03 Introduction to Python

CONTENTS

Python was developed by Guido van Rossum at the National Research Institute for Mathematics and Computer Science in Netherlands during 1985-1990. Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, Unix shell and other scripting languages. Rossum was inspired by Monty Python's Flying Circus, a BBC comedy series and he wanted the name of his new language to be short, unique and mysterious. Hence he named it Python. It is a general-purpose interpreted, interactive, object-oriented and high-level Programming language. Python source code is available under the GNU General Public License (GPL) and it is now maintained by a core development team at the National Research Institute.

3.1 Features of Python

a) **Simple and easy-to-learn** – Python is a simple language with few keywords, simple structure and its syntax is also clearly defined. This makes Python a beginner's language.

b) **Interpreted and Interactive** - Python is processed at runtime by the interpreter. We need not compile the program before executing it. The Python prompt interact with the interpreter to interpret the programs that you have written. Python has an option namely interactive mode which allows interactive testing and debugging of code.

c) **Object-Oriented** – Python supports Object Oriented Programming (OOP) concepts that encapsulate code within objects. All concepts in OOPs like data hiding, operator overloading, inheritance etc. can be well written in Python. It supports functional as well as structured programming.

d) **Portable** - Python can run on a wide variety of hardware and software platforms and has the same interface on all platforms. All variants of Windows, Unix, Linux and Macintosh are to name a few.

e) **Scalable** - Python provides a better structure and support for large programs than shell scripting. It can be used as a scripting language or can be compiled to bytecode (intermediate code that is platform independent) for building large applications.

f) **Extendable** - You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient. It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

g) **Dynamic** - Python provides very high-level dynamic data types and supports dynamic type checking. It also supports automatic garbage collection.

h) **GUI Programming and Databases** - Python supports GUI applications that can be created and ported to many libraries and windows systems, such as Windows Microsoft Foundation Classes (MFC), Macintosh and the X Window system of Unix. Python also provides interfaces to all major commercial databases.

i) **Broad Standard Library** - Python's library is portable and cross platform compatible on UNIX, Linux, Windows and Macintosh. This helps in the support and development of a wide range of applications from simple text processing to browsers and complex games.

3.2 How to Run Python

There are three different ways to start Python.

a) Using Interactive Interpreter

You can start Python from Unix, DOS, or any other system that provides you a command-line interpreter or shell window. Get into the command line of Python. For Unix/Linux, you can get into interactive mode by typing `$python or python%`. For Windows/Dos it is `C:>python`.

Invoking the interpreter without passing a script file as a parameter brings up the following prompt–

```
$ python
Python 2.7.10 (default, Sep 27 2015, 18:11:38)
[GCC 5.1.1 20150422 (Red Hat 5.1.1-1)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

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

```
>>> print("Programming in Python!")
```

The result will be as given below.

```
Programming in Python!
>>>
```

b) Script from the Command Line

This method invokes the interpreter with a script parameter which begins the execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active. A Python script can be executed at command line by invoking the interpreter on your application, as follows.

For Unix/Linux it is `$python script.py` or `python% script.py`. For Windows/Dos it is `C:>python script.py`

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

```
print("Programming in Python!")
```

Now, try to run this program as follows.

```
$ python first.py
```

This produces the following result:

```
Programming in Python!
```

c) Integrated Development Environment

You can run Python from a Graphical User Interface (GUI) environment as well, if you have a GUI application on your system that supports Python. IDLE is the Integrated Development Environment (IDE) for UNIX and PythonWin is the first Windows interface for Python. All the programs in this book are tested in Python 3.4.3 for Windows.

3.3 Identifiers

A Python identifier is a name used to identify a variable, function, class, module or any other object. Python is case sensitive and hence uppercase and lowercase letters are considered distinct. The following are the rules for naming an identifier in Python.

a) Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_). For example Total and total is different.

b) Reserved keywords (explained in section 3.4) cannot be used as an identifier.

c) Identifiers cannot begin with a digit. For example 2more, 3times etc. are invalid identifiers.

d) Special symbols like @, !, #, $, % etc. cannot be used in an identifier. For example sum@, #total are invalid identifiers.

e) Identifier can be of any length.

Some examples of valid identifiers are total, max_mark, count2, Student etc. Here are some naming conventions for Python identifiers.

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter. Eg: Person
- Starting an identifier with a single leading underscore indicates that the identifier is private. Eg: _sum
- Starting an identifier with two leading underscores indicates a strongly private identifier. Eg: __sum
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name. foo__

3.4 Reserved Keywords

These are keywords reserved by the programming language and prevent the user or the programmer from using it as an identifier in a program. There are 33 keywords in Python 3.3. This number may vary with different versions. To retrieve the keywords in Python the following code can be given at the prompt. All keywords except True, False and None are in lowercase. The following list in Table 3.1 shows the Python keywords.

```
>>> import keyword
>>> print(keyword.kwlist)
['False', 'None', 'True', 'and', 'as', 'assert', 'break',
'class', 'continue', 'def', 'del', 'elif', 'else', 'except',
'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is',
'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return',
'try', 'while', 'with', 'yield']
```

Table 3.1: Python Keywords

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

3.5 Variables

Variables are reserved memory locations to store values. Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables. The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

Example Program

```
a = 100          # An integer assignment
b = 1000.0       # A floating point
name = "John"    # A string

print(a)
print(b)
print(name)
```

Here, 100, 1000.0 and "John" are the values assigned to a, b, and *name* variables, respectively. This produces the following result.

```
100
1000.0
John
```

Python allows you to assign a single value to several variables simultaneously. For example –

```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example

```
a, b, c = 1, 2, "Tom"
```

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "Tom" is assigned to the variable c.

3.6 Comments in Python

Comments are very important while writing a program. It describes what the source code has done. Comments are for programmers for better understanding of a program. In Python, we use the hash (#) symbol to start writing a comment. A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment. It extends up to the newline character. Python interpreter ignores comment.

Example Program

```
>>>#This is demo of comment
>>>#Display Hello
>>>print("Hello")
```

This produces the following result: `Hello`

For multiline comments one way is to use (#) symbol at the beginning of each line. The following example shows a multiline comment. Another way of doing this is to use triple quotes, either ''' or """.

Example 1

```
>>>#This is a very long sentence
>>>#and it ends after
>>>#Three lines
```

Example 2

```
>>>"""This is a very long sentence
>>>and it ends after
>>>Three lines"""
```

Both Example 1 and Example 2 given above produces the same result.

3.7 Indentation in Python

Most of the programming languages like C, C++ and Java use braces { } to define a block of code. Python uses indentation. A code block (body of a function, loop etc.) starts with indentation and ends with the first unindented line. The amount of indentation can be decided by the programmer, but it must be consistent throughout the block. Generally four whitespaces are used for indentation and is preferred over tabspace. For example

```
if True:
    print("Correct")
else:
  print("Wrong")
```

But the following block generates error as indentation is not properly followed.

```
if True:
    print("Answer")
    print("Correct")
else:
    print("Answer")
  print("Wrong")
```

3.8 Multi-Line Statements

Instructions that a Python interpreter can execute are called statements. For example, a=1 is an assignment statement. if statement, for statement, while statement etc. are other kinds of statements which will be discussed in coming Chapters. In Python, end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character(\). For example:

```
grand_total = first_item + \
              second_item + \
              third_item
```

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

```
months = ['January', 'February', 'March', 'April',
          'May', 'June', 'July', 'August', 'September',
          'October', 'November', 'December']
```

We could also put multiple statements in a single line using semicolons, as follows.

```
a = 1; b = 2; c = 3
```

3.9 Multiple Statement Group (Suite)

A group of individual statements, which make a single code block is called a **suite** in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite. Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite. For example

```
if expression :
   suite
elif expression :
   suite
else :
   suite
```

3.10 Quotes in Python

Python accepts single (') , double (") and triple (''' or """) quotes to denote string literals. The same type of quote should be used to start and end the string. The triple quotes are used to span the string across multiple lines. For example, all the following are legal.

```
word = 'single word'
sentence = "This is a short sentence."
paragraph = """This is a long paragraph. It consists of
          several lines and sentences."""
```

3.11 Input, Output and Import Functions

3.11.1 Displaying the Output

The function used to print output on a screen is the print statement where you can pass zero or more expressions separated by commas. The print function converts the expressions you pass into a string and writes the result to standard output.

Example 1

```
>>>print("Learning Python is fun and enjoy it.")
```

This will produce the following output on the screen.

```
Learning Python is fun and enjoy it.
```

Example 2

```
>>>a=2
>>>print("The value of a is",a)
```

This will produce the following output on the screen.

```
The value of a is 2
```

By default a space is added after the text and before the value of variable a. The syntax of *print* function is given below.

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

Here, `objects` are the values to be printed. `Sep` is the separator used between the values. Space character is the default separator. `End` is printed after printing all the values. The default value for end is the new line. `File` is the object where the values are printed and its default value is `sys.stdout` (screen). Flush determines whether the output stream needs to be flushed for any waiting output. A True value forcibly flushes the stream. The following shows an example for the above syntax.

Example

```
>>>print(1,2,3,4)
>>>print(1,2,3,4,sep='+')
>>> print(1,2,3,4,sep='+',end='%')
```

The output will be as follows.

```
1 2 3 4
1+2+3+4
1+2+3+4%
```

The output can also be formatted to make it attractive according to the wish of a user. This will be discussed in the subsequent Chapters.

3.11.2 Reading the Input

Python provides two built-in functions to read a line of text from standard input, which by default comes from the keyboard. These functions are **raw_input** and **input.**

```
raw_input function
```

The raw_input([prompt]) function reads one line from standard input and returns it as a string (removing the trailing newline). This prompts you to enter any string and it would display same string on the screen. raw_input function is not supported by Python 3.4.3 for Windows.

Example

```
str = raw_input("Enter your name: ");
print("Your name is : ", str)
```

Output

```
Enter your name: Collin Mark
Your name is : Collin Mark
```

input function

The input([prompt]) function is equivalent to raw_input, except that it assumes the input is a valid Python expression and returns the evaluated result to you.

Example 1

```
n = input("Enter a number ");
print("The number is : ", n)
```

Output

```
Enter a number: 5
The number is : 5
```

Example 2

```
n = input("Enter an expression ");
print("The result is : ", n)
```

Output

```
Enter an expression: 5*2
The result is : 10
```

3.11.3 Import function

When the program grows bigger or when there are segments of code that is frequently used, it can be stored in different modules. A module is a file containing Python definitions and statements. Python modules have a filename and end with the extension `.py`. Definitions inside a module can be imported to another module or the interactive interpreter in Python. We use the `import` keyword to do this. For example, we can import the `math` module by typing in `import math`.

```
>>> import math
>>> math.pi
3.141592653589793
```

3.12 Operators

Operators are the constructs which can manipulate the value of operands. Consider the expression a = b + c. Here a, b and c are called the operands and +, = are called the operators. There are different types of operators. The following are the types of operators supported by Python.

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

- Bitwise Operators
- Membership Operators
- Identity Operators

Let us have a look on all operators one by one.

3.12.1 Arithmetic Operators

Arithmetic operators are used for performing basic arithmetic operations. The following are the arithmetic operators supported by Python. Table 3.2 shows the arithmetic operators in Python.

Table 3.2: Arithmetic Operators in Python

Operator	Operation	Description
+	Addition	Adds values on either side of the operator.
-	Subtraction	Subtracts right hand operand from left hand operand.
*	Multiplication	Multiplies values on either side of the operator.
/	Division	Divides left hand operand by right hand operand.
%	Modulus	Divides left hand operand by right hand operand and returns remainder.
**	Exponent	Performs exponential (power) calculation on operators.
//	Floor Division	The division of operands where the result is the quotient in which the digits after the decimal point are removed.

Example Program

```
a, b, c = 10, 5, 2
print("Sum=",(a+b))
print("Difference=",(a-b))
print("Product=",(a*b))
print("Quotient=",(a/b))
print("Remainder=",(b%c))
print("Exponent=",(b**2))
print("Floor Division=",(b//c))
```

Output

```
Sum= 15
Difference= 5
Product= 50
Quotient= 2
Remainder= 1
Exponent= 25
Floor Division= 2
```

3.12.2 Comparison Operators

These operators compare the values on either sides of them and decide the relation among them. They are also called relational operators. Python supports the following relational operators. Table 3.3 shows the comparison or relational operators in Python.

Table 3.3: Comparison (Relational Operators) in Python

Operator	Description
==	If the values of two operands are equal, then the condition becomes true.
!=	If values of two operands are not equal, then condition becomes true.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.

Example Program

```
a,b=10,5
print("a==b is",(a==b))
print("a!=b is",(a!=b))
print("a>b is",(a>b))
print("a<b is",(a<b))
print("a>=b is",(a>=b))
print("a<=b is",(a<=b))
```

Output

```
a==b is False
a!=b is True
a>b is True
a<b is False
a>=b is True
a<=b is False
```

3.12.3 Assignment Operators

Python provides various assignment operators. Various shorthand operators for addition, subtraction, multiplication, division, modulus, exponent and floor division are also supported by Python. Table 3.4 provides the various assignment operators.

Table 3.4: Assignment Operators

Operator	Description
=	Assigns values from right side operands to left side operand.
+=	It adds right operand to the left operand and assign the result to left operand.
-=	It subtracts right operand from the left operand and assign the result to left operand.
*=	It multiplies right operand with the left operand and assign the result to left operand.
/=	It divides left operand with the right operand and assign the result to left operand.
%=	It takes modulus using two operands and assign the result to left operand.
**=	Performs exponential (power) calculation on operators and assign value to the left operand.
//=	It performs floor division on operators and assign value to the left operand.

The assignment operator = is used to assign values or values of expressions to a variable. Example: a = b+ c. For example c+=a is equivalent to c=c+a. Similarly c-=a is equivalent to c=c-a, c*=a is equivalent to c=c*a, c/=a is equivalent to c=c/a, c%=a is equivalent to c=c%a, c**=a is equivalent to c=c**a and c//=a is equivalent to c=c//a.

Example Program

```
a,b=10,5
a+=b
print(a)
a,b=10,5
a-=b
print(a)
a,b=10,5
a*=b
print(a)
a,b=10,5
a/=b
print(a)
b,c=5,2
b%=c
print(b)
b,c=5,2
b**=c
print(b)
b,c=5,2
b//=c
print(b)
```

Output

```
15
5
50
2
1
25
2
```

3.12.4 Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. The following are the bitwise operators supported by Python. Table 3.5 gives a description of bitwise operators in Python.

Table 3.5: Bitwise Operators

Operator	Operation	Description
&	Binary AND	Operator copies a bit to the result if it exists in both operands.
|	Binary OR	It copies a bit if it exists in either operand.
^	Binary XOR	It copies the bit if it is set in one operand but not both.
~	Binary Ones Complement	It is unary and has the effect of 'flipping' bits.
<<	Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.
>>	Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.

Let $a = 60$ (`0011 1100` in binary) and $b = 2$ (`0000 0010` in binary)

Example Program

```
a,b=60,2
print(a&b)
print(a|b)
print(a^b)
print(~a)
print(a>>b)
print(a<<b)
```

Output

```
0
62
62
```

```
-61
15
240
```

Consider the first print statement `a&b`. Here a = 0011 1100
b = 0000 0010

When a bitwise and is performed, the result is 0000 0000 . This is 0 in decimal. Similar is the case for all the print statements given above.

3.12.5 Logical Operators

Table 3.6 shows the various logical operators supported by Python language.

Table 3.6: Logical Operators

Operator	Operation	Description
and	Logical AND	If both the operands are true then condition becomes true.
or	Logical OR	If any of the operands are true then condition becomes true.
not	Logical NOT	Used to reverse the logical state of its operand.

Example Program

```
a,b,c,d=10,5,2,1
print((a>b)and(c>d))
print((a>b)or(d>c))
print(not(a>b))
```

Output

```
True
True
False
```

3.12.6 Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below in Table 3.7. Strings, lists and tuples will be covered in the forthcoming Chapter.

Table 3.7: Membership Operators

Operator	Description
in	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.

Example Program

```
s='abcde'
print('a' in s)
print('f' in s)
print('f'not in s)
```

Output

```
True
False
True
```

3.12.7 Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators as shown in below Table 3.8.

Table 3.8: Identity Operators

Operator	Description
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.

Example Program

```
a,b,c=10,10,5
print(a is b)
print(a is c)
print(a is not b)
```

Output

```
True
False
False
```

3.12.8 Operator Precedence

The following Table 3.9 lists all operators from highest precedence to lowest. Operator associativity determines the order of evaluation when they are of the some precedence and are not grouped by parenthesis. An operator may be left-associative or right-associative. In left-associative, the operator falling on the left side will be evaluated first, while in right-associative, operator falling on the right will be evaluated first. In Python '=' and '**' are right-associative while all other operators are left-associative.

Table 3.9: Operator Precedence

Operator	Description
**	Exponentiation (raise to the power)
~,+,-	Complement, unary plus and minus
*, /, %, //	Multiply, divide, modulo and floor division

Operator	Description
+, -	Addition and subtraction
>>, <<	Right and left bitwise shift
&	Bitwise 'AND'
^, |	Bitwise exclusive 'OR' and regular 'OR'
<=, < >, >=	Comparison operators
<>, ==, !=	Equality operators
= ,%= ,/=, //= ,-= ,+=, *= ,**=	Assignment operators
is, is not	Identity operators
in, not in	Membership operators
not, or, and	Logical operators

3.13 Conclusion

This Chapter gives an introduction to Python, its features, programming constructs like identifiers, reserved keywords, variables and various operators with illustrated examples. This Chapter also covers the purpose of indentation, the usage of comments, multi-line statements, multiple statements and quotes in Python, each one with illustrated examples. The basic Input/Output and import functions covered in this Chapter helps to handle the input, output and import operations in Python. Moreover how to execute Python is explained in this Chapter.

3.14 Review Questions

1. List some of the features of Python.
2. Is Python a case sensitive language?
3. How can you run Python?
4. Give the rules for naming an identifier in Python.
5. How can you give comments in Python?
6. What are multi-line statements?
7. What do you mean by suite in Python?
8. Briefly explain the Input and Output functions used in Python.
9. What is the purpose of import function?
10. What is the purpose of // operator?
11. What is the purpose of ** operator?
12. What is the purpose of is operator?
13. What is the purpose of not in operator?
14. Briefly explain the various types of operators in Python.
15. List out the operator precedence in Python.

04 Data Types and Operations

CONTENTS

The data stored in memory can be of many types. For example, a person's name is stored as alphabets, age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has the following standard data types.

- Numbers
- String
- List
- Tuple
- Set
- Dictionary

4.1 Numbers

Number data types store numeric values. Number objects are created when you assign a value to them. For example `a = 1, b = 20`. You can also delete the reference to a number object by using the del statement. The syntax of the del statement is as follows.

```
del variable1[,variable2[,variable3[....,variableN]]]
```

You can delete a single object or multiple objects by using the del statement. For example

```
del a
del a,b
```

Python supports four different numerical types.

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

Integers can be of any length, it is only limited by the memory available. A floating point number is accurate up to 15 decimal places. Integer and floating points are separated by decimal points. 1 is integer, 1.0 is floating point number. Complex numbers are written in the form, x + yj, where x is the real part and y is the imaginary part.

Example Program

```
a,b,c,d=1,1.5,231456987,2+9j
print("a=",a)
print("b=",b)
print("c=",c)
print("d=",d)
```

Output

```
a= 1
b= 1.5
c= 231456987
d= (2+9j)
```

4.1.1 Mathematical Functions

Python provides various built-in mathematical functions to perform mathematical calculations. The following Table 4.1 provides various mathematical functions and its purpose. For using the below functions, all functions except abs(x), max(x1,x2,...xn), min(x1,x2,...xn), round(x[,n]) and pow(x,y) need to import the math module because these functions reside in the math module. The math module also defines two mathematical constants `pi` and `e`. Modules will be discussed in Chapter 7.

Table 4.1: Mathematical Functions

Function	Description
abs(x)	Returns the absolute value of x.
sqrt(x)	Finds the square root of x.
ceil(x)	Finds the smallest integer not less than x.
floor(x)	Finds the largest integer not greater than x.

Function	Description
pow(x,y)	Finds x raised to y.
exp(x)	Returns e^x ie exponential of x.
fabs(x)	Returns the absolute value of x.
log(x)	Finds the natural logarithm of x for x>0.
log10(x)	Finds the logarithm to the base 10 for x>0.
max(x1,x2,...xn)	Returns the largest of its arguments.
min(x1,x2,...xn)	Returns the smallest of its arguments.
round(x,[n])	In case of decimal numbers, x will be rounded to n digits.
modf(x)	Returns the integer and decimal part as a tuple. The integer part is returned as a decimal.

Example Program

```
import math
print("Absolute value of -120:",abs(-120))
print("Square root of 25:",math.sqrt(25))
print("Ceiling of 12.2:", math.ceil(12.2))
print("Floor of 12.2:", math.floor(12.2))
print("2 raised to 3:", pow(2,3))
print("Exponential of 3:",math.exp(3))
print("Absolute value of -123:", math.fabs(-123))
print("Natural Logarithm of 2:", math.log(2))
print("Logarithm to the Base 10 of 2:",math.log10(2))
print("Largest among 10, 4, 2:",max(10,4,2))
print("Smallest among 10,4, 2:",min(10,4,2))
print("12.6789 rounded to 2 decimal places:",round(12.6789,2))
print("Decimal part and integer part of 12.090876:", math.
modf(12.090876))
```

Output

```
Absolute value of -120: 120
Square root of 25: 5.0
Ceiling of 12.2: 13
Floor of 12.2: 12
2 raised to 3: 8
Exponential of 3: 20.085536923187668
Absolute value of -123: 123.0
Natural Logarithm of 2: 0.6931471805599453
```

```
Logarithm to the Base 10 of 2: 0.3010299956639812
Largest among 10, 4, 2: 10
Smallest among 10,4, 2: 2
12.6789 rounded to 2 decimal places: 12.68
Decimal part and integer part of 12.090876:
(0.09087599999999973, 12.0)
```

4.1.2 Trigonometric Functions

There are several built-in trigonometric functions in Python. The function names and its purpose are listed in Table 4.2.

Table 4.2: Trigonometric Functions

Function	Description
sin(x)	Returns the sine of x in radians.
cos(x)	Returns the cosine of x in radians.
tan(x)	Returns the tangent of x in radians.
asin(x)	Returns the arc sine of x, in radians.
acos(x)	Returns the arc cosine of x, in radians.
atan(x)	Returns the arc tangent of x, in radians.
atan2(y,x)	Returns atan(y / x), in radians.
hypot(x,y)	Returns the Euclidean form, sqrt(x*x + y*y).
degrees(x)	Converts angle x from radians to degrees.
radians(x)	Converts angle x from degrees to radians.

Example

```
import math
print("Sin(90):",math.sin(90))
print("Cos(90):",math.cos(90))
print("Tan(90):",math.tan(90))
print("asin(1):",math.asin(1))
print("acos(1):",math.acos(1))
print("atan(1):",math.atan(1))
print("atan2(3,2):",math.atan2(3,2))
print("Hypotenuse of 3 and 4:",math.hypot(3,4))
print("Degrees of 90:",math.degrees(90))
print("Radians of 90:",math.radians(90))
```

Output

```
Sin(90): 0.893996663601
```

```
Cos(90): -0.448073616129
Tan(90): -1.99520041221
asin(1): 1.57079632679
acos(1): 0.0
atan(1): 0.785398163397
atan2(3,2): 0.982793723247
Hypotenuse of 3 and 4: 5.0
Degrees of 90: 5156.62015618
Radians of 90: 1.57079632679
```

4.1.3 Random Number Functions

There are several random number functions supported by Python. Random numbers have applications in research, games, cryptography, simulation and other applications. Table 4.3 shows the commonly used random number functions in Python. For using the random number functions, we need to import the module `random` because all these functions reside in the `random` module.

Table 4.3: Random Number Functions

Function	Description
choice(sequence)	Returns a random value from a sequence like iist, tuple, string etc.
shuffle(list)	Shuffles the items randomly in a list. Returns none.
random()	Returns a random floating-point number which lies between 0 and 1.
randrange([start,] stop [,step])	Returns a randomly selected number from a range where start shows the starting of the range, stop shows the end of the range and step decides the number to be added to decide a random number.
seed([x])	Gives the starting value for generating a random number. Returns none. This function is called before calling any other random module function.
uniform(x,y)	Generates a random floating-point number n such that n>x and n<y.

Example

```
import random
s='abcde'
print("choice(abcde):", random.choice(s))
list = [10,2,3,1,8,19]
print("shuffle(list):", random.shuffle(list))
print("random.seed(20):", random.seed(20))
print("random():",random.random())
print("uniform(2,3):", random.uniform(2,3))
print("randrange(2,10,1):",random.randrange(2,10,1))
```

Output

```
choice(abcde): b
shuffle(list): None
random.seed(20): None
random(): 0.905639676175
uniform(2,3): 2.68625415703
randrange(2,10,1): 8
```

4.2 Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and ending at -1.The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. The % operator is used for string formatting which will be discussed in Section 4.2.2.

Example Program

```
str = 'Welcome to Python Programming '
print(str) # Prints complete string
print(str[0]) # Prints first character of the string
print(str[11:17]) # Prints characters starting from 11th to 17th
print(str[11:]) # Prints string starting from 11th character
print(str * 2) # Prints string two times
print(str + "Session") # Prints concatenated string
```

Output

```
Welcome to Python Programming
W
Python
Python Programming
Welcome to Python Programming Welcome to Python Programming
Welcome to Python Programming Session
```

4.2.1 Escape Characters

An escape character is a character that gets interpreted when placed in single or double quotes. They are represented using backslash notation. These characters are non printable. The following Table 4.4 shows the most commonly used escape characters and their description.

Table 4.4: Escape Characters

Escape Characters	Description
\a	Bell or alert
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage Return
\s	Space
\t	Tab
\v	Vertical Tab

4.2.2 String Formatting Operator

This operator is unique to strings and is similar to the formatting operations of the printf() function of the C Programming language. The following Table 4.5 shows various format symbols and their conversion.

Table 4.5: Format Symbols and their Conversion

Format Symbols	Conversion
%c	Character
%s	String
%i	Signed Decimal Integer
%d	Signed Decimal Integer
%u	Unsigned Decimal Integer
%o	Octal Integer
%x	Hexadecimal Integer (lowercase letters)
%X	Hexadecimal Integer(uppercase letters)
%e	Exponential notation with lowercase letter 'e'
%E	Exponential notation with uppercase letter 'E'
%f	Floating-point real number

The following example shows the usage of various string formatting functions.

Example Program

```
#Demo of String Formatting Operators
print("The first letter of %s is %c " %('apple', 'a')) #usage of
%s and %c
print("The sum=%d" %(-12)) #usage of %d
print("The sum=%i" %(-12)) #usage of %i
print("The sum=%u" %(12)) #usage of %u
```

```
print("%o is the octal octal equivalent of %d" %(8,8)) #usage of %o
print("%x is the hexadecimal equivalent of %d" %(10,10))#usage
of %x
print("%X is the hexadecimal equivalent of %d" %(10,10))#usage
of %X
print("%e is the exponential equivalent of %f"
%(10.98765432,10.98765432)) #usage of %e and %f
print("%E is the exponential equivalent of %f"
%(10.98765432,10.98765432)) #usage of %E and %f
print( "%.2f" %(32.1274)) #usage of %f for printing upto two
decimal places
```

Output

```
The first letter of apple is a
The sum=-12
The sum=-12
The sum=12
10 is the octal octal equivalent of 8
a is the hexadecimal equivalent of 10
A is the hexadecimal equivalent of 10
1.098765e+01 is the exponential equivalent of 10.987654
1.098765E+01 is the exponential equivalent of 10.987654
32.13
```

4.2.3 String Formatting Functions

Python includes a large number of built-in functions to manipulate strings.

1. **len(string)**- Returns the length of the string

   Example Program

   ```
   #Demo of len(string)
   s='Learning Python is fun!'
   print("Length of", s, "is", len(s))
   ```

   Output

   ```
   Length of Learning Python is fun! is 23
   ```

2. **lower()** – Returns a copy of the string in which all uppercase alphabets in a string are converted to lowercase alphabets.

   Example Program

   ```
   #Demo of lower()
   s='Learning Python is fun!'
   print(s.lower())
   ```

```
print("%o is the octal octal equivalent of %d" %(8,8)) #usage of %o
print("%x is the hexadecimal equivalent of %d" %(10,10))#usage
of %x
print("%X is the hexadecimal equivalent of %d" %(10,10))#usage
of %X
print("%e is the exponential equivalent of %f"
%(10.98765432,10.98765432)) #usage of %e and %f
print("%E is the exponential equivalent of %f"
%(10.98765432,10.98765432)) #usage of %E and %f
print( "%.2f" %(32.1274)) #usage of %f for printing upto two
decimal places
```

Output

```
The first letter of apple is a
The sum=-12
The sum=-12
The sum=12
10 is the octal octal equivalent of 8
a is the hexadecimal equivalent of 10
A is the hexadecimal equivalent of 10
1.098765e+01 is the exponential equivalent of 10.987654
1.098765E+01 is the exponential equivalent of 10.987654
32.13
```

4.2.3 String Formatting Functions

Python includes a large number of built-in functions to manipulate strings.

1. **len(string)**- Returns the length of the string

   Example Program

```
#Demo of len(string)
s='Learning Python is fun!'
print("Length of", s, "is", len(s))
```

   Output

```
Length of Learning Python is fun! is 23
```

2. **lower()** – Returns a copy of the string in which all uppercase alphabets in a string are converted to lowercase alphabets.

   Example Program

```
#Demo of lower()
s='Learning Python is fun!'
print(s.lower())
```

Output

```
learning python is fun!
```

3. **upper()** – Returns a copy of the string in which all lowercase alphabets in a string are converted to uppercase alphabets.

Example Program

```
#Demo of upper()
s='Learning Python is fun!'
print(s.upper())
```

Output

```
LEARNING PYTHON IS FUN!
```

4. **swapcase()** – Returns a copy of the string in which the case of all the alphabets are swapped. ie. all the lowercase alphabets are converted to uppercase and vice versa.

Example Program

```
#Demo of swapcase()
s='LEARNing PYTHON is fun!'
print(s.swapcase())
```

Output

```
learnING python IS FUN!
```

5. **capitalize()** – Returns a copy of the string with only its first character capitalized.

Example Program

```
#Demo of capitalize()
s='learning Python is fun!'
print(s.capitalize())
```

Output

```
Learning python is fun!
```

6. **title()** - Returns a copy of the string in which first character of all the words are capitalized.

Example Program

```
#Demo of title()
s='learning Python is fun!'
print(s.title())
```

Output

```
Learning Python Is Fun!
```

7. **lstrip()** – Returns a copy of the string in which all the characters have been stripped(removed) from the beginning. The default character is whitespaces.

Example Program

```
#Demo of lstrip()
s='       learning Python is fun!'
print(s.lstrip())
s='*******learning Python is fun!'
print(s.lstrip('*'))
```

Output

```
learning Python is fun!
learning Python is fun!
```

8. **rstrip()** - Returns a copy of the string in which all the characters have been stripped(removed) from the ~~beginning~~ ending. The default character is whitespaces.

Example Program

```
#Demo of rstrip()
s='learning Python is fun!     '
print(s.rstrip())
s='learning Python is fun!******'
print(s.rstrip('*'))
```

Output

```
learning Python is fun!
learning Python is fun!
```

9. **strip()** - Returns a copy of the string in which all the characters have been stripped(removed) from the beginning and end . It performs both `lstrip()` and `rstrip()`. The default character is whitespaces.

Example Program

```
#Demo of strip()
s='   learning Python is fun!     '
print(s.strip())
s='******learning Python is fun!******'
print(s.strip('*'))
```

Output

```
learning Python is fun!
learning Python is fun!
```

10. **max(str)** – Returns the maximum alphabetical character from string `str`.

Example Program

```
#Demo of max(str)
```

```
s='learning Python is fun'
print('Maximum character is :', max(s))
```

Output

```
Maximum character is : y
```

11. **min(str)** - Returns the minimum alphabetical character from string `str`.

Example Program

```
#Demo of min(str)
s='learning-Python-is-fun'
print('Minimum character is :', min(s))
```

Output

```
Minimum character is : -
```

12. **replace(old, new [,max])** – Returns a copy of the string with all the occurrences of substring `old` is replaced by `new`. The `max` is optional and if it is specified, the first occurrences specified in `max` are replaced.

Example Program

```
# Demo of replace(old, new[,max])
s="This is very new.This is good"
print(s.replace('is', 'was'))
print(s.replace('is', 'was', 2))
```

Output

```
Thwas was very new.Thwas was good
Thwas was very new.This is good
```

13. **center(width, fillchar)** – Returns a string centered in a length specified in the `width` variable. Padding is done using the character specified in the `fillchar`. Default padding is space.

Example Program

```
# Demo of center(width, fillchar)
s="This is Python Programming"
print(s.center(30, '*'))
print(s.center(30))
```

Output

```
**This is Python Programming**
This is Python Programming
```

14. **ljust(width[,fillchar])** – Returns a string left-justified in a length specified in the `width` variable. Padding is done using the character specified in the `fillchar`. Default padding is space.

Example Program

```
# Demo of ljust(width[,fillchar])
s="This is Python Programming"
print(s.ljust(30, '*'))
print(s.ljust(30))
```

Output

```
This is Python Programming****
This is Python Programming
```

15. **rjust(width[,fillchar])** – Returns a string right-justified in a length specified in the `width` variable. Padding is done using the character specified in the `fillchar`. Default padding is space.

Example Program

```
# Demo of rjust(width[,fillchar])
s="This is Python Programming"
print(s.rjust(30, '*'))
print(s.rjust(30))
```

Output

```
****This is Python Programming
     This is Python Programming
```

16. **zfill(width)** - The method `zfill()` pads string on the left with zeros to fill `width`.

Example Program

```
# Demo of zfill(width)
s="This is Python Programming"
print(s.zfill(30))
```

Output

```
0000This is Python Programming
```

17. **count(str,beg=0,end=len(string))** – Returns the number of occurrences of `str` in the range `beg` and `end`.

Example Program

```
# Demo of count(str,beg=0,end=len(string))
s="This is Python Programming"
print(s.count('i',0,10))
print(s.count('i',0,25))
```

Output

```
2
3
```

18. **find(str,beg=0,end=len(string))** – Returns the index of `str` if `str` occurs in the range `beg` and `end` and returns -1 if it is not found.

Example Program

```
# Demo of find(str,beg=0,end=len(string))
s="This is Python Programming"
print(s.find('thon',0,25))
print(s.find('thy'))
```

Output

```
10
-1
```

19. **rfind(str,beg=0,end=len(string))** – Same as **find**, but searches backward in a string.

Example Program

```
# Demo of rfind(str,beg=0,end=len(string))
s="This is Python Programming"
print(s.rfind('thon',0,25))
print(s.rfind('thy'))
```

Output

```
10
-1
```

20. **index(str,beg=0,end=len(string))** – Same as that of find but raises an exception if the item is not found.

Example Program

```
# Demo of index(str,beg=0,end=len(string))
s="This is Python Programming"
print(s.index('thon',0,25))
print(s.index('thy'))
```

Output

```
10
Traceback (most recent call last):
File "main.py", line 4, in <module>
  print s.index('thy')
ValueError: substring not found
```

21. **rindex(str,beg=0,end=len(string))** – Same as index but searches backward in a string.

Example Program

```
# Demo of rindex(str,beg=0,end=len(string))
```

```
s="This is Python Programming"
print(s.rindex('thon',0,25))
print(s.rindex('thy'))
```

Output

```
10
Traceback (most recent call last):
File "main.py", line 4, in <module>
  print s.index('thy')
ValueError: substring not found
```

22. **startswith(suffix, beg=0,end=len(string))-** It returns `True` if the string begins with the specified suffix, otherwise return `False`.

Example Program

```
# Demo of startswith(suffix,beg=0,end=len(string))
s="Python Programming is fun"
print(s.startswith('is',10,21))
s="Python Programming is fun"
print(s.startswith('is',19,25))
```

Output

```
False
True
```

23. **endswith(suffix, beg=0,end=len(string))-** It returns `True` if the string ends with the specified suffix, otherwise return `False`.

Example Program

```
# Demo of endswith(suffix,beg=0,end=len(string))
s="Python Programming is fun"
print(s.endswith('is',10,21))
s="Python Programming is fun"
print(s.endswith('is',0,25))
```

Output

```
True
False
```

24. **isdecimal()** - Returns `True` if a unicode string contains only decimal characters and `False` otherwise. To define a string as unicode string, prefix 'u' to the front of the quotation marks.

Example Program

```
# Demo of isdecimal()
s=u"This is Python 1234"
```

Output

```
['Python', 'Programming', 'is', 'fun']
['Python', 'Programming', 'is', 'fun']
['Python\n', 'Programming\n', 'is\n', 'fun']
```

4.3 List

List is an ordered sequence of items. It is one of the most used data type in Python and is very flexible. All the items in a list do not need to be of the same type. Items separated by commas are enclosed within brackets [ ]. To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type. The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indices starting at 0 in the beginning of the list and ending with -1. The plus (+) sign is the list concatenation operator and the asterisk (*) is the repetition operator.

Example Program

```
first_list = ['abcd', 147, 2.43, 'Tom', 74.9]
small_list = [111, 'Tom']
print(first_list)# Prints complete list
print(first_list[0])# Prints first element of the list
print(first_list[1:3])# Prints elements starting from 2nd till 3rd
print(first_list[2:])# Prints elements starting from 3rd element
print(small_list * 2)# Prints list two times
print(first_list + small_list) # Prints concatenated lists
```

Output

```
['abcd', 147, 2.43, 'Tom', 74.9]
abcd
[147, 2.43]
[2.43, 'Tom', 74.9]
[111, 'Tom', 111, 'Tom']
[['abcd', 147, 2.43, 'Tom', 74.9, 111, 'Tom']
```

We can update lists by using the slice on the left hand side of the assignment operator. Updates can be done on single or multiple elements in a list.

Example Program

```
#Demo of List Update
list = ['abcd', 147, 2.43, 'Tom', 74.9]
print("Item at position 2=", list[2])
list[2]=500
print("Item at position 2=", list[2])
print("Item at Position 0 and 1 is", list[0],list[1])
list[0]=20; list[1]='apple'
print("Item at Position 0 and 1 is", list[0],list[1])
```

Output

```
Item at position 2= 2.43
Item at position 2= 500
Item at Position 0 and 1 is abcd 147
Item at Position 0 and 1 is 20 apple
```

To remove an item from a list, there are two methods. We can use `del` statement or `remove()` method which will be discussed in the subsequent session.

Example Program

```
#Demo of List Deletion
list = ['abcd', 147, 2.43, 'Tom', 74.9]
print(list)
del list[2]
print("List after deletion: ", list)
```

Output

```
['abcd', 147, 2.43, 'Tom', 74.9]
List after deletion:  ['abcd', 147, 'Tom', 74.9]
```

4.3.1 Built-in List Functions

1. **len(list)** – Gives the total length of the list.

Example Program

```
#Demo of len(list)
list1 = ['abcd', 147, 2.43, 'Tom']
print(len(list1))
```

Output

```
4
```

2. **max(list)** – Returns item from the list with maximum value.

Example Program

```
#Demo of max(list)
list1 = [1200, 147, 2.43, 1.12]
list2 = [213, 100, 289]
print("Maximum value in: ", list1, "is",  max(list1))
print("Maximum value in: ", list2, "is",  max(list2))
```

Output

```
Maximum value in:  [1200, 147, 2.43, 1.12] is 1200
Maximum value in:  [213, 100, 289] is 289
```

3. **min(list)** – Returns item from the list with minimum value.

Example Program

```
#Demo of min(list)
list1 = [1200, 147, 2.43, 1.12]
list2 = [213, 100, 289]
print("Minimum value in: ", list1, "is",  min(list1))
print("Minimum value in: ", list2, "is",  min(list2))
```

Output

```
Minimum value in:  [1200, 147, 2.43, 1.12] is 1.12
Minimum value in:  [213, 100, 289] is 100
```

4. **list(seq)** – Returns a tuple into a list.

Example Program

```
#Demo of list(seq)
tuple = ('abcd', 147, 2.43, 'Tom')
print("List:", list(tuple))
```

Output

```
List: ['abcd', 147, 2.43, 'Tom']
```

5. **map(aFunction,aSequence)** - One of the common things we do with list and other sequences is applying an operation to each item and collect the result. The **map(aFunction, aSequence)** function applies a passed-in function to each item in an iterable object and returns a list containing all the function call results.

Example Program

```
str=input("Enter a list(space separated):")
lis=list(map(int,str.split()))
print(lis)
```

Output

```
Enter a list(space separated):1 2 3 4
[1, 2, 3, 4]
```

In the above example, a string is read from the keyboard and each item is converted into int using map(aFunction,aSequence) function.

4.3.2 Built-in List Methods

1. **list.append(obj)** – This method appends an object obj passed to the existing list.

Example Program

```
#Demo of list.append(obj)
list = ['abcd', 147, 2.43, 'Tom']
```

```
print("Old List before Append:", list)
list.append(100)
print("New List after Append:", list)
```

Output

```
Old List before Append: ['abcd', 147, 2.43, 'Tom']
New List after Append: ['abcd', 147, 2.43, 'Tom', 100]
```

2. **list.count(obj)** – Returns how many times the object obj appears in a list.

Example Program

```
#Demo of list.count(obj)
list = ['abcd', 147, 2.43, 'Tom',147,200,147]
print("The number of times", 147, "appears in", list, "=",list.
count(147))
```

Output

```
The number of times 147 appears in['abcd', 147, 2.43, 'Tom',
147, 200, 147]= 3
```

3. **list.remove(obj)** – Removes object obj from the list.

Example Program

```
#Demo of list.remove(obj)
list1 = ['abcd', 147, 2.43, 'Tom']
list1.remove('Tom')
print(list1)
```

Output

```
['abcd', 147, 2.43]
```

4. **list.index(obj)** - Returns index of the object obj if found, otherwise raise an exception indicating that value does not exist.

Example Program

```
#Demo of list.index(obj)
list1 = ['abcd', 147, 2.43, 'Tom']
print(list1.index(2.43))
```

Output

```
2
```

5. **list.extend(seq)** – Appends the contents in a sequence seq passed to a list.

Example Program

```
#Demo of list.extend(seq)
```

```
list1 = ['abcd', 147, 2.43, 'Tom']
list2 = ['def', 100]
list1.extend(list2)
print(list1)
```

Output

```
['abcd', 147, 2.43, 'Tom', 'def', 100]
```

6. **list.reverse()** – Reverses objects in a list.

Example Program

```
#Demo of list.reverse()
list1 = ['abcd', 147, 2.43, 'Tom']
list1.reverse()
print(list1)
```

Output

```
['Tom', 2.43, 147, 'abcd']
```

7. **list.insert(index,obj)** – Returns a list with object obj inserted at the given index.

Example Program

```
#Demo of list.insert(index,obj)
list1 = ['abcd', 147, 2.43, 'Tom']
print("List before insertion:",list1)
list1.insert(2,222)
print("List after insertion:",list1)
```

Output

```
List before insertion: ['abcd', 147, 2.43, 'Tom']
List after insertion: ['abcd', 147, 222, 2.43, 'Tom']
```

8. **list.sort([func])** – Sorts the items in a list and returns the list. If a function is provided, it will compare using the function provided.

Example Program

```
#Demo of list.sort([func])
list1 = [890, 147, 2.43, 100]
print("List before sorting:",list1)
list1.sort()
print("List after sorting:",list1)
```

Output

```
List before sorting: [890, 147, 2.43, 100]
List after sorting: [2.43, 100, 147, 890]
```

9. **list.pop(obj=list[-1])** – Removes or returns the last object obj from a list.

Example Program

```
#Demo of list.pop(obj=list[-1])
list1 = ['abcd', 147, 2.43, 'Tom']
print("List before poping",list1)
list1.pop(-1)
print("List after poping:",list1)
```

Output

```
List before poping ['abcd', 147, 2.43, 'Tom']
List after poping: ['abcd', 147, 2.43]
```

4.4 Tuple

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. The main differences between lists and tuples are lists are enclosed in square brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be considered as read-only lists.

Example Program

```
first_tuple = ('abcd', 147, 2.43, 'Tom', 74.9)
small_tuple = (111, 'Tom')

print(first_tuple)# Prints complete tuple
print(first_tuple[0])# Prints first element of the tuple
print(first_tuple[1:3])# Prints elements starting from 2nd till 3rd
print(first_tuple[2:])# Prints elements starting from 3rd element
print(small_tuple * 2)# Prints tuple two times
print(first_tuple + small_tuple) # Prints concatenated tuples
```

Output

```
('abcd', 147, 2.43, 'Tom', 74.9)
abcd
(147, 2.43)
(2.43, 'Tom', 74.9)
(111, 'Tom', 111, 'Tom')
('abcd', 147, 2.43, 'Tom', 74.9, 111, 'Tom')
```

The following code is invalid with tuple whereas this is possible with lists.

```
first_list = [ 'abcd', 147 , 2.43, 'Tom', 74.9 ]
first_tuple = ('abcd', 147, 2.43, 'Tom', 74.9)
tuple[2] = 100     # Invalid syntax with tuple
list[2] = 100      # Valid syntax with list
```

To delete an entire tuple we can use the del statement. It is not possible to remove individual items from a tuple. Eg. del tuple

4.4.1 Built-in Tuple Functions

1. **len(tuple)** – Gives the total length of the tuple.

   Example Program

```
#Demo of len(tuple)
tuple1 = ('abcd', 147, 2.43, 'Tom')
print(len(tuple1))
```

   Output

```
4
```

2. **max(tuple)** – Returns item from the tuple with maximum value.

   Example Program

```
#Demo of max(tuple)
tuple1 = (1200, 147, 2.43, 1.12)
tuple2 = (213, 100, 289)
print("Maximum value in: ", tuple1, "is",  max(tuple1))
print("Maximum value in: ", tuple2, "is",  max(tuple2))
```

   Output

```
Maximum value in:  (1200, 147, 2.43, 1.12) is 1200
Maximum value in:  (213, 100, 289) is 289
```

3. **min(tuple)** – Returns item from the tuple with minimum value.

   Example Program

```
#Demo of min(tuple)
tuple1 = (1200, 147, 2.43, 1.12)
tuple2 = (213, 100, 289)
print("Minimum value in: ", tuple1, "is",  min(tuple1))
print("Minimum value in: ", tuple2, "is",  min(tuple2))
```

   Output

```
Minimum value in:  (1200, 147, 2.43, 1.12) is 1.12
Minimum value in:  (213, 100, 289) is 100
```

4. **tuple(seq)** – Returns a list into a tuple.

   Example Program

```
#Demo of tuple(seq)
list = ['abcd', 147, 2.43, 'Tom']
print("Tuple:", tuple(list))
```

Output

```
Tuple: ('abcd', 147, 2.43, 'Tom')
```

4.5 Set

Set is an unordered collection of unique items. Set is defined by values separated by comma inside braces { }. It can have any number of items and they may be of different types (integer, float, tuple, string etc.). Items in a set are not ordered. Since they are unordered we cannot access or change an element of set using indexing or slicing. We can perform set operations like union, intersection, difference on two sets. Set have unique values. They eliminate duplicates. The slicing operator [] does not work with sets.

Example Program

```
# Demo of Set Creation
s1={1,2,3} #set of integer numbers
print(s1)
s2={1,2,3,2,1,2}#output contains only unique values
print(s2)
s3={1, 2.4, 'apple', 'Tom', 3}#set of mixed data types
print(s3)
#s4={1,2,[3,4]}#sets cannot have mutable items
#print s4 # Hence not permitted
s5=set([1,2,3,4])#using set function to create set from a list
print(s5)
```

Output

```
set([1, 2, 3])
set([1, 2, 3])
set([1, 3, 2.4, 'apple', 'Tom'])
set([1, 2, 3, 4])
```

4.5.1 Built-in Set Functions

1. **len(set)** – Returns the length or total number of items in a set.

Example Program

```
#Demo of len(set)
set1 = {'abcd', 147, 2.43, 'Tom'}
print(len(set1))
```

Output

```
4
```

2. **max(set)** – Returns item from the set with maximum value.

Example Program

```
#Demo of max(set)
set1 = {1200, 147, 2.43, 1.12}
set2 = {213, 100, 289}
print("Maximum value in: ", set1, "is",  max(set1))
print("Maximum value in: ", set2, "is",  max(set2))
```

Output

```
Maximum value in:  {1200, 1.12, 2.43, 147} is 1200
Maximum value in:  {289, 100, 213} is 289
```

3. **min(set)** – Returns item from the set with minimum value.

Example Program

```
#Demo of min(set)
set1 = {1200, 147, 2.43,1.12}
set2 = {213, 100, 289}
print("Minimum value in: ", set1, "is",  min(set1))
print("Minimum value in: ", set2, "is",  min(set2))
```

Output

```
Minimum value in:  {1200, 1.12, 2.43, 147} is 1.12
Minimum value in:  {289, 100, 213} is 100
```

4. **sum(set)** – Returns the sum of all items in the set.

Example Program

```
#Demo of sum(set)
set1 = {147, 2.43}
set2 = {213, 100, 289}
print("Sum of elements in", set1, "is",  sum(set1))
print("Sum of elements in", set2, "is",  sum(set2))
```

Output

```
Sum of elements in {147, 2.43} is 149.43
Sum of elements in {289, 100, 213} is 602
```

5. **sorted(set)**- Returns a new sorted set. The set does not sort itself.

Example Program

```
#Demo of sorted(set)
set1 = {213, 100, 289, 40, 23, 1, 1000}
```

```
set2 = sorted(set1)
print("Sum of elements before sorting", set1)
print("Sum of elements after sorting", set2)
```

Output

```
Sum of elements before sorting {1, 100, 289, 1000, 40, 213, 23}
Sum of elements after sorting {1, 23, 40, 100, 213, 289, 1000}
```

6. **enumerate(set)** – Returns an enumerate object. It contains the index and value of all the items of set as a pair.

Example Program

```
#Demo of enumerate(set)
set1 = {213, 100, 289, 40, 23, 1, 1000}
print("enumerate(set):", enumerate(set1))
```

Output

```
enumerate(set): <enumerate object at 0x7f0a73573690>
```

7. **any(set)** – Returns True, if the set contains at least one item, False otherwise.

Example Program

```
#Demo of any(set)
set1 = set()
set2={1,2,3,4}
print("any(set):", any(set1))
print("any(set):", any(set2))
```

Output

```
any(set): False
any(set): True
```

8. **all(set)** – Returns True, if all the elements are true or the set is empty.

Example Program

```
#Demo of all(set)
set1 = set()
set2={1,2,3,4}
print("all(set):", all(set1))
print("all(set):", all(set1))
```

Output

```
all(set): True
all(set): True
```

15. set1.issuperset(set2) – Returns True, if set1 is a super set of set2.

Example Program

```
#Demo of set1.issuperset(set2)
set1={3,8,4,6}
print("Set1:",set1)
set2={3,8,}
print("Set2:",set2)
print("Result of set1.issuperset(set2):", set1.issuperset(set2))
```

Output

```
Set1: {8, 3, 4, 6}
Set2: {8, 3}
Result of set1.issuperset(set2): True
```

4.5.3 Frozenset

Frozenset is a new class that has the characteristics of a set, but its elements cannot be changed once assigned. While tuples are immutable lists, frozensets are immutable sets. Sets being mutable are unhashable, so they can't be used as dictionary keys which will be discussed in the next section. On the other hand, frozensets are hashable and can be used as keys to a dictionary.

Frozensets can be created using the function frozenset(). This datatype supports methods like difference(), intersection(), isdisjoint(), issubset(), issuperset(), symmetric_difference() and union(). Being immutable it does not have methods like add(), remove(), update(), difference_update(),intersection_update(),symmetric_difference_update() etc.

Example Program

```
#Demo of frozenset()
set1= frozenset({3,8,4,6})
print("Set1:",set1)
set2=frozenset({3,8})
print("Set2:",set2)
print("Result of set1.intersection(set2):", set1.
intersection(set2))
```

Output

```
Set1: frozenset({8, 3, 4, 6})
Set2: frozenset({8, 3})
Result of set1.intersection(set2): frozenset({8, 3})
```

4.6 Dictionary

Dictionary is an unordered collection of key-value pairs. It is generally used when we have a huge

amount of data. We must know the key to retrieve the value. In Python, dictionaries are defined within braces {} with each item being a pair in the form key:value. Key and value can be of any type. Keys are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).

Example Program

```
dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"
tinydict = {'name': 'john','code':6734, 'dept': 'sales'}

print(dict['one'])          # Prints value for 'one' key
print(dict[2])              # Prints value for 2 key
print(tinydict)             # Prints complete dictionary
print(tinydict.keys())      # Prints all the keys
print(tinydict.values())    # Prints all the values
```

Output

```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```

We can update a dictionary by adding a new key-value pair or modifying an existing entry.

Example Program

```
#Demo of updating and adding new values to dictionary
dict1= {'Name':'Tom','Age':20,'Height':160}
print(dict1)
dict1['age']=25 #updating existing value in Key-Value pair
print("Dictionary after update:",dict1)
dict1['Weight']=60 #Adding new Key-value pair
print("Dictionary after adding new Key-value pair:",dict1)
```

Output

```
{'Age': 20, 'Name': 'Tom', 'Height': 160}
Dictionary after update: {'age': 25, 'Age': 20, 'Name': 'Tom',
'Height': 160}
Dictionary after adding new Key-value pair: {'age': 25, 'Age':
20, 'Name': 'Tom', 'Weight': 60, 'Height': 160}
```

We can delete the entire dictionary elements or individual elements in a dictionary. We can use del statement to delete the dictionary completely. To remove entire elements of a dictionary, we can use the clear() method which will be discussed in the built-in methods of dictionary.

Example Program

```
#Demo of Deleting Dictionary
dict1= {'Name':'Tom','Age':20,'Height':160}
print(dict1)
del dict1['Age'] #deleting Key-value pair 'Age':20
print("Dictionary after deletion:",dict1)
dict1.clear() #Clearing entire dictionary
print(dict1)
```

Output

```
{'Age': 20, 'Name': 'Tom', 'Height': 160}
Dictionary after deletion: {'Name': 'Tom', 'Height': 160}
{}
```

Properties of Dictionary Keys

1. More than one entry per key is not allowed.ie, no duplicate key is allowed. When duplicate keys are encountered during assignment, the last assignment is taken.
2. Keys are immutable. This means keys can be numbers, strings or tuple. But it does not permit mutable objects like lists.

4.6.1 Built-in Dictionary Functions

1. **len(dict)** – Gives the length of the dictionary.

   Example Program

   ```
   #Demo of len(dict)
   dict1= {'Name':'Tom','Age':20,'Height':160}
   print(dict1)
   print("Length of Dictionary=",len(dict1))
   ```

   Output

   ```
   {'Age': 20, 'Name': 'Tom', 'Height': 160}
   Length of Dictionary= 3
   ```

2. **str(dict)** - Produces a printable string representation of the dictionary.

   Example Program

   ```
   #Demo of str(dict)
   dict1= {'Name':'Tom','Age':20,'Height':160}
   print(dict1)
   print("Representation of Dictionary=",str(dict1))
   ```

Output

```
{'Age': 20, 'Name': 'Tom', 'Height': 160}
Representation of Dictionary= {'Age': 20, 'Name': 'Tom',
'Height': 160}
```

3. **type(variable)** - The method type() returns the type of the passed variable. If passed variable is dictionary then it would return a dictionary type. This function can be applied to any variable type like number, string, list, tuple etc.

Example Program

```
#Demo of type(variable)
dict1= {'Name':'Tom','Age':20,'Height':160}
print(dict1)
print("Type(variable)=",type(dict1))
s="abcde"
print("Type(variable)=",type(s))
list1= [1,'a',23,'Tom']
print("Type(variable)=",type(list1))
```

Output

```
{'Age': 20, 'Name': 'Tom', 'Height': 160}
Type(variable)= <type 'dict'>
Type(variable)= <type 'str'>
Type(variable)= <type 'list'>
```

4.6.2 Built-in Dictionary Methods

1. **dict.clear()** – Removes all elements of dictionary dict.

Example Program

```
#Demo of dict.clear()
dict1= {'Name':'Tom','Age':20,'Height':160}
print dict1
dict1.clear()
print dict1
```

Output

```
{'Age': 20, 'Name': 'Tom', 'Height': 160}
{}
```

2. **dict.copy()** – Returns a copy of the dictionary dict().

Example Program

```
#Demo of dict.copy()
dict1= {'Name':'Tom','Age':20,'Height':160}
```

```
print(dict1)
dict2=dict1.copy()
print(dict2)
```

Output

```
{'Age': 20, 'Name': 'Tom', 'Height': 160}
{'Age': 20, 'Name': 'Tom', 'Height': 160}
```

3. **dict.keys()** – Returns a list of keys in dictionary dict.

Example Program

```
#Demo of dict.keys()
dict1= {'Name':'Tom','Age':20,'Height':160}
print(dict1)
print("Keys in Dictionary:",dict1.keys())
```

Output

```
{'Age': 20, 'Name': 'Tom', 'Height': 160}
Keys in Dictionary: ['Age', 'Name', 'Height']
```

4. **dict.values()** – This method returns list of all values available in a dictionary.

Example Program

```
#Demo of dict.values()
dict1= {'Name':'Tom','Age':20,'Height':160}
print(dict1)
print("Values in Dictionary:",dict1.values())
```

Output

```
{'Age': 20, 'Name': 'Tom', 'Height': 160}
Values in Dictionary: [20, 'Tom', 160]
```

5. **dict.items()** – Returns a list of dictionary dict's(key,value) tuple pairs.

Example Program

```
#Demo of dict.items()
dict1= {'Name':'Tom','Age':20,'Height':160}
print(dict1)
print("Items in Dictionary:",dict1.items())
```

Output

```
{'Age': 20, 'Name': 'Tom', 'Height': 160}
Items in Dictionary: [('Age', 20), ('Name', 'Tom'), ('Height',
160)]
```

Output

```
{'Age': 20, 'Name': 'Tom', 'Height': 160}
Dict1.get(key) : 20
Dict1.get(key) : 0
```

9. **dict.setdefault(key,default=None)** – Similar to dict.get(key,default=None) but will set the key with the value passed and if key is not in the dictionary, it will set with the default value.

Example Program

```
#Demo of dict1.set(key,default='Name')
dict1= {'Name':'Tom','Age':20,'Height':160}
print(dict1)
print("Dict1.setdefault('Age') :",dict1.setdefault('Age'))
print("Dict1.setdefault('Phone') :",dict1.
setdefault('Phone',0))#'Phone not a #key, hence 0 is given as
default value.
```

Output

```
{'Age': 20, 'Name': 'Tom', 'Height': 160}
Dict1.setdefault('Age') : 20
Dict1.setdefault('Phone') : 0
```

10. **dict.fromkeys(seq,[val])** – Creates a new dictionary from sequence seq and values from val.

Example Program

```
#Demo of dict.fromkeys(seq,[val])
list= ['Name','Age','Height']
dict= dict.fromkeys(list)
print("New Dictionary:",dict)
```

Output

```
New Dictionary: {'Age': None, 'Name': None, 'Height': None}
```

4.7 Mutable and Immutable Objects

Objects whose value can change are said to be mutable and objects whose value is unchangeable once they are created are called immutable. An object's mutability is determined by its type. For instance, numbers, string and tuples are immutable, while lists, sets and dictionaries are mutable. The following example illustrates the mutability of objects.

Example Program

```
#Mutability illustration
#Numbers
```

```
a=10
print(a)#Value of a before subtraction
print(a-2)#value of a-2
print(a)#Value of a after subtraction
#Strings
str="abcde"
print(str)#Before applying upper()function
print(str).upper())#result of upper()
print(str)#After applying upper()function
#Lists
list=[1,2,3,4,5]
print(list)#Before applying remove()method
list.remove(3)
print(list)#After applying remove()method
```

Output

```
10
8
10
abcde
ABCDE
abcde
[1,2,3,4,5]
[1,2,4,5]
```

From the above example, it is clear that the values of numbers and string are not changed even after applying a function or operation. But in the case of list, the value is changed or the changes are reflected in the original variable and hence called mutable.

4.8 Data Type Conversion

We may need to perform conversions between the built-in types. To convert between different data types, use the type name as a function. There are several built-in functions to perform conversion fromone data type to another. These functions return a new object representing the converted value. Table 4.6 shows various functions and their descriptions used for data type conversion.

Table 4.6: Functions for Data Type Conversions

Function	Description
int(x [,base])	Converts x to an integer. base specifies the base if x is a string.
long(x [,base])	Converts x to a long integer. base specifies the base if x is a string.
float(x)	Converts x to a floating-point number.

Function	Description
complex(real [,imag])	Creates a complex number.
str(x)	Converts object x to a string representation.
repr(x)	Converts object x to an expression string.
eval(str)	Evaluates a string and returns an object.
tuple(s)	Converts s to a tuple.
list(s)	Converts s to a list.
set(s)	Converts s to a set.
dict(d)	Creates a dictionary. d must be a sequence of (key,value) tuples.
frozenset(s)	Converts s to a frozen set.
chr(x)	Converts an integer to a character.
unichr(x)	Converts an integer to a Unicode character.
ord(x)	Converts a single character to its integer value.
hex(x)	Converts an integer to a hexadecimal string.
oct(x)	Converts an integer to an octal string.

Example Program 1

```
# Demo of Data Type Conversions
x= 12.8
print("Integer x=",int(x))
x=12000
print("Long x=", long(x))
x=12
print("Floating Point x=", float(x))
real,img=5,2
print("Complex number =",complex(real,img))
x=12
print("String conversion of", x ,"is", str(x))
x=12
print("Expression String of", x ,"is", repr(x))
```

Output 1

```
Integer x= 12
Long x= 12000
Floating Point x= 12.0
Complex number = (5+2j)
String conversion of 12 is 12
Expression String of 12 is 12
```

Example Program 2

```
# Demo of Data Type Conversions
s='abcde'
s1={1:'a',2:'b',3:'c'}
print("Conversion of", s ,"to tuple is ",tuple(s))
print("Conversion of", s ,"to list is ",list(s))
print("Conversion of", s ,"to set is ",set(s))
print("Conversion of", s ,"to frozenset is ",frozenset(s))
print("Conversion of", s1 ,"to dictionary is ",dict(s1))
```

Output 2

```
Conversion of abcde to tuple is ('a', 'b', 'c', d', 'e')
Conversion of abcde to list is ['a', 'b', 'c','d', 'e']
Conversion of abcde to set is set(['a', 'c', 'b', 'e', 'd'])
Conversion of abcde to frozenset is frozenset(['a', 'c', 'b',
'e', 'd'])
Conversion of {1: 'a', 2: 'b', 3: 'c'} to dictionary is {1: 'a',
2: 'b', 3: 'c'}
```

Example Program 3

```
# Demo of Data Type Conversions
a=120
s='a'
print("Conversion of", a ,"to character is",chr(a) )
print("Conversion of", a ,"to unicode character is",unichr(a))
print("Conversion of", s ,"to integer is",ord(s))
print("Conversion of", a ,"to hexadecimal string is",hex(a))
print("Conversion of", a ,"to octal string is",oct(a))
```

Output 3

```
Conversion of 120 to character is x
Conversion of 120 to unicode character is x
Conversion of a to integer is 97
Conversion of 120 to hexadecimal string is 0x78
Conversion of 120 to octal string is 0170
```

05 Flow Control

CONTENTS

5.1 Decision Making

Decision making is required when we want to execute a code only if a certain condition is satisfied. The `if...elif...else` statement is used in Python for decision making.

5.1.1 *if* statement

Syntax

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

The following Fig. 5.1 shows the flowchart of the `if` statement.

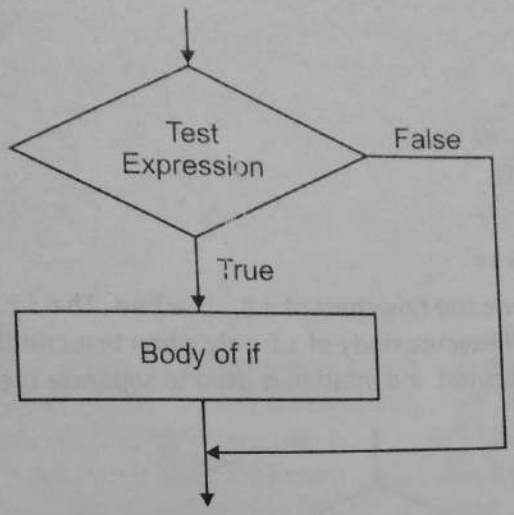

Fig. 5.1: Flowchart of *if* Statement

Here, the Program evaluates the `test expression` and will execute statement(s) only if the text expression is `True`. If the text expression is `False`, the statement(s) is not executed. In Python, the body of the `if` statement is indicated by the indentation. Body starts with an indentation and ends with the first unindented line. Python interprets non-zero values as `True`. `None` and `0` are interpreted as `False`.

Example Program

```
num =int(input("Enter a number: "))
if num == 0:
    print("Zero")
print("This is always printed")
```

Output 1

```
Enter a number: 0
Zero
This is always printed
```

Output 2

```
Enter a number: 2
This is always printed
```

In the above example, `num == 0` is the test expression. The body of `if` is executed only if this evaluates to `True`. When user enters 0, test expression is True and body inside `if` is executed. When user enters 2, test expression is False and body inside `if` is skipped. The statement `This is always printed` because the `print` statement falls outside the `if` block. The statement outside the `if` block is shown by unindentation. Hence, it is executed regardless of the test expression or it is always executed.

5.1.2 *if....else* statement

Syntax

```
if test expression:
    Body of if
else:
    Body of else
```

The following Fig. 5.2 shows the flowchart of `if.....else`. The `if..else` statement evaluates `test expression` and will execute body of `if` only when test condition is `True`. If the condition is `False`, body of `else` is executed. Indentation is used to separate the blocks.

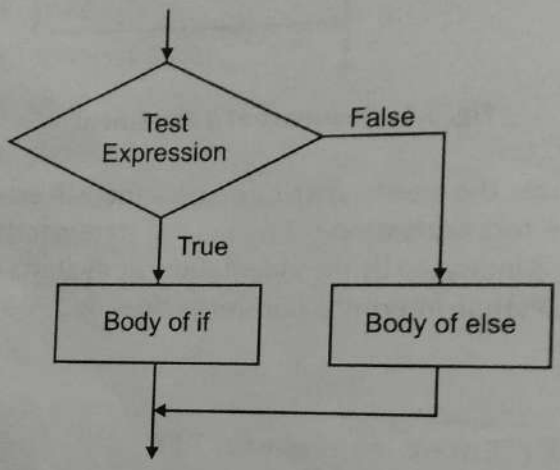

Fig. 5.2: Flow chart of `if.....else`

Example Program

```
num =int(input("Enter a number: "))
if num >= 0:
    print("Positive Number or Zero")
else:
    print("Negative Number")
```

Output 1

```
Enter a number: 5
Positive Number or Zero
```

Output 2

```
Enter a number: -2
Negative Number
```

In the above example, when user enters 5, the test expression is `True`. Hence the body of `if` is executed and body of `else` is skipped. When user enters -2, the test expression is `False` and body of `else` is executed and body of `if` is skipped.

5.1.3 if...elif...else statement

Syntax

```
if test expression:
    Body of if
elif test expression:
    Body of elif
else:
    Body of else
```

The `elif` is short for else if. It allows us to check for multiple expressions. If the condition for `if` is `False`, it checks the condition of the next `elif` block and so on. If all the conditions are `False`, body of else is executed. Only one block among the several `if...elif...else` blocks is executed according to the condition. A `if` block can have only one `else` block. But it can have multiple `elif` blocks. The following Fig. 5.3 shows the flow chart for `if...elif..else` statement.

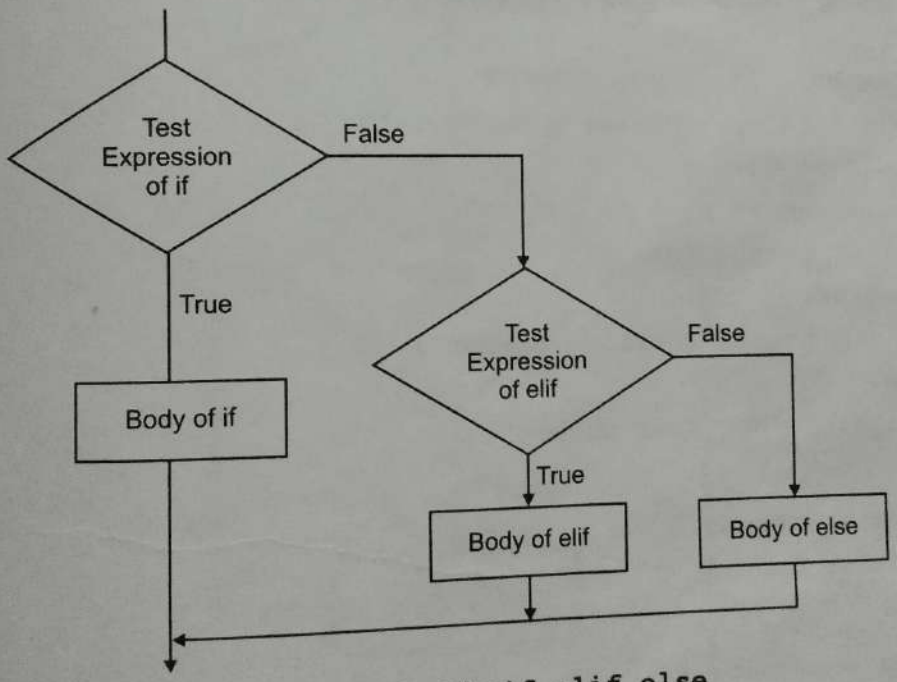

Fig. 5.3: Flow chart for `if...elif...else`

Example Program

```
num =int(input("Enter a number: "))
if num > 0:
    print("Positive number")
elif num == 0:
    print("Zero")
else:
    print("Negative number")
```

Output 1

```
Enter a number: 5
Positive Number
```

Output 2

```
Enter a number: 0
Zero
```

Output 3

```
Enter a number: -2
Negative Number
```

5.1.4 Nested *if* statement

We can have a `if...elif...else` statement inside another `if...elif...else` statement. This is called nesting in computer programming. Indentation is the only way to identify the level of nesting.

Example Program

```
num = int(input("Enter a number: "))
if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive number")
else:
    print("Negative number")
```

Output 1

```
Enter a number: 5
Positive Number
```

Output 2

```
Enter a number: 0
Zero
```

Output 3

```
Enter a number: -2
Negative Number
```

5.2 Loops

Generally, statements are executed sequentially. The first statement in a function is executed first, followed by the second, and so on. There will be situations when we need to execute a block of code

several number of times. Python provides various control structures that allow for repeated execution. A loop statement allows us to execute a statement or group of statements multiple times.

5.2.1 for loop

The `for` loop in Python is used to iterate over a sequence (list, tuple, string) or other objects that can be iterated. Iterating over a sequence is called traversal. Fig. 5.4 shows the flow chart of `for` loop.

Syntax

```
for item in sequence:
    Body of for
```

Here, `item` is the variable that takes the value of the item inside the sequence of each iteration. The sequence can be list, tuple, string, set etc. Loop continues until we reach the last item in the sequence. The body of `for` loop is separated from the rest of the code using indentation.

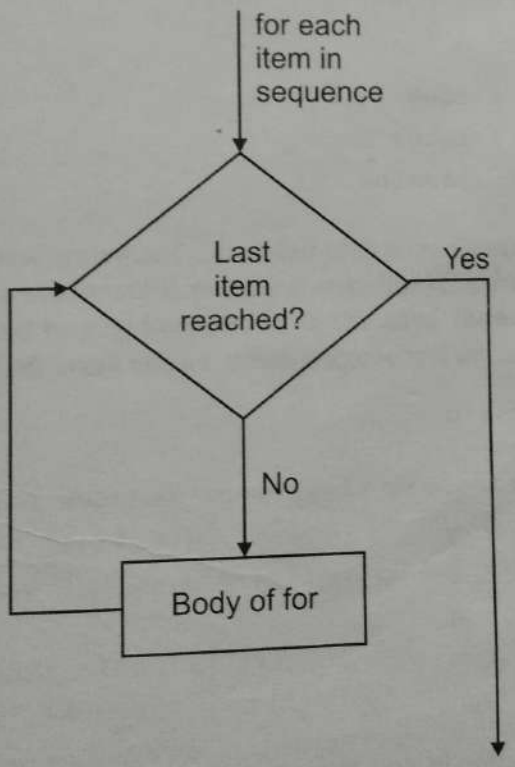

Fig. 5.4: Flow chart of `for` loop

Example Program 1

```
#Program to find the sum of all numbers stored in a list
# List of numbers
numbers = [2,4,6,8,10]
# variable to store the sum
```

```
sum = 0
# iterate over the list
for item in numbers:
    sum = sum+item
# print the sum
print("The sum is",sum)
```

Output 1

```
The sum is 30
```

Example Program 2

```
flowers = ['rose', 'lotus',  'jasmine']
for flower in flowers:
  print('Current flower :', flower)
```

Output 2

```
Current flower : rose
Current flower : lotus
Current flower : jasmine
```

Example Program 3

```
for letter in 'Program':
  print('Current letter :', letter)
```

Output 3

```
Current letter : p
Current letter : r
Current letter : o
Current letter : g
Current letter : r
Current letter : a
Current letter : m
```

range() function

We can use the `range()` function in `for` loops to iterate through a sequence of numbers. It can be combined with the `len()` function to iterate though a sequence using indexing. `len()` function is used to find the length of a string or number of elements in a list, tuple, set etc.

Example Program

```
flowers = ['rose', 'lotus',  'jasmine']
for i in range(len(flowers)):
print('Current flower :', flowers[i])
```

Output

```
Current flower : rose
Current flower : lotus
Current flower : jasmine
```

We can generate a sequence of numbers using `range()` function. `range(10)` will generate numbers from 0 to 9 (10 numbers). We can also define the `start`, `stop` and `step_size` as `range(start,stop,step_size)`. The default value of `step_size` is 1, if not provided. This function does not store all the values in memory. It keeps track of the the `start`, `stop`, `step_size` and generates the next number.

Example Program

```
for num in range(2, 10, 2):print("Number = ", num)
```

Output

```
Number = 2
Number = 4
Number = 6
Number = 8
```

5.2.2 *for* loop with *else*

Python supports to have an else statement associated with a loop statement. If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has finished iterating the list. A `break` statement can be used to stop a `for` loop. In this case, the `else` part is ignored. Hence, a `for` loop's `else` part runs if no break occurs. Break statement is discussed in the Section 5.4 Control Statements.

Example Program

```
#Program to find whether an item is present in the list
list=[10,12,13,34,27,98]
num = int(input("Enter the number to be searched in the list:
"))
for item in range(len(list)):
 if list[item] == num:
   print("Item found at: ",(item+1))
   break
else:
   print("Item not found in the list")
```

Output 1

```
Enter the number to be searched in the list:34
Item found at:4
```

Output 2

```
Enter the number to be searched in the list:99
Item not found in the list
```

5.2.3 *while* loop

The `while` loop in Python is used to iterate over a block of code as long as the test expression (condition) is true. We generally use this loop when we don't know the number of times to iterate in advance. Fig. 5.5 shows the flow chart of a `while` loop.

Syntax

```
while test_expression:
    Body of while
```

In `while` loop, test expression is checked first. The body of the loop is entered only if the `test_expression` evaluates to `True`. After one iteration, the test expression is checked again. This process is continued until the `test_expression` evaluates to `False`. In Python, the body of the `while` loop is determined through indentation. Body starts with indentation and the first unindented line shows the end. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

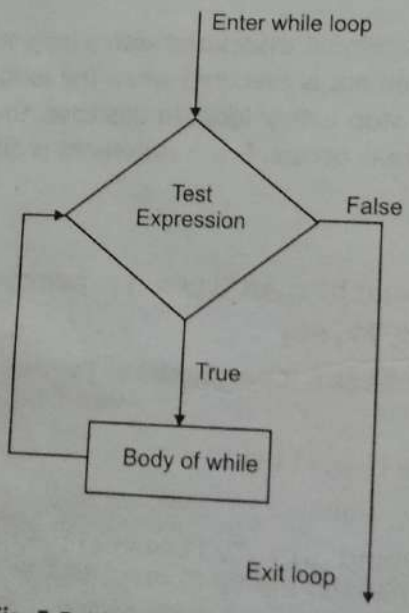

Fig. 5.5: Flow chart for while loop

Example Program

```
# Program to find the sum of first N natural numbers
n =int(input("Enter the limit: "))
sum=0
i=1
```

```
while i<=n:
    sum=sum+i
    i=i+1
print("Sum of first", n, "natural numbers is ", sum)
```

Output

```
Enter the limit: 5
Sum of first 5 natural numbers is 15
```

Illustration

In the above program, when the text `Enter the limit` appears, 5 is given as the input. ie. n=5. Initially, a counter i is set to 1 and sum is set to 0. When the condition is checked for the first time i<=n, it is True. Hence the execution enters the body of while loop. Sum is added with i and i is incremented. Since the next statement is unindented, it assumes the end of while block. This process is repeated when the condition given in the while loop is False and the control goes to the next print statement to print the sum.

5.2.4 *while* loop with *else* statement

If the `else` statement is used with a `while` loop, the `else` statement is executed when the condition becomes False. `while` loop can be terminated with a `break` statement. In such case, the `else` part is ignored. Hence, a `while` loop's `else` part runs if no break occurs and the condition is False.

Example Program 1

```
# Demo of While with else
count=1
while count<=3:
    print("Python Programming")
    count=count+1
else:
    print("Exit")
print("End of Program")
```

Output 1

```
Python Programming
Python Programming
Python Programming
Exit
End of Program
```

Illustration

In this example, initially the count is 1. The condition in the while loop is checked and since it is True, the body of the while loop is executed. The loop is terminated when the condition is False and it

goes to the else statement. After executing the else block it will move on to the rest of the Program statements. In this example the print statement after the else block is executed.

Example Program 2

```
# Demo of While with else
count=1
while count<=3:
    print("Python Programming")
    count=count+1
    if count==2:break
else:
    print("Exit")
print("End of Program")
```

Output 2

```
Python Programming
End of Program
```

Illustration

In this example, initially the count is 1. The condition in the `while` loop is checked and since it is True, the body of the `while` loop is executed. When the `if` condition inside the `while` loop is True, ie. when count becomes 2, the control is exited from the `while` loop. It will not execute the `else` part of the loop. The control will moves on to the next statement after the `else`. Hence the print statement `End of Program` is executed.

5.3 Nested Loops

Sometimes we need to place a loop inside another loop. This is called nested loop. We can have nested loops for both `while` and `for`.

Syntax for nested for loop

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

Syntax for nested while loop

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

Q: Write a program to print the prime numbers starting from 1 upto a limit entered by the user.

Program using **nested for**

```
#Program to generate prime numbers between 2 Limits
import math
n =int(input("Enter a Limit:"))
for i in range(1,n):
  k=int(math.sqrt(i))
  for j in range(2,k+1):
   if i%j==0:break
  else: print(i)
```

Output

```
Enter a Limit:12
1
2
3
5
7
11
```

The limit entered by the user is stored in n. To find whether a number is prime, the logic used is to divide that number from 2 to square root of that number (Since all numbers are completely divisible by 1, we started from 2). If the remainder of this division is zero at any time, that number is skipped and moved to next number. A complete division shows that the number is not prime. If the remainder is not zero at any time, it shows that it is a prime number.

The outer `for` loop starts from 1 to the input entered by the user. Initially i=1. A variable k is used to store the square root of the number. The inner `for` loop is used to find whether the number is completely divisible by any number between 2 and square root of that number(k). If it is completely divisible by any number between 2 and k, the number is not prime, else the number is considered prime. The same steps are repeated until the limit entered by the user is reached.

The above program is rewritten using nested while and is given below.

Program using **nested while**

```
#Program to generate prime numbers between 2 Limits
import math
n =int(input("Enter a Limit:"))
i=1
while i<=n:
  k=int(math.sqrt(i))
  j=2
  while j<=k:
```

```
        if i%j==0:break
        j+=1
    else: print(i)
    i+=1
```

Output

```
Enter a Limit:12
1
2
3
5
7
11
```

5.4 Control Statements

Control statements change the execution from normal sequence. Loops iterate over a block of code until test expression is False, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression. The `break` and `continue` statements are used in these cases. Python supports the following three control statements.

1. `break`
2. `continue`
3. `pass`

5.4.1 *break* statement

The `break` statement terminates the loop containing it. Control of the Program flows to the statement immediately after the body of the loop. If it is inside a nested loop (loop inside another loop), `break` will terminate the innermost loop. It can be used with both `for` and `while` loops. Fig. 5.6 shows the flow chart of `break` statement.

Example Program 1

```
#Demo of break statement in Python
for i in range(2,10,2):
 if i==6: break
 print(i)
print("End of Program")
```

Output 1

```
2
4
End of Program
```

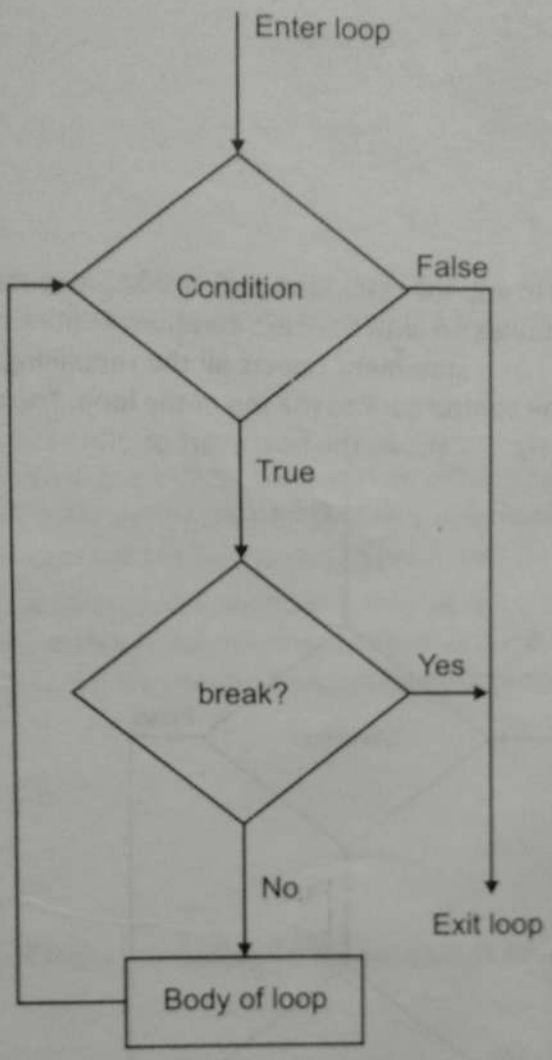

Fig 5.6: Flow chart of `break` statement

Illustration

In this the `for` loop is intended to print the even numbers from 2 to 10. The `if` condition checks whether i=6. If it is 6, the control goes to the next statement after the `for` loop. Hence it goes to the print statement `End of Program.`

Example Program 2

```
#Demo of break statement in Python
i=5
while i>0:
 if i==3: break
 print(i)
 i=i-1
print("End of Program")
```

```
Output 2
    5
    4
    End of Program
```

5.4.2 continue statement

The `continue` statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration. Continue returns the control to the beginning of the loop. The `continue` statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop. The **continue** statement can be used in both *while* and *for* loops. Fig. 5.7 shows the flow chart of `continue` statement.

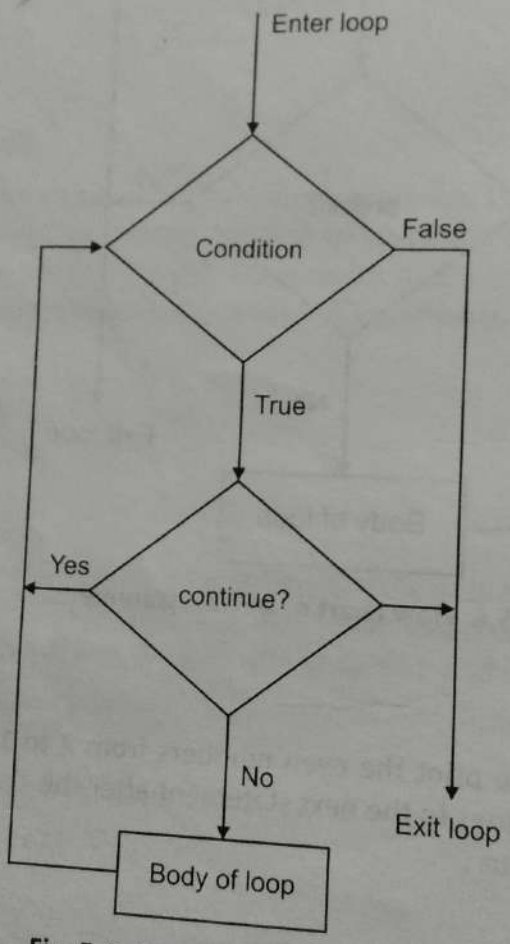

Fig. 5.7: Flow chart of `continue`

Example Program

```
# Demo of continue in Python
for letter in 'abcd':
 if letter =='c': continue
 print(letter)
```

Output

```
a
b
d
```

Illustration

When the letter=c, the continue statement is executed and the control goes to the beginning of the loop, bypasses the rest of the statements in the body of the loop. In the output the letters from "a" to "d" except "c" gets printed.

5.4.3 *pass* statement

In Python Programming, `pass` is a null statement. The difference between a comment and `pass` statement in Python is that, while the interpreter ignores a comment entirely, `pass` is not ignored. But nothing happens when it is executed. It results in no operation.

It is used as a placeholder. Suppose we have a loop or a function that is not implemented yet, but we want to implement it in the future. The function or loop cannot have an empty body. The interpreter will not allow this. So, we use the `pass` statement to construct a body that does nothing.

Example

```
for val in sequence:
    pass
```

5.5 Types of Loops

There are different types of loops depending on the position at which condition is checked.

5.5.1 Infinite Loop

A loop becomes infinite loop if a condition never becomes `False`. This results in a loop that never ends. Such a loop is called an infinite loop. We can create an infinite loop using `while` statement. If the condition of `while` loop is always `True`, we get an infinite loop. We must use `while` loops with caution because of the possibility that the condition may never resolves to a `False` value.

Example Program

```
count=1
while count==1:
  n=input("Enter a Number:")
  print("Number=", n)
```

This will continue running unless you give CTRL+C to exit from the loop.

5.5.2 Loops with condition at the top

This is a normal `while` loop without `break` statements. The condition of the `while` loop is at the top and the loop terminates when this condition is `False`. This loop is already explained in Section 5.2.3

5.5.3 Loop with condition in the middle

This kind of loop can be implemented using an infinite loop along with a conditional break in between the body of the loop. Fig. 5.8 shows the flow chart of a loop with condition in the middle.

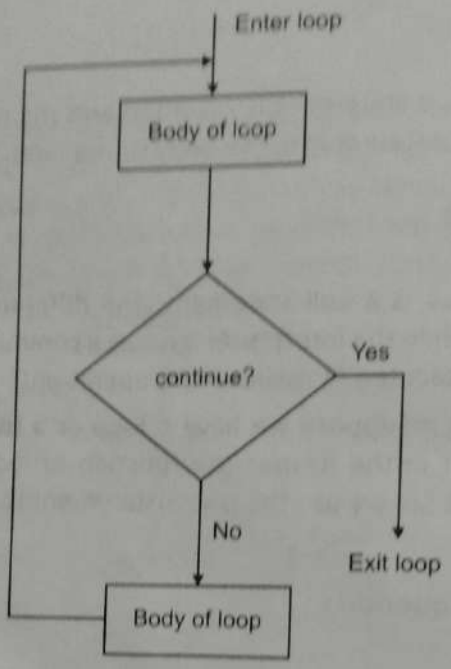

Fig. 5.8: Loop with condition in the middle

Example Program

```
vowels='aeiou'
# Infinite Loop
while True:
    v = input ("Enter a letter:")
    if v in vowels:
        print(v, "is a vowel")
        break
    print("This is not a vowel, Enter another letter")
print("End of Program")
```

Output

```
Enter a letter:t
This is not a vowel, Enter another letter
Enter a letter:o
o is a vowel
End of Program
```

5.5.4 Loop with condition at the bottom

This kind of loop ensures that the body of the loop is executed at least once. It can be implemented using an infinite loop along with a conditional break at the end. This is similar to the `do...while` loop in C. Fig. 5.9 shows the flow chart of loop with condition at the bottom.

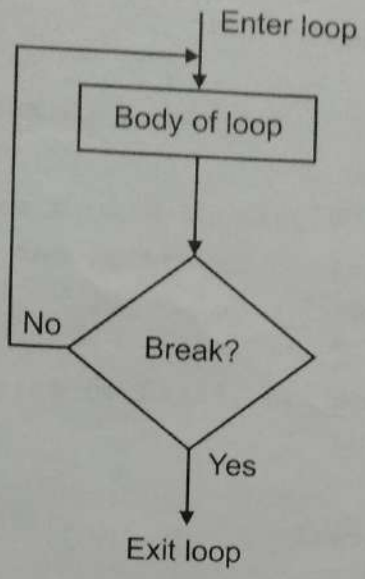

Fig 5.9: Loop with condition at the bottom

Example Program

```
# Demo of loop with condition at the bottom
choice=0
a,b=6,3
while choice !=5:
 print("Menu")
 print("1. Addition")
 print("2. Subtraction")
 print("3. Multiplication")
 print("4. Division")
 print("5. Exit")
 choice =int(input("Enter your choice:"))
 if choice==1: print("Sum=",(a+b))
 if choice==2: print("Difference=", (a-b))
 if choice==3: print("Product=",(a*b))
 if choice==4: print("Quotient=", (a/b))
 if choice==5: break
print("End of Program")
```

06 Functions

CONTENTS

A group of related statements to perform a specific task is known as a function. Functions help to break the program into smaller units. Functions avoid repetition and enhance code reusability. Functions provide better modularity in programming. Python provides two types of functions.

a) Built-in functions
b) User-defined functions

Functions like input(), print() etc. are examples of built-in functions. The code of these functions will be already defined. We can create our own functions to perform a particular task. These functions are called user-defined functions.

6.1 Function Definition

The following specifies simple rules for defining a function.

- Function block or Function header begins with the keyword def followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. We can also define parameters inside these parentheses.

- The first string after the function header is called the `docstring` and is short for documentation string. It is used to explain in brief, what a function does. Although optional, documentation is a good programming practice.
- The code block within every function starts with a colon (:) and is indented.
- The `return` statement [expression] exits a function, optionally passing back an expression to the caller. A `return` statement with no arguments is the same as `return None`.

Syntax

```
def functionname( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

Example Program

```
def sum_of_two_numbers(a,b): #Function Header with 2 parameters
a and b
 "This Function is to find the sum of two numbers" #Docstring
 sum=a+b
 return sum
```

6.2 Function Calling

After defining a function, we can call the function from another function or directly from the Python prompt. The order of the parameters specified in the function definition should be preserved in function call also.

Example Program

```
#Main Program Code
a = int(input("Enter the first number:"))
b = int(input("Enter the second number:"))
s = sum_of_two_numbers(a,b)#Function calling
print("Sum of",a,"and",b,"is", s)
```

Output

```
Enter the first number:4
Enter the second number:3
Sum of 4 and 3 is 7
```

All the parameters in Python are passed by reference. It means if we change a parameter which refers to within a function, the change also reflects back in the calling function. Parameters and variables defined inside a function is not visible from outside. Hence, they have a local scope. Lifetime of a variable is the period throughout which the variable exists in the memory. The lifetime of variables inside a function is as long as the function executes. All variables in a program may not be accessible at all locations in that program. This depends on where we have declared a variable. The scope of a

variable determines the portion of the program where we can access a particular identifier. They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

Example Program

```
def value_change(a): #Function Header
 a=10
 print("Value inside function=",a) #prints the value of a inside
function
 return
#Main Program Code
a =int(input("Enter a number:"))
value_change(a)#Function calling
print("Value outside function=",a) #prints value of a outside
function
```

Output

```
Enter a number:3
Value inside function = 10
Value outside function = 3
```

Here in the main program a number is read and stored in variable `a`. Even though this value is passed to the function `value_change`, `a` is assigned another value inside the function. This value is displayed inside the function. After returning from the function, it will display the value read from the main function. The variable `a` declared inside function `value_change` is local to that function and hence after returning to the main program, `a` will display the value stored in the main program.

6.3 Function Arguments

We can call the function by any of the following 4 arguments.

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

6.3.1 Required Arguments

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

Example Program

```
def value_change(a): #Function Header
 a=10
```

```
    print("Value inside function=",a) #prints the value of a inside
    function
    return
#Main Program Code
a =int(input("Enter a number:"))
value_change()#Function calling without passing parameter
print("Value outside function=",a) #prints value of a outside
function
```

Output

```
Enter a number:4
Traceback (most recent call last):
  File "main.py", line 7, in <module>
    value_change()#Function calling without passing parameter
TypeError: value_change() takes exactly 1 argument (0 given)
```

The above example contains a function which requires 1 parameter. In the main program code, the function is called without passing the parameter. Hence it resulted in an error.

6.3.2 Keyword Arguments

When we use keyword arguments in a function call, the caller identifies the arguments by the parameter name. This allows us to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

Example Program

```
# Function definition
def studentinfo(rollno,name,course):
    print("Roll Number:",rollno)
    print("Name:",name)
    print("Course:",course)
#Function calling
#order of parameters in function call is shuffled
studentinfo(course="UG",rollno=50,name="Jack")
```

Output

```
Roll Number: 50
Name: Jack
Course: UG
```

6.3.3 Default Arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example shows how default arguments are invoked.

Example Program

```
# Function definition
def studentinfo(rollno,name,course="UG"):
   "This prints a passed info into this function"
   print("Roll Number:",rollno)
   print("Name:",name)
   print("Course:",course)
#Function calling
#Order of parameters is shuffled
studentinfo(course="UG", rollno=50, name="Jack")
#The parameter course is omitted
studentinfo(rollno=51,name="Tom")
```

Output

```
Roll Number: 50
Name: Jack
Course: UG
Roll Number: 51
Name: Tom
Course: UG
```

In the above example, the first function call to `studentinfo` passes the three parameters. In the case of second function call to `studentinfo`, the parameter `course` is omitted. Hence it takes the default value "UG" given to `course` in the function definition.

6.3.4 Variable-Length Arguments

In some cases we may need to process a function for more arguments than specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments. An asterisk (*) is placed before the variable name that holds the values of all non-keyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. The following shows the syntax of a function definition with *variable-length* arguments.

Syntax

```
def functionname([formal_arguments,] *variable_arguments_tuple
):
   "function_docstring"
   function_statements
   return [expression]
```

Example Program

```
# Function definition
def variablelengthfunction(*argument):
```

```
    print("Result:",)
    for i in argument: print(i)
#Function call with 1 argument
variablelengthfunction(10)
#Function call with 4 arguments
variablelengthfunction(10,30,50,80)
```

Output

```
Result: 10 #Result of function call with 1 argument
Result: 10 #Result of function call with 4 arguments
30
50
80
```

6.4 Anonymous Functions (Lambda Functions)

In Python, anonymous function is a function that is defined without a name. While normal functions are defined using the def keyword, in Python anonymous functions are defined using the lambda keyword. Hence, anonymous functions are also called lambda functions. The following are the characteristics of lambda functions.

1. Lambda functions can take any number of arguments but return only one value in the form of an expression.
2. It cannot contain multiple expressions.
3. It cannot have comments.
4. A lambda function cannot be a direct call to print because lambda requires an expression.
5. Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
6. Lambda functions are not equivalent to inline functions in C or C++.

Syntax

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

Example Program

```
# Lambda Function definition
square=lambda x : x*x
# Usage of lambda function
n=int(input("Enter a number:"))
print("Square of", n,"is", square(n))#Lambda function call
```

Output

```
Enter a number:5
Square of 5 is 25
```

In the above example, lambda is the keyword, x shows the argument passed, and x*x is the expression to be evaluated and stored in the variable square. In the case of calculating the square of a number, only one argument is passed. More than one argument is possible for lambda functions. The following example shows a lambda function which takes two arguments and returns the sum.

Example Program

```
# Lambda Function definition
sum=lambda x,y : x+y
# Usage of lambda function
m=int(input("Enter first number:"))
n=int(input("Enter second number:"))
print("Sum of", m, "and", n,"is", sum(m,n))#Lambda function call
```

Output

```
Enter first number:10
Enter second number:20
Sum of 10 and 20 is 30
```

6.4.1 Uses of lambda function

We use lambda functions when we require a nameless function for a short period of time. In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like filter(), map() etc.

Example Program with filter()

The filter() function in Python takes in a function and a list as arguments. The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.

```
# Lambda Function to filter out only the odd numbers from a list
oldlist=[2,31,42,11,6,5,23,44]
# Usage of lambda function
newlist=list(filter(lambda x: (x%2!=0),oldlist))
print(oldlist)
print(newlist)
```

Output

```
[2, 31, 42, 11, 6, 5, 23, 44]
[31, 11, 5, 23]
```

Example Program with map()

The map() function in Python takes in a function and a list. The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

```
# Lambda Function to increment the items in list by 2
oldlist=[2,31,42,11,6,5,23,44]
```

```
print(oldlist)
# Usage of lambda function
newlist=list(map(lambda x: x+2,oldlist))
print("List after incrementation by 2")
print(newlist)
```

Output

```
[2, 31, 42, 11, 6, 5, 23, 44]
List after incrementation by 2
[4, 33, 44, 13, 8, 7, 25, 46]
```

6.5 Recursive Functions

Recursion is the process of defining something in terms of itself. A function can call other functions. It is possible for a function to call itself. This is known as recursion. The following example shows a recursive function to find the factorial of a number.

Example Program

```
def recursion_fact(x):
 #This is a recursive function to find the factorial of an
integer
  if x == 1: return 1
  else: return (x * recursion_fact(x-1))#Recursive calling
num = int(input("Enter a number: "))
if num >= 1:
  print("The factorial of", num, "is", recursion_fact(num))
```

Output

```
Enter a number: 5
The factorial of 5 is 120
```

Explanation

In the above example, recursion_fact() is a recursive function as it calls itself. When we call this function with a positive integer, it will recursively call itself by decreasing the number. Each function call multiplies the number with the factorial of number-1 until the number is equal to one. This recursive call can be explained in the following steps. Our recursion ends when the number reduces to 1. This is called the base condition. Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely. We must avoid infinite recursion.

```
recursion_fact(5)
5 * recursion_fact(4)              # 1st call with 5
5 * 4 * recursion_fact(3)          # 2nd call with 4
5 * 4 * 3 *  recursion_fact(2)     # 3rd call with 3
                                   # 4th call with 2
```

```
5 * 4 * 3 * 2 * recursion_fact(1)   # 5th call with 1
5 * 4 * 3 * 2 * 1                   # return from 5th call as
number=1
5 *4 * 3 * 2                       # return from 4th call
5 * 4 * 6                           # return from 3rd call
5*24                               # return from 2nd call
120                                 #return from 1st call
```

6.6 Function with more than one return value

Python has a strong mechanism of returning more than one value at a time. This is very flexible when the function needs to return more than one value. Instead of writing separate functions for returning individual values, we can return all the values within same function. The following shows an example for function returning more than one value.

Example Program

```
#Python Function Returning more than One value
def calc(a,b):
    sum=a+b
    diff=a-b
    prod=a*b
    quotient=a/b
    return sum, diff, prod, quotient
a=int(input("Enter first number:"))
b=int(input("Enter second number:"))
s,d,p,q=calc(a,b)
print("Sum=",s)
print("Difference=",d)
print("Product=",p)
print("Quotient=",q)
```

Output

```
Enter first number:10
Enter second number:5
Sum= 15
Difference= 5
Product= 50
Quotient= 2
```

6.7 Solved Lab Exercises

1. Write a Python function to check whether a number is even or odd.

07 Modules and Packages

CONTENTS

Modules refer to a file containing Python statements and definitions. A module is a Python object with arbitrarily named attributes that we can bind and reference. We use modules to break down large programs into small manageable and organized files. Further, modules provide reusability of code. A module can define functions, classes and variables. A module can also include runnable code. Python has a lot of standard modules (built-in modules) available. A full list of Python standard modules is available in the Lib directory inside the location where you installed Python.

7.1 Built-in Modules

We have already seen the built-in module `math`. The following Table 7.1 shows the list of various built-in modules and their description.

Scanned with OKEN Scanner

Table 7.1: Built-in modules and their description

a	
abc	Abstract base classes according to PEP 3119.
aifc	Read and write audio files in AIFF or AIFC format.
argparse	Command-line option and argument parsing library.
array	Space efficient arrays of uniformly typed numeric values.
ast	Abstract Syntax Tree classes and manipulation.
asynchat	Support for asynchronous command/response protocols.
asyncio	Asynchronous I/O, event loop, coroutines and tasks.
asyncore	A base class for developing asynchronous socket handling services.
atexit	Register and execute cleanup functions.
audioop	Manipulate raw audio data.
b	
base64	RFC 3548: Base16, Base32, Base64 Data Encodings; Base85 and Ascii85
bdb	Debugger framework.
binascii	Tools for converting between binary and various ASCII-encoded binary representations.
binhex	Encode and decode files in binhex4 format.
bisect	Array bisection algorithms for binary searching.
builtins	The module that provides the built-in namespace.
bz2	Interfaces for bzip2 compression and decompression.
c	
calendar	Functions for working with calendars, including some emulation of the Unix cal Program.
cgi	Helpers for running Python scripts via the Common Gateway Interface.
cgitb	Configurable traceback handler for CGI scripts.
chunk	Module to read IFF chunks.
cmath	Mathematical functions for complex numbers.
cmd	Build line-oriented command interpreters.
code	Facilities to implement read-eval-print loops.
codecs	Encode and decode data and streams.
codeop	Compile (possibly incomplete) Python code.
collections	Container datatypes
colorsys	Conversion functions between RGB and other color systems.
compileall	Tools for byte-compiling all Python source files in a directory tree.
concurrent	

Scanned with OKEN Scanner

configparser	Configuration file parser.
contextlib	Utilities for with-statement contexts.
copy	Shallow and deep copy operations.
copyreg	Register pickle support functions.
cProfile	
crypt (Unix)	The crypt() function used to check Unix passwords.
csv	Write and read tabular data to and from delimited files.
ctypes	A foreign function library for Python.
curses (Unix)	An interface to the curses library, providing portable terminal handling.
d	
datetime	Basic date and time types.
dbm	Interfaces to various Unix "database" formats.
decimal	Implementation of the General Decimal Arithmetic Specification.
difflib	Helpers for computing differences between objects.
dis	Disassembler for Python bytecode.
distutils	Support for building and installing Python modules into an existing Python installation.
doctest	Test pieces of code within docstrings.
dummy_threading	Drop-in replacement for the threading module.
e	
email	Package supporting the parsing, manipulating, and generating email messages, including MIME documents.
encodings	
ensurepip	Bootstrapping the "pip" installer into an existing Python installation or virtual environment.
enum	Implementation of an enumeration class.
errno	Standard errno system symbols.
f	
faulthandler	Dump the Python traceback.
fcntl (Unix)	The fcntl() and ioctl() system calls.
filecmp	Compare files efficiently.
fileinput	Loop over standard input or a list of files.
fnmatch	Unix shell style filename pattern matching.
formatter	Deprecated: Generic output formatter and device interface.
fpectl (Unix)	Provide control for floating point exception handling.
fractions	Rational numbers.

Scanned with OKEN Scanner

u	
unicodedata	Access the Unicode Database.
unittest	Unit testing framework for Python.
urllib	
uu	Encode and decode files in uuencode format.
uuid	UUID objects (universally unique identifiers) according to RFC 4122
v	
venv	Creation of virtual environments.
w	
warnings	Issue warning messages and control their disposition.
wave	Provide an interface to the WAV sound format.
weakref	Support for weak references and weak dictionaries.
webbrowser	Easy-to-use controller for Web browsers.
winreg (Windows)	Routines and objects for manipulating the Windows registry.
winsound (Windows)	Access to the sound-playing machinery for Windows.
wsgiref	WSGI Utilities and Reference Implementation.
x	
xdrlib	Encoders and decoders for the External Data Representation (XDR).
xml	Package containing XML processing modules
xmlrpc	
z	
zipapp	Manage executable python zip archives
zipfile	Read and write ZIP-format archive files.
zipimport	support for importing Python modules from ZIP archives.
zlib	Low-level interface to compression and decompression routines compatible with gzip.

7.2 Creating Modules

We can define our most used functions in a module and import it, instead of copying their definitions into different programs. A file containing Python code, for e.g.: `prime.py`, is called a module and its module name would be `prime`. The Python code for a module named `test` normally resides in a file named `test.py`.

Let us create a module for finding the sum of two numbers. The following code creates a module in Python. Type the code and save it as test.py.

Example

```
# Python Module example
```

Scanned with OKEN Scanner

```
"""This Program adds two
   numbers and return the result"""
def sum(a, b):
   result = a + b
   return result
```

Here we have a function `sum()` inside a module named `test`. The function takes in two numbers and returns their sum.

7.3 *import* Statement

We can use any Python source file as a module by executing an import statement in some other Python source file. User-defined modules can be imported the same way as we import built-in modules.

We use the `import` keyword to do this. The import has the following syntax.

```
import module1[, module2[,... moduleN]
```

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module `test.py`, you need to put the following command at the top of the script.

Example

```
import test
print(test.sum(2,3))
```

Output

```
5
```

When the above code in the example is executed, we get the result 5.

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.

7.3.1 *import* with renaming

We can import built-in or user-defined modules with an alias name. The following code shows how the built-in module math can be imported using an alias name.

Example Program

```
import math as m
print("The value of pi is", m.pi)
```

Output

```
The value of pi is 3.141592653589793
```

We have renamed the `math` module as `m`. This can save typing time in some cases. It is important to note that the name `math` is not recognized in our scope. Hence, `math.pi` is invalid, `m.pi` is the correct implementation.

Scanned with OKEN Scanner

7.3.2 *from...import* statement

We can import specific names from a module without importing the module as a whole. The module math contains a lot of built-in functions. But if we want to import only the `pi` function, the `from... import` statement can be used. The following example illustrates importing `pi` from `math` module.

Example Program

```
from math import pi
print("The value of pi is :", pi)
```

Output

```
The value of pi is : 3.14159265359
```

We have imported only the `pi` function from the `math` module. Hence no need to use the dot operator as given in example 7.3.1. We can also import multiple attributes from the module using `from...import` statement. The following example illustrates the import of `pi` as well as `sqrt()` from the `math` module using `from...import` statement.

Example Program

```
from math import pi,sqrt
print("The value of pi is :", pi)
print("The square root of 4 is:",sqrt(4))
```

Output

```
The value of pi is: 3.14159265359
The square root of 4 is: 2.0
```

7.3.3 *import* all names

We can import all names(definitions) from a module using the following construct. The following example shows how all the definitions from the `math` module can be imported. This makes all names except those beginning with an underscore, visible in our scope.

Example Program

```
from math import *
print("The value of pi is :", pi )
print("The square root of 4 is:",sqrt(4))
```

Output

```
The value of pi is: 3.14159265359
The square root of 4 is: 2.0
```

Here all functions in the module `math` are imported. But we have used only `pi` and `sqrt`. Importing everything with the asterisk (*) symbol is not a good programming practice. This can lead to duplicate definitions for an identifier. It also reduces the readability of our code.

7.4 Locating Modules

When we import a module, the Python interpreter searches for the module in the following sequence.

Scanned with OKEN Scanner

1. The current directory.
2. If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
3. If the above two mentioned fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

The module search path is stored in the system module `sys` as the `sys.path` variable. The `sys.path` variable contains the current directory, PYTHONPATH and the installation-dependent default.

7.4.1 PYTHONPATH variable

The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH. The following shows a typical PYTHONPATH from a Windows operating system.

```
set PYTHONPATH=c:\python34\lib
```

And here is a typical PYTHONPATH from a UNIX system.

```
set PYTHONPATH=/usr/local/lib/python
```

7.5 Namespaces and Scope

Variables are names (identifiers) that map to objects. A namespace is a dictionary of variable names (keys) and their corresponding objects (values). A Python statement can access variables in a local namespace and in the global namespace. If a local and a global variable have the same name, the local variable shadows the global variable.

Each function has its own local namespace. Python makes certain guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local. Therefore, in order to assign a value to a global variable within a function, you must first use the global statement. The statement `global variable_name` tells Python that `variable_name` is a global variable. Once a variable is declared global, Python stops searching the local namespace for that variable. Consider the following example.

Example Program

```
#Program to illustrate local and global namespace
a=10
def Add():
 a=a+1
 print(a)
Add()
print(a)
```

Output

```
Traceback (most recent call last):
  File "main.py", line 5, in <module>
    Add()
```

Scanned with OKEN Scanner

```
  File "main.py", line 3, in Add
    a=a+1
UnboundLocalError: local variable 'a' referenced before
assignment
```

When we run the above program, the output is listed above with errors. The reason is that we are trying to add 1 to a before a is assigned a value. This is because even though a is assigned a value 10, its scope is outside the function Add(). Further Python always searches for variables in the local namespace. Since a is not assigned a value inside the function, it resulted in an error. The below program is re-written using global variable concept.

Example Program

```
#Program to illustrate local and global namespace
a=10
print a
def Add():
  global a
  a=20
  a=a+1
  print(a)
Add()
print(a)
```

Output

```
10
21
21
```

In the above example, a is assigned a value 10 and it is printed. Inside the function, a is declared global and it is incremented by one. The result is 21. Even after, exiting the function, when we try to print the value of a, it is printing it as 21 since a is declared a global variable.

7.6 The *dir()* function

The dir() built-in function returns a sorted list of strings containing the names defined by a module. The list contains the names of all the modules, variables and functions that are defined in a module. The following example illustrates the use of dir() function for the built-in module called math.

Example Program

```
import math
c=dir(math)
print(c)
```

Scanned with OKEN Scanner

Output

```
['__doc__', '__file__', '__name__', '__package__', 'acos',
'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
'copysign',, 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc',
'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
'fsum', 'gamma', 'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma',
'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin',
'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

In the above output, _doc_is present in every namespace. Initially it will be None, we can store documentation here.__file__ is the filename from which the module was loaded. __name__ is the name of the current module (minus the .py extension). In the top-level script or in conversational mode, this name is set to the string '__main__'. The module's __package__ attribute should be set. Its value must be a string, but it can be the same value as its __name__. If the attribute is set to None or is missing, the import system will fill it in with a more appropriate value. When the module is a package, its __package__ value should be set to its __name__. When the module is not a package, __package__ should be set to the empty string for top-level modules, or for submodules, to the parent package's name. The rest shows the built-in functions available in the math module.

Another example illustrates the use of dir() function along with user-defined modules.

Example Program

```
import test
c=dir(math)
print(c)
```

Output

```
['__builtins__', '__doc__', '__file__', '__name__', '__
package__', 'sum']
```

The output contains built-in strings and we have defined only one function called sum in that module. _builtins_ contain Python's built-in error handling functions which will be discussed in Chapter 10.

7.7 The *reload()* function

When the module is imported into a script, the code in the top-level portion of a module is executed only once. Therefore, if you want to reexecute the top-level code in a module, you can use the reload() function. The reload() function imports a previously imported module again. The syntax of the reload() function is as follows.

```
reload(module_name)
```

Here, module_name is the name of the module we want to reload and not the string containing the module name. For example, to reload `test` module, do the following

```
reload(test)
```

The following example illustrate the use of *reload()* function in Python.

Scanned with OKEN Scanner

7.8 Packages in Python

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and so on. As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages. This makes a project (program) easy to manage and conceptually clear. Similar, as a directory can contain sub-directories and files, a Python package can have sub-packages and modules. A directory must contain a file named `__init__.py` in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file.

7.8.1 Importing modules from a Package

Consider a file `Asample.py` available in `pack` directory. This file has following line of source code.

```
def a:
 print("This is from A")
```

Similarly we have two more files `Bsample.py` and `Csample.py` available in `pack` directory with the following codes.

```
Bsample.py
 def b:
  print("This is from B")
 Csample.py
 def c:
  print("This is from C")
```

Now we have three files in the same directory `pack`. To make all of our functions available when we are importing `pack`, we need to put explicit import statements in __init__.py as follows.

```
from Asample import a
from Bsample import b
from Csample import c
```

After adding these lines to __init__.py, we have all of these functions available when we import the `pack` package. The following code shows how the package `pack` can be imported.

```
import pack
pack.a()
pack.b()
pack.c()
```

When the above code is executed, it gives the following result.

```
This is from A
This is from B
This is from C
```

In the above example, we have defined only one function in each module. It is also possible to have multiple definitions (more than one function) in each module and all these functions can be imported

Scanned with OKEN Scanner

using the wildcard character '*'. Likewise if we have subfolders each subfolder can be separated by using dot operator while importing the package. For example, if the package `pack` resides in `/user/abc/level1`, we can give the `import` statement as follows.

```
import user.abc.level1.pack
```

7.9 Date and Time Modules

Conversion between various date and time formats is a common task to be done in almost all applications. Python provides various modules like time, calendar, datetime for parsing formatting and performing calculations on date and time.

7.9.1 The `time` Module

The time module contains a lot of functions to process time. Time intervals are represented as floating-point numbers in seconds. The time since January 1, 1970, 12:00 a.m is available and is termed as an epoch. The following example shows how many seconds have elapsed since the epoch.

Example Program

```
import time
seconds = time.time()
print("Number of seconds since 12:00am, January 1, 1970:",
seconds)
```

Output

```
Number of seconds since 12:00am, January 1, 1970:
1448080451.357909
```

The float representation is useful when storing or comparing dates, but not as useful for producing human readable representations. For printing the current time `ctime()` may be more useful. The following shows an example for `ctime()` method.

Example Program

```
import time
print("The time is:", time.ctime())
later = time.time() + 30
print("30 secs from now:", time.ctime(later))
```

Output

```
The time is: Sat Nov 21 10:10:08 2015
30 secs from now: Sat Nov 21 10:10:38 2015
```

7.9.1.1 struct_time Structure

Storing time as elapsed seconds is useful in some situations, but there are times when you need to have access to the individual fields of a date (year, month, etc.). The time module defines `struct_time` for holding date and time values with components broken out so they are easy to access. There are several functions that work with `struct_time` values instead of float. The `struct_time` has the following attributes listed in Table 7.2.

Scanned with OKEN Scanner

Table 7.2: Attributes of struct_time

Attribute	Values
tm_year	Current year
tm_mon	1 to 12
tm_mday	1 to 31
tm_hour	0 to 23
tm_min	0 to 59
tm_sec	0 to 61(60 and 61 are leap seconds)
tm_yday	0 to 6 (0 is Monday)
tm_wday	1 to 366 (Julian day)
tm_isdst	-1,0,1 -1 if library determines DST

Daylight saving time (DST) or summer time is the practice of advancing clocks during summer months by one hour so that in the evening, daylight is experienced an hour longer, while sacrificing normal sunrise times.

Example Program

```
import time
print("gmtime:", time.gmtime())
print("localtime:", time.localtime())
print("mktime:", time.mktime(time.localtime()))
t = time.localtime()
print("Current Year:",t.tm_year)
print("Current Month:",t.tm_mon)
print("Current Day:",t.tm_mday)
print("Day of month:", t.tm_mday)
print("Day of week:", t.tm_wday)
print("Day of year:", t.tm_yday)
print("DST:",t.tm_isdst)
```

Output

```
gmtime:time.struct_time(tm_year=2015, tm_mon=11, tm_mday=21,
tm_hour=5, tm_min=19, tm_sec=44, tm_wday=5, tm_yday=325, tm_
isdst=0)
localtime: time.struct_time(tm_year=2015, tm_mon=11, tm_mday=21,
tm_hour=10, tm_min=49, tm_sec=44, tm_wday=5, tm_yday=325, tm_
isdst=0)
mktime: 1448083184.0
Current Year: 2015
Current Month: 11
```

Scanned with OKEN Scanner

```
Current Day: 21
Day of month: 21
Day of week: 5
Day of year: 325
DST: 0
```

`gmtime()` returns the current time in Universal Time Clock. `localtime()` returns the current time with the current time zone applied. `mktime()` takes a `struct_time` and converts it to the floating point representation.

7.9.1.2 Parsing and Formatting Time

The two functions strptime() and strftime() convert between struct_time and string representations of time values. There is a long list of formatting instructions available to support input and output in different styles. The following example converts the current time from a string, to a struct_time instance, and back to a string.

Example Program

```
import time
now = time.ctime()
print(now)
parsed = time.strptime(now)
print(parsed)
print(time.strftime("%a %b %d %H:%M:%S %Y", parsed))
```

Output

```
Sat Nov 21 11:16:24 2015
time.struct_time(tm_year=2015, tm_mon=11, tm_mday=21, tm_
hour=11, tm_min=16, tm_sec=24, tm_wday=5, tm_yday=325, tm_
isdst=-1)
Sat Nov 21 11:16:24 2015
```

7.9.2 The `calendar` Module

The calendar module has many built-in functions for printing and formatting the output. By default, calendar takes Monday as the first day of the week and Sunday as the last one. To change this, call calendar.setfirstweekday() function. The following example shows a formatted text calendar.

Example Program

```
import calendar
c = calendar.TextCalendar(calendar.SUNDAY)
c.prmonth(2015,10)
```

Scanned with OKEN Scanner

Output

```
      October 2015
Su Mo Tu We Th Fr Sa
                 1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

If we want to print the entire calendar for a year, the following code needs to be executed.

Example Program

```
import calendar
print(calendar.TextCalendar(calendar.SUNDAY).formatyear(2015))
```

Output

```
                                  2015
      January                   February                   March
Su Mo Tu We Th Fr Sa      Su Mo Tu We Th Fr Sa      Su Mo Tu We Th Fr Sa
                 1  2  3       1  2  3  4  5  6  7       1  2  3  4  5  6  7
 4  5  6  7  8  9 10       8  9 10 11 12 13 14       8  9 10 11 12 13 14
11 12 13 14 15 16 17      15 16 17 18 19 20 21      15 16 17 18 19 20 21
18 19 20 21 22 23 24      22 23 24 25 26 27 28      22 23 24 25 26 27 28
25 26 27 28 29 30 31                                29 30 31

      April                     May                       June
Su Mo Tu We Th Fr Sa      Su Mo Tu We Th Fr Sa      Su Mo Tu We Th Fr Sa
                 1  2  3  4                 1  2          1  2  3  4  5  6
 5  6  7  8  9 10 11       3  4  5  6  7  8  9       7  8  9 10 11 12 13
12 13 14 15 16 17 18      10 11 12 13 14 15 16      14 15 16 17 18 19 20
19 20 21 22 23 24 25      17 18 19 20 21 22 23      21 22 23 24 25 26 27
26 27 28 29 30            24 25 26 27 28 29 30      28 29 30
                          31

      July                      August                    September
Su Mo Tu We Th Fr Sa      Su Mo Tu We Th Fr Sa      Su Mo Tu We Th Fr Sa
                 1  2  3  4                    1                 1  2  3  4  5
 5  6  7  8  9 10 11       2  3  4  5  6  7  8       6  7  8  9 10 11 12
12 13 14 15 16 17 18       9 10 11 12 13 14 15      13 14 15 16 17 18 19
19 20 21 22 23 24 25      16 17 18 19 20 21 22      20 21 22 23 24 25 26
26 27 28 29 30 31         23 24 25 26 27 28 29      27 28 29 30
                          30 31
```

Scanned with OKEN Scanner

```
      October
Su Mo Tu We Th Fr Sa
                1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

```
     November
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30
```

```
     December
Su Mo Tu We Th Fr Sa
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

7.9.2.1 Built-in functions

1. calendar.firstweekday( ) - Returns the current setting for the weekday that starts each week. By default, when calendar is first imported, first week day is 0, i. e Monday.
2. calendar.setfirstweekday(weekday) - Sets the first day of each week to weekday with the parameter passed. Weekday codes are 0 (Monday) to 6 (Sunday).
3. calendar.isleap(year)- returns True if year is a leap year, False otherwise.
4. calendar.leapdays(y1,y2)- Returns the total number of leap days in the years within range(y1,y2).

   Example Program

```
import calendar
print("Default first weekday:",calendar.firstweekday())
calendar.setfirstweekday(calendar.SUNDAY)
print("New first weekday:",calendar.firstweekday())
print("Leap Year:",calendar.isleap(2014))
print("Leap days between 2010 and 2015:",calendar.
leapdays(2010,2015))
```

   Output

```
Default first weekday: 0
New first weekday: 6
Leap Year: False
Leap days between 2010 and 2015: 1
```

5. calendar.calendar(year,w=2,l=1,c=6) - Returns a multiline string with a calendar for year year formatted into three columns separated by c spaces. w is the width in characters of each date. l is the number of lines for each week. This is equivalent to the example given in Section 7.9.2

   Example Program

```
import calendar
print(calendar.calendar(2015,2,1,6))
```

Scanned with OKEN Scanner

Output

```
                                  2015

      January                   February                   March
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                1  2  3  4                   1                         1
 5  6  7  8  9 10 11       2  3  4  5  6  7  8       2  3  4  5  6  7  8
12 13 14 15 16 17 18       9 10 11 12 13 14 15       9 10 11 12 13 14 15
19 20 21 22 23 24 25      16 17 18 19 20 21 22      16 17 18 19 20 21 22
26 27 28 29 30 31         23 24 25 26 27 28         23 24 25 26 27 28 29
                                                    30 31

      April                     May                       June
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
          1  2  3  4  5                1  2  3       1  2  3  4  5  6  7
 6  7  8  9 10 11 12       4  5  6  7  8  9 10       8  9 10 11 12 13 14
13 14 15 16 17 18 19      11 12 13 14 15 16 17      15 16 17 18 19 20 21
20 21 22 23 24 25 26      18 19 20 21 22 23 24      22 23 24 25 26 27 28
27 28 29 30               25 26 27 28 29 30 31      29 30

      July                    August                   September
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
          1  2  3  4  5                   1  2          1  2  3  4  5  6
 6  7  8  9 10 11 12       3  4  5  6  7  8  9       7  8  9 10 11 12 13
13 14 15 16 17 18 19      10 11 12 13 14 15 16      14 15 16 17 18 19 20
20 21 22 23 24 25 26      17 18 19 20 21 22 23      21 22 23 24 25 26 27
27 28 29 30 31            24 25 26 27 28 29 30      28 29 30
                          31

      October                   November                 December
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
             1  2  3  4                      1                1  2  3  4  5  6
 5  6  7  8  9 10 11       2  3  4  5  6  7  8       7  8  9 10 11 12 13
12 13 14 15 16 17 18       9 10 11 12 13 14 15      14 15 16 17 18 19 20
19 20 21 22 23 24 25      16 17 18 19 20 21 22      21 22 23 24 25 26 27
26 27 28 29 30 31         23 24 25 26 27 28 29      28 29 30 31
                          30
```

Scanned with OKEN Scanner

6. calendar.month(year,month,w=2,l=1) -Returns a multiline string with a calendar for month, one line per week plus two header lines. w is the width in characters of each date l is the number of lines for each week.
7. calendar.prcal(year,w=2,l=1,c=6) – Same as that of calendar.calendar(year,w,l,c).
8. calendar.prmonth(year,month,w=2,l=1)–Same as that of calendar.month(year,month,w,l).

Example Program

```
import calendar
print(calendar.month(2015,11))
```

Output

```
    November 2015
Mo Tu We Th Fr Sa Su
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30
```

7.9.3 The datetime Module

The datetime module includes functions and classes for doing date and time parsing, formatting, and arithmetic. It contains functions and classes for working with dates and times, separately and together.

7.9.3.1 The time Class

Time values are represented with the `time` class. Times have attributes for hour, minute, second, and microsecond. They can also include time zone information. A time instance only holds values of time, and not a date associated with the time.

Example Program

```
import datetime
t = datetime.time(1, 2, 3)
print(t)
print('hour  :', t.hour)
print('minute:', t.minute)
print('second:', t.second)
print('microsecond:', t.microsecond)
print('tzinfo:', t.tzinfo)
```

Output

```
01:02:03
hour  : 1
```

Scanned with OKEN Scanner

```
minute: 2
second: 3
microsecond: 0
tzinfo: None
```

7.9.3.2 The date Class

Calendar date values are represented with the date class. Instances have attributes for year, month, and day. It is easy to create a date representing today's date using the today() class method.

Example Program

```
import datetime
today = datetime.date.today()
print(today)
print('ctime:', today.ctime())
print('Year:', today.year)
print('Mon :', today.month)
print('Day :', today.day)
```

Output

```
2015-11-22
ctime: Sun Nov 22 00:00:00 2015
Year: 2015
Mon : 11
Day : 22
```

Another way to create new date instances uses the `replace()` method of an existing date. For example, we can change the year, leaving the day and month alone.

Example Program

```
import datetime
d1 = datetime.date(2015, 3, 12)
print('d1:', d1)
d2 = d1.replace(year=2016)
print('d2:', d2)
```

Output

```
d1: 2015-03-12
d2: 2016-03-12
```

7.9.3.3 timedeltas

Using `replace()` is not the only way to calculate future/past dates. You can use datetime to perform basic arithmetic on date values via the timedelta class. Subtracting dates produces a timedelta, and a timedelta can be added or subtracted from a date to produce another date. The internal values for a timedelta are stored in days, seconds, and microseconds. Intermediate level values passed to the constructor are converted into days, seconds, and microseconds.

Scanned with OKEN Scanner

Example Program

```
import datetime
print("microseconds:", datetime.timedelta(microseconds=1))
print("milliseconds:", datetime.timedelta(milliseconds=1))
print("seconds     :", datetime.timedelta(seconds=1))
print("minutes     :", datetime.timedelta(minutes=1))
print("hours       :", datetime.timedelta(hours=1))
print("days        :", datetime.timedelta(days=1))
print("weeks       :", datetime.timedelta(weeks=1))
```

Output

```
microseconds: 0:00:00.000001
milliseconds: 0:00:00.001000
seconds     : 0:00:01
minutes     : 0:01:00
hours       : 1:00:00
days        : 1 day, 0:00:00
weeks       : 7 days, 0:00:00
```

Date math uses the standard arithmetic operators. This example with date objects illustrates using timedelta objects to compute new dates, and subtracting date instances to produce timedeltas (including a negative delta value).

Example Program

```
import datetime
today = datetime.date.today()
print('Today    :', today)
one_day = datetime.timedelta(days=1)
print('One day  :', one_day)
yesterday = today - one_day
print('Yesterday:', yesterday)
tomorrow = today + one_day
print('Tomorrow :', tomorrow)
print('tomorrow - yesterday:', tomorrow - yesterday)
print('yesterday - tomorrow:', yesterday - tomorrow)
```

Output

```
Today    : 2015-11-22
One day  : 1 day, 0:00:00
Yesterday: 2015-11-21
```

Scanned with OKEN Scanner

```
Tomorrow : 2015-11-23
tomorrow - yesterday: 2 days, 0:00:00
yesterday - tomorrow: -2 days, 0:00:00
```

Both date and time values can be compared using the standard operators to determine which is earlier or later.

Example Program

```
import datetime
import time
print('Times:')
t1 = datetime.time(12, 55, 0)
print('\tt1:', t1)
t2 = datetime.time(13, 5, 0)
print('\tt2:', t2)
print('\tt1 < t2:', t1 < t2)
print('Dates:')
d1 = datetime.date.today()
print('\td1:', d1)
d2 = datetime.date.today() + datetime.timedelta(days=1)
print('\td2:', d2)
print('\td1 > d2:', d1 > d2)
```

Output

```
Times:
    t1: 12:55:00
    t2: 13:05:00
    t1 < t2: True
Dates:
    d1: 2015-11-23
    d2: 2015-11-24
    d1 > d2: False
```

In addition, we can use combine() a date instance and time instance to create a datetime instance.

Example Program

```
import datetime
t = datetime.time(1, 2, 3)
print('t :', t)
d = datetime.date.today()
print('d :', d)
```

Scanned with OKEN Scanner

```
dt = datetime.datetime.combine(d, t)
print('dt:', dt)
```

Output

```
t : 01:02:03
d : 2015-11-23
dt: 2015-11-23 01:02:03
```

The default string representation of a datetime object uses the ISO 8601 format (YYYY-MM-DDTHH:MM:SS.mmmmmm). Alternate formats can be generated using `strftime()`. Similarly, if your input data includes timestamp values parsable with `time.strptime()`, then `datetime.strptime()` is a convenient way to convert them to datetime instances.

Example Program

```
import datetime
format = "%a %b %d %H:%M:%S %Y"
today = datetime.datetime.today()
print('ISO      :', today)
s = today.strftime(format)
print('strftime:', s)
d = datetime.datetime.strptime(s, format)
print('strptime:', d.strftime(format))
```

Output

```
ISO      : 2015-11-23 09:52:29.427414
strftime: Mon Nov 23 09:52:29 2015
strptime: Mon Nov 23 09:52:29 2015
```

7.10 Solved Lab Exercises

1. Write a Python function to reverse a string. Import the module to reverse a string input by the user.

Program

```
StringReverse.py
def string_reverse(str1):
print("Reversed string:",str1[::-1])

#Main program importing the module
import StringReverse
str1=input("Enter a string:")
print("Original string:",str1)
StringReverse.string_reverse(str1)
```

Scanned with OKEN Scanner

08 File Handling

CONTENTS

File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk). Since, random access memory (RAM) is volatile which loses its data when computer is turned off, we use files for future use of the data.

When we want to read from or write to a file we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed. Hence, in Python, a file operation takes place in the following order.

a. Open a file
b. Read or write (perform operation)
c. Close the file

We have been reading and writing to the standard input and output. Now, we will see how to use actual data files. Python provides basic functions and methods necessary to manipulate files by default. We can do most of the file manipulation using a file object.

Scanned with OKEN Scanner

8.1 Opening a File

Before we can read or write a file, we have to open it using Python's built-in open() function. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

We can specify the mode while opening a file. In mode, we specify whether we want to read 'r', write 'w' or append 'a' to the file. We also specify if we want to open the file in text mode or binary mode. The default is reading in text mode. In this mode, we get strings when reading from the file. On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like image or exe files. The following shows the syntax for opening a file.

Syntax

```
file object = open(filename [, accessmode][, buffering])
```

The following gives the explanation of the parameters for opening a file.

`filename`: The `filename` argument is a string value that contains the name of the file that we want to access.

`accessmode`: The `accessmode` determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in Section 8.1.1. This is optional parameter and the default file access mode is read (r).

`buffering`: If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If we specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default. This is also an optional parameter.

Example

```
f = open("abc.txt",'r')      # open file in current directory
f = open("C:/Python34/sample.txt",'r')# specifying full path
```

In the above example, the file `abc.txt` is opened in the current working directory in read mode and `sample.txt` is opened in read mode inside `C:\Python34` folder.

8.1.1 Modes for Opening a File

1. `'r'` - Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
2. `'rb'` – Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file.
3. `'r+'` – Opens a file for both reading and writing. The file pointer is placed at the beginning of the file.
4. `'rb+'` - Opens a file for both reading and writing in binary format. The file pointer is placed at the beginning of the file.
5. `'w'` – Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
6. `'wb'` - Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.

Scanned with OKEN Scanner

7. 'w+' - Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
8. 'wb+' - Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
9. 'a' - Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
10. 'ab' - Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
11. 'a+' - Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
12. 'ab+' - Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
13. 'x' - Open a file for exclusive creation. If the file already exists, the operation fails.

Since the version 3.x, Python has made a clear distinction between `str` (text) and `bytes` (8-bits). Unlike other languages, the character 'a' does not imply the number 97 until it is encoded using ASCII (or other equivalent encodings). Hence, when working with files in text mode, it is recommended to specify the encoding type. Files are stored in bytes in the disk, we need to decode them into `str` when we read into Python. Similarly, encoding is performed while writing texts to the file.

The default encoding is platform dependent. In Windows, it is 'cp1252' but 'utf-8' in Linux. Hence, we must not rely on the default encoding otherwise, our code will behave differently in different platforms. Thus, this is the preferred way to open a file for reading in text mode.

```
f = open("abc.txt", mode = 'r', encoding = 'utf-8')
```

8.1.2 Attributes of *file* object

Once a file is opened, we have one file object, we can get various information related to that file. The following shows various attributes for the file object.

1. file.closed - Returns True if file is closed, False otherwise.
2. file. mode - Returns access mode with which file was opened.
3. file.name - Returns name of the file.
4. file.softspace – Returns 0 if space is explicitly required with print, or 1 otherwise.

The following example shows the working of the above described attributes of file object.

Example Program

```
#Demo of file object attribute in Python
fo = open("abc.txt", "w")
print("Name of the file:", fo.name)
print("Closed or not:", fo.closed)
```

Scanned with OKEN Scanner

```
print("Opening mode:", fo.mode)
print("Softspace flag:", fo.softspace)
```

Output

```
Name of the file:  abc.txt
Closed or not: False
Opening mode: w
Softspace flag:0
```

8.2 Closing a File

When we have finished the operations to a file, we need to properly close it. Python has a garbage collector to clean up unreferenced objects. But we must not rely on it to close the file. Closing a file will free up the resources that were tied with the file and is done using the `close()` method. The following shows the syntax for closing a file.

Syntax

```
fileObject.close()
```

Example Program

```
# Open a file for writing in binary format
fo = open("bin.txt", "wb")
print("Name of the file: ", fo.name)
# Close opened file
fo.close()
print("File Closed")
```

Output

```
Name of the file:  bin.txt
File Closed
```

8.3 Writing to a File

In order to write into a file we need to open it in write 'w', append 'a' or exclusive creation 'x' mode. We need to be careful with the 'w' mode as it will overwrite into the file if it already exists. All previous data will be erased.

Writing a string or sequence of bytes (for binary files) is done using `write()` method. This method returns the number of characters written to the file. The `write()` method does not add a newline character ('\n') to the end of the string. The following shows the syntax of `write()` method.

Syntax

```
fileObject.write(string)
```

Here, passed parameter `string` is the content to be written into the opened file.

Scanned with OKEN Scanner

Example Program

```
# Open a file in writing mode
fo = open("test.txt", "w")
fo.write("Programming with Python is Fun.\nLet's try
Python!\n");
# Close opened file
fo.close()
print("File", fo.name, "closed.")
```

Output

```
File test.txt closed.
```

In the above program, we have opened a file `test.txt` in writing mode. The string to be written is passed to the `write()` method. This file is opened to see the contents in the example given in Section 8.4.

8.4 Reading from a File

To read the content of a file, we must open the file in reading mode. The `read()` method reads a string from an open file. It is important to note that Python strings can have binary data, apart from text data. The following gives the syntax for reading from a file.

Syntax

```
fileObject.read([size])
```

Here, the passed parameter size is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if size is missing, then it tries to read as much as possible, maybe until the end of file. The following example shows how to read from the file test.txt which we have already created.

Example Program

```
# Open a file in read mode
fo = open("test.txt", "r")
str = fo.read(11)
print("String Read is : ", str)
# Close opened file
fo.close()
print("File", fo.name, "is closed.")
```

Output

```
String Read is: Programming
File test.txt is closed.
```

Scanned with OKEN Scanner

8.5 File Methods

1. file.close() – This method is already explained in Section 8.2.
2. file.fileno() - Returns an integer number (file descriptor) of the file.

   Example Program

```
# Example for fileno() in File Handling
fo=open("abc.txt","w")
str="Python Programming"
fo.write(str)
print("The file number is:",fo.fileno())
fo.close()
```

   Output

```
The file number is: 3
```

3. file.seek(offset,from=SEEK_SET)- Change the file position to offset bytes, in reference to from (start, current, end).
4. file.tell() – Returns the current file location.

   Example Program

```
#Demo Program for seek and tell
# Creating a file
str="Python Programming is fun"
fo=open("test.txt","w")
fo.write(str)
fo.close()
fo = open("test.txt", "r+")
s = fo.read(10);
print("Read String is : ", s)
# Check current position
pos = fo.tell();
print("Current file position : ", pos)
# Reposition pointer at the beginning once again
pos = fo.seek(0, 0);
s = fo.read(10);
print("Again read String is : ", s)
# Close opened file
fo.close()
```

5. file.write() – This method is already explained in Section 8.3.
6. file.read() – This method is already explained in Section 8.4.

Scanned with OKEN Scanner

7. file.readline()-The method readline() reads one entire line from the file. A trailing newline character is kept in the string. An empty string is returned only when EOF is encountered immediately.

8. file.truncate([size] - The method truncate() truncates the file's size. If the optional size argument is present, the file is truncated to (atmost) that size.

Example Program

```
# Open a file
fo = open("test.txt", "r+")
line = fo.readline()
print("Read Line:",line)
# Now truncate remaining file.
fo.truncate()
# Try to read file now
line = fo.readline()
print("Read Line: ",line)
# Close opened file
fo.close()
```

Output

```
Read Line: Python Programming is fun
Read Line:
```

9. file.readlines([sizeint]) - The method readlines() reads until EOF using readline() and returns a list containing the lines. sizeint is an optional argument and if it is present, instead of reading up to EOF, whole lines approximate to sizeint bytes are read. An empty string is returned only when EOF is encountered immediately.

10. file.writelines(sequence) - The method writelines() writes a sequence of strings to the file. The sequence can be any iterable object containing strings, typically a list of strings. There is no return value.

Example Program

```
# Open a file in write mode
fo = open("Demo.txt", "w")
seq=["First Line\n", "Second Line\n", "Third Line\n", "Fourth
Line\n", "Fifth Line\n"]
fo.writelines(seq)
#close the file
fo.close()
#Open the file in read mode
fo = open("Demo.txt", "r")
line = fo.readlines()
```

Scanned with OKEN Scanner

8. file.readline()-The method readline() reads one entire line from the file. A trailing newline character is kept in the string. An empty string is returned only when EOF is encountered immediately.

9. file.truncate([size] - The method truncate() truncates the file's size. If the optional size argument is present, the file is truncated to (atmost) that size.

Example Program

```
# Open a file
fo = open("test.txt", "r+")
line = fo.readline()
print("Read Line:",line)
# Now truncate remaining file.
fo.truncate()
# Try to read file now
line = fo.readline()
print("Read Line: ",line)
# Close opened file
fo.close()
```

Output

```
Read Line: Python Programming is fun
Read Line:
```

9. file.readlines([sizeint]) - The method readlines() reads until EOF using readline() and returns a list containing the lines. sizeint is an optional argument and if it is present, instead of reading up to EOF, whole lines approximate to sizeint bytes are read. An empty string is returned only when EOF is encountered immediately.

10. file.writelines(sequence) - The method writelines() writes a sequence of strings to the file. The sequence can be any iterable object containing strings, typically a list of strings. There is no return value.

Example Program

```
# Open a file in write mode
fo = open("Demo.txt", "w")
seq=["First Line\n", "Second Line\n", "Third Line\n", "Fourth
Line\n", "Fifth Line\n"]
fo.writelines(seq)
#close the file
fo.close()
#Open the file in read mode
fo = open("Demo.txt", "r")
line = fo.readlines()
```

Scanned with OKEN Scanner

```
print("readlines():", line)
line= fo.readlines(2)
print("readlines(2):",line)
# Close opened file
fo.close()
```

Output

```
readlines(): ['First Line\n', 'Second Line\n', 'Third Line\n',
'Fourth Line\n', 'Fifth Line\n']
readlines(2): []
```

8.6 Renaming a File

The os module Python provides methods that help to perform file-processing operations, such as renaming and deleting files. To use this os module, we need to import it first and then we can call any related functions. To rename an existing file, we can use the rename() method. This method takes two arguments, current filename and new filename. The following shows the syntax for rename() method.

Syntax

```
os.rename(current_file_name, new_file_name)
```

Example Program

```
import os
# Rename a file from test.txt to Newtest.txt
os.rename( "test.txt", "Newtest.txt" )
print("File renamed.")
```

Output

```
File renamed.
```

8.7 Deleting a File

We can delete a file by using the remove() method available in os module. This method receives one argument which is the filename to be deleted. The following gives the syntax for remove() method.

Syntax

```
os.remove(filename)
```

Example Program

```
import os
# Delete a file
os.remove("Newtest.txt")
print("File Deleted.")
```

Scanned with OKEN Scanner

Output

```
File Deleted.
```

8.8 Directories in Python

All files will be saved in various directories, and Python has efficient methods for handling directories and files. The os module has several methods that help to create, remove and change directories.

8.8.1 mkdir() method

The mkdir() method of the `os` module is used to create directories in the current directory. We need to supply an argument to this method which contains the name of the directory to be created. The following shows the syntax of `mkdir()` method.

Syntax

```
os.mkdir("dirname")
```

Example

```
import os
# Create a directory "Fruits"
os.mkdir("Fruits")
```

The above example creates a directory named `Fruits`.

8.8.2 chdir() method

To change the current directory, we can use the `chdir()` method. The `chdir()` method takes an argument, which is the name of the directory that we want to make the current directory. The following shows the syntax of `chdir()` method.

Syntax

```
os.chdir("dirname")
```

Example

```
import os
# Change a directory
os.chdir("/home/abc")
```

The above example goes to the directory `/home/abc`.

8.8.3 getcwd() method

The `getcwd()` method displays the current working directory. The following shows the syntax of `getcwd()` method.

Syntax

```
os.getcwd()
```

Scanned with OKEN Scanner

Example

```
import os
# Displays the location of current directory
os.getcwd()
```

8.8.4 rmdir() method

The rmdir() method deletes the directory, which is passed as an argument in the method.

Syntax

```
os.rmdir("dirname")
```

Example

```
import os
# Removes the directory
os.rmdir("Fruits")
```

It is intended to give the exact path of a directory. Otherwise it will search the current working directory.

8.9 Solved Lab Exercises

1. Write a program to copy a text file to another file.

Program

```
# Open the file to be copied in read mode
file1=input("Enter the source file to be copied:")
file2=input("Enter the destination file name:")
fr=open(file1, "r")
# Open a file to be copied in write mode
fw=open(file2, "w")
for line in fr.readlines():
    fw.write(line)
#close the files
fr.close()

fw.close()
print("1 File Copied")
```

Output

```
1 File Copied
```

2. **Program to count the number of lines in a file.**

Scanned with OKEN Scanner

Program

```
fr=open("Demo.txt", "r")
countlines=0
for line in fr.readlines():
    countlines=countlines+1
print("Number of lines:",countlines)
#close the file
fr.close()
```

Output

```
Number of lines: 2
```

3. Write a program to count the frequencies of each word from a file.

Program

```
# Open the file to be read in read mode
fr=open("Demo.txt","r")
wordcount={}
for word in fr.read().split():
    if word not in wordcount:
        wordcount[word] = 1
    else:
        wordcount[word] += 1
for k,v in wordcount.items():
    print(k, v)
#close the file
fr.close()
```

Output

```
Handling 1
easy 1
Python 2
is 2
Programming 1
File 1
fun 1
```

4. Write a program to append a file with the contents of another file.

Program

```
file1=input("Enter the file to be opened for appending:")
file2=input("Enter the file name to be appended:")
```

Scanned with OKEN Scanner

```
# Open the file to in append mode
fa=open(file1, "a")
# Open a file in read mode
fr=open(file2, "r")
for line in fr.readlines():
    fa.write(line)
#close the files
fr.close()
fa.close()
print("1 File Appended")
```

Output

```
Enter the file to be opened for appending:TruncateDemoCopy.py
Enter the file name to be appended:TruncateDemo.py
1 File Appended
```

5. Write a program, that counts the number of times the first three letters of the alphabet (A, a, B, b, C, c) occur in a file. Do not distinguish between the lowercase and uppercase letters.

Program

```
file1=input("Enter the file to be opened :")
# Open the file in read mode
fr=open(file1, "r")
counta,countb,countc=0,0,0
for ch in fr.readlines():
 counta=counta+(ch.count('A',0,len(ch))or
ch.count('a',0,len(ch)))
 countb=countb+(ch.count('B',0,len(ch)) or
ch.count('b',0,len(ch)))
 countc=countc+(ch.count('C',0,len(ch)) or
ch.count('c',0,len(ch)))
#close the file
fr.close()
print("Number of occurences of 'A' or 'a'=",counta)
print("Number of occurencesof 'B' or 'b'=",countb)
print("Number of occurences of 'C' or 'c'=",countc)
```

Output

```
Enter the file to be opened :test.txt
Number of occurences of 'A' or 'a'= 3
Number of occurencesof 'B' or 'b'= 2
Number of occurences of 'C' or 'c'= 3
```

Scanned with OKEN Scanner

6. Write a program to compare two files.

Program

```
import filecmp
file1=input("Enter the first file to be compared:")
file2=input("Enter the second file to be compared:")
b=filecmp.cmp(file1,file2)
if b: print("Files are equal")
else: print("Files are not equal")
```

Output

```
Enter the first file to be compared:test.txt
Enter the second file to be compared:test1.txt
Files are equal
```

7. Write a program to delete a sentence from the specified position in a file.

Program

```
#Program to delete a sentence from a specified position from a
file
def filedel(n):
  with open(str,'r') as file :
   lis = file.readlines()
   file.close()
   print(lis)
   del lis[n]
   print(lis)
  with open(str,'w') as file:
   file.writelines(lis)
   file.close()
#main Program
str=input("Enter the filename:")
n=int(input("Enter the position of the word to be deleted:"))
filedel(n)
print("Sentence Deletion Successful")
```

Output

```
Enter the filename:abc.txt
Enter the position of the word to be deleted:1
['Python Programming is Fun\n', 'Learning Python is Fun\n',
'Welcome to python\n', 'Guido van Rossum\n', 'Open Source']
```

Scanned with OKEN Scanner

```
['Python Programming is Fun\n', 'Welcome to python\n', 'Guido
van Rossum\n', 'Open Source']
File Deletion Successful
```

8. Write a program to delete the sentences from a file, if the file contains a specific word.

Program

```
#Program for deleting the line if the line contains a particular
word
import os
def fileDEL(str,word):
    file = open(str,"r")
    newfile = open("new.txt","w")
    while True:
        line = file.readline()
        if not line :
         break
        elif word in line :
         pass
        else :
         print(line)
         newfile.write(line)
    file.close()
    newfile.close()
    os.remove(str)
    os.rename("new.txt",str)
#main Program
str=input("Enter the name of the file:")
word=input("Enter the word:")
fileDEL(str,word)
```

Output

```
Enter the name of the file:abc.txt
Enter the word:fun
Learn Python
```

9. Write a program to delete comment lines (lines beginning with a #) from a file.

Program

```
#Program for deleting comment lines or lines beginning with a #
import os
def fileDEL(str):
```

Scanned with OKEN Scanner

```
    file = open(str,"r")
    newfile = open("new.txt","w")
    while True:
        line = file.readline()
        if not line :
         break
        elif line.startswith('#',0,len(line)):
         pass
        else :
         newfile.write(line)
    file.close()
    newfile.close()
    os.remove(str)
    os.rename("new.txt",str)
#Main Program
str=input("Enter the name of the file:")
fileDEL(str)
print("Lines beginning with # deleted")
```

Output

```
Enter the name of the file:abc.txt
Lines beginning with # deleted
```

10. Write a Python function to capitalize each word in a file.

Program

```
#Program for deleting the line if the line contains a particular
word
import os
def fileCapitalize(str):
    file = open(str,"r")
    newfile = open("new.txt","w")
    while True:
        line = file.readline()
        if not line :
         break
        else :
         newfile.write(line.title())
    file.close()
    newfile.close()
```

Scanned with OKEN Scanner

```
    os.remove(str)
    os.rename("new.txt",str)
#main Program
str=input("Enter the name of the file:")
fileCapitalize(str)
print("File Capitalized")
```

Output

```
Enter the name of the file:abc.txt
File Capitalized
```

11. Write a Python program to search a word and replace with another word for all the occurrences.

Program

```
#Program for deleting the line if the line contains a particular
word
import os
def fileSearchandReplace(str,oldword,newword):
   file = open(str,"r")
   newfile = open("new.txt","w")
   while True:
      line = file.readline()
      if not line :
       break
      else :
       newfile.write(line.replace(oldword,newword))
   file.close()
   newfile.close()
   os.remove(str)
   os.rename("new.txt",str)
#main Program
str=input("Enter the name of the file:")
oldword=input("Enter the word to be replaced:")
newword=input("Enter the new word:")
fileSearchandReplace(str,oldword,newword)
print("Replacements Successful")
```

Output

```
Enter the name of the file:abc.txt
Enter the word to be replaced:Is
```

Scanned with OKEN Scanner

```
Enter the new word:was
Replacement Successful
```

12. Write a Python program to insert a sentence into a specified position in a file.

Program

```
#program to insert a sentence in a specified position in a file
def fileinsert(str,n,line):
  with open(str,'r') as file :
    lis = file.readlines()
    lis.insert(n,line)
    lis.insert(n+1,'\n')
    file.close()
    print(lis)

  with open(str,'w') as file:
    file.writelines(lis)
    print(lis)
    file.close()
#main program
str=input("Enter the filename:")
n=int(input("Enter the position of the sentence to be
inserted:"))
line=input("Enter the Sentence to be inserted:")
fileinsert(str,n,line)
print("Sentence Insertion Successful.")
```

Output

```
Enter the filename:abc.txt
Enter the position of the sentence to be inserted:1
Enter the Sentence to be inserted:Python Program
['Why Python\n', 'Python Program', '\n', 'Python is fun\n',
'Guido van Rossum']
['Why Python\n', 'Python Program', '\n', 'Python is fun\n',
'Guido van Rossum']
Sentence Insertion Successful
```

13. Write a program to read a file in reverse order. The last sentence should be read first and continue till the first sentence is read.

Program

```
#Program to read a file and display contents backwards
```

```
def filebackward(str):
  with open(str,'r') as file :
   lis = file.readlines()
   lis.reverse()
   file.close()
   print(lis)
   for line in lis:
    print(line)
#main Program
str=input("Enter the filename:")
filebackward(str)
```

Output

```
Enter the filename:abc.txt
['Guido van Rossum', 'Python is fun\n', 'Python Program\n', 'Why
Python\n']
Guido van Rossum
Python is fun
Python Program
Why Python
```

8.10 Conclusion

This Chapter covers the concepts of file handling which covers how to open a file, different modes of opening a file, how to close a file, how to read and write data to files, various built-in methods available in file handling, how to rename and delete a file. This Chapter also covers various directory methods like mkdir(), chdir(), getcwd and rmdir() methods available in Python.

8.11 Review Questions

1. Briefly explain various modes for opening a file.
2. What are the various attributes for a file object?
3. Explain the built-in methods available in Python for file handling.
4. How will you rename a file in Python?
5. How will you delete a file in Python?
6. Briefly explain the various methods available in Python for processing directories.
7. Explain the seek() and tell() method in Python.
8. What is the purpose of truncate() method in Python?
9. Explain readline(), readlines() and writelines() method in Python.
10. Explain the methods fileno() and filename() with examples.

Scanned with OKEN Scanner

09 Object Oriented Programming

CONTENTS

Object Oriented Programming is a powerful method for creating programs. So far we have discussed the procedural concepts in programming. Before going into Object Oriented Programming (OOPs), we need to understand the basic terminologies used in Object Oriented Programming. While procedure oriented programming gives importance to functions, Object Oriented Programming focuses on objects. A class is the base of Object Oriented Programming.

A class contains a collection of data (variables) and methods (functions) that act on those data. Class is considered as a blueprint for an object. For example consider the design of a house with details of windows, doors, roofs and floors. We can build the house based on the descriptions. This can be considered as a class while the object is the house itself. An object is said to be an instance of a class and the process of creating a class is called instantiation. The basic concepts related to OOP are as follows:

Scanned with OKEN Scanner

1. Classes
2. Objects
3. Encapsulation
4. Data Hiding
5. Inheritance
6. Polymorphism

Advantages of Object Oriented Programming

Object Oriented Programming has following advantages:

- **Simplicity:** The objects in case of OOP are close to the real world objects, so the complexity of the program is reduced making the program structure very clear and simple. For example by looking at the class Student, we can simply identify with the properties and behavior of a student. This makes the class Student very simple and easy to understand.
- **Modifiability:** It is easy to make minor changes in the data representation or the procedures in an OOP. Changes inside a class do not affect any other part of a program, since the only public interface that the external world has to a class is through the use of methods.
- **Extensibility and Maintainability:** It is quite easy to add new features and extend the program in case of Object Oriented Programming. It can be simply done by introducing a few new objects and modifying some existing ones. The original base class need not be modified at all. Even objects can be maintained separately, there by making locating and fixing problems easier. For example if a new attribute of the class Student needs to be added, a new derived class of the class Student may be created and no other class in the class hierarchy need to be modified.
- **Re-usability:** Objects can be reused in different programs. The class definitions can be reused in various applications. Inheritance makes it possible to define subclasses of data objects that share some or all of the main class characteristics. It forces a more thorough data analysis, reduces development time and ensures more accurate coding.
- **Security:** Since a class defines only the data it needs to be concerned with, when an instance of that class (an object) is run, the code will not be able to accidentally access other program data. This characteristic of data hiding provides greater system security and avoids unintended data corruption.

9.1 Class Definition

The class definition in Python begins with the keyword class. The first statement after class definition will be the docstring which gives a brief description about the class. The following shows the syntax for defining a class.

Syntax

```
class ClassName:
    'Optional class documentation string'
    class_suite
```

Scanned with OKEN Scanner

A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions. There are also special attributes in it that begins with double underscores (__). For example, __doc__ gives us the docstring of that class. As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

Example Program

```
class Student:
 "Common base class for all students"
 studentcount = 0
 def __init__(self, rollno,name,course):
      self.rollno=rollno
      self.name = name
      self.course = course
      Student.studentcount += 1
 def displayCount(self):
     print("Total Students=", Student.studentcount)
 def displayStudent(self):
      print("Roll Number:", self.rollno)
      print("Name: ", self.name)
      print("Course: ", self.course)
```

In the above program, we have created a class called Student. The variable `studentcount` is a class variable whose value is shared among all instances of this class. This can be accessed from inside the class or outside the class by giving `Student.studentcount`.

The first method __init__() is a special method, which is called class constructor or initialization method that Python calls when we create a new instance of this class.

Class methods have only one specific difference from ordinary functions - they must have an extra argument in the beginning of the parameter list. This particular argument is `self` which is used for referring to the instance. But we need not give any value for this parameter when we call the method. Python provides it automatically. `self` is not a reserved word in Python but just a strong naming convention and it is always convenient to use conventional names as it makes the program more readable. So while defining our class methods, we must explicitly list `self` as the first argument for each method, including __init__.

It also implies that if we have a method which takes no arguments, then we still have to define the method to have a `self` argument. `Self` is an instance identifier and is required so that the statements within the methods can have automatic access to the current instance attributes.

9.2 Creating Objects

An object is created by calling the class name with the arguments defined in the _init_ method. We can access a method by using the dot operator along with the object. Similarly a class name can be accessed by specifying the method name with the dot operator with the class.

Scanned with OKEN Scanner

Example Program

```
# First student object is created
stud1=Student(10, "Jack", "MS")
# Second student object is created.
stud2=Student(20, "Jill", "BE")
# Displays the details of First Student
stud1.displayStudent()
# Displays the details of Second Student.
stud2.displayStudent()
print("Total Number of Students:", Student.studentcount)
```

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

```
Roll Number: 10
Name:  Jack
Course:  MS
Roll Number: 20
Name:  Jill
Course:  BE
Total Number of Students: 2
```

9.3 Built-in Attribute Methods

There are built-in functions to access the attributes. We can access the attribute value, check whether an attribute exists or not, modify an attribute value or delete an attribute. The following gives the built-in attribute methods.

1. getattr(obj, name[, default]) : This method is used to access the attribute of object.
2. hasattr(obj,name) : This attribute is used to check if an attribute exists or not.
3. setattr(obj,name,value) : This method is used to set an attribute. If attribute does not exist, then it would be created.
4. delattr(obj, name) : This method is used to delete an attribute.

Example Program

```
class Student:
"Common base class for all students"
def __init__(self, rollno,name,course):
    self.rollno=rollno
    self.name = name
    self.course = course
def displayStudent(self):
    print("Roll Number:", self.rollno)
```

Scanned with OKEN Scanner

```
        print("Name: ", self.name)
        print("Course: ", self.course)
stud1=Student(10, "Jack", "MS")
stud1.displayStudent()
at=getattr(stud1,'name')
print("getattr(stud1,'name'):",at)
print("hasattr(stud1,age):" , hasattr(stud1,'age'))
#New attribute inserted
print("setattr(stud1,age,21):",setattr(stud1, 'age', 21))
stud1.displayStudent()
print("Age:", stud1.age)
# Attribute age deleted
print("delattr(stud1,age):", delattr(stud1, 'age'))
stud1.displayStudent()
```

Output

```
Roll Number: 10
Name:  Jack
Course:  MS
getattr(stud1,'name'): Jack
hasattr(stud1,age): False
setattr(stud1,age,21): None
Roll Number: 10
Name:  Jack
Course:  MS
Age: 21
delattr(stud1,age): None
Roll Number: 10
Name:  Jack
Course:  MS
```

9.4 Built-in Class Attributes

Python contains several built-in class attributes. These attributes are accessed using the dot operator. The following gives the list of built-in class attributes.

1. __dict__: This attribute contains the dictionary containing the class's namespace.
2. __doc__: It describes the class or it contains the class documentation string. If undefined, it contains None.
3. __name__: This attribute contains the class name.

Scanned with OKEN Scanner

4. __module__: This contains the module name in which the class is defined. This attribute is "__main__" in interactive mode.
5. __bases__: This contains an empty tuple containing the base classes, in the order of their occurrence in the base class list.

The following program shows the usage of built-in class attributes.

Example Program

```
class Student:
 "Common base class for all students"
 def __init__(self, rollno,name,course):
       self.rollno=rollno
       self.name = name
       self.course = course
 def displayStudent(self):
      print("Roll Number:", self.rollno)
      print("Name: ", self.name)
      print("Course: ", self.course)
stud1=Student(10, "Jack", "MS")
stud1.displayStudent()
print("Student.__doc__:", Student.__doc__)
print("Student.__name__:", Student.__name__)
print("Student.__module__:", Student.__module__)
print("Student.__bases__:", Student.__bases__)
print("Student.__dict__:", Student.__dict__)
```

Output

```
Roll Number: 10
Name:  Jack
Course:  MS
Student.__doc__: Common base class for all students
Student.__name__: Student
Student.__module__: __main__
Student.__bases__: ()
Student.__dict__: {'displayStudent': <function displayStudent at
0x7efdcc047758>, '__module__': '__main__', '__doc__': 'Common ba
se class for all students', '__init__': <function __init__ at
0x7efdcc0476e0>}
```

Scanned with OKEN Scanner

9.5 Destructors in Python

Python automatically deletes an object that is no longer in use. This automatic destroying of objects is known as garbage collection. Python periodically performs the garbage collection to free the blocks off memory that are no longer in use. But a class can implement the special method __del__(), called a destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any non memory resources used by an instance. The following program shows the destructors in Python.

Example Program

```
class Student:
 "Common base class for all students"
 def __init__(self, rollno,name,course):
      self.rollno=rollno
      self.name = name
      self.course = course
 def displayStudent(self):
      print("Roll Number:", self.rollno)
      print("Name: ", self.name)
      print("Course: ", self.course)
 def __del__(self):
      class_name = self.__class__.__name__
      print(class_name, "destroyed")
stud1=Student(10, "Jack", "MS")
stud1.displayStudent()
del stud1
```

Output

```
Roll Number: 10
Name:  Jack
Course:  MS
Student destroyed
```

9.6 Encapsulation

Encapsulation is the most basic concept of OOP. It is the combining of data and the functions associated with that data in a single unit. In most of the languages including Python, this unit is called a class. In simple terms we can say that encapsulation is implemented through classes. In fact the data members of a class can be accessed through its member functions only. It keeps the data safe from any external interference and misuse. The only way to access the data is through the functions of the class. In the example of the class Student, the class encapsulates the data (rollno, name, course) and the associated functions into a single independent unit.

Scanned with OKEN Scanner

9.7 Data Hiding

We can hide data in Python. For this we need to prefix double underscore for an attribute. Data hiding can be defined as the mechanism of hiding the data of a class from the outside world or to be precise, from other classes. This is done to protect the data from any accidental or intentional access. In most of the Object Oriented Programming languages, encapsulation is implemented through classes. In a class, data may be made private or public. Private data or function of a class cannot be accessed from outside the class while public data or functions can be accessed from anywhere. So data hiding is achieved by making the members of the class private. Access to private members is restricted and is only available to the member functions of the same class. However the public part of the object is accessible outside the class. Once the attributes are prefixed with the double underscore, it will not be visible outside the class. The following program shows an example for data hiding in Python.

Example Program

```
class HidingDemo:
 "Program for Hiding Data"
 __num=0
 def numbercount(self):
     self.__num+=1
     print("Number Count=", self.__num)
number=HidingDemo()
number.numbercount()
print(number.__num)
```

Output

```
Number Count= 1
Traceback (most recent call last):
  File "main.py", line 9, in <module>
    print number.__num
AttributeError: HidingDemo instance has no attribute '__num'
```

The output shows that numbercount is displayed once. When it is attempted to call the variable number.__num from outside the function, it resulted in an error.

9.8 Inheritance

The mechanism of deriving a new class from an old class is known as inheritance. The old class is known as base class or super class or parent class. The new one is called the subclass or derived class or child class. Inheritance allows subclasses to inherit all the variables and methods to their parent class. The advantage of inheritance is the reusability of code. Fig. 9.1 illustrates the single inheritance. The attributes and methods of parent class will be available in child class after inheritance.

Scanned with OKEN Scanner

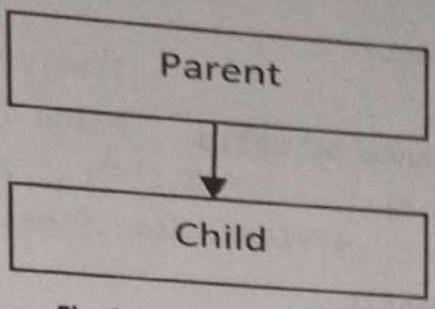

Fig. 9.1: Single Inheritance

9.8.1 Deriving a Child Class

The following shows the syntax for deriving a child class in Python.

Syntax

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
   'Optional class documentation string'
class_suite
```

The following program shows an example for single inheritance.

Example Program

```
class Student:
 "Common base class for all students"
 def getData(self,rollno,name,course):
        self.rollno=rollno
        self.name = name
        self.course = course
 def displayStudent(self):
       print("Roll Number:", self.rollno)
       print("Name:", self.name)
       print("Course:", self.course)
 #Inheritance
class Test(Student):
 def getMarks(self, marks):
   self.marks=marks
 def displayMarks(self):
   print("Total Marks:", self.marks)
r = int(input("Enter Roll Number:"))
n = input("Enter Name:")
c = input("Enter Course Name:")
m = int(input("Enter Marks:"))
```

Scanned with OKEN Scanner

```
#creating the object
print("Result")
stud1=Test()#instance of child
stud1.getData(r,n,c)
stud1.getMarks(m)
stud1.displayStudent()
stud1.displayMarks()
```

Output

```
Enter Roll Number:20
Enter Name:Smith
Enter Course Name:MS
Enter Marks:200
Result
Roll Number: 20
Name: Smith
Course: MS
Total Marks: 200
```

9.8.2 Multilevel Inheritance

We have already discussed the mechanism of deriving a new class from one parent class. We can further inherit the derived class to form a new child class. Inheritance which involves more than one parent class but at different levels is called multilevel inheritance. Figure 9.2 illustrates multilevel inheritance.

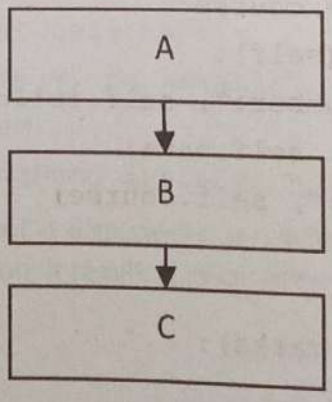

Fig. 9.2: Multilevel Inheritance

Here B is derived from A. Hence all attributes and methods that are available in A is now available in B also. C is derived from B. Hence all methods and attributes of B and A is now available in C. The following program shows an example for multilevel inheritance.

Scanned with OKEN Scanner

Example Program

```
    def displayStudent(self):
        print("Roll Number:", self.rollno)
        print("Name:", self.name)
        print("Course:", self.course)
    #Inheritance
    class Test(Student):
     def getMarks(self, marks):
       self.marks=marks
     def displayMarks(self):
       print("Total Marks:", self.marks)
    #Multilevel Inheritance
    class Result(Test):
     def calculateGrade(self):
       if self.marks>480: self.grade="Distinction"
       elif self.marks>360: self.grade="First Class"
       elif self.marks>240: self.grade="Second Class"
       else: self.grade="Failed"
       print("Result:", self.grade)
    #Main Program
    r = int(input("Enter Roll Number:"))
    n = input("Enter Name:")
    c = input("Enter Course Name:")
    m = int(input("Enter Marks:"))
    #creating the object
    print("Result")
    stud1=Result()#instance of child
    stud1.getData(r,n,c)
    stud1.getMarks(m)
    stud1.displayStudent()
    stud1.displayMarks()
    stud1.calculateGrade()
```

Output

```
Enter Roll Number:12
Enter Name:Jones
Enter Course Name:MS
```

Scanned with OKEN Scanner

```
Enter Marks:400
Result
Roll Number: 12
Name: Jones
Course: MS
Total Marks: 400
Result: First Class
```

9.8.3 Multiple Inheritance

It is possible to inherit from more than one parent class. Such type of inheritance is called multiple inheritance. In this case all the attributes and methods of both the parent class will be available in the child class after inheritance. Fig 9.3 illustrates multiple inheritance.

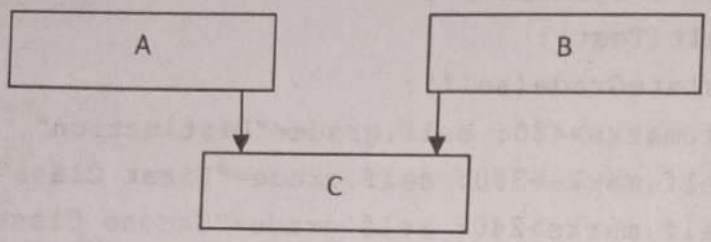

Fig. 9.3: Multiple Inheritance

Here A and B are the parent classes and C is the child class. The attributes and methods of both classes A and B are now available in C after inheritance.

Example Program

```
class Student:
 "Common base class for all students"
 def getData(self,rollno,name,course):
       self.rollno=rollno
       self.name = name
       self.course = course
 def displayStudent(self):
      print("Roll Number:", self.rollno)
      print("Name:", self.name)
      print("Course:", self.course)
 #Inheritance
class Test(Student):
 def getMarks(self, marks):
   self.marks=marks
 def displayMarks(self):
```

Scanned with OKEN Scanner

```
    print("Total Marks:", self.marks)
class Sports:
 def getSportsMarks(self, spmarks):
   self.spmarks=spmarks
 def displaySportsMarks(self):
   print("Sports Marks:", self.spmarks)

#Multiple Inheritance
class Result(Test,Sports):
 def calculateGrade(self):
     m=self.marks+self.spmarks
     if m>480: self.grade="Distinction"
     elif m>360: self.grade="First Class"
     elif m>240: self.grade="Second Class"
     else: self.grade="Failed"
     print("Result:", self.grade)
#Main Program
r = int(input("Enter Roll Number:"))
n = input("Enter Name:")
c = input("Enter Course Name:")
m = int(input("Enter Marks:"))
s = int(input("Enter Sports marks:"))
#creating the object
print("Result")
stud1=Result()#instance of child
stud1.getData(r,n,c)
stud1.getMarks(m)
stud1.getSportsMarks(s)
stud1.displayStudent()
stud1.displayMarks()
stud1.displaySportsMarks()
stud1.calculateGrade()
```

Output

```
Enter Roll Number:10
Enter Name:Bob
Enter Course Name:MS
Enter Marks:190
```

Scanned with OKEN Scanner

```
Enter Sports marks:200
Result
Roll Number: 10
Name: Bob
Course: MS
Total Marks: 190
Sports Marks: 200
Result: First Class
```

9.8.4 Invoking the Base Class Constructor

In the above examples of inheritance, we have used member functions. Instead we can use constructors to pass value to the objects. In Python the constructor of the base class can be invoked by extending __init__ ().

The class Student and class Teacher both have __init__ () method. The __init__ () method is defined in class Student and Teacher is extended in class School. In Python, name of the base class can also be used to access the method of the base class which has been extended in derived class.

Example Program

```
class Student(object):
 def __init__(self,Id,name):
  self.Id=Id
  self.name=name
 def showStudent(self):
  print("Id number of the Student:",self.Id)
  print("Name of the Student:",self.name)
class Teacher(object):
 def __init__(self,tec_Id,tec_name, subject):
  self.tec_Id=tec_Id
  self.tec_name=tec_name
  self.subject=subject
 def showTeacher(self):
  print("Id of the Teacher:",self. tec_Id)
  print("Name of the Teacher:",self.tec_name)
  print("Subject:",self.subject)
class School(Student,Teacher):
  def __init__(self, ID, name, tec_Id, tec_name, subject, sch_
Id):
   Student.__init__ (self,ID,name)
   Teacher. __init__(self,tec_Id,tec_name, subject)
```

Scanned with OKEN Scanner

```
        self.sch_Id= sch_Id
      def showSchool(self):
        print("Id of the School:",self.sch_Id)
#main Program
sc=School(10,"Jones",100,"Jack","Math",80)
sc.showStudent()
sc.showTeacher()
sc.showSchool()
```

Output

```
Id number of the Student: 10
Name of the Student: Jones
Id of the Teacher: 100
Name of the Teacher: Jack
Subject: Math
Id of the School: 80
```

9.9 Method Overriding

Polymorphism is an important characteristic of Object Oriented Programming language. In Object-Oriented Programming, polymorphism refers to a programming language's ability to process objects differently depending on their data type or class. More specifically, it is the ability to redefine methods for derived classes. We can override methods in the parent class.

Method overriding is required when we want to define our own functionality in the child class. This is possible by defining a method in the child class that has the same name, same arguments and same return type as a method in the parent class. When this method is called, the method defined in the child class is invoked and executed instead of the one in the parent class. The following shows an example program for method overriding.

Example Program

```
class Parent:
 "Base class"
 def __init__(self,name):
        self.name = name
 def displayName(self):
       print("Name: ", self.name)
 def __del__(self):
       class_name = self.__class__.__name__
       print(class_name, "destroyed")
class Child(Parent):
 def __init__(self,name,address):
```

Scanned with OKEN Scanner

```
        self.name = name
        self.address= address
    def displayName(self):
        print("Name: ", self.name)
        print("Address:",self.address)
    def __del__(self):
        class_name = self.__class__.__name__
        print(class_name, "destroyed")
#Main Program
n=input("Enter Name:")
a=input("Enter Address:")
obj=Child(n,a)#instance of child
obj.displayName() #Calling child's method
del obj
```

Output

```
Enter Name:Jill
Enter Address:Klos
Name:  Jill
Address: Klos
Child destroyed
```

In the above program, even though there are two methods with the same name displayName() is available in child class, the method in the child class will be invoked. The method in the child class has overridden the method in the parent class.

The following shows some generic functionality that can be overriden.

```
1__init__ ( self [,args...] )
Constructor (with any optional arguments)
Sample Call : obj = className(args)
2__del__( self )
Destructor, deletes an object
Sample Call : del obj
3__repr__( self )
Evaluatable string representation
Sample Call : repr(obj)
4__str__( self )
Printable string representation
Sample Call : str(obj)
5__cmp__ ( self, x )
```

Scanned with OKEN Scanner

```
Object comparison
Sample Call : cmp(obj, x)
```

9.10 Polymorphism

The word polymorphism is formed from two words - poly and morph where poly means many and morph means forms. So polymorphism is the ability to use an operator or function in various forms. That is a single function or an operator behaves differently depending upon the data provided to them. Polymorphism can be achieved in two ways: operator overloading and function overloading. Unlike in C++ and Java, Python does not support function or method overloading.

9.10.1 Operator Overloading

Operator overloading is one of the important features of Object Oriented Programming. C++ and Python supports operator overloading, while Java does not support operator overloading. The mechanism of giving special meanings to an operator is known as operator overloading. For example the operator '+' is used for adding two numbers. In Python this operator can also be used for concatenating two strings. This '+' operator can be used for adding member variables of two different class. The following example shows how '+' operator can be overloaded.

Example Program

```
class Abc:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def __str__(self):
        return "Abc (%d, %d)" % (self.a, self.b)
    def __add__(self,other):
        return Abc(self.a + other.a, self.b + other.b)
a1 = Abc(2,4)
a2 = Abc(5,-2)
print(a1 + a2)
```

Output

```
Abc (7, 2)
```

9.11 Solved Lab Exercises

1. Write a Python class to convert an integer to a roman numeral.

Program

```
class py_solution:
def int_to_Roman(self, num):
    val = [
```

Scanned with OKEN Scanner

10 Exception Handling

CONTENTS

An exception is an abnormal condition that is caused by a runtime error in the program. It disturbs the normal flow of the program. An example for an exception is division by 0. When a Python script encounters a situation that it cannot deal with, it raises an exception. An exception is a Python object that represents an error.

If the exception object is not caught and handled properly, the interpreter will display an error message. If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display appropriate messages for taking corrective actions. This task is known as exception handling.

10.1 Built-in Exceptions

Python has a collection of built-in exception classes. If the runtime error belongs to any of the pre-defined built-in exception, it will throw the object of the appropriate exception. The following Table 10.1 gives a detailed explanation of built-in exceptions and the situation in which they are invoked.

Table 10.1: Built-in Exceptions

Exception Name	When it is raised
AssertionError	Raised in case of failure of the Assert statement.
AttributeError	Raised in case of failure of attribute reference or assignment.

Scanned with OKEN Scanner

Exception Name	When it is raised
EnvironmentError	Base class for all exceptions that occur outside the Python environment.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
Exception	Base class for all exceptions
ImportError	Raised when an import statement fails.
IndentationError	Raised when indentation is not specified properly.
IndexError	Raised when an index is not found in a sequence.
IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
KeyboardInterrupt	Raised when the user interrupts Program execution, usually by pressing Ctrl+c.
KeyError	Raised when the specified key is not found in the dictionary.
LookupError	Base class for all lookup errors.
NameError	Raised when an identifier is not found in the local or global namespace.
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.
OSError	Raised for operating system-related errors.
RuntimeError	Raised when a generated error does not fall into any category.
SyntaxError	Raised when there is an error in Python syntax.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
ArithmeticError	Base class for all errors that occur for numeric calculation.
AssertionError	Raised in case of failure of the Assert statement
AttributeError	Raised in case of failure of attribute reference or assignment.
Exception	Base class for all exceptions
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
EnvironmentError	Base class for all exceptions that occur outside the Python environment.
FloatingPointError	Raised when a floating point calculation fails.

Scanned with OKEN Scanner

Exception Name	When it is raised
IndentationError	Raised when indentation is not specified properly.
ImportError	Raised when an import statement fails.
IndexError	Raised when an index is not found in a sequence.
IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
KeyboardInterrupt	Raised when the user interrupts Program execution, usually by pressing Ctrl+c.
NameError	Raised when an identifier is not found in the local or global namespace.
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
OSError	Raised for operating system-related errors.
RuntimeError	Raised when a generated error does not fall into any category.
StopIteration	Raised when the next() method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
SyntaxError	Raised when there is an error in Python syntax.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
ZeroDivisonError	Raised when division or modulo by zero takes place for all numeric types.

10.2 Handling Exceptions

Python uses a keyword `try` to prepare a block of code that is likely to cause an error and throw an exception. An `except` block is defined which catches the exception thrown by the `try` block and handles it. The `try` block can have one or more statements that could generate an exception. If anyone statement generates an exception, then the remaining statements in the block are skipped and execution jumps to the `except` block that is placed next to the `try` block.

The `except` block can have more than one statement and if the except parameter matches with the type of the exception object, then the exception is caught and statements in the `except` block will be executed. Every `try` block should be followed by atleast one `except` statement.

Scanned with OKEN Scanner

10.2.1 try.... except

The following gives the syntax of try....except statement.

Syntax

```
try:
    suite

except Exception1:
    Exception1_suite#Executes when Exception1 occurs.

except Exception2:
    Exception2_suite#Executes when Exception2 occurs.

else:
    else_suite #Executes if there is no exception in the try
block.
```

When an exception occurs inside a try block, it will go to the corresponding except block. If no exception is raised inside a try block then after the try block, the statements in the else block is executed. The else block can also be considered a place to put the code that does not raise any exception.

Example Program 1

```
#Exception Handling
try:
    a=int(input("First Number:"))
    b=int(input("Second Number:"))
    result=a/b
    print("Result=",result)
except ZeroDivisionError:
    print("Division by Zero")
else:
    print("Successful Division")
```

Output

```
First Number:10
Second Number:0
Division by Zero
```

In the above example, the second number is 0. Since division by zero is not possible, an exception is thrown and the execution goes to the except block. All the rest of the statements is bypassed in the try block. Hence the output is displayed as Division by Zero.

Scanned with OKEN Scanner

Example Program 2

```
#Exception Handling
try:
    a=int(input("First Number:"))
    b=int(input("Second Number:"))
    result=a/b
    print("Result=",result)
except ZeroDivisionError:
    print("Division by Zero")
else:
    print("Successful Division")
```

Output

```
First Number:20
Second Number:10
Result= 2
Successful Division
```

In the above example exception is not thrown and hence all the statements in the `try` block is executed. After executing the `try` block, the control passes on to the `else` block. Hence the print statement in the `else` block is executed.

10.2.2 except clause with no Exception

We can use except statement with no exceptions. This kind of try-except statement catches all the exceptions. Since it catches all the exceptions, it will not help the programmer to exactly identify what is the cause for the error occurred. Hence it is not considered as a good programming practice. The following shows the syntax for except clause with no exception.

Syntax

```
try:
    suite

except:
    Exception_suite #Executes when whatever Exception occurs.

else:
    else_suite #Executes if there is no exception in the try
block.
```

Scanned with OKEN Scanner

Example Program

```
#Exception Handling
try:
    a=int(input("First Number:"))
    b=int(input("Second Number:"))
    result=a/b
    print("Result=",result)
except:
    print("Error Occured")
else:
    print("Successful Division")
```

Output

```
First Number: 2
Second Number: 0
Error Occurred
```

10.2.3 except clause with multiple Exceptions

This is used when we want to give multiple exceptions in one `except` statement. The following shows the syntax of the `except` clause with multiple exceptions.

Syntax

```
try:
   suite

except(Exception1[, Exception2[,...ExceptionN]]]):

   Exception_suite #Executes when whatever Exception specified
occurs.

else:
   else_suite #Executes if there is no exception in the try
block.
```

Example Program

```
#Exception Handling
try:
    a=int(input("First Number:"))
    b=int(input("Second Number:"))
    result=a/b
    print("Result=",result)
except(ZeroDivisionError,TypeError):
    print("Error Occured")
```

Scanned with OKEN Scanner

```
    else:
        print("Successful Division")
```

Output

```
First Number:10
Second Number:0
Error Occurred
```

10.2.4 try...finally

A finally block can be used with a `try` block. The code placed in the `finally` block is executed no matter exception is caused or caught. We cannot use `except` clause and `finally` clause together with a `try` block. It is also not possible to use `else` clause and `finally` together with a `try` block. The following gives the syntax for a `try...finally` block.

Syntax

```
try:
    suite

finally:
    finally_suite #Executed always after the try block.
```

Example Program 1

```
#Exception Handling
try:
    a=int(input("First Number:"))
    b=int(input("Second Number:"))
    result=a/b
    print("Result=",result)
finally:
    print("Executed Always")
```

Output

```
First Number:20
Second Number:0
Executed Always
Traceback (most recent call last):
  File "main.py", line 5, in <module>
    result=a/b
ZeroDivisionError: integer division or modulo by zero
```

In the above example, an error occurred in the try block. It is caught by the Python's error handling mechanism. But the statement in the finally block is executed even though an error has occurred.

Scanned with OKEN Scanner

Example Program 2

```
#Exception Handling
try:
    a=int(input("First Number:"))
    b=int(input("Second Number:"))
    result=a/b
    print("Result=",result)
finally:
    print("Executed Always")
```

Output

```
First Number:10
Second Number:5
Result= 2
Executed Always
```

In the above example there was no exception thrown. Hence after the try block, the finally block is executed.

10.3 Exception with Arguments

It is possible to have exception with arguments. An argument is a value that gives additional information about the problem. This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number and an error location. The syntax of exception with arguments is as follows.

Syntax

```
try:
   suite

except ExceptionType, argument:
   except_suite #Executed when exception occurs.
```

Example Program

```
def display(a):
  try:
        return int(a)
  except ValueError,argument1:
print("Argument does not contain numbers", argument1)
#Function Call
display("a")
```

Scanned with OKEN Scanner

Output

```
Argument does not contain numbers invalid literal for int() with
base 10: 'a'
```

In the above example, the function `display` is called with string arguments. But the actual function is defined with integer parameter. Hence an exception is thrown and it is caught by the `except` clause.

10.4 Raising an Exception

We have discussed about raising built-in exceptions. It is also possible to define a new exception and raise it if needed. This is done using the `raise` statement. The following shows the syntax of `raise` statement. An exception can be a string, a class or an object. Most of the exceptions that the Python raises are classes, with an argument that is an instance of the class. This is equivalent to `throw` clause in Java.

Syntax

```
raise [Exception [, argument [, traceback]]]
```

Here, Exception is the type of exception (for example, IOError) and `argument` is a value for the exception argument. This argument is optional. The exception argument is `None` if we does not supply any argument. The argument, `traceback`, is also optional. If this is used, then the `traceback` object is used for the exception. This is rarely used in programming.

In order to catch the exception defined using the `raise` statement, we need to use the `except` clause which is already discussed. The following code shows how an exception defined using the `raise` statement can be used.

Example Program

```
# Raising an Exception
a=int(input("Enter the parameter value:"))
try:
  if a<=0:
    raise  ValueError("Not a Positive Integer")
except ValueError as err:
    print(err)
else:print("Positive Integer=",a)
```

Output

```
Enter the parameter value:-9
Not a Positive Integer
```

10.5 User-defined Exception

Python also allows us to create our own exceptions by deriving classes from the standard built-in exceptions. In the try block, the user-defined exception is raised and caught in the except block. This is

Scanned with OKEN Scanner

useful when we need to provide more specific information when an exception is caught. The following shows an example for user-defined exception.

Example Program

```
# define Python user-defined exceptions
class Error(Exception):
    """Base class for other exceptions"""
    pass
class ValueTooSmallError(Error):
    """Raised when the input value is too small"""
    pass
class ValueTooLargeError(Error):
    """Raised when the input value is too large"""
    pass
# Main Program
number = 10
while True:
    try:
        i_num = int(input("Enter a number: "))
        if i_num < number:
            raise ValueTooSmallError
        elif i_num > number:
            raise ValueTooLargeError
        break
    except ValueTooSmallError:
        print("This value is too small, try again!")
    except ValueTooLargeError:
        print("This value is too large, try again!")
print("Congratulations! You guessed it correctly.")
```

Output

```
Enter a number: 13
This value is too large, try again!
Enter a number: 9
This value is too small, try again!
Enter a number: 10
Congratulations! You guessed it correctly.
```

Here, we have defined a base class called Error. The other two exceptions (ValueTooSmallError and ValueTooLargeError) that are actually raised by our program are derived from this class. This is the standard way to define user-defined exceptions in Python programming, but you are not limited to this way only.

Scanned with OKEN Scanner

10.6 Assertions in Python

An assertion is a checking in Python that can be turned on or off while testing a program. In assertion, an expression is tested and if the result is false, an exception is raised. Assertions are done using the assert statement. The application of assertion is to check for a valid input or for a valid output. The following shows the syntax for assert statement.

When Python interpreter encounters an assertion statement, it evaluates the accompanying expression. If the expression is evaluated to `False`, Python raises an AssertionError exception. AssertionError exceptions can be caught and handled like any other exception using the `try-except` statement, but if not handled, they will terminate the program and produce a traceback.

Example Program

```
#Python Assertions
def sum(a,b):
 sum=a+b
 assert(sum>0), "Too low value"
 return(sum)
a=int(input("First Number:"))
b=int(input("Second Number:"))
print(sum(a,b))
```

Output

```
First Number:-8
Second Number:-2
Traceback (most recent call last):
  File "main.py", line 8, in <module>
    print sum(a,b)
  File "main.py", line 4, in sum
    assert(sum>0), "Too low value"
AssertionError: Too low value
```

10.7 Conclusion

This Chapter covers exception handling mechanisms, built-in exceptions, how to handle exceptions using try....except, except clause with no exception, except clause with multiple exceptions and try... finally statements. It also covers exception with arguments, how to raise an exception, user-defined exception and assertions in Python.

10.8 Review Questions

1. Briefly explain exception handling in Python.
2. What are built-in exceptions?
3. Explain try....except statement in Python.

Scanned with OKEN Scanner

11 Regular Expressions

CONTENTS

Regular expressions (called REs, or regexes, or regex patterns) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the `re` module. A regular expression helps to match or find other strings or sets of strings, using a specialized syntax held in a pattern. They are widely used in UNIX world.

The `re` module raises the exception `re.error` if an error occurs while compiling or using a regular expression. There are various characters, which have special meaning when they are used in regular expression. While dealing with regular expressions, we use raw strings as r'expression'.

11.1 The *match()* function

The purpose of *match()* function is to match regular expression pattern to string with optional flags. For using this function, the `re` module needs to be imported. The following shows the syntax of *match()* function.

Scanned with OKEN Scanner

Syntax

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

where `pattern` is the regular expression to be matched, `string` is searched to match the pattern at the beginning of string, `flags` are modifiers specified using bitwise OR(|). The modifiers will be explained in Section 11.4. The re.match() function returns a match object if the matching was success and returns None, if matching was a failure.

Two functions group(n) and groups() of match object are used to get matched expression. The group(n) method returns entire match (or specific subgroup n) and groups() method returns all matching subgroups in a tuple or empty if there weren't any matches.

Example Program

```
#Example Program for match() function
import re
line = "Python Programming is fun";
matchObj = re.match( r'fun', line, re.M|re.I)
if matchObj:
   print("match --> matchObj.group() : ", matchObj.group())
else:
   print("No match!!")
```

Output

```
No match!!
```

The match() function always checks the beginning of the string. Even though the keyword "fun" is there in the string it is not considered as a match since it does not appear at the beginning of the string. Consider the below example.

Example Program

```
#Example Program for match() function
import re
line = "Python Programming is fun";
matchObj = re.match( r'python', line, re.M|re.I)
if matchObj:
   print("match --> matchObj.group() : ", matchObj.group())
else:
   print("No match!!")
```

Output

```
match --> matchObj.group() :  Python
```

In the above example the word "python" is found as a match since it appears at the beginning of the string.

Scanned with OKEN Scanner

11.2 The *search()* function

The *search()* function searches for first occurrence of regular expression pattern within a string with optional flags. The following shows the syntax for *search()* function.

Syntax

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

where `pattern`, `string` and `flags` have the same meaning as that of `match()` function. The re.search() function returns a match object if the matching was success and returns None, if matching was a failure.

The difference between `match()` and `search()` is that `match()` checks for a match only at the beginning of the string, while `search()` checks for a match anywhere in the string. The following shows an example between `match()` and `search()` functions.

Example Program

```
#Example Program for difference between match() & search()
function
import re
line = "Python Programming is fun";
matchObj = re.match( r'fun', line, re.M|re.I)
if matchObj:
   print("match --> matchObj.group() : ", matchObj.group())
else:
   print("No match!!")
searchObj = re.search( r'fun', line, re.M|re.I)
if searchObj:
   print("search --> searchObj.group() : ", searchObj.group())
else:
   print("Nothing found!!")
```

Output

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

In the above example, match() function returns nothing while search() function searches the entire string to find a match.

11.3 Search and Replace

Search and replace is done in Python with the help of `sub` method in `re` module. The following shows the syntax of `sub` method.

Scanned with OKEN Scanner

Syntax

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

This method replaces all occurrences of the regular expression pattern in string with `repl`, substituting all occurrences. If value for `max` is provided, it will replace only the number of occurrences specified in `max`. This method returns modified string. The following example shows the application of `sub` method.

Example Program

```
import re
zipcode = "2004-959-559 # This is zipcode"
# Remove anything other than digits
num = re.sub(r'\D', "", zipcode)
print("Zip Code: ", num)
```

Output

```
Zip Code:  2004959559
```

11.4 Regular Expression Modifiers: Option Flags

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. We can provide multiple modifiers using exclusive OR (|), as shown in the previous examples. The following Table 11.1 shows various modifiers and their description available in the `re` module.

Table 11.1: Modifiers and Descriptions in re Module

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.
re.X	Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker.

11.5 Regular Expression Patterns

All characters match themselves except for control characters, (+ ? . * ^ $ ( ) [ ] { } | \). We can escape a control character by preceding it with a backslash. Table 11.2 shows the regular expression patterns and their descriptions.

Scanned with OKEN Scanner

Table 11.2: Regular Expression Patterns and Descriptions

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.
[^...]	Matches any single character not 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.

Scanned with OKEN Scanner

\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.
\10	Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.

11.6 Character Classes

The Table 11.3 shows example of various character classes and their description used in regular expression of Python.

Table 11.3: Example of Character Classes

Example	Description
[Pp]ython	Match "Python" or "python"
rub[ye]	Match "ruby" or "rube"
[aeiou]	Match any one lowercase vowel
[0-9]	Match any digit; same as [0123456789]
[a-z]	Match any lowercase ASCII letter
[A-Z]	Match any uppercase ASCII letter
[a-zA-Z0-9]	Match any of the above
[^aeiou]	Match anything other than a lowercase vowel
[^0-9]	Match anything other than a digit

11.7 Special Character Classes

Table 11.4 shows example of various special character classes and their description used in regular expression of Python.

Table 11.4: Example of Special Character Classes

Example	Description
.	Match any character except newline
\d	Match a digit: [0-9]
\D	Match a nondigit: [^0-9]
\s	Match a whitespace character: [\t\r\n\f]
\S	Match nonwhitespace: [^ \t\r\n\f]
\w	Match a single word character: [A-Za-z0-9_]
\W	Match a nonword character: [^A-Za-z0-9_]

Scanned with OKEN Scanner

11.8 Repetition Cases

Table 11.5 shows example of various repetition cases and their description used in regular expression of Python.

Table 11.5: Repetition cases with examples

Example	Description
ruby?	Match "rub" or "ruby": the y is optional
ruby*	Match "rub" plus 0 or more ys
ruby+	Match "rub" plus 1 or more ys
\d{3}	Match exactly 3 digits
\d{3,}	Match 3 or more digits
\d{3,5}	Match 3, 4, or 5 digits

11.9 findall() method

The findall() method searches all patterns and returns a list. The following shows an example for findall() method.

Example Program

```
import re
a = "hello 234789 world 63678 ok 1 ok 1115 alpha 88 beta 999g"
print(re.findall("\d+", a))
```

Output

```
['234789', '63678', '1', '1115', '88', '999']
```

In the above example, \d is any numeric characters ie 0,1,2,3,4,5,6,7,8,9, "+" represents one or more previous character. Hence the regular expression searches for only numeric characters.

11.10 compile() method

The compile method compiles a regular expression pattern into a regular expression object, which can be used for matching using its match() and search() methods. The syntax of compile is as follows.

```
re.compile(pattern, flags=0)
```

The expression's behaviour can be modified by specifying a flags value. Values can be any of the following variables, combined using bitwise OR (the | operator). The sequence

```
prog = re.compile(pattern)
result = prog.match(string)
```

is equivalent to `result = re.match(pattern, string)`

Similarly the sequence

```
prog = re.compile(pattern)
```

Scanned with OKEN Scanner

```
result = prog.search(string)
```

is equivalent to `result = re.search(pattern, string)`

Using re.compile() and saving the resulting regular expression object for reuse is more efficient when the expression will be used several times in a single program.

Example Program

```
import re
file = open("Abc.txt","r")
text = file.readlines()
file.close()
# compiling the regular expression:
keyword = re.compile(r'a ')
# searching the file content line by line:
for line in text:
    if keyword.search (line):
        print(line)
```

Output

```
#Only lines containing a is displayed
Ram is a boy
Geeta is a girl
```

11.11 Solved Lab Exercises

1. Given a string like, "1, 3-7, 12, 15, 18-21", produce the list [1,3,4,5,6,7,12,15,18,19,20,21]

Program

```
import re
listString = []
string = "1,4,6-9,12-19"
def matches(l):
    start, end = l.groups()
    return ','.join(str(i) for i in range(int(start),
int(end)+1))
listString = (re.sub('(\d+)-(\d+)', matches, string)).split(',')
print(listString)
```

Output

```
['1', '4', '6', '7', '8', '9', '12', '13', '14', '15', '16',
'17', '18', '19']
```

Scanned with OKEN Scanner