comp/numeric/_init __.py

```
print("Hello from numeric init")
```

Listing 8.5 File mathproj/comp/numeric/n1.py

```
from mathproj import version from mathproj.comp import c1
from mathproj.comp.numeric.n2 import h
def g():
        print("version is", version)
```

```
5         print(h())
```

Listing 8.6 File mathproj/comp/numeric/n2.py

```
1 def h():
2         return "Called function h in module n2"
```

These files are created in a mathproj directory that's on the Python search path. (It's sufficient to ensure that the current working directory for Python is the directory containing mathproj when executing these examples.)

8.0.1 __init.py files in packages

An __init.py file serves following purposes:

- It's automatically executed by Python the first time a package or sub-package is loaded. This permits whatever package initialization you may desire. Python requires that a directory contain an init .py file before it can be recognized as a package. This prevents directories containing miscellaneous Python code from being accidentally imported as if they defined a package.

- Automatic initialization. For many packages, you won't need to put anything in the package's init .py file—just make sure an empty init .py file is present.

8.1 Graphical User Interface

The Python core language has no built-in support for GUIs. It's a pure programming language, like C, Perl, or Pascal. As such, any support for GUIs must come from libraries external to Python, and many such libraries have been developed.

Of all the GUI packages currently available to Python programmers, Tkinter is the one commonly used. Tkinter is an object-oriented layer on top of the Tcl/Tk graphics libraries. The code that drives it is stable, efficient, and well supported.

8.1.1 Installing Tkinter

If you're using IDLE, you already have Tkinter installed. On Windows and Mac OS X, Tkinter comes as part of the Python distribution. Some Linux distributions don't come with Tkinter by default, but they usually have Tkinter packages available. On Ubuntu Linux, for example, installing the IDLE package also installs Tkinter. If you don't have Tkinter installed, or are having problems making it run, go to the Python home page and search for "tkinter" to find the link to the Tkinter page, which contains documentation, tutorials, troubleshooting information, and instructions on how download the latest version for your platform. Don't be confused by the fact that you'll be installing something probably called Tcl/Tk, followed by some version number. Tcl is a scripting language, and Tk is a GUI extension. Python uses Tcl to access Tk, but in a transparent fashion; you'll never need to worry about the Tcl aspect of the package.

8.1.2 Starting Tk and using Tkinter

To test the working of Tkniter type the following command

```
1        from tkinter import *
```

If you receive another Python command prompt ¿¿¿ and no errors, then everything is working okay, and Tk has been started automatically by the importation of Tkinter.

The following code creates a dialog box like the one in Figure 8.3

Figure 8.3: A minimal Tkinter application

```
1 import sys win = Tk()
2 b1=Button(win, text="Goodbye",command=sys.exit) b1.pack()
3 mainloop()
```

Note that if you're in IDLE, you may need to omit the last line.

Consider the application shown in Figure 8.3, when you click the Goodbye button, the sys.exit command is executed, and Python quits. This window is only a little larger than the button it contains and may appear behind another window. But as long as you didn't get any error messages, it should be there.

8.1.3 Principles of Tkinter

The Tkinter GUI package is based on a small number of basic principles and ideas. Tkinter is a relatively easy way for you to learn GUI and event-driven programming.

8.1.3.1 Widgets

The first basic idea behind Tkinter is the concept of a widget, which is short for window gadget. A widget is a data structure that also has a visible, onscreen representation. When the program changes the internal data structure of the widget, that change is automatically displayed on the screen. Various user actions on the visible representation of the widget (mouse clicks and so forth) can, in turn, cause internal changes or actions within the widget's data structure.

Tkinter is a collection of widget definitions, together with commands for operating on them, and a few extra commands that don't apply to any spe-

cific widget but that are still relevant to GUI programming. In Python, each different type of widget is represented by a different Python class. A Button widget is of the Button class, a Label widget is of the Label class, and so forth.

A Python program that creates and uses a Button widget and a Label widget looks something like this:

```
from tkinter import *
...
button = Button(...optional arguments...)
label = Label(...optional arguments...)
...
```

This style of mapping Python classes to widgets is common in Python GUI environments, although the exact names of the widgets and their parameters will naturally vary.

8.1.3.2 Named attributes

The second basic idea behind Tkinter is the availability and use of named attributes to fine-tune widget behavior. To understand this consider simple task—creating a button. The simplest way to do this is to specify a class, say AButton, with a one-argument object constructor, whose single argument is a string that will become the name displayed on the button. Following statement creates a button:

```
button1 = AButton(name)
```

But this provides no way to associate a command with the button, the name of a function that should be executed when the user clicks the button. To do this, change AButton to have a two-argument constructor, with the command as the second argument:

```
button1 = AButton(name, command)
```

To set the background color following syntax is used:

```
1 button1=AButton(name,cmd, foreground_clr ,background_clr)
```

A button ca be created by passing 20 different parameters according to the requirements.

The Python command to create a button that displays the string "Hello!" is

```
1 button1 = Button(text="Hello!")
```

Following example shows overriding of the default attributes of a widget, i.e changing the foreground color of the button

```
1 button1 = Button(text="Hello!", foreground="red")")
```

8.1.4 Geometry management and widget placement

Geometry management means how widgets are placed on the screen. It isn't obvious in the previous example. Consider the following Python code:

```
1 button-align= Button(text="Alright!")
```

This isn't enough to display the button onscreen. Tk doesn't know where you want the button to show up, and until it's told the desired position, it will keep the button hidden. Deciding where to display the button onscreen is a function of the window hierarchy and associated Tk geometry managers.

To understand the idea of the Tk window hierarchy, you need to know about two special Tk widget classes, called Toplevel and Frame. Both Toplevel widgets and Frame widgets may contain other Tk widgets (including other frames) and are the basic building blocks for constructing complex GUIs. A Frame is a container for other widgets and can be either the main window of an application or contained in another frame. Nesting frames within frames can be useful in laying out and grouping widgets.

Figure 8.4: A Simple Application

Tk uses a Toplevel widget to represent a complete window in a GUI, complete with title bar, close and zooming buttons, and so forth. A Toplevel widget is useful for cus- tom dialog boxes and other situations where you want a window that's independent from the main Frame. The subwidgets contained in any particular frame are arranged for display accord- ing to one of Tkinter's three built-in geometry managers. These managers permit you to specify the arrangement of the subwidgets in various ways, ranging from giving exact coordinates for each widget within a window to giving only relative placement, leaving the precise sizing of each widget to the geometry manager.

Grid manager works by placing all widgets in an implicit grid, similar to a spreadsheet layout. If you start using Tkinter, you'll want to learn the pack and place geometry managers as well, which you can use to specify that widgets should be placed relative to one another (pack), or in absolute locations in a window (place).

A simple Tkinter application

Let's start with an example that introduces all of the basic Tkinter concepts: window hierarchies, geometry management, Tkinter attributes, and a couple of the most basic widgets. The example is a simple one. When run, it produces a window that resembles Figure 8.4. It may look different on your machine, because Tk provides a native look and feel for whatever operating system it's running on. This example was produced under the Windows XP operating system. Clicking the Increment button adds 1 to the number shown in the Count field, and clicking the Quit button quits the application.

Creating widgets Widgets can be created in Python by instantiating an object of that widget's class, using the name of the type of widget being created. Button and Label widgets were created in the previous example, but it is also possible to create Menu, Scrollbar, Listbox, Text, and many other types.

The widget-creation commands all follow the same general form. They all have one mandatory argument, the parent window (or parent widget), followed by zero or more optional named widget attributes, which determine the precise appearance and behavior of the new widget. Each creation command returns the new widget as a result. You'll usually want to store this new widget somewhere so that you can modify it later if necessary. A line in a Python program that creates a widget usually looks something like this:

```
1 new_widget=WidgetCreationCommand(parent, attribute1=value1,
2 attribute2=value2, . . .)
```

The parent window of a widget called w is the window (or widget) that contains w. It's important to define a parent for following reasons.

- Widgets are always displayed inside and relative to their parent window
- Tk provides a powerful event mechanism, and a widget may pass events to its parent if it can't handle them itself
- Tk can have widgets that act as windows, in that they themselves are the parent window for (and contain) other widgets, and the widget-creation commands need to be able to set up this sort of relationship

All the other optional arguments in a widget-creation command define widget attributes and control different aspects of the widget. Some widget attributes are common to several different widget types (for example, the text attribute applies to all types of simple widgets that can display a label or a line of text of some sort), whereas other attributes are unique to certain widgets.

Figure 8.5: A widget window

One characteristic that makes Tk special, compared to other GUI packages, is that it gives you a great deal of control over your widgets. There are many attributes for each widget. To give you an idea of this, here's a program that uses some of the attributes that control the appearance of widgets. Figure 8.5 is the resulting window.

```
1 from tkinter import * main_window = Tk()
2 label=Label(main_window, text="Hello", background='white',
3 foreground='red',font='Times 20',relief='groove',
4  borderwidth=3)
5 label.grid(row=0, column=0)
6 mainloop()
```

8.1.5 Widget placement

Creating a widget does not automatically draw it on the screen. Before this can be done, Tk needs to know where the widget should be drawn. This is done using the grid command. Tk is more sophisticated in the way it handles widget placement than most GUI packages. Under Windows, the standard way of setting the locations of widgets is to specify an absolute position in their parent window. This can also be done in Tkinter (using the place rather than the grid command) but usually is not, because this technique is not very flexible. For instance, if you set up a window for use on a monitor that has 640X480 resolution, and a user uses it on a monitor that has 1600X1200 resolution, the window uses only a small amount of the available screen space and cannot be resized (unless you write the code to resize the window). This

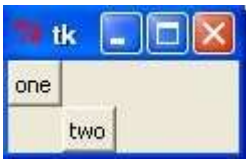


Figure 8.6: A two- button window

is a common problem with many programs.

Instead, Tkinter usually makes use of the notion of relative placement, where widgets are placed in such a manner that their positions relative to one another are maintained no matter what size the enclosing window happens to be. The grid command places widgets in a window by considering a window as an infinite grid of cells. You place a widget into this grid by specifying row and column arguments to the grid command, which tell it in which cell to place the widget. The rows and columns will expand as needed to display the widgets they contain, and any rows or columns that don't display any widgets aren't displayed. As a simple example which illustates the widget placement :

```
1 from tkinter import * win = Tk()
2 button1 = Button(win, text="one")
3 button2 = Button(win, text="two")
4 button1.grid(row=0, column=0)
5 button2.grid(row=1, column=1)
6 mainloop()
```

This program produces a window that looks like Figure 8.6 The cells of the grid are automatically sized large enough to display what they contain, although you can override this and place constraints on the maximum sizes of the cells. This makes it easy to set up a text window with scroll-bars and, with the proper placement as shown in the Figure refoutput1.

The program to do this is as shown below:

```
1 from tkinter import * main = Tk()
2 main.columnconfigure(0, weight=1)
```

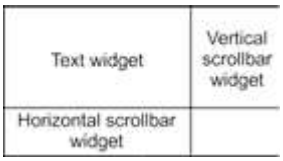


Figure 8.7: Grid usage

```
main.rowconfigure(0, weight=1)
text = Text(main)
text.grid(row=0, column=0, sticky='nesw')
vertical_scroller = Scrollbar(main, orient='vertical')
vertical_scroller.grid(row=0, column=1, sticky='ns')
horizontal_scroller=Scrollbar(main, orient='horizontal')
horizontal_scroller.grid(row=1, column=0, sticky='ew')
mainloop()
```

8.1.6 Using classes to manage Tkinter applications

One problem with creating Tkinter applications as we have so far is that they quickly become hard to read and maintain as you add more widgets and code. Using the OOP principles introduced in the previous chapter and making an application class that inherits from Frame can make your code much more organized and easy to read and maintain. Following program, shows the counter application, refactored as a class.

```
from tkinter import *
class Application(Frame):
  def init (self, master=None):
          Frame. init (self, master)
      self.grid()
          self.create_widgets()
          self.count_value = 0
```

```
8
9  def create_widgets(self):
10            self.count_label=Label(self,text="Count:0")
11            self.count_label.grid(row=0, column=1)
12            self.incr_button=Button(self,text="Increment",
13           command=self.increment_count)
14            self.incr_button.grid(row=0, column=0)
   self.quit_button = Button(self, text="Quit",
15 command=self.master.destroy)
16                    self.quit_button.grid(row=1, column=0)
17          def increment_count(self):
18                    self.count_value += 1
19                    self.count_label.configure(text='Count:'+
20                    str(self.count_value))
21
22 app = Application()
23 app.mainloop()
```

This code is much better organized and easier to read. Basic initialization, widget setup, and counter incrementing are now much easier to pick out; and because increment-counter is now an instance variable, we no longer need to make it global. Even though you almost certainly won't want more than one instance of Application at a time, creating the class is worth because of the improved organization i.e.readability and maintainability of the code.

8.1.7 Event Handling in Python

A Tkinter application runs most of its time inside an event loop, which is entered via the mainloop method. It waiting for events to happen. Events can be key presses or mouse operations by the user.

Tkinter provides a mechanism to bind Python functions and methods to handle an event. Syntax of bind function:

```
widget.bind(event, handler)
```

Consider the follwing simple example for event handling:

```
#!/usr/bin/python3
from tkinter import *
def hello(event):
        print("Single Click, Button-1")
def quit(event):
        print("Double Click, so let's stop")
import sys; sys.exit()

widget = Button(None, text='Mouse Clicks')
widget.pack()
widget.bind('<Button-1>', hello)
widget.bind('<Double-1>', quit)
widget.mainloop()
```

Some of the possible events are mouse click, mouse double click, key pressed, focus and so on..

8.2 Networking Using Python

Python provides two levels of access to network services. At a low level, you can access the basic socket support in the underlying operating system, which allows you to implement clients and servers for both connection-oriented and connectionless protocols.

8.2.1 Sockets

Sockets are the endpoints of a bidirectional communications channel. Sockets may communicate within a process, between processes on the same machine, or between processes on different continents. Sockets may be implemented over a number of different channel types: Unix domain sockets, TCP, UDP, and so on.

Socket Creation

To create a socket, you must use the socket.socket() function available in socket module, which has the general syntax :

```
1 s=socket.socket (socket_family,socket_type, protocol=0)
```

Parameters description is as follows :

- socket _family : This is either AF _UNIX or AF _INET.
- socket_type : This is either SOCK_STREAM or SOCK_DGRAM.
- protocol : This is usually left out, defaulting to 0.

Server Socket Methods

- s.bind() - Binds address (hostname, port number pair) to socket.
- s.listen() - Sets up and start TCP listener.
- s.accept()- Accept TCP client connection, blocks until connection arrives. Client Socket Method
- s.connect() - Actively initiates TCP server connection

General Socket Methods

- s.recv() - Receives TCP message
- s.send() - Tansmits TCP message

- s.recvfrom() - Receives UDP message
- s.sendto() - Transmits UDP message
- s.close() - Closes socket
- socket.gethostname() - Returns the hostname.

A Simple Server

Socket function is used to create a socket object. Then call bind(hostname, port) function to specify a port for your service on the given host. After this call the accept method of the returned object. This method waits until a client connects to the port specified, and then returns a connection object that represents the connection to that client.

```
1 #!/usr/bin/python           # This is server.py file
2 import socket               # Import socket module
3 s = socket.socket()         # Create a socket object
4 host = socket.gethostname() # Get local machine name
5 port = 12345                # Reserve a port
6 s.bind((host, port))        # Bind to the port
7 s.listen(5)       # Now wait for client connection.
8 while True:
9 c, addr = s.accept()  # Establish connection with client.
10 print 'Got connection from', addr
11 c.send('Connection successful - Hello from Client')
12 c.close()                 # Close the connection
```

A Simple Client

```
1 #!/usr/bin/python           # This is client.py file
2 import socket               # Import socket module
3 s = socket.socket()         # Create a socket object
4 host = socket.gethostname() # Get local machine name
```

```
port = 12345          # Reserve a port for your service.
s.connect((host, port))
print s.recv(1024)
s.close()             # Close the socket when done
```

To execute run the server in background then execute the client.

Output

```
$ python server.py &
$ python client.py
Got connection from ('127.0.0.1', 48437)
'Connection successful - Hello from Client
```

Lab exercises

1. Write a Python program to print alphabetically sorted list of all the function names havig the substring *find* in the *re* module.

2. Write a Python program for calculator application with basic operations (+,-,*,/,%) (With GUI).

3. Write two separate Python programs, one for server and another for client using TCP socket APIs, in which the client accepts a string from the user and sends it to the server. The server checks if the string is palindrome or not and sends the result along with the length of the string and the number of occurrences of each vowel in the string to the client. The client displays the same on the client screen. The process repeats until user enters the string "Stop".

Additional Exercises

1. Develop a quiz application with GUI.

2. Write two separate programs one for server and another for client using socket APIs for TCP. The client sends a set of integers along with a choice 1) to search for a number 2) sort the given set in ascending/descending order 3) split the given set to odd & even to the server. The server performs the relevant operation according to the choice and relevant input. Client displays the results obtained from the server. The process continues until the user selects the choice "exit".

9. Memory Management and Exception Handling

Objectives

- To understand the role of destructors in Python memory management
- To understand the basic concept of exception handling in Python
- To write Python programs that handle exceptions

A destructor is used to remove an object and perform the final clean up in Python. Python provides automatic memory management through a reference-counting mechanism. Hence creating and calling a destructor is not necessary to ensure that the memory used by the instance is freed or not. That is, it keeps track of the number of references to each instance; when this reaches zero, the memory used by instance is reclaimed, and any Python objects referenced by this instance have their reference counts decremented by one.

A destructor has this format: **def __del__(self):**
It is always part of a class and is called implicitly. Python assumes an empty destructor, if it is not defined. The following simple class illustrates this:

```
1 class SpecialFile:
2         def __init__(self, file_name):
3                 self.__file = open(file_name, 'w')
4                 self.__file.write('Start Special File\n')
5         def __init__(self, file_name):
6                 self.__file = open(file_name, 'w')
7                 self.__file.write('Start Special File\n')
8         def write(self, str):
9                 self.__file.write(str)
```

```
10          def writelines(self, str_list):
11                  self.__file.writelines(str_list)
```

```
1           def __del__(self):
2                   print("entered __del__")
3           self.close()
4           def close(self):
5                   if self.__file:
6                     self.__file.write('End Special File')
7                     self.__file.close()
8                     self.__file = None
```

Notice that close is written so that it can be called more than once without complaint. This is what you'll generally want to do. Also, the __del __function has a print expression in it. But this is just for demonstration purposes. Consider the following test function:

```
1 >>> def test():
2 ...   f = SpecialFile('testfile')
3 ...   f.write('111111\n')
4 ...   f.close()
5 ...
6 >>> test()
7 entered __del__
```

When the function test exits, f's reference count goes to zero and __del__is called. Thus, in the normal case close is called twice, which is why we want close to be able to handle this. If we forgot the f.close() at the end of test, the file would still be closed properly because we're backed up by the call to the destructor. This also happens if we reassign to the same variable without first closing the file:

In the Example 1, the del statement is used to delete the references of object *obj*, the destructor__del__is invoked automatically. The destructor was called after the program ended or when all the references to an object are deleted i.e. when the reference count becomes zero, not when the object went out of scope.

```
# Python program to illustrate destructor Example 1
class Employee:
# Initializing
def __init__(self):
print('Employee created.')

# Deleting (Calling destructor)
def __del__(self):
print('Destructor called, Employee deleted.')
obj = Employee()
del obj
```

output

```
Employee created.
Destructor called, Employee deleted.
```

In Example 2 notice that, the destructor is called after the 'Program End...' printed.

```
# Python program to illustrate destructor

class Employee:
# Initializing
def __init__(self):
print('Employee created')

```

```
# Calling destructor
def __del__(self):
print("Destructor called")

def Create_obj():
print('Making Object...')
obj = Employee()
print('function end...')
return obj

print('Calling Create_obj() function...')
obj = Create_obj()
print('Program End...')
```

Output

```
Calling Create_obj() function...
Making Object...
Employee created
function end...
Program End...
Destructor called
```

The following example illustrates the effect of a cyclical reference in Python and how you might break it. The purpose of the __del__method in this example is only to indicate when an object is removed:

```
>>> class Circle:
...     def __init__(self, name, parent):
...         self.name = name
...         self.parent = parent
```

```
5 ...                 self.child = None
6 ...                 if parent:
7 ...                         parent.child = self
8 ...         def cleanup(self):
9 ...                 self.child = self.parent = None
10 ...        def __del__(self):
11 ...                print("__del__ called on", self.name)
12 >>> def test1():
13 ...     a = Circle("a", None)
14 ...     b = Circle("b", a)
15 ...
16 >>> def test2():
17 ...         c = Circle("c", None)
18 ...         d = Circle("d", c)
19 ...         d.cleanup()
20 ...
21 >>> test1()
22 >>> test2()
23 __del__ called on c
24 __del__ called on d
```

Objects *a* and *b* are not removed when test1 exits q because they still refer each other. This is a memory leak. That is, each time test1 is called, it leaks two more objects. The explicit call to the cleanup method is necessary to avoid this. The cycle is broken in the cleanup method, not the destructor, and we only had to break it in one place. Python's reference-counting mechanism took over from there. This approach is not only more reliable, but also more efficient, because it reduces the amount of work that the garbage collector must do. A more robust method to perform this is use of try-finally compound

statement. It takes the following form:

```
try:
        body
finally:
        cleanup-statements
```

It ensures that cleanup-statements is executed regardless of how or from where body is exited. We can easily see this by writing and executing another test function for the Circle class defined earlier:

```
>>> def test3(x):
        try:
                c = Circle("c", None)
                d = Circle("d", c)
                if x == 1:
                  print("leaving-test3-via-return")
                return
                if x == 2:
                print("leaving-test3-via-exception")
                        raise RuntimeError
                        print("leaving-test3-end")
                finally:
                        d.cleanup()

>>> test3(0)
leaving-test3-off-the-end
__del__ called on c
__del__ called on d
>>> test3(1)
leaving-test3-via-return
```

```
__del__ called on c
__del__ called on d
>>> try:
...     test3(2)
...     except RuntimeError:
pass
leaving-test3-via-exception
__del__ called on c
__del__ called on d
```

Here, with the addition of three lines of code, it can be ensured that the cleanup method is called immediately after executing the function. This happens via an exception, a return statement, or returning after its last statement.

Lab Exercises

1. Design a system using a class called books with suitable member functions, constructors, and destructors. Implement a complete Python program for the system to incorporate the following: A book shop maintains the inventory of books that are being sold at the shop. The list includes details such as author, title, price, publisher and stock position. Whenever a customer wants a book, system searches the list and displays the availability based on the title and author. If it is not available, an appropriate message is displayed. If it is available, then the system displays the book details and requests for the number of copies required. If the requested copies are available, the total cost of it is displayed; otherwise the message "Required copies not in stock" is displayed.

2. Write a complete Python program to design a calculator. The program must read two real valued numbers from the user, perform basic opera-

tions (+,-,*,/,%) and catch exception for the following errors:

- If input number(s) is/are not real valued
- If the first number is lesser than second for subtraction operation
- Division by zero

3. Create a class Bank having data members: Name, Age, AccountNumber, AccountType and Balance. Add member functions, constructors, and destructors in order to perform the follwoing functions: create a new account, deposit given amount to the account, withdraw given amount from the account and print complete account details. The menu-driven program should handle the bank operations. Consider at least 5 bank account details.

4. Extend the Bank account class by adding following features:

   - Age (Should be greater than 18 and less than 100)
   - AccountNumber (should be a number having exactly a 5 digits)
   - AccountType (should be either S or C, S:Savings, C: current)

Additional Exercises

1. Add the following user-defined exceptions to Bank class:

   - Balance (should be non-negative and greater then Rs.1000)
   - Deposit (Daily deposite limit is Rs.10000)
   - Withdraw (Daily withdrawal limit is Rs. 5000 and after withdrawal the balance must not go below Rs.1000)

2. Extend the Book class by adding the user defined Exception "BookNotAvailable" and "InsufficientCopies".

10. MINI PROJECT

Objectives

- To design and develop a mini project employing all the concepts learnt.

Lab exercises

1. Design and develop a Python application considering network, web, and/or database concepts.

2. The mini project supposed to be carried out in two labs.

REFERENCES

1. Eric M., Python Crash Course: A Hands-On, Project-Based Introduction to Programming (1e), William Pollock Publication, 2016.

2. Mark L., Learning Python (5e), O'REILLY publication, 2013.

3. Daryl H., Kenneth M. M. and Vernon C., The Quick Python Book (2e), Manning Publications, 2011.

4. Allen B. D., Think Python: How to think like a Computer Scientist (2e), O'REILLY publication, 2015.