PYTHON PROGRAMMING
MCA DISTANCE EDUCATION
COURSE MATERIAL

Bama Srinivasan

JULY 2021

Contents

Chapter 1

PYTHON BASICS

Learning Objectives

- Introduce the basics of Python Programming
- Familiarize with the syntax of Python programming
- Solve simple problems with Python programming

1.1 Introduction to Python Programming

Python was first created by Guido van Rossum in early 1990. It has gained popularity in the recent years because of the simple syntax and easy to understand language. It is modular, has several programming features embedded in scientific applications and above all it is Free and Open source. This makes it easier for the interested individuals to work on the specific libraries as modules and release those. The major modules used like tensorflow, numpy, scipy, matplotlib are developed on the top of python. Many organizations like Google and NASA use python for their applications.

Why python?

The name python comes from the 1970s British comedy series *Monty Python's Flying Circus*.

1.2 Python Interpreter and Interactive mode

To get started with python, you can start with IDLE, which comes by default when you install Python. IDLE is an Integrated Development Environment (IDE). There are many IDEs available nowadays (Section 1.13), but for beginners it is better to start with IDLE. IDE usually consits of program editor, translator and debugger.

- Editor is used to create and modify program with ease. For python, the most important aspect is indentation, which is included in most IDEs.
- Translator helps to translate the text in editor to machine readable form so as to execute the code.
- Debugger helps in the execution of the code and to find the errors in programming structure.

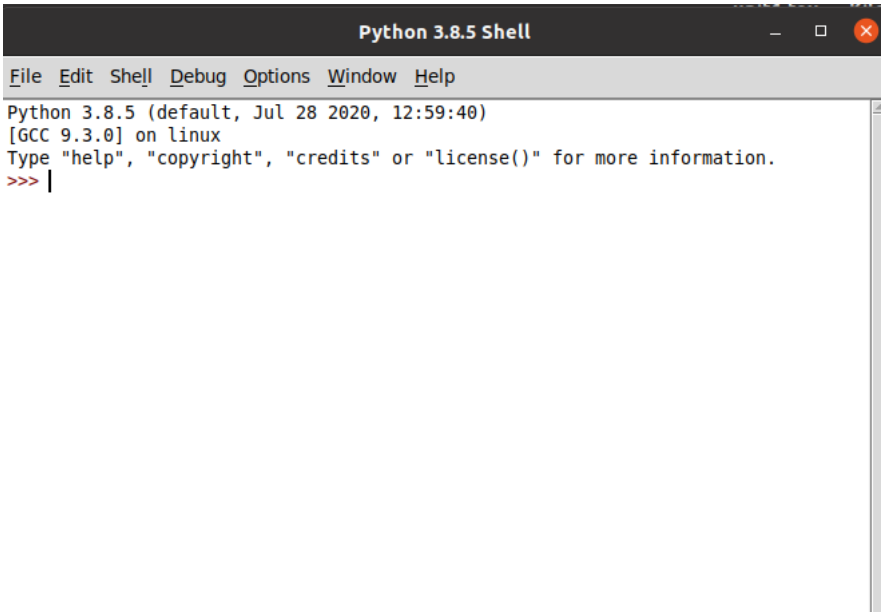


Figure 1.1: Python Shell

1.2.1 Interactive Shell

One of the basic features in Python is the interactive shell, that come with IDLE environment. You can interactively work with code and see instant output through interactive shell. Example of this shell is shown in Figure 1.1.

The shell can be used as a calculator. Try out these in your shell.

```
\texttt{
>>> 75*4 \\
300\\
>>> "hello" \\
'hello'\\
>>> "hello" + "world"\\
'helloworld'\\
>>> \\
}
```

The first line multiplies 75*4 with the output as 300. "+" concatenates the two strings "hello" and "world" to give the output as 'hello world'.

1.2.2 Execution of python program

While the interactive shell gives an immediate output after an enter statement, for programming it is always a good practice to use an IDE and save the program. To write as a python program, open IDLE or any python IDE of your choice. From IDLE, click File–> New. An 'untitled' window appears as shown in Figure 1.2. In this window, the program can be written and executed. Refer to Figure 1.3 and enter these lines in your editor. Save as 'hello.py'. The extension of any python program is '.py'. Here, a, b, c are variables and the strings of "hello", space " " , and "world" are assigned. The print statement displays "hello world" in the screen. This program can be executed by clicking "Run" from the window. The output should be

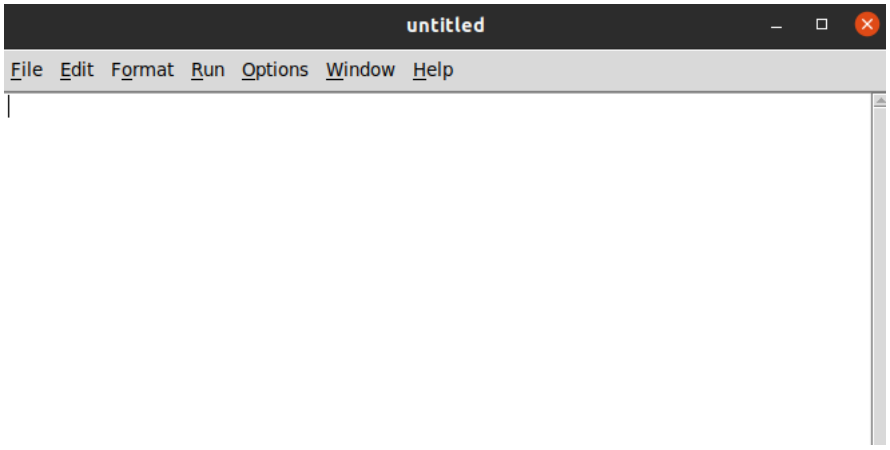


Figure 1.2: Area to write python program in IDE

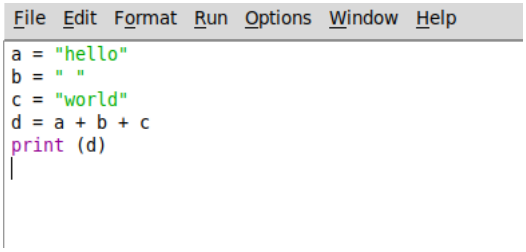


Figure 1.3: IDE with python program

visible in the IDLE python interactive shell. Alternately, any python program can be executed from the terminal by entering the command `python3 <filename.py>` from the directory where the file is residing. For this example, the command is `python3 hello.py`.

Nowadays, python version used by default is 3 and above. So, python3 is used. If python 2.7 is used, 3 can be dropped

1.2.3 Output and Input Statements

The output statment is given using **`print`** statement. For example, if **`print`** `''helloworld''` is given, the output results in `helloworld`.

To get the input from the user, **`input`** keyword is used. To restrict only integers **`int(input)`** can be used. Example of input and output statements are given below.

```
print (''This course is on python programming'')
n = int(input(''Enter the present year: ''))
print (''The present year is'', n)
```

OUTPUT

```
Enter the present year: 2020
This course is on python programming
Enter the present year: 2020
The present year is 2020
```

1.3 Variables and Identifiers

(*Identifier identifies the Variables. In this context, for this course Variables and Identifiers are interchangeably used.*)

Variables as the name specify can be varied over the values. You can assume variables as some holder or bucket that hold the values and are changeable.

1.3.1 Correct usage of Variables

In Python, Variables always start with:

- letters 'A' to 'Z' or a set of characters from these
- leters 'a' to'z' or a set of characters from these
- combination of uppercase or lowercase letters
- _(underscore) followed by 0(zero) or more letters
- _(underscore)followed by digits(0 to 9)

Variable names cannot have special characters.

One of the beauty with Python is that the variables need not be declared at the beginning of the program. A value can be assigned automatically.

Examples of variable declaration are shown below. Practice these in your interactive shell.

```
>>> x = 100
>>> num2 = 200
>>> x+num2
300
>>> y = "hello"
>>> Cap = "WORLD"
>>> y+Cap
'helloWORLD'
```

Here, the variables are "x,num2,y and Cap". 100 is assigned to 'x' and 200 to num2. Adding the variables gives the result of 300. The string of "hello" is assigned to 'y' and "WORLD" to 'Cap'.

A few more examples are shown below.

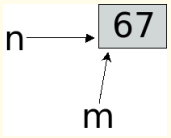
```
example1 = 10
example_1 = 10
_example1 = 10
ex_ample = 10
Xample = 10
eXamPLe = 10
example_ = 10
_example_= 10
```

Now that some value is assigned to a variable, what really happens in the machine when we assign a value to the variable?

Python is an object oriented language (you will see more details of it in Chapter 5) and all values are objects itself. When you create a variable say `n=67`, an integer object of 67 is created and then assigned to n. In other words, n points to the value of 67. Now consider these two lines.

```
1. n = 67
2. m = n
```

The first line indicates n pointing to 67. The second line has two variables m and n, with m assigning to the variable n, which itself is a variable. Here, there are not two objects created, but the variable m points to the value of n, which is equal to 67 as shown below.



1.3.2 Incorrect Usage of Variables

When using variables, be careful about the keywords and special characters. Some examples where variables cannot be used as special characters are shown below.

```
>>> 3eg=5
SyntaxError: invalid syntax
>>> ex-ample = 5
SyntaxError: cannot assign to operator
>>> ex ample = 5
SyntaxError: invalid syntax
>>> !example = 6
SyntaxError: invalid syntax
>>> and = 5
SyntaxError: invalid syntax
>>> assert = 6
SyntaxError: invalid syntax
>>> from = 67
SyntaxError: invalid syntax
>>> global = 8
SyntaxError: invalid syntax
>>> lambda = 8
SyntaxError: invalid syntax
>>>
```

Note: Variables should be readable. Do not give any names that are bizarre and irrelevant.

1.4 Arithmetic Operators and Precedence

The standard mathematical operations of addition, subtraction, multiplication and division are supported in python. In addition, the truncated division, modulo and exponential are well supported. While all these operations are binary, the unary operator of negation (negative of a positive value) is also included. The symbols are shown in Table 1.1. Examples are shown

Table 1.1: Operators in Python

Name	Operator	Example
Addition	+	87+67
Subtraction	-	87-67
Multiplication	*	87*67
Division	/	87/67
Truncated Division	//	87//67
Modulo	%	87 % 5
Exponential	**	5**2
Negative value	−	-87

below. Practice these in your Python Shell.

```
>>> n = 87
>>> m = 67
>>> addition = m+n
>>> addition
154
>>> subtraction = m − n
>>> subtraction
−20
>>> multiplication = m∗5
>>> multiplication
335
>>> division = m/2
>>> division
33.5
>>> truncated_division = m//2
>>> truncated_division
33
>>> modulo_op = m%2
>>> modulo_op
1
>>> exp_op = 5∗∗2
>>> exp_op
25
>>> neg_val = −m
>>> neg_val
−67
>>>
```

The operator precedence is given as follows.
exponential (**) >negation (-) > multiplication (*) >division (/) >truncated division (//) > modulo (%) >addition (+) subtraction (-)

Try these and check for yourself

```
1. 6-4+2
2. 8**2-5*3+4
3. -10*5
4. 4+6**6//2
5. 10**6//2
6. (2**3)**3+5//2-1
```

1.5 Boolean Expressions and Operations

Python employs two values **True** and **False** for the evaluation of Boolean expressions. The expression can be in the form of relational, membership or Boolean operators.

1.5.1 Relational Operators

The relational operators compare two objects. These are given in Table 1.2. Numerical comparison is straight forward. But when characters and strings are used, the alphabetical order is used to compare. For example, since the character of b comes before d, we can write in python as ''b''<''d''> or ''cat''<''dog''. Here, the first letter of c is before d and therefore evaluates to **True** value.

Table 1.2: Relational Operators

Name	Relational Operator	Example	Boolean value
Equal	==	5 == 5	True
Not equal	!=	8 !=5	False
Less than	<	5 < 7	True
Less than or equal to	<=	5 <=5	True
Greater than	>	5 > 8	False
Greater than or equal to	>=	100>=9	True

Examples employing these operators are given below. Practice these in your shell.

```
>>> a = 5
>>> b = 6
>>> a == b
False
>>> a<b
True
>>> a<=b
True
>>> b>a
```

```
True
>>> b>=a
True
>>> a!=b
True
>>> "cat" == "cat"
True
>>> "cat" != "dog"
True
>>> "cat" < "dog"
True
```

1.5.2 Membership Operators

Python provides an elegant method to determine whether a member is in the set of values using keyword **in** and **not in**. For example, take the even integers less than 10. This can be represented as a list [0,2,4,6,8]. To check whether 2 and 5 are in this list, you can use these keywords as shown below.

```
>>> even = [0,2,4,6,8]
>>> 2 in even
True
>>> 5 not in even
True
>>> 5 in even
False
>>> 2 not in even
False
>>> 100 in even
False
```

1.5.3 Boolean Operators

The operators of *not*, *and* and *or* can be used in Python. The evaluation of these operators are:

- True and True = True
- True and False = False
- False and False = False
- True or True = True
- True or False = True
- False or False = False
- not True = False
- not False = True

A few examples are shown below. Practice all types of combinations similar to these.

```
>>> True and False
False
>>> True or False
True
```

```
>>> not False
True
>>> (5<10) and (100>6)
True
>>> ('cat'<'dog') and ('dog' < 'zebra')
True
>>> not (('cat'<'dog') and ('dog' < 'zebra'))
False
```

1.6 Types and Values

"A data type is a set of values and a set of operators that may be applied to those values". Example: Integer datatype involves set of integers, operators like addition, subtraction, multiplication and division which could be applied appropriately. In python there are a set of built-in data types. These are:

- Integers - represented as 'int'
- Float - these are decimal numbers represented as float
- Strings - represented as 'str'

The representation of data types is essential because these act as fool proof method in programming. For example, assume you are entering these in python interpreter.

```
text1 = ''this course is on python programming''
num = text1/2
```

As a human you know that the text cannot be divided. But how does a machine know? To encounter these types of problems, data types are used. For the above codes, 'text1' is a string type variable - anything that is within quotation is string. This cannot be divided by two, which throws an error as:
`TypeError: unsupported operand type(s) for /: 'str'and 'int'.`

By default python understands the variable as integer and float. For example, if you assign the variable as `n = 10`, 'n' is assigned with the integer value of 10. If you assign `n=10.0`, 'n' is assigned with the decimal value of 10.0. You need not declare explicitly as integer and float. If you assign `n=''10''`, then n is considered as string. In this case, arithmetic operations cannot be performed. To find out the datatype of the value, 'type' is used. Try these in your python shell and check for yourself.

```
>>> n = 10
>>> type(n)
<class 'int'>
>>> n=10.0
>>> type(n)
<class 'float'>
>>> n="10"
>>> type(n)
<class 'str'>
>>> n/2
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
```

```
    n/2
TypeError: unsupported operand type(s) for /: 'str' and 'int'
>>>
```

Data types can be converted from one form to another. For example, an integer can be converted to float or vice-versa. Consider the following piece of code. Here, 'n' is assigned an integer value of 15, then 'm' is assigned to the float value of 'n', which gives the output of 15.0. Here, integer is converted to float.

```
>>> n = 15
>>> type(n)
<class 'int'>
>>> m = float(n)
>>> m
15.0
>>> type(m)
<class 'float'>
>>>
```

Now, consider the following code. Here, 'n' is assigned with a value of 15.8, which is of float type. On conversion to integer as assigned in the variable 'm', the value is 15. When you convert float to integer, the decimal values are ignored.

```
>>> n = 15.8
>>> type(n)
<class 'float'>
>>> m = int(n)
>>> m
15
```

If you want to round off the value to the next integer, you have to use the keyword of round. For example, to round off the value of `n = 15.82` to the nearest integer, use `int(round(n,0))`. Here, 0 indicates the number of places to be rounded and 'int; indicates the conversion of float to integer type.

The conversion of integer to string and string to integer can be done in a similar manner. The following example illustrates the use of conversion from integer to string and string to float value.

Integer to String conversion

```
>>> m = 15
>>> type(m)
<class 'int'>
>>> n = str(m)
>>> type(n)
<class 'str'>
```

String to float (*Note here that there is an error when converting a string value of 15.2 to integer*).

```
>>> n = "15.2"
>>> type(n)
<class 'str'>
```

```
>>> m = int(n)
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    m = int(n)
ValueError: invalid literal for int() with base 10: '15.2'
>>> m = float(n)
>>> m
15.2
```

1.7 Control Statements (Conditionals)

Control statements prescribe a condition in the form of selective and iterative operations. Selective is usually denoted by *if - else statements* and iterative includes *while* and *for loops*.

Note: *When you are working with control statements and functions, be careful about indentation (spaces) in python.*

1.7.1 If - else statements

If statement is a decision control statement based on the value of some arithmetic or boolean expressions. The syntax is given by:

```
  if (expression):
    block 1
    block 2
  else:
    block 3
    block 4
```

The expression should be within simple brackets and with the indentation space, the appropriate statements are written. Example: Assume you have a class at 8.30 a.m. on mondays, wednesdays and fridays. Write a program to check whether there is a class at 8.30 a.m. on any particular day.

```
day = input("Enter the day as mon, tues, wed, thurs, fri,
                                                sat, sun")
if (day == "mon" or day == "wed" or day == "fri"):
    print("You have a class at 8.30 a.m. Get ready")
else:
    print("You do not have a class at 8.30 a.m.")
```

```
OUTPUT
Enter the day as mon, tues, wed, thurs, fri, sat, sun: fri
You have a class at 8.30 a.m. Get ready
```

For multiple selective control statements, you can use *else if* statements, given by the following syntax.

```
  if (expression):
    block 1
    block 2
  elif (expression):
    block 3
    block 4
```

Extending the same example to other days, the code can be written as follows.

```
 day = input("Enter the day as mon, tues, wed, thurs,
                                          fri, sat, sun: ")
if (day == "mon" or day == "wed" or day == "fri"):
    print("You have a class at 8.30 a.m. Get ready")
elif (day == "tues"):
    print("You have a class at 11.20 a.m.")
elif (day == "thurs"):
    print("You have a class at 10.30 a.m.")
else:
    print("Either you have entered a wrong string or it is holiday")
```

A *nested if* statement includes two or more *if* statements. These are used to check multiple expressions within themselves. The syntax for nested-if statement is:

```
 if (expression 1):
    if (expression 2):
        if (expression 3):
            block 1
            block 2
```

Extending the above example, the user is asked for the input of course. If the course is MCA, then another condition is checked for day and print the class timings.

```
course = input("Enter your course: ")
day = input("Enter the day as mon, tues, wed, thurs,
                                  fri, sat, sun: ")
if (course == "MCA"):
    if (day == "mon" or day == "wed" or day == "fri"):
        print("You have a class at 8.30 a.m. Get ready!")
```

1.7.2 Iterative Control - Loops

To execute certain statements repeatedly, loops are used. Basically, iterative loops of *while* and *for* are used in python.

Iterative control statement provides a repeated execution of instructions. A *while* statement is an iterative control statement. In *while* loop, the statements are executed when a particular expression is true. The syntax of *while* loop is:

```
statement s1
while (condition):
    block 1
```

```
    block 2
statement s2
```

Here, statement 1 is the initial condition before the *while* condition. If the condition is true, blocks 1 and 2 gets executed. If the condition is false, statement 2 gets executed.

In this example, let us print the even numbers below or equal to the value of 10 using *while* loop.

```
#print even numbers below or equal to the value of 10
1. n = 0
2. while (n<=10):
3.     print (n, end = '\t')
# 't' indicates the space of tab separated while printing on screen
4.     n += 2
5. print ('\n')#newline
6. print("Even numbers less than or equal to 10 are printed")
```

OUTPUT
0 2 4 6 8 10
Even numbers less than or equal to 10 are printed

The first line initializes the value of 0 to n. The condition of $n < 10$ is checked within *while* loop. If the condition is true, the value is printed at line 3 and then increments by 2 at line 4. This repeats until the value of n equals 10. When the condition is false, the newline and the statement are printed at lines 5 and 6.

Let us write another program with *while* statement. Assume you are programming the microwave oven to heat a bowl of vegetables and tea. If vegetables, cooking time is 2 minutes and if tea, time is 40 seconds. Without importing time, we will just count down the time and print the end statement for each of it.

```
 dish = str(input("enter the item whether veg or tea: "))
if dish == "veg":
    n = 2*60
    time = n
    while (n>0):
        n = n - 1
        print (n,end=',')
    print ("\n")
    print("Veg is cooked in", time, "seconds")
elif dish == "tea":
    n = 40
    time = n
    while (n>0):
        n = n-1
        print (n,end=",")
    print ("\n")
    print ("Tea is heated in", time, "seconds")
```

```
enter the item whether veg or tea: tea
39,38,37,36,35,34,33,32,31,30,29,28,27,26,25,
24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,
9,8,7,6,5,4,3,2,1,0,
Tea is heated in 40 seconds
```

Try these.

1. Write a program to find the sum of first n integers.

2. Write a program to find the GCD of two given numbers.

Similar to the *while* loop, *for* loop also executes the statement if a statement is true. But, the increment or decrement value is available in a single line. The syntax is:

```
for variable in range(initial_value, final_value,
                      increment or decrement steps):
    block 1
    block 2
```

A program to list the odd numbers from 1 to 20 using *for* loop is given below.

```
#program to list odd numbers from 1 to 20
for i in range(1,20,2):
    print(i,end = " ")
print ()
print ("Odd numbers 1 to 20 are printed")
```

```
OUTPUT
1 3 5 7 9 11 13 15 17 19
Odd numbers 1 to 20 are printed
```

Here, the first line indicates the numbers are in the range 1 to 20 and should be incremented by 2. The second line prints the number each time in the loop with space in between. Once the loop is complete, both print statements are executed. An other example with strings is given below. Here, the number of occurrence of "l" in "classical" is determined.

```
#number of l in classical
word = "classical"
count = 0
for i in word:
    if i == "l":
        count += 1
print ("Number of l in classical is ", count)
```

```
OUTPUT
Number of l in classical is  2
```

1.7.3 Break and continue Statements

In some cases, certain programs need to be terminated and should not be going through the loop. In such cases, **break** statement is used. Sometimes you might encounter a case where the loop has to go through without doing anything. In those cases, **continue** is used. Examples for these two statements are given below.

In the following program, a **for** loop is constructed to loop from 1 to 10. The second line prints the number and if the value reaches 5, the program is terminated because of **break** statement.

```
#example of break statement
for i in range(1,10):
    print (i)
    if i == 5:
        break
```

```
OUTPUT
1
2
3
4
5
```

For the use of **continue** statement, the above program is slightly modified such that when the loop encounters 5 or 6, the values are not displayed. But the loop continues till the end.

```
#example of continue statement
for i in range(1,10):
    if i == 5 or i == 6:
        continue
    print (i)
```

```
OUTPUT
1
2
3
4
7
8
9
```

1.8 Function

A program should never have any repeated statements. Suppose you write a program with certain blocks and want to use that block again and again, function can be used. For instance, take the program of heating *veg* or *tea* using **while** loop. If you see the program carefully, two blocks are used and the other one for tea and one for veg, while the logic for both looks similar. Is there a case where you could simplify those? The answer is yes. We can use functions for these cases.

Use of function is done in two ways. First you have to define the function block and then call it. The syntax of function definition is given by:

```
def function_name(parameters):
    block 1
    block 2
```

Here, **def** is the keyword and should always be started with this word when you start writing a function. The particular name of the function is indicated by 'function_name', which can be declared by the user. The parameters are variables that are entered within round brackets and these are finite in number. The function name and parameters are associated with function call statement, which is given by:

```
function_name (parameters)
```

The same function name should be used and the number of parameters should be identical with the number of variables used in the function block. An example to add two numbers using function is given below. The function name is addition and the parameters in the function block are x and y. Here, two numbers with variables num1 and num2 are assigned the values of 100 and 200, respectively (line 6,7). In line 8, the function with name addition is called with two paramenters num1 and num2. This line indicates the values of 100 and 200 are the parameters for addition function.

In line 2, the function block of addition begins with two parameters x and y. The values of $num1$ is passed to the variable of x and $num2$ to y. Hence, x and y point to the values of 100 and 200, respectively. In line 3, total is computed with these values and in line 4 **print** statement is executed.

```
1. #example of function for addition
2.  def addition(x,y):
3.      total = x+y
4.      print ("Addition of two numbers ", total)
5.
6. num1 = 100
7. num2 = 200
8. addition(num1,num2)
```

```
OUTPUT
Addition of two numbers  300
```

Note: Always, write function blocks at the beginning of the program and call the particular function after that.

The same example of heating tea and veg using microwave oven is given below, but using function. Here, the parameters are the dish represented as string and time as integers.

```
#example of function
def heat(item,time):
    n = time
    while (time>0):
        time = time -1
```

```
        print (time, end = ',')
    print()
    print(item, "is ready in ", n, "seconds")

dish = str(input("enter the item whether veg or tea: "))
if dish == "veg":
    heat ("veg",200)
elif dish == "tea":
    heat ("tea",40)
```

```
OUTPUT
enter the item whether veg or tea: veg
199,198,197,196,195,194,193,192,191,190,189,..0,
veg is ready in  200 seconds

enter the item whether veg or tea: tea
39,38,37,36,35,34,33,32,31,30,29,28,...,1,0,
tea is ready in  40 seconds
```

Sometimes, the value that the function computes should be returned to the main program. In such cases, the keyword of **return** is used. The syntax is:

```
def function_name(parameters):
    block 1
    computed_value = {operation on parameters}
    return {value from function}
```

The called function in this can be represented as:

```
print function_name (parameters)
```

Extending the example of addition, here is a program performing addition and subtraction of two numbers in two separate functions.

```
def addition(n1,n2):
    total = n1+n2
    return total

def subtraction(n1,n2):
    sub = n1-n2
    return sub

x = 78
y = 10
print ("Addition of two numbers ")
print (addition(x,y))
print ("Subtraction of two numbers")
print (subtraction(x,y))
```

```
OUTPUT
Addition of two numbers
```

```
88
Subtraction of two numbers
68
```

In the above example, the values of 78 and 10 are available in the main program. But the variables within a particular function is accessible within that function itself. In other words, it is called as local variable. The main program cannot access this variable. In the above example, the variable of total is limited only to the function of addition and sub to subtraction. You cannot use total or sub in main program. You cannot also use total in subtraction function and sub in addition function. Let us see what happens when we use total in main function.

```
  OUTPUT
Traceback (most recent call last):
File "fn_arithmetic.py", line 15, in <module>
print (total)
NameError:  name 'total' is not defined
```

The program throws an error that the variable 'total' is not declared although it has been used in the function of addition. Thus, variable of total in this case is local in nature.

Suppose you want the program to recognize the total variable in the main program, you can do it by declaring it as global. Refer to the following example for the usage of global.

```
 def addition(n1,n2):
    global total
    total = n1+n2
    return total

def subtraction(n1,n2):
    global sub
    sub = n1-n2
    return sub

x = 78
y = 10
print ("Addition of two numbers ")
a = addition(x,y)
print (total)
print ("Subtraction of two numbers")
b = subtraction(x,y)
print (sub)
```

```
 OUTPUT
 Addition of two numbers
88
Subtraction of two numbers
68
```

1.9 Recursive Functions

A function calling by itself is a recursive function. The process of calling repeats several times and is useful in different applications. In real life, fractals, leaves and mountains with hills are some examples where you can see the pattern are recursive in nature. In computer science, recursion is found useful in solving problems. A few examples include Tower of Hanoi puzzle, Fibonacci series and Ackermann functions.

An example of recursion is given in this example, which counts from 1 to 6. Here, count_recursion is the function name and the number 6 is passed to the variable k. This value is decremented by one in the function and calls by itself as specified in `count_recursion(k-1)`. The program works such that, the value decrements until it reaches 0 with an addition of 1 in each step. The value is then displayed in the screen. This is the reason why, the count is appearing in ascending order although we have specified 6 and function decrements the value.

```
 def count_recursion(k):
  if(k>0):
    result = count_recursion(k-1)+1
    print(result)
  else:
    result = 0
  return result

print("Example of Recursion Count")
count_recursion(6)
```

```
 OUTPUT
 Example of Recursion Count
1
2
3
4
5
6
```

Another common example is the factorial which can be programmed with recursion as shown below.

```
 #facorial on a number
def fact(n):
   if n == 1:
       return n
   else:
       return n*fact(n-1)

num = int(input("Enter a number: "))

if num < 0:
   print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
   print("The factorial of 0 is 1")
```

```
else:
   print("The factorial of",num,"is",fact(num))
```

```
 OUTPUT
 Enter a number: 6
The factorial of 6 is 720
```

1.10 Worked out Programming Problems

Example 1.1. Write a program to calculate the area of triangle.

```
 #program to find the area of triangle
base = 5.0
height = 10.0
area = 0.5 * base * height
print ("The area of triangle is", area)
```

```
 OUTPUT
 The area of triangle is 25.0
```

Example 1.2. With python print a pattern like the following structure.

```
******
*****
****
***
**
*
*
**
***
****
*****
******
```

```
 for i in range (6,0,-1):
    print (i*"*")
for i in range (0,7,1):
    print (i*"*")
```

Example 1.3. Write a program to calculate compound interest

```
#program for calculating compound interest
p = 1000.0
r = 0.05
t = 2
y = 5
amount = float(p*(1+(r/t))**(t*y))
amount = round(amount,2)
print ("Amount is Rs. ", amount)
```

```
OUTPUT
Amount is Rs.  1280.08
```

Example 1.4. Write a program to get the number from the user and check whether it is positive or negative.

```
  #Check whether the given number is positive or negative
n = int(input("enter the number: "))
if n>0:
    print (n, "is positive")
elif n == 0:
    print (n,"is equal to 0")
else:
    print (n, "is negative")
```

```
 enter the number: -98
-98 is negative
```

Example 1.5. Write a program to find the sum of first n natural numbers.

```
  #Sum of first n numbers
total = 0
n = int(input("enter the number "))
for i in range(n+1):
    total = total+i
print ("Total of first ",n, "numbers is ", total)
```

```
  OUTPUT
enter the numer 10
Total of first  10 numbers is  55
```

Example 1.6. Write a program to find the factorial of a given number.

```
#facorial on a number
def fact(n):
   if n == 1:
       return n
   else:
       return n*fact(n-1)

num = int(input("Enter a number: "))

if num < 0:
   print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
   print("The factorial of 0 is 1")
else:
   print("The factorial of",num,"is",fact(num))
```

```
OUTPUT
The factorial of 5 is 120
```

Example 1.7. Write a program to print the range of odd and even numbers. Get the range from the user and print the numbers.

```
#odd/even numbers from 1 to 100

print("Enter a range")
x=int(input("From: "))
n=int(input("To: "))

print("Even numbers from",x,"to",n)
for i in range(x,n+1):
    if(i%2==0):
        print(i)

print("Odd numbers from",x,"to",n)
for i in range(x,n+1):
    if(i%2!=0):
        print(i)
```

```
 OUTPUT
 Enter a range
From: 4
To: 20
Even numbers from 4 to 20
4
6
8
10
12
14
16
18
20
Odd numbers from 4 to 20
5
7
9
11
13
15
17
19
```

Example 1.8. Write a program to print the series or $1 + x + x^2 + x^3 + ...x^n$

```
#sum of series 1+x+x^2+...
x = 2
n = 3
total = 0

for i in range(0,n+1):
    term = pow(x,i)
    print (term)
    total = total+term
print (total)
```

```
OUTPUT
 enter the value for x: 2
enter the value for n: 3
The sum is 15
```

Example 1.9. Write a program to print the series or $1 - x + x^2 - x^3 + ...x^n$

```
# Series to find 1 -x + x^2-x^3-..
n = int(input("enter the value for n: "))
x = int(input("enter the value for x: "))
total = 0
for i in range(0,n+1):
    term = pow(x,i)
    neg = pow(-1,i) # alternate + and - ve terms
    full_term = term*neg
    total = total+full_term
print ("Series is ", total)
```

```
OUTPUT
enter the value for n: 3
enter the value for x: 3
Series is  -20
```

Example 1.10. What is the output of the following code?

```
  n = 3
while n > 0:
    if n == 5:
        n = -99
    print(n)
    n = n+10
```

```
  OUTPUT
3
4
-99
```

Example 1.11. Write a program to print the multiplication table.

```
n = int(input("enter the integer number to generate multiplication table
print("Multiplication table for", n)
print ("————————————————————————————")
for i in range(1,11):
    m = n*i
    print (n,"X",i,"=",m)
```

```
 OUTPUT
 enter the integer number to generate multiplication table 5
Multiplication table for 5
————————————————————————————
5 X 1 = 5
5 X 2 = 10
5 X 3 = 15
5 X 4 = 20
5 X 5 = 25
5 X 6 = 30
5 X 7 = 35
5 X 8 = 40
5 X 9 = 45
5 X 10 = 50
```

Example 1.12. For a given integer, separate the digits and print those.

```
  #program to separate the digits
n = int(input("enter an integer "))
while n>0:
    rem = n%10
    n = n//10
    print("digit is ", rem)
```

```
 OUTPUT
enter an integer 456
digit is  6
digit is  5
digit is  4
```

Example 1.13. Extend the above example 1.12 such that the given number is reversed.

```
  #program to reverse the number
n = int(input("enter an integer "))
reverse = 0
while n>0:
    rem = n%10
    n = n//10
    reverse = (reverse*10) + rem
print ("The reversed number is ", reverse)
```

```
 OUTPUT
enter an integer 456
The reversed number is  654
```

Example 1.14. Write a program to print all prime numbers from 1 to 100

```
  #print prime numbers within 100

for i in range(1,100):
    if i>1:
        for j in range(2,i):
            if (i%j) == 0:
                break
        else:
            print (i, end = ",")
print()
```

```
OUTPUT
2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,
71,73,79,83,89,97,
```

Example 1.15. Write a program to print the length of the given string.

```
  #print the total number of characters in the string
word = str(input("Enter the string "))
count = 0
for i in word:
    count += 1
print ("Number of letters in the word ", word, "is ", count)
```

```
  OUTPUT
Enter the string university
Number of letters in the word  university is  10
```

Example 1.16. Write a program to print the number in each row such that it corresponds to the count. For example for 1, 1 should be printed, in the second row 2 2 should be printed, in third row 3 3 3 should be printed.

```
  #numbers in each row according to the count
for i in range(1,6):
    for j in range(1,i+1):
        print (i, end = " ")
    print()
```

```
 1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

Example 1.17. Write a program to print the Floyd triangle.

```
  #print Floyd triange

n = int(input("enter the number of rows "))
num = 1
for i in range(1,n+1):
    for j in range(1, i+1):
        print(num, end = ' ')
        num = num+1
    print()
```

```
OUTPUT
 enter the number of rows 6
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
16 17 18 19 20 21
```

Example 1.18. Write a program to print the Fibonacci numbers using recursion

```
  #print fibonacci series

def fib(n):
    if n <= 1:
        return n
    elif n > 1:
        return (fib(n-1)+fib(n-2))


for i in range(11):
    print(fib(i), end = ",")
print()
```

OUTPUT

```
  0,1,1,2,3,5,8,13,21,34,55,
```

Example 1.19. Write a python program to print the multiplication table of an integer in a recursive manner.

```
  #recursive multiplication of 5
def mul(n):
    count = 0
    if n<=1:
        return 1
    else:
        count  = count +1
```

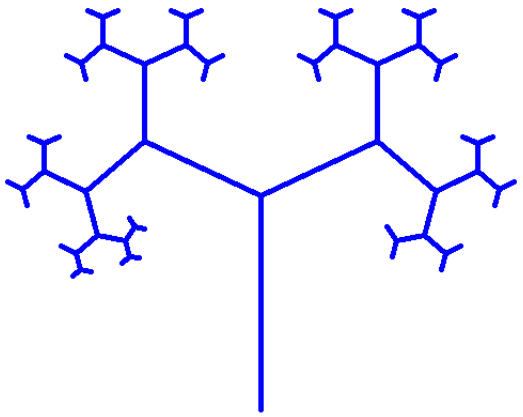
```
        return ("5X",count,"=",(n-1)*5)
n = 12
for i in range (n):
    print (mul(i))
```

```
  OUTPUT
1
1
('5X', 1, '=', 5)
('5X', 1, '=', 10)
('5X', 1, '=', 15)
('5X', 1, '=', 20)
('5X', 1, '=', 25)
('5X', 1, '=', 30)
('5X', 1, '=', 35)
('5X', 1, '=', 40)
('5X', 1, '=', 45)
('5X', 1, '=', 50)
```

Example 1.20. This is an example taken from `https://openbookproject.net/thinkcs/python/english3e/recursion.html`. Try it out changing l,a and f.

```
  from turtle import Turtle, mainloop
def tree(plist, l, a, f):
    """ plist is list of pens
    l is length of branch
    a is half of the angle between 2 branches
    f is factor by which branch is shortened
    from level to level."""
    if l > 6:
        lst = []
        for p in plist:
            p.forward(l)
            q = p.clone()
            p.left(a)
            q.right(a)
            lst.append(p)
            lst.append(q)
        tree(lst, l*f, a, f)
def main():
    p = Turtle()
    p.color("blue")
    p.pensize(5)
    p.hideturtle()
    p.speed(100)
    p.left(90)
    p.penup()
    p.goto(0,-200)
    p.pendown()
```

Figure 1.4: OUTPUT



```
    t = tree([p], 200, 65, 0.6)

main()
```

Summary

1. Python can be executed in interactive mode through IDE such as IDLE
2. With an extension of `*.py`, Python can be saved and executed with the command `python3 <filename.py>`
3. Python supports variables extensively and a proper mechanism of variable format should be provided.
4. Different datatypes such as integers, float and strings are automatically recognized with python without any declaration
5. Arithmetic operators and boolean operators are supported in Python
6. Conditional statements are supported through `'if-then-else'` constructs
7. The loops of 'for' and 'while' help to repeat the logic multiple times
8. Functions are declared through 'def' statement
9. Functions can be also of recursive type, which helps in a number of real life example problems

1.11 Learning Outcomes

After reading and working out the examples provided in this chapter, you should be able to:

1. Understand the basic syntax of Python Programming
2. Use arithmetic and boolean operators in programming problems
3. Understand the logic of python programming structures with control statements
4. Solve simple mathematical problems using Python programming structures
5. Provide a modular approach of programming through functions
6. Understand the intricacies of recursive function

1.12 Chapter Exercises

1.12.1 Multiple choice Questions

1. Indicate which of the following are valid numeric literals in python.
   (a) 1024
   (b) 1,024
   (c) 1024.0
   (d) 0.25+10
2. Which of the following are valid string literals in python.
   (a) “Hello”
   (b) ’hello’
   (c) “Hello”
   (d) ‘Hello there’
3. Which of the following expressions evaluate to True?
   (a) 80<=8
   (b) 80==80
   (c) 80!=90
   (d) ’8’<=’10’
4. What is the output of the following piece of code.

```
num=100
num = num+50
num = 20
num = 20+50
```

   (a) 70

(b) 20

(c) 100

(d) 50

5. All if statements must contain an **`if`** or **`elif`** header - State True or False.

6. Which is not an example of iterative control statement.

   (a) while loop

   (b) for with increment loop

   (c) if-else

   (d) for with decrement

7. A boolean flag is

   (a) A variable

   (b) Has the value of True or False

   (c) Is used as a condition for control statements

   (d) All of the above

8. 15%2 equals

   (a) 1

   (b) 2

   (c) 0

   (d) 7

9. What is the purpose of the “def” keyword in Python?

   (a) It is slang that means "the following code is really cool"

   (b) It is the start of the function

   (c) It indicates that the following indented section of code is to be stored for later

   (d) b and c are true

10. Which of the following will not work as variable name?

    (a) if

    (b) text

    (c) atefc

    (d) t_80

11. How many numbers will be printed in the following piece of code?

```
i = 5
while i>0:
    print(i)
    i = i - 1
```

(a) 4

(b) 5

(c) 6

(d) 0

12. Which statement ends the current iteration of the loop and continues to the next one?

    (a) break

    (b) continue

    (c) skip

    (d) pass

13. Which of the following is placed after the *if* condition?

    (a) ;

    (b) :

    (c) !

    (d) ::

14. Which statement is used to terminate the current loop?

    (a) break

    (b) continue

    (c) skip

    (d) pass

15. Which statement is used to stop the current iteration of the loop and continues with the next loop?

    (a) break

    (b) continue

    (c) skip

    (d) pass

1.12.2 Descriptive Questions

1. With examples explain the correct and incorrect usage of variables in python.
2. What are the different arithmetic and boolean operators that are used in python.
3. How will you convert one datatype to another? Give examples.
4. With an example explain iterative control statements.
5. Differentiate Break and Continue statement. When will you use Continue statement and when will you use Break statement?

6. Explain the local and global scope in functions with an example.
7. Differentiate iteration and recursion.
8. Evaluate the Boolean expressions for n = 10 and k = 20
   (a) (n>10) and (k == 20)
   (b) (n>10) or (k == 20)
   (c) not ((n>10) and (k == 20))
   (d) (not (n>10)) and (not(k == 20))
   (e) (n>10) or (k==10 or k!=5)
9. Give an appropriate if statement for each of the following.
   (a) An if statement that displays 'within range' if num is between 0 and 100 , inclusive
   (b) An if statement that displays 'within range' if num is between 0 and 100, inclusive, and displays 'out of range' otherwise.
10. What is the difference between (/) and (//) operator in python? Explain with an example.

1.12.3 Programming Questions

1. Write a python program to print the area of triangle.
2. Write a python program to find the area of trapezoid.
3. Write a python program to print the amount for simple interest, given principal amount, year and rate of interest.
4. Write a python program to indicate the type of different values eg: if 345, entered value 345 is integer. if "abc", entered value is string.
5. Write a python program to swap two values.
6. Write a python program to find the salary of an employee computing basic + DA (120% of basic) + cca (20% of basic).
7. Write a python program to find the roots of quadratic equation
8. Write a python program to print ones place of an integer.
9. Write a python program to convert the given temperature as degree to farenheit.
10. Write a python program to find whether the given year is a leap year.
11. Write a python program to find the largest of three numbers.
12. Write a python program to find the sum and average of first n numbers.
13. Write a python program to find the sum of odd numbers and even numbers within a certain range.

14. Write a python program to find whether the given number is an Armstrong number. Armstrong number is the number when the sum of the squares of the integer is equal to the same number.

15. Write a python program to compute whether the given integer is palindrome or not.

16. Write a python program to convert the given decimal number to binary and binary to decimal.

17. Write a python function to find the series $1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + ... + \frac{x^n}{n!}$

18. Write a python program to emulate rock, paper and scissors game.

19. Write a python program to get the value of LCM of two integers.

20. Write a python program to generate first N perfect numbers.

1.13 References

1. Reema Thareja, Python Programming using Problem Solving Approach, Oxford University Press, 2019

2. Charles Dierbach, Introduction to Computer Science using Python A Computational Problem-Solving Focus, Wiley Publications, 2015

3. Installation of Python - `https://www.python.org/downloads/`

4. Beginners guide - `https://wiki.python.org/moin/BeginnersGuide/Download`

5. Online browser based tutorial - `https://www.w3schools.com/python/python_intro.asp`

6. IDE for python - `https://realpython.com/python-ides-code-editors-guide/`

7. Installation of pip - `https://pip.pypa.io/en/stable/installing/`

Chapter 2

DATA TYPES IN PYTHON

Objectives

- To understand the data structures of Lists, Tuples, Sets and Dictionaries in Python
- To understand how Strings are handled in Python.

In this chapter, different data structures that are available in python are discussed. These include Lists, Tuples, Sets and Dictionary. Some of the important concepts and methods are provided with examples. A separate topic of Strings is included to make you comfortable with the processing of strings in Python.

2.1 Lists

We use list in our daily life. You might have used a to-do list for tasks or a list for shopping items. Assume that you have a set of to-do tasks that are to be performed. To this list, you might have to add, delete or modify the tasks. This analogy is very much applicable for lists in Python.

A list is a linear sequence of data, where each of it can be referred to the corresponding indices. For example, consider the data of the five subject marks. These can be considered as a list with the name of subjects. Each mark is specified with an index 0,1,2,3,4 as subjects[0], subjects[1], ..., subjects[4], as shown in Figure 2.1. It indicates that subjects[0] = 70, subjects[1] = 75, subjects[2] = 80, subjects[3] = 86 and subjects[4] = 68.

Note: The indices always range from 0 to $n-1$, where n is the number of elements in the list.

Common operations of list are *retrieve, update, insert, remove and append*. For example, consider the above list.

1. Retrieve subjects[3] will get the value of 86
2. Update subjects[1] with the value of 78 will replace 78, instead of 75
3. Insert subjects[2] with the value of 95 will result in subjects[2] = 95, subjects[3] = 80, subjects[4]=86 and subjects[5]=68
4. Remove the value of 68 from the list will result in the range of list from subjects[0] to subjects[3]
5. Append 89 to the list will result in subjects[5] = 89

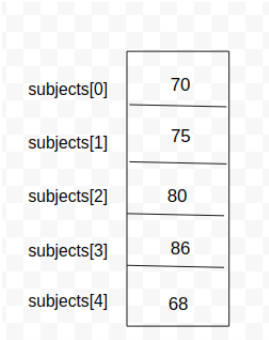


Figure 2.1: List in Python

2.1.1 Common Operations

In Python, a list is denoted with square bracket and elements are separated by a comma. The same example from Figure 2.1 in python can be written as:

```
>>> subjects = [70,75,80,86,68]
>>> subjects
[70, 75, 80, 86, 68]
```

Here, the elements are integers. Instead of integers, strings can also be included as elements. An empty list is indicated as `[]`. The operations described above can be performed with python as follows.

```
 >>> subjects = [70,75,80,86,68]
>>> subjects
[70, 75, 80, 86, 68]
>>> subjects[3]#Retreive subjects[3]
86
>>> subjects[1] = 78 # replace subjects[1] value with 78
>>> subjects
[70, 78, 80, 86, 68]
>>> subjects.insert(2,95) #insert 95 at index 2
>>> subjects
[70, 78, 95, 80, 86, 68]
>>> del subjects[4] # delete index 4
>>> subjects
[70, 78, 95, 80, 68]
>>> subjects.append(89) # append 89
>>> subjects
[70, 78, 95, 80, 68, 89]
>>>
```

Two additional opertions that you would use are sorting and reverse reorder, which is given below.

```
 >>> subjects
[68, 70, 78, 80, 89, 95]
>>> subjects.reverse()
```

```
>>> subjects
[95, 89, 80, 78, 70, 68]
```

Note: An important feature of list in python is that it is **mutable** - meaning, the values in the list can be changed.

Operations of list with strings is given below.

```
 >>> text
['dog', 'cat', 'horse', 'binary', 'oxen']
>>> del text[2]
>>> text
['dog', 'cat', 'binary', 'oxen']
>>> text.append('zebra')
>>> text
['dog', 'cat', 'binary', 'oxen', 'zebra']
>>> text.sort()
>>> text
['binary', 'cat', 'dog', 'oxen', 'zebra']
>>> text.reverse()
>>> text
['zebra', 'oxen', 'dog', 'cat', 'binary']
>>>
```

2.1.2 Sequence Operations

Python provides several ready made operations that can be performed on a list. These include finding the total number of elements, slicing of indices to determine specific data, counting the frequency of specific element, finding the index of the data, membership determination, concatenation, finding minimum and maximum value. An operation of sum is also provided which calculates the total value in the list.

Assume you have two lists, list1 and list2.

```
 >>> list1 = [20,20,40,10]
>>> list2 = [60,70,70,70]
```

To find the total number of elements in the list, `len` can be used as `len(list1)`, which will result in 4. Different slicing operations can be performed such as retrieving the elements between the indices 2 to 4 as `list1[2:4]`. This will be dealt in Section 2.1.3. The frequency of the elements in the list can be obtained using the keyword 'count'. To determine the index of the data, the keyword of index can be used, which will return the index value of that particular data. Two lists can be concatenated using the operator of '+'. Minimum and maximum of the elements can be determined using 'min' and 'max' operators, respectively. If numerals are used in lists, the keyword of 'sum' can be used to find the total value of elements. These operations are provided in the code below.

```
>>> list1 = [20,20,40,10]
>>> list2 = [60,70,70,70]
>>> len
```

```
<built-in function len>
>>> len(list1)
4
>>> list1[2:]
[40, 10]
>>> list1.count(20)
2
>>> list1.index(40)
2
>>> list1.index(10)
3
>>> 70 in list1
False
>>> 70 in list2
True
>>> list2.count(70)
3
>>> list3 = list1+list2
>>> list3
[20, 20, 40, 10, 60, 70, 70, 70]
>>> len(list3)
8
>>> min(list3)
10
>>> max(list3)
70
>>> sum(list3)
360
```

2.1.3 Slicing Operation

One of the most useful operation is to get the specific values from the list. This is done with slicing operator ':'. Suppose the list is denoted as L, then the syntax is:

```
L[start:end:step]
```

For example, refer Figure 2.1. If you want to get the the values of subjects with indices 1,2 and 3, use the following expression

```
subjects[1:4]
```

Different ways of slicing is shown below. Try and practice with more examples.

```
>>> words = ["apple", "banana", "orange", "pineapple",
                                        "mango","papaya"]
>>> len(words)
6
>>> words[2]
'orange'
>>> words[2:4] #get words from 2 to 4. Index 4 is
```

```
                                        not included
['orange', 'pineapple']
>>> words[2:] # get words from 2 to end of the list
['orange', 'pineapple', 'mango', 'papaya']
>>> words[2:5] #get words from 2 to 5
['orange', 'pineapple', 'mango']
>>> words[-1] # get the last word
'papaya'
>>> words[:-1] # get all words till the last word
['apple', 'banana', 'orange', 'pineapple', 'mango']
>>> words[:-2] # get all words before the last 2nd word
                                          from right
['apple', 'banana', 'orange', 'pineapple']
>>> words[3:-1] #get the words from 3rd index to
                                        the last word
['pineapple', 'mango']
>>> words[-3]# get the last 3rd word from the right
'pineapple'
>>> words[:]# get all words
['apple', 'banana', 'orange', 'pineapple', 'mango', 'papaya']
```

The operator '::' is used to get the elements with a jump in sequence. For example, to get 0, 2 and 4 elements from the list `L[0::2]` can be used (An element is skipped while reading the sequence). A few more scenarios are presented below.

```
 >>> words[:]
['apple', 'banana', 'orange', 'pineapple', 'mango', 'papaya']
>>> words[0::4]
['apple', 'mango']
>>> words[1::3]
['banana', 'mango']
>>> words[1::4]
['banana', 'papaya']
>>> words[0::2]
['apple', 'orange', 'mango']
>>> words[1::2]
['banana', 'pineapple', 'papaya']
```

The operations discussed above such as updating, deleting can also be used with slicing. The following program starts with an empty list, adds the even numbers to the list and deletes one by one.

```
 #program to enter even numbers to a list and delete
even = []#initialize the list
for i in range(0,20):
    if i % 2 == 0:
        even = even+[i] #convert i to list and concatenate
                                        with even at every step
print ("Even numbers between 0 and 20 are: ")
print (even)
```

```
print ("Deleting the list now")
del even[:]
print ("The list is empty")
print (even)
```

```
 OUTPUT
 Even numbers between 0 and 20 are:
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
Deleting the list now
The list is empty
[]
```

2.1.4 Nested List

A nested list is a list within a list. In this case, the nested list will take one value of index and the nested list itself has index values. Assume you have a list 'L' with two lists 'L1' and 'L2' within it. This can be represented as `L = [[L1],[L2]]`. As an example, let us take the list as person with elements of roll number, name, courses they are taking and programming languages known. This can be expressed as:

```
person = [''2020115223'',''Selvi'',[''maths'',''operating systems'',
''data structures''], [''c'',''c++'',''python'',''java'']]
```

Accessing the list is given below.

```
 >>> len(person)
4
>>> person[0]
'2020115223'
>>> person[1]
'Selvi'
>>> person[2]
['maths', 'operating systems', 'data structures']
>>> person[3]
['c', 'c++', 'python', 'java']
```

To access the list within a list [][] can be used, as shown in the following example.

```
 >>> len(person[2])
3
>>> person[2][0] # access the first course of the person list
'maths'
>>> person[2][1] # access the second course
'operating systems'
>>> person[2][2] # access the third course
'data structures'
>>> person[3][1:3] # get the 1st and 2nd value from the list
                                    #of programming languages
['c++', 'python']
```

2.1.5 Looping within Lists

Items within list can be iterated with loops such as *for* and *while*. With *for* loops, the iteration is done in three different ways.

- items within a list
- using an index
- using enumerate

These three types of iteration are explained using an example of list.

Looping as items within list

Here, 'item' is a variable that iterates each time within the list of fruits. During the first loop, item is initiated to "apple", second time "orange" and so on.

```
fruits = ["apple","orange","banana","pineapple","mango"]
for item in fruits:
    print (item)
```

```
 OUTPUT
apple
orange
banana
pineapple
mango
```

Looping using index

With the same list, using index the code is given below. To iterate through indices, first the length of the list should be determined, as shown in the second line as 'len(fruits)', which is assinged to the variale 'n'. For looping, this integer is used (line 6) and the loop iterates through index with variable 'i'. In this case 'fruits[i]' indicates each item.

First time, when the loop starts, i = 0 and fruits[0] = "apple". During the second loop, i =1 and fruits[1] = "orange" and this continues till the lenth of the list.

```
1. fruits = ["apple","orange","banana","pineapple","mango"]
2. n = len(fruits)
3. print ("_______","________")
4. print ("index","| ","item")
5. print ("_______","________")
6. for i in range(n):
7.     print (" ",i,"   |   ",fruits[i])
8. print ("_______","________")
```

```
 OUTPUT
 _______ ________
index |   item
______ ________
  0    |    apple
  1    |    orange
```

```
  2    |    banana
  3    |    pineapple
  4    |    mango
```

Looping with enumerate

You have seen in the earlier example how to access through the index using the length of the list. You can get the same result without finding the actual number of items in the list using the keyword 'enumerate'. In this case, two variables separated by comma can be used in the *for* loop. The syntax is:

```
for index,value in enumerate(list_name):
```

The same example that you have seen above is illustrated below using enumerate. Here, the variables are index and item, which indicate the index and items within the list, respectively.

```
fruits = ["apple","orange","banana","pineapple","mango"]
print ("_______","________")
print ("index","| ","item")
print ("_______","________")
for index,item in enumerate(fruits):
    print (" ",index,"   |   ",item)
print ("_______","________")
```

```
 OUTPUT
 _______ ________
index |  item
______ ________
  0    |    apple
  1    |    orange
  2    |    banana
  3    |    pineapple
  4    |    mango
_______ ________
```

A quick way of iterating can be done in one line (list comprehension) as shown below. Here, the print statement with *for* loop is included in square brackets.

```
 fruits = ["apple","orange","banana","pineapple","mango"]
[print(item) for item in fruits]
```

```
OUTPUT
apple
orange
banana
pineapple
mango
```

With *while*, the example of looping is illustrated below.

```
fruits = ["apple","orange","banana","pineapple","mango"]
n = len(fruits)
```

```
i = 0
while (i<n):
    print (fruits[i])
    i+=1
```

```
OUTPUT
apple
orange
banana
pineapple
mango
```

2.1.6 Assignment and Copy of Lists

Like variables, you can assign one variable to another. In these cases, the list that assigns point to the same location. For example, consider the following code showing two lists numbers_1 and numbers_2. numbers_1 is assigned to numbers_2 and those give the same result.

```
>>> numbers_1 = [10,20,30]
>>> numbers_2 = numbers1
>>> numbers_1
[10, 20, 30]
>>> numbers_2
[10, 20, 30]
```

Now, the value of 30 in numbers_2 should be updated as 200. So, the code is `numbers_2[2]=200`. Try printing the values of numbers_1 and numbers_2 after this change. You will be surprised that values in numbers_1 also is changed.

```
 >>> numbers_2[2]=200
 >>> numbers_2
[10, 20, 200]
>>> numbers_1
[10, 20, 200]
```

What really happens when you assign a variable or a list?

When you assign a variable or a list to another variable or a list, the same memory address location is pointed where the value is stored. Take for example, numbers_1 and numbers_2 and see the memory locations where the lists are stored. In python, to get the memory location address in hexadecimal, use **hex(id(variable_name))**. Let us print the location of numbers_1 and numbers_2.

```
  >>> hex(id(numbers_1))
'0x7fa0b987fec0'
>>> hex(id(numbers_2))
'0x7fa0b987fec0'
>>> numbers_2[2]=200
```

Here, both locations are one and the same. That is, data in numbers_1 are not copied to numbers_2, but numbers_2 link to memory location of numbers_1.

When you are programming, you may come across some cases where you have to copy the list and make changes only in the copied list and do not want just an assignment. In such cases, you can use slicing operator [:]. To illustrate this case, let us take the above example and instead of just assigning, use [:] and see the difference.

```
 >>> numbers_1 = [10,20,30]
 >>> numbers_2 = numbers_1[:]
>>> numbers_1
[10, 20, 30]
>>> numbers_2
[10, 20, 30]
>>> numbers_2[2] = 200
>>> numbers_1
[10, 20, 30]
>>> numbers_2
[10, 20, 200]
```

Here, value of numbers_2 alone changes without any effect on numbers_1. Another interesting feature to explore is what happens to the memory location of the two lists. Try this code and you will see that the two lists point to different memory locations, indicating that the second list has a copy of the first list.

```
 >>> hex(id(numbers_1))
'0x7fa0b97b3880'
>>> hex(id(numbers_2))
'0x7fa0b9a2f300'
```

Copying with [:] works well when there is a single element in a list. But if there is a nested list, you should be careful. Consider the following example with list num1 which is a nested list. num1[:] is assigned to num2 to copy the values of num1 to num2.

```
 >>> num1 = [10,20,[5,15,25,35],50]
>>> num2 = num1[:]
>>> num1
```

```
[10, 20, [5, 15, 25, 35], 50]
>>> num2
[10, 20, [5, 15, 25, 35], 50]
>>> num2[2]
[5, 15, 25, 35]
```

If you want to update the value of 35 to 2005 in num2, you will probably code like this.

```
>>> num2[2][3] = 2005
>>> num2
[10, 20, [5, 15, 25, 2005], 50]
>>> num1
[10, 20, [5, 15, 25, 2005], 50]
```

But on doing so, the values in both lists num1 and num2 changes to 2005.

Inorder to have the second list with the update without any modification in the first list, you have to use the module of deepcopy as shown below. The first line imports a module called 'deepcopy' from 'copy' package. The third line uses this module to copy the contents of num1 to num2.

```
>>> from copy import deepcopy
>>> num1 = [10,20,[5,15,25,35],50]
>>> num2 = deepcopy(num1)
>>> num1
[10, 20, [5, 15, 25, 35], 50]
>>> num2
[10, 20, [5, 15, 25, 35], 50]
>>> num2[2][3] = 2005
>>> num2
[10, 20, [5, 15, 25, 2005], 50]
>>> num1
[10, 20, [5, 15, 25, 35], 50]
>>>
```

In this case, the num2 values are updated without any effect on num1.

2.2 Tuples

Tuples are similar to that of lists, but are immutable - meaning the values cannot be changed. Tuples can be created using parantheses () separated by a comma ',' and can include different data types such as integers, floats, strings, lists and tuples too. With tuples, the similar slicing operations are possible as seen in list.

```
 >>> text = ('dog','cat','horse','zebra')
>>> text[0]
'dog'
>>> text[1]
'cat'
>>> text[2]
'horse'
```

```
>>> text[3]
'zebra'
>>> text[3][0]
'z'
>>> text[2][1:-1]
'ors'
```

As said earlier, tuples do not accept update or changes in values. Suppose in the example, you have to change the value of 'horse' to 'ox', in list you can assign as `text[2]='ox'`. Try assigning the same to this tuple and you will get an error statement as shown below. This means that values in tuple cannot be changed.

```
 >>> text[2] = 'ox'
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    text[2] = 'ox'
TypeError: 'tuple' object does not support item assignment
```

Some operations on tuples are listed below.

Two tuples can be concatenated using the operator of '+'. Tuples of birds and mammals contain strings which are concatenated to animals. Another tuples of num1 and num2 contain integers which are concatenated to form another tuple.

```
 >>> birds = (''crow'',''sparrow'')
>> birds
('crow', 'sparrow')
>>> mammals = (''camel'',''cat'',''dog'')
>>> mammals
('camel', 'cat', 'dog')
>>> animals = birds+mammals
>>> animals
('crow', 'sparrow', 'camel', 'cat', 'dog')
>>> num1 = (1,2,3,4,5)
>>> num2 = (6,7,8,9,10)
>>> num1+num2
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Other operations such as finding the number of elements, membership, index numbers and count operations are similar to that of a list. Examples illustrating these operations are given below.

```
 >>> animals = ('crow', 'sparrow', 'camel', 'cat', 'dog')
 >>> len(animals)
5
>>> num3 = num1+num2
>>> len(num3)
10
#find the index of cat
>>> animals.index("cat")
3
# find the index of dog
```

```
>>> animals.count("dog")
1
# concatenate animals with animals and assign to animals
>>> animals = animals+animals
>>> animals
('crow', 'sparrow', 'camel', 'cat', 'dog',
                'crow', 'sparrow', 'camel', 'cat', 'dog')
#count the number of dogs in animals tuple
>>> animals.count("dog")
2
# check whether cat is in animals tuple
>>> 'cat' in animals
True
# check whether camel is in animals tuple
>>> 'camel' in animals
True
# check whether 3 is in animals tuple
>>> 3 in animals
False
# check whether 3 is not in animals tuple
>>> 3 not in animals
True
>>>
```

Loops can be used to iterate within tuples like that of lists. An example is shown below.

```
animals = ("cat","dog","horse","sparrow","crow",[2,4,6])
for index,item in enumerate(animals):
    print (index,item)
```

```
 OUTPUT
0 cat
1 dog
2 horse
3 sparrow
4 crow
5 [2,4,6]
```

2.3 Sets

A set is a datatype with the following features and provides the standard mathematical operations such as union, intersection and set difference.

1. Set is mutable
2. No duplicates in Sets
3. Unordered - that is there are no indices for reference to items

The set is represented using the syntax {} or set(). It can have any type of data such as integer, float, string and tuple. But lists cannot be included in a set. Refer to the following example. set1 has integers and set2 has datatype of integers, float, strings and tuple.

```
 >>> set1 = {0,2,4}
>>> set1
{0, 2, 4}
>>> set2 = {0,2,4,6.8,"hello world",(7,10.5,7)}
>>> set2
{0, 2, 4, 6.8, 'hello world', (7, 10.5, 7)}
```

As mentioned earlier, sets cannot have duplicates. In the next example `set3 = 0,2,4,4,6,4,7` is entered, but the result is unique.

```
 >>> set3 = {0,2,4,4,6,4,7}
>>> set3
{0, 2, 4, 6, 7}
```

To add values in the set, you can either use the keyword `add` or `update`. The former is used to add a single element and the latter to add the set of elements. An example of using both these keywords are given below.

```
 >>> set3 = {0,2,4,4,6,4,7}
>>> set3
{0, 2, 4, 6, 7}
>>> set3.add(50)
>>> set3
{0, 2, 4, 6, 7, 50}
>>> set3.add(50)
>>> set3
{0, 2, 4, 6, 7, 50}
>>> set3.update([60,70,80,90])
>>> set3
{0, 2, 4, 6, 7, 70, 80, 50, 90, 60}
>>> set3.update({200.6,300.7,400.8,500.9})
>>> set3
{0, 2, 4, 6, 7, 70, 200.6, 300.7, 80, 400.8, 50, 500.9, 90, 60}
```

To delete an element in the set, use the keyword `remove` or `discard`. The difference between these is that the former raises an error if the element is not present in the set, while the latter does not. An example demonstrating these two keywords is shown below.

```
 >>> set3
{0, 2, 4, 6, 7, 70, 200.6, 300.7, 80, 400.8, 50, 500.9, 90, 60}
#discard the element 200.6
>>> set3.discard(200.6)
>>> set3
{0, 2, 4, 6, 7, 70, 300.7, 80, 400.8, 50, 500.9, 90, 60}
#discard the element 1000, which is not in the set
>>> set3.discard(1000)
>>> set3
{0, 2, 4, 6, 7, 70, 300.7, 80, 400.8, 50, 500.9, 90, 60}
```

```
#remove the element 300.7
>>> set3.remove(300.7)
>>> set3
{0, 2, 4, 6, 7, 70, 80, 400.8, 50, 500.9, 90, 60}
#remove the element 1000
>>> set3.remove(1000)
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    set3.remove(1000)
KeyError: 1000
```

The basic operations of Union, Intersection and Set difference are illustrated below with two sets `set1 = {1,2,3,4,5,6,7}` and `set2 = {0,2,4,6}`.

```
 >>> set1 = {1,2,3,4,5,6,7}
>>> set2 = {0,2,4,6}
>>> #Union operator with |
>>> set1|set2
{0, 1, 2, 3, 4, 5, 6, 7}
>>> #Union operator with union
>>> set1.union(set2)
{0, 1, 2, 3, 4, 5, 6, 7}
>>> #intersection operator with & and intersection
>>> set1&set2
{2, 4, 6}
>> set1.intersection(set2)
{2, 4, 6}
>>>#set difference with - and difference
>>> set1-set2
{1, 3, 5, 7}
>>> set1.difference(set2)
{1, 3, 5, 7}
>>> set2.difference(set1)
{0}
>>> #symmetric difference - elements not in both set1 and set2
>>> set1.symmetric_difference(set2)
{0, 1, 3, 5, 7}
>>> set2.symmetric_difference(set1)
{0, 1, 3, 5, 7}
```

Other methods are given in Table 2.3. Assume `set1 = {1,3,5,7}`, `set3 = {2,4,6,8}` and `set4 = {1,2,3,4,5,6,7,8}`.

Table 2.1: Methods in Sets

Method	Description	Example	Result
clear()	Removes all elements from the set	set1.clear()	set()
copy()	Copies elements from one set to another	set2 = set1.copy()	`set2 = {1,3,5,7}`
isdisjoint()	Checks whether the two sets are disjoint	set3.isdisjoint(set1)	`True`
issubset()	Checks whether one set is a subset of another	set3.issubset(set4)	`True`
issuperset()	Checks whether one set is a superset of another	set4.issuperset(set1)	`True`
in	membership	0 in set1	`False`

How to create an empty set?

To initialize a list, you will use an empty list [ ]. Similarly for tuple, you will use ( ). But for set, **do not use** {}, because {} is also used to indicate dictionary in python. Hence, to create an empty set use the keyword **set( )**. In the example given below, set5 is initialized to an empty set and then 2 is added.

```
>>> set5 = set ()
>>> set5
set()
>>> set5. add(2)
>>> set5
{2}
```

While **set( )** is mutable without any duplicates and unordered, there is another variety which is immutable known as **frozenset**. An example to demonstrate this set is given below.

```
>>> fruits = frozenset(["apple","banana"])
>>> fruits.add("orange")
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    fruits.add("orange")
AttributeError: 'frozenset' object has no attribute 'add'
```

Like other types, sets also can be iterated with loops and an example is shown below.

```
 fruits = {"apple","orange","banana","pineapple","mango"}
for item in fruits:
    print (item)
```

```
OUTPUT
banana
pineapple
orange
```

```
apple
mango
```

2.4 Dictionary

A dictionary is used to indicate a key, value pair and is an unordered data structure. It is denoted by {:}, where the key and the value are separated by ':'.

You can take the analogy of how the dictionary is organized as word and the corresponding meaning.

For example, if you want to represent roll number and names of students, it can be done with the help of dictionary as:

```
>>> students = {2013110022: "Raman", 2013110023: "Ragavan",
2013110024:"Sandra",2013110025:"Sita Lakshmi", 2013110026: "Siva"}
>>> students
{2013110022: 'Raman', 2013110023: 'Ragavan', 2013110024:
'Sandra', 2013110025: 'Sita Lakshmi', 2013110026: 'Siva'}
```

The dictionary is unordered, meaning there are no indices like that of lists and tuples. But the value can be accessed based on the key as shown below.

```
 >>> students[2013110022]
'Raman'
```

Dictionaries cannot have duplicate keys. Try these in your editor and check what happens when you give duplicate keys.

```
 >>> {2013110022: 'Raman', 2013110023: 'Ragavan', 2013110024: 'Sandra',
  2013110025: 'Sita Lakshmi', 2013110026: 'Siva', 2013110026: "Syed"}
{2013110022: 'Raman', 2013110023: 'Ragavan', 2013110024: 'Sandra',
2013110025: 'Sita Lakshmi', 2013110026: 'Syed'}
```

Note: The syntax of dictionary and set looks similar as both are represented using {}. But the dictionary has a key value pair separated by :, whereas a set is a single entity. By default, if you initialize a variable with {}, python takes it as dictionary and not set. See the following example, where the variable num is assigned to the empty dictionary and the keyword `type(num)` indicates as dictionary.

```
>>> num = {}
>>> type(num)
<class 'dict'>
```

Some methods used in Dictionary are given below. These include the length of the elements, string representation, determination of the key for that particular item, membership with **in**, displaying all keys, values and updating the dictionary.

```
>>> students = {2013110022: 'Raman', 2013110023: 'Ragavan',
2013110024: 'Sandra',2013110025: 'Sita Lakshmi',
2013110026: 'Siva', 2013110026: "Syed"}
>>> # length
>>> len(students)
5
>>> #string representation of students dictionary
>>> str(students)
"{2013110022: 'Raman', 2013110023: 'Ragavan', 2013110024: 'Sandra',
2013110025: 'Sita Lakshmi', 2013110026: 'Syed'}"
>>> #copy students to names
>>> names = students.copy()
>>> names
{2013110022: 'Raman', 2013110023: 'Ragavan', 2013110024: 'Sandra',
2013110025: 'Sita Lakshmi', 2013110026: 'Syed'}
>>> names[2013110022]= "Robert"
>>> names
{2013110022: 'Robert', 2013110023: 'Ragavan', 2013110024: 'Sandra',
2013110025: 'Sita Lakshmi', 2013110026: 'Syed'}
>>> students
{2013110022: 'Raman', 2013110023: 'Ragavan', 2013110024: 'Sandra',
2013110025: 'Sita Lakshmi', 2013110026: 'Syed'}
>>> # get the value of the key
>>> students.get(2013110023)
'Ragavan'
>>> # get the items
>>> students.items()
dict_items([(2013110022, 'Raman'), (2013110023, 'Ragavan'),
(2013110024, 'Sandra'), (2013110025, 'Sita Lakshmi'),
(2013110026, 'Syed')])
>>> students.keys()
dict_keys([2013110022, 2013110023, 2013110024, 2013110025, 2013110026])
>>> #update
>>> dict2={2014110022:"Gerald", 2014110023: "Greshma",
    2014110024: "Geiko"}
>>> students.update(dict2)
>>> students
{2013110022: 'Raman', 2013110023: 'Ragavan', 2013110024: 'Sandra',
2013110025: 'Sita Lakshmi', 2013110026: 'Syed', 2014110022: 'Gerald',
2014110023: 'Greshma', 2014110024: 'Geiko'}
>>> #display values
>>> students.values()
dict_values(['Raman', 'Ragavan', 'Sandra', 'Sita Lakshmi',
'Syed', 'Gerald', 'Greshma', 'Geiko'])
>>> #set a default value
>>> students.setdefault("rollnumber","name")
'name'
```

```
>>> students
{2013110022: 'Raman', 2013110023: 'Ragavan',
2013110024: 'Sandra', 2013110025: 'Sita Lakshmi', 2013110026: 'Syed',
2014110022: 'Gerald', 2014110023: 'Greshma',
2014110024: 'Geiko', 'rollnumber': 'name'}
>>> # membership
>>> 2013110026 in students
True
>>> 2015112034 in students
False
```

To iterate in dictionary, the keyword **dict**.items() can be used.

```
>>> for key,value in students.items():
        print (key,value)


2013110022 Raman
2013110023 Ragavan
2013110024 Sandra
2013110025 Sita Lakshmi
2013110026 Syed
2014110022 Gerald
2014110023 Greshma
2014110024 Geiko
rollnumber name
```

A dictionary can have nested dictionary as illustrated below. Here, each student is the key and the value is the marks of the course which itself is in a dictionary.

```
>>> mca = {"Raman":{"OS": 90, "networks":85, "maths": 85},
           "Ragavan": {"OS": 80, "networks":75, "maths": 85},
           "Sita Lakshmi" : {"OS": 95, "networks":80, "maths": 75}}
>>> for i, j in mca.items():
        print(i,j)


Raman {'OS': 90, 'networks': 85, 'maths': 85}
Ragavan {'OS': 80, 'networks': 75, 'maths': 85}
Sita Lakshmi {'OS': 95, 'networks': 80, 'maths': 75}
```

2.5 Strings

Python supports strings extensively. A number of natural language processing techniques are implemented using python. Strings are expressed using inverted comma “ ”. (Here, a single of double quote can be used). For multiline strings, three quotes can be used. An example with multiline string is given below.

```
>>> str1 = '''Hello, this is the course on python.
```

```
 Here you will the basics  of python in Unit 1 and
 data types in Unit II. Python is used in many
 applications worldwide.'''
>>> str1
'Hello, this is the course on python.
Here you will the basics  of python in Unit 1
and data types in Unit II. Python is used in many
applications worldwide.'
```

Strings like lists can be concatenated and appended using '+' operator. If '*' operator is used, then the string can be seen as a multiple. A few examples are shown below.

```
 >>> str1 = "hello"
>>> str2 = "world"
>>> # concatenation of str1 and str2
>>> str1+str2
'helloworld'
>>> str1+ " " + str2
'hello world'
>>> #append with += operator
>>> str1 += " world"
>>> str1
'hello world'
>>> # multiply str1 three times
>>> str1
'hello world'
>>> str1*3
'hello worldhello worldhello world'
```

There are several inbuilt methods available in strings. For the purose of this course, the methods are arranged into these categories:

1. Frequency of the particular string and the total number of characters
2. Case conversion (Capitalise, Upper and Lower case)
3. Determination of the type of string
4. Search of particular character or string and replace conditions
5. String formatting
6. Slicing operation
7. Maximum and minimum characters in string
8. Split and join methods

2.5.1 Length and count of particular occurrences

The total number of characters can be determined using the keyword of **len**. To count the number of occurrences of the particular character or the string, **str**.count(**chr**,beg,end) is

used. **str** is the variable of the string and **chr** represent the character or string for which the count is to be determined. beg represents the beginning index of the string and end represent the index of the last element until which the string is to be traversed.

```
 >>> string1 = "Python Programming"
>>> #length of string
>>> len(string1)
18
>>> #count the number of 'n'
>>> string1.count(string1,0,len(string1))
1
>>> string1.count('n',0,len(string1))
2
>>> string2 = string1*2
>>> string2
'Python ProgrammingPython Programming'
>>> #count the occurrences of 'mm'
>>> string2.count('mm',0,len(string2))
2
```

2.5.2 String Cases

Several methods such as capitalizing the first character in the string, change of cases and checking whether the string is in a particular string are included in Python. These are illustrated below.

```
 >>> #capitalize
>>> str1.capitalize()
'This is a course on python'
>>> str2 = "THIS IS A COURSE ON PYTHON"
>>> str2.capitalize()
'This is a course on python'
>>> str1
'This is a course on Python'
>>> str3 = "this is a course on python"
>>> str3.capitalize()
'This is a course on python'
>>> #upper case
>>> str3.upper()
'THIS IS A COURSE ON PYTHON'
>>> #lower case
>>> str2.lower()
'this is a course on python'
>>> #check for lower or upper case
>>> str2.islower()
False
>>> str2.isupper()
True
>>> #title case
```

```
>>> str2
'THIS IS A COURSE ON PYTHON'
>>> str2.title()
'This Is A Course On Python'
>>> str3
'this is a course on python'
>>> str3.title()
'This Is A Course On Python'
>>> #swapcase
>>> str4 = "ThIs iS a CoUrSe oN PyThOn"
>>> str4
'ThIs iS a CoUrSe oN PyThOn'
>>> str4.swapcase()
'tHiS Is A cOuRsE On pYtHoN'
```

2.5.3 Type of String

Methods such as `isalpha()`, `isalnum()`, `isdigit()`, `isspace()`, `isidentifier()` can be used to check the type of the string. These are illustrated below.

```
 >>> str1 = "Chennai600025"
>>> #If string has alphabets and numbers, return true
>>> str1.isalnum()
True
>>> str2 = "C1_"
>>> str2.isalnum()
False
>>> str3 = "C1"
>>> str3.isalnum()
True
>>> #If string has alphabets, return true
>>> str1.isalpha()
False
>>> str4 = "Chennai"
>>> str4.isalpha()
True
>>> #Check if the string consist of space only
>>> str5 = "     "
>>> str5.isspace()
True
>>> str1.isspace()
False
>>> #Check if the string is only of digits
>>> str1.isdigit()
False
>>> str6 = "80930"
>>> str6.isdigit()
True
```

```
>>> #check for identifier
>>> str1.isidentifier()
True
>>> str2.isidentifier()
True
>>> str7 = ":45er"
>>> str7.isidentifier()
False
```

2.5.4 Search and replace of strings

Finding a part of the string with variants from the beginning or from the end are available in Python. The returning value in these cases is the index value. If the particular index is not found, then the `find` option returns -1 and `index` returns an error value. A replace option is also provided which can be used to replace the string. The following example illustrate these methods.

```
 >>> str1 = "This is a course on Python"
>>> #find "is" from str1
>>> str1.find("is",0,len(str1))
2
>>> #find "course" from str1
>>> str1.find("course",0,len(str1))
10
>>> str1.find("hello",0,len(str1))
-1
>>> #find the particular string using index keyword.
>>> str1.index("is",0,len(str1))
2
>>> str1.index("hello",0,len(str1))
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    str1.index("hello",0,len(str1))
ValueError: substring not found
>>> #Search from end using find
>>> str1.rfind("course",0,len(str1))
10
>>> str1.rfind("is",0,len(str1))
5
>>> str1.rindex("hello",0,len(str1))
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    str1.rindex("hello",0,len(str1))
ValueError: substring not found
>>> #replace "course" with "suject"
>>> str1.replace("course","subject")
'This is a subject on Python'
```

2.5.5 Formatting Strings

To format strings for displaying in the output screen, $\%s$ symbol is used. This symbol substitutes the corresponding variables. The syntax is:

```
''<FORMAT>'' % (<VALUES>)
```

An example is given below to print the rollnumber and name.

```
>>> rollnumber = 2013110022
>>> name = "Raman"
>>> print("Roll number = %d and Name = %s" %(rollnumber,name))
Roll number = 2013110022 and Name = Raman
```

The strings can be formatted to center justified, left or right justified as shown below. Here, a total of 10 character length is given with a symbol of '*' to indicate justification. To fill with zeros, keyword of `zfill` can be used.

```
>>> #center justified
>>> str1 = "hello"
>>> print(str1.center(10,'*'))
**hello***
>>> #left justified
>>> str1.ljust(10,'*')
'hello*****'
>>> #right justified
>>> str1.rjust(10,'*')
'*****hello'
>>>#fill with zeros
>>> str2 = "6789"
>>> str2.zfill(10)
'0000006789'
```

2.5.6 Slicing and Removal of spaces

Like lists, strings can be sliced according to indices. White spaces in strings can be manipulated as per the needs. A few examples are shown below.

```
>>> str1 = "This is a Python Course"
>>> len(str1)
23
>>> #slicing
>>> str1[5:]
'is a Python Course'
>>> str1[-1]
'e'
>>> str1[5:-7]
'is a Python'
>>> #strip white space
>>> str1.strip()
'This is a Python Course'
```

```
>>> str2 = "  Hello World  "
>>> str2.strip()
'Hello World'
>>> #remove white space at the beginning
>>> str2
'  Hello World  '
>>> str2.lstrip()
'Hello World  '
>>> #remove white space at the end
>>> str2
'  Hello World  '
>>> str2.rstrip()
'  Hello World'
```

2.5.7 Maximum and Minium characters in Strings

The inbuilt methods of max and min can be used to determine the alphabetic ordering of strings. An example is shown below.

```
 >>> str4 = "Python"
>>> max(str4)
'y'
>>> min(str4)
'P'
```

2.5.8 Split and Join methods

One of the important methods, that you might come across is to convert a string to list and vice-versa. These can be done using split and join methods. Split is used to convert a string to a list and join is to convert the list to a string. In the following example, the string "Hello World" is converted to ["Hello", "World"] using split method. The list of ["Python", "Course"] is converted to string using join method.

```
>>> #split - convert string to list
>>> str1= "Hello World"
>>> str1.split()
['Hello', 'World']
>>> #convert from list to string
>>> list1 = ["Python", "Course"]
>>> str3 = " ".join(list1)
>>> str3
'Python Course'
```

Another useful method is enumerate, which is used to list the index numbers along with the characters as shown below. In this example, list is included because the output is generated as a list.

```
>>> str1 = "Python"
>>> list(enumerate(str1))
```

```
[(0, 'P'), (1, 'y'), (2, 't'), (3, 'h'), (4, 'o'), (5, 'n')]
```

2.5.9 Comparing and Iterating Strings

The regular Boolean operators of $==, !=, >, <, >=, <=$ can be used to compare strings as shown below.

```
 >>> str1 = "abc"
>>> str2 = "def"
>>> str3 = "abc"
>>> str4 = "bbc"
>>> str1 == str4 #equal
False
>>> str1 == str3
True
>>> str4<str3 #less than
False
>>> str4>str3 # greater than
True
>>> str4>=str3 # greater than or equal to
True
>>> str1 != str2 # not equal to
True
```

Iteration can be performed with *for* or *while* loops. An example is given below, which iterates through the string "Python Course".

```
 >>> str1 = "Python Course"
>>> for i in str1:
        print (i, end="")

Python Course
```

2.6 Worked out Programming Problems

Example 2.1. Write a python program to draw the four sides using turtle module of different colors. The colors can be accessed through a list.

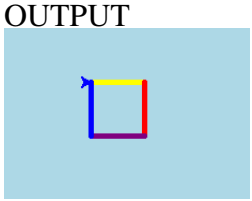
```
import turtle
wn = turtle.Screen()
wn.bgcolor("lightblue")
t1 = turtle.Turtle()
t1.pensize(5)

clrs = ["yellow","red","purple","blue"]
for i in clrs:
    t1.color(i)
    t1.forward(50)
```

```
    t1.right(90)

wn.mainloop()
```

OUTPUT



Example 2.2. Write a python program to get the positive integers from the user and add those to a list. Then, find the cube of the numbers in the list.

```
#get integers from user and save it in the list. Find the cube.

numlist = []
n = 1
while n>0:
    num = int(input("Enter the positive number to be added to the list,
                                        to stop enter -1:  "))
    if num > 0:
        numlist = numlist+[num]
        n = n+1
    else:
        n = -1

print ("The entered numbers are ", numlist)

cube = []
for i in numlist:
    cube = cube +[i**3]
print ("The cube of entered numbers ", cube)
```

```
OUTPUT
Enter the positive number to be added to the list, to stop enter -1:
3
Enter the positive number to be added to the list, to stop enter -1:
4
Enter the positive number to be added to the list, to stop enter -1:
5
Enter the positive number to be added to the list, to stop enter -1:
6
Enter the positive number to be added to the list, to stop enter -1:
-1
The entered numbers are  [3, 4, 5, 6]
The cube of entered numbers  [27, 64, 125, 216]
```

Example 2.3. Write a python program to get the fixed numbers of integers from the user and store as a list. Then arrange in ascending order.

```
  #sort in ascending order

n = int(input("enter the number of elements in the list: "))
numlist = []
for i in range(n):
    num = int(input("enter number " ))
    numlist.append(num)
print ("Entered numbers are: ")
print (numlist)
ascendlist = sorted(numlist)
print ("Sorted numbers are ")
print(ascendlist)
```

```
 OUTPUT
 enter the number of elements in the list: 5
enter number 20
enter number 10
enter number 39
enter number 45
enter number 12
Entered numbers are:
[20, 10, 39, 45, 12]
Sorted numbers are
[10, 12, 20, 39, 45]
```

Example 2.4. Write a python program to interchange the first and last element in a list.

```
  #interchange first and last element in the list

def interchange(listn):
    #find the length of the list
    num = len(listn)
    #print before interchange
    print ("Original list before interchange: ", listn)
    first_element = listn[0]
    last_element = listn[-1]
    #interchange the elements
    listn[0] = last_element
    listn[-1] = first_element
    print ("List after interchange: ", listn)

nlist = [2,3,4,5,6,100]
interchange(nlist)
```

```
 OUTPUT
Original list before interchange:  [2, 3, 4, 5, 6, 100]
List after interchange:  [100, 3, 4, 5, 6, 2]
```

Example 2.5. Write a python program to reverse the numbers in a list.

```
 #reverse the numbers in the list
def revslice(listn):
    print ("Original list before reverse: ", listn)
    #use slicing operation
    revlist = listn[::-1]
    print ("List after reverse using slice operation: ", revlist)

def revbuiltin(listn):
    print ("Original list before reverse: ", listn)
    #use built in function
    listn.reverse()
    print ("List after reverse using builtin: ", listn)

nlist = [2,3,4,5,6,100]
revslice(nlist)
revbuiltin(nlist)
```

```
 OUTPUT
Original list before reverse:  [2, 3, 4, 5, 6, 100]
List after reverse using slice operation:  [100, 6, 5, 4, 3, 2]
Original list before reverse:  [2, 3, 4, 5, 6, 100]
List after reverse using builtin:  [100, 6, 5, 4, 3, 2]
```

Example 2.6. Write a python program to find the sum of numbers in the list.

```
#sum of elements in a list

def total(listn):
    print ("Original list: ", listn)
    #find the total in the list using for loop
    total = 0
    for i in listn:
        total = total + i
    print ("Total using for loop =  ", total)

def totalbuiltin(listn):
    print ("Original list: ", listn)
    #find the total in the list
    total = sum(listn)
    print ("Total with builtin sum =  ", total)

nlist = [2,3,4,5,6,100]
total(nlist)
totalbuiltin(nlist)
```

```
OUTPUT
Original list:  [2, 3, 4, 5, 6, 100]
```

```
Total using for loop =   120
Original list:  [2, 3, 4, 5, 6, 100]
Total with builtin sum =   120
```

Example 2.7. Write a python program to find the smallest number in a list.

```
#smallest number in the list

def smallloop(listn):
    print ("Original list: ", listn)
    #find the smallest number in the list using for loop
    minimum = listn[0]
    for i in listn:
        if minimum > i:
            minimum = i
    print ("Minimum using for loop =  ", minimum)

def smallbuiltin(listn):
    print ("Original list: ", listn)
    #find the smallest number using builtin function
    minimum = min(listn)
    print ("Minimum with builtin function =  ", minimum)

def smallsort(listn):
    print ("Original list: ", listn)
    #find the smallest number using sort function
    listn.sort()
    print ("Sorted list ",listn )
    print ("Minimum with sort function =  ", listn[0])

def dashedline():
    print ("-----------------------------------------------------")


nlist = [2,3,4,5,6,100]
smallloop(nlist)
dashedline()
smallbuiltin(nlist)
dashedline()
smallsort(nlist)
dashedline()
```

```
OUTPUT
Original list:  [2, 3, 4, 5, 6, 100]
Minimum using for loop =   2
-----------------------------------------------------
Original list:  [2, 3, 4, 5, 6, 100]
Minimum with builtin function =   2
-----------------------------------------------------
```

```
Original list:  [2, 3, 4, 5, 6, 100]
Sorted list  [2, 3, 4, 5, 6, 100]
Minimum with sort function =   2
```

Example 2.8. Write a python program to print the items of even indices in a list.

```
  #print the even numbers in the list

def even(listn):
    print ("Original list: ", listn)
    for i in range(0,len(listn),2):
        print ("Index =  ", i, " number = ", listn[i])

nlist = [2,3,4,5,6,100]
even(nlist)
```

```
  OUTPUT
  Original list:  [2, 3, 4, 5, 6, 100]
Index =   0  number =  2
Index =   2  number =  4
Index =   4  number =  6
```

Example 2.9.
Write a python program to add two matrices with nested list.

```
#addition of two matrices
mat1 = [[1,2,3],[1,2,1],[2,3,5]]
mat2 = [[1,2,3],[1,2,1],[2,3,5]]
total = [[0,0,0],[0,0,0],[0,0,0]]

for i in range(len(mat1)):
    for j in range(len(mat1[0])):
        total[i][j] = mat1[i][j]+mat2[i][j]
print ("Matrix 1", mat1)
print ("Matrix 2", mat2)
print ("Addition of matrix", total)
```

```
 OUTPUT
Matrix 1 [[1, 2, 3], [1, 2, 1], [2, 3, 5]]
Matrix 2 [[1, 2, 3], [1, 2, 1], [2, 3, 5]]
Addition of matrix [[2, 4, 6], [2, 4, 2], [4, 6, 10]]
```

Example 2.10. Write a python program to get the k^{th} column of the matrix.

```
#get kth column of a matrix
mat1 = [[1,2,3,4],[1,2,1,4],[2,3,5,4]]
#number of rows
print ("The matrix is ", mat1)
rows = len(mat1)
```

```
print ("Number of rows = ", rows)
colnum = int(input("Enter the column number to be printed "))
print ("The column ", colnum, "is: ")
for i in range(0,rows):
    print (mat1[i][colnum])
```

```
OUTPUT
The matrix is  [[1, 2, 3, 4], [1, 2, 1, 4], [2, 3, 5, 4]]
Number of rows =  3
Enter the column number to be printed 3
The column  3 is:
4
4
4
```

Example 2.11. Write a python program to convert the list elements to a tuple and vice versa.

```
#convert list to tuple
listn = [2,3,4,5,6]
print ("The list elements are ", listn)
#convert to tuple
tuplen = tuple(listn)
print ("The converted tuple elments are", tuplen)
#converting back to list
converted_list = list(tuplen)
print ("Converting tuple back to list ", converted_list)
```

```
 OUTPUT
The list elements are  [2, 3, 4, 5, 6]
The converted tuple elments are (2, 3, 4, 5, 6)
Converting tuple back to list  [2, 3, 4, 5, 6]
```

Example 2.12. Write a Python program to join two tuples if the last element of first tuple is equal to the first element of second tuple and print only unique elements.

```
#join two tuple and print unique elements

tuple1 = (2,3,4,5)
tuple2 = (5,6,7,8)

print ("Tuple1 is ", tuple1)
print ("Tuple2 is ", tuple2)

if tuple1[-1] == tuple2[0]:
    tuple3 = tuple1+ tuple2
    unique = set(tuple3)
    print ("The unique elements are: ", unique)
else:
    print ("The last element of one tuple is not equal to the
```

```
    first element of second tuple")
```

```
 OUTPUT
Tuple1 is  (2, 3, 4, 5)
Tuple2 is  (5, 6, 7, 8)
The unique elements are:  {2, 3, 4, 5, 6, 7, 8}
```

Example 2.13. Write a Python program to check whether the even number indices of a tuple are same.

```
#check whether the tuple of immediate even number indices are same

tuple1 = (1,0,1,2,1,3,1)
for i in range(0,len(tuple1)-2):
    if tuple1[i] == tuple1[i+2]:
        print ("indices ", i, i+1, "are same with ")
        print (tuple1[i], " ", tuple1[i+2])
```

```
OUTPUT
 indices  0 1 are same with
1   1
indices  2 3 are same with
1   1
indices  4 5 are same with
1   1
bama@bama-Veriton-M4660G
```

Example 2.14. Write a Python program to generate the jumbled words of a given string.

```
#permutate the given string
from itertools import permutations
word = str(input("Enter the word: "))
word = word.lower()
perm_list = []
for i in permutations(word):
    perm_list = perm_list+ ["". join(i)]

perm_num = len(perm_list)
print ("Total number of jumbled words are ", perm_num)
print ("The jumbled words are ")
print (perm_list)
```

```
OUTPUT
Enter the word: HELLO
Total number of jumbled words are  120
The jumbled words are
['hello', 'helol', 'hello', 'helol', 'heoll', 'heoll', 'hlelo',
'hleol', 'hlleo', 'hlloe', 'hloel', 'hlole', 'hlelo', 'hleol',
```

```
'hlleo', 'hlloe', 'hloel', 'hlole', 'hoell', 'hoell', 'holel',
'holle', 'holel', 'holle', 'ehllo', 'ehlol', 'ehllo', 'ehlol',
'eholl', 'eholl', 'elhlo', 'elhol', 'ellho', 'elloh', 'elohl',
'elolh', 'elhlo', 'elhol', 'ellho', 'elloh', 'elohl', 'elolh',
'eohll', 'eohll', 'eolhl', 'eollh', 'eolhl', 'eollh', 'lhelo',
'lheol', 'lhleo', 'lhloe', 'lhoel', 'lhole', 'lehlo', 'lehol',
'lelho', 'leloh', 'leohl', 'leolh', 'llheo', 'llhoe', 'lleho',
'lleoh', 'llohe', 'lloeh', 'lohel', 'lohle', 'loehl', 'loelh',
'lolhe', 'loleh', 'lhelo', 'lheol', 'lhleo', 'lhloe', 'lhoel',
'lhole', 'lehlo', 'lehol', 'lelho', 'leloh', 'leohl', 'leolh',
'llheo', 'llhoe', 'lleho', 'lleoh', 'llohe', 'lloeh', 'lohel',
'lohle', 'loehl', 'loelh', 'lolhe', 'loleh', 'ohell', 'ohell',
'ohlel', 'ohlle', 'ohlel', 'ohlle', 'oehll', 'oehll', 'oelhl',
'oellh', 'oelhl', 'oellh', 'olhel', 'olhle', 'olehl', 'olelh',
'ollhe', 'olleh', 'olhel', 'olhle', 'olehl', 'olelh', 'ollhe',
'olleh']
```

Example 2.15. Write a python program to simulate a game of jumbled word. Ask the user to enter the correct word and try for 10 times. If correct, print the user is correct with the number of attempts. The program should print the user has lost the game, if the user has tried for 10 attempts.

```
  #Simulate a word jumble game for 10 attempts
word = "reuosc"
print ("The word in jumbled form is ", word)
print ("The game starts ")
print ("————————————————————")
print ("You have ten chances to type the correct word")
n = 10
attempt = 1
while n>0:
    w = str(input("enter the word "))
    wrd = w.lower()
    if w == "course":
        print ("You are correct")
        print ("Number of attempts ", attempt)
        break
    else:
        print ("Try again")
        attempt +=1
    n = n-1
    if n ==0:
        print("You have lost the game")
```

```
 OUTPUT
 The word in jumbled form is  reuosc
The game starts
————————————————————
You have ten chances to type the correct word
```

```
enter the word resocu
Try again
enter the word rescor
Try again
enter the word scourse
Try again
enter the word course
You are correct
Number of attempts  4
```

Example 2.16. Write a Python program to check whether the string starts with a particular sequence of characters. Also check whether the string ends with a different sequence of characters.

```
  #program that starts and ends with particular string
words = ["character","class","remark","channel", "miss"]
for word in words:
    if word.startswith("ch"):
        print("The word starts with ch in ", word)
    if word.endswith("ss"):
        print("The word ends with ss in ", word)
```

```
 OUTPUT
The word starts with ch in  character
The word ends with ss in  class
The word starts with ch in  channel
The word ends with ss in  miss
```

Example 2.17. Write a Python program to get a sentence from the user and find the frequency of the words. Use words and the count as keys and values, respectively.

```
#Count the frequency of word in a sentence. Use dictionary
#with key as word occurrence as value

sen = str(input("Enter the sentence: "))
word = sen.split()
freq = {}
for w in word:
    freq[w] = freq.get(w,0)+1
print ("Frequency of words in the sentence: ")
print (freq)
```

```
OUTPUT
Enter the sentence: The cat went to the hut and
saw another cat in the same hut
Frequency of words in the sentence:
{'The': 1, 'cat': 2, 'went': 1, 'to': 1, 'the': 2,
'hut': 2, 'and': 1, 'saw': 1, 'another': 1,
'in': 1, 'same': 1}
```

Example 2.18. Write a Python program to get the digit in number and print the corresponding letters. Use dictionary.

```
#Get the digit from the user and print the
#corresponding number in letters. Use dictionary
n_dict = {0:"Zero",1:"One",2:"Two",3:"Three",
4:"Four",5:"Five",6:"Six",7:"Seven",
8: "Eight", 9: "Nine"}
n = int(input("Enter the number from 0 to 9: "))
print ("The entered number is ", n)
for i,j in n_dict.items():
    if i == n:
        print ("In letters ", n_dict[i])
```

```
OUTPUT
Enter the number from 0 to 9: 9
The entered number is  9
In letters  Nine
```

Example 2.19. Write a Python program to find the unique values in a dictionary.

```
  #print unique values in a dictionary
dict1 = {"a":199,"b":299,"c":399,"d":199,"e":299}
print ("The dictionary is ")
print (dict1)
values = []
for key,value in dict1.items():
    values = values + [dict1[key]]
print ("The values from dictionary are ", values)
unique_values = set(values)
print ("The unique values are ", unique_values)
```

```
OUTPUT
The dictionary is
{'a': 199, 'b': 299, 'c': 399, 'd': 199, 'e': 299}
The values from dictionary are  [199, 299, 399, 199, 299]
The unique values are  {399, 299, 199}
```

Example 2.20. Write a Python program to sort the dictionary according to the values. Use any relevant library.

```
 #Sort by value
from collections import Counter
dict1 = {"a":499,"b":299,"c":399,"d":199,"e":99}
X = Counter(dict1)
print ("The dictionary is ")
print (dict1)
print ("The sorted values are ")
print (X.most_common())
```

```
OUTPUT
The dictionary is
{'a': 499, 'b': 299, 'c': 399, 'd': 199, 'e': 99}
The sorted values are
[('a', 499), ('c', 399), ('b', 299), ('d', 199), ('e', 99)]
```

2.7 Application of Lists, Tuples and Strings

Lists and tuples can be used extensively owing to the indexing and slicing property. Some common applications include system level administration and development, data analysis and machine learning libraries. Strings are used in text mining applications and natural language processing libraries.

2.8 Summary

1. Lists in Python are ordered, mutable and can have duplicate elements.
2. Tuple is a collection that are ordered and non mutable. It can have duplicate elements.
3. Set is a collection that is unordered and not indexed. It does not have duplicate elements.
4. Dictionary is a collection that is unordered and mutable. It cannot have duplicate key elements.

2.9 Learning Outcomes

On completion of this chapter, you should be able to:

1. Apply lists to solve problems which has an array like structure.
2. Program with tuples where the elements remain permanent.
3. Program using Sets for mathematical and other real world problems.
4. Implement dictionary for problems involving key and value pair.
5. Program using strings in the real world settings.

2.10 Chapter Exercises

2.10.1 Multiple Choice Questions

1. What is the range of index values for the values from 1 to 10.
   (a) 0 to 9
   (b) 0 to 10
   (c) 1 to 10
   (d) 1 to 9
2. Which one of the following is not a common operation on lists?
   (a) retreive
   (b) delete
   (c) interleave

(d) append

3. For nums = [10,20,30,40], what is the output of the following code?

```
for k in nums:
    print (k, end = ',')
```

(a) 10,20,30,40,

(b) 20,10,30,40,

(c) 30,40,10,20,

(d) 30,40,20,10,

4. For nums = [10,20,30,40], what is the output of the following code?

```
for k in range(1,4):
    print (k, end = ',')
```

(a) 10,20,30,

(b) 10,20,30,40,

(c) 10,30,20,

(d) 30,40,20,10,

5. For nums = [12,4,11,23,18,41,27], what is the value of k when the loop terminates?

```
k = 0
while k< len(nums) and nums[k]!=18:
    k = k+1
```

(a) 3

(b) 4

(c) 5

6. Which of these is a mutable type?

(a) strings

(b) lists

(c) tuples

7. Which of the following is the correct way to denote a tuple of one element?

(a) 9

(b) (9)

(c) (9,)

(d) 9

8. For string `s = 'abcdef'`, s[2:4] results in:

(a) 'cd'

(b) 'bcd'

(c) 'bc'

(d) 'cde'

9. For a dictionary `d = ''apples'': 20.00, ''pears'': 25.00, ''bananas'':10.00`, which of the following correctly updates the price of bananas?

   (a) d[2] = 12.00

   (b) d[1] = 12.00

   (c) d[0] = 12.00

   (d) d.update(12)

10. Which of the following can be used as a key in Python dictionaries?

    (a) Strings

    (b) Lists

    (c) Tuples

    (d) Range of Numerical values

11. What is the output of the following piece of code?

```
>>> str1 = "This class is good"
>>> str1.find("is")
```

    (a) 2

    (b) 1

    (c) 3

    (d) 0

12. What is the output of the following piece of code?

```
>>> list1 = [5, 37, "a", [2, 4, "d"], [ ], "pi", False]
>>> list1[2:4]
```

    (a) [5,37, 'a',[2, 4, 'd']]

    (b) ['a', [2, 4, 'd']

    (c) ['a', [2, 4, 'd']]

    (d) [5, 37, "a"]

13. What is the output of the following segment?

```
 >>> t1 = "apple","banana","carrot"
>>> a,b,c = tuple(t1)
>>> a
```

(a) a

(b) “apple”

(c) “banana”

(d) “abc”

14. What is the output of the following segment?

```
 >>> t1 = (100,200,300)
>>> t1[1] = 2
>>>t1
```

(a) Error

(b) (100,2,300)

(c) 2

(d) 200

15. What happens with `(66, 4, 17, 4) < (66, 4, 16, 5)`?

(a) True

(b) False

2.10.2 Descriptive Questions

1. What happens to the memory location when you assign a variable to another variable?
2. What happens to the memory location when you assign a list to another list without slicing operator?
3. What happens to the memory location when you assign a list that has a nested list with slicing operator?
4. What happens to the memory location when you use the module deepcopy to copy the items from one list to another?
5. Explain how elements are sliced with an example?
6. Explain the nested list with examples?
7. What is the difference between list and tuple?
8. Explain the use of sets in Python programming.
9. With examples, explain any five built in methods available in Strings.
10. How will you convert the string to a list and vice-versa?
11. With examples, explain the data structure of dictionary. How is it iterated?

2.10.3 Programming Questions

List

1. Write a python program to find the square of all numbers in the list.
2. Write a python program to multiply the numbers in a list.
3. Write a python program to find the largest number in the list.
4. Write a python program to print all even and all odd numbers in the list.
5. Write a python program to print positive numbers in the list.
6. Write a python program to print negative numbers in the list.
7. Write a python program to remove a particular element from the list.
8. Write a python program to copy the list of elements.
9. Write a python program to count the number of occurrences in a list.
10. Write a python program to print the duplicates of elements.
11. Write a python program to find the cumulative sum in a list.
12. Write a python program to break the list into N chunks.
13. Write a python program to multiply two matrices.
14. Write a python program to transpose a given matrix.

Tuple

1. Write a python program to find the size of the tuple.
2. Write a python program to find the maximum and minimum k elements in tuple.
3. Write a python program to extract digits from tuple elements.
4. Write a python program to find Permutations and combinations.
5. Write a python program to check whether two tuples are same.

Sets

1. Write a Python program to create a set.
2. Write a Python program to iterate over sets.
3. Write a Python program to add member(s) in a set.
4. Write a Python program to remove item(s) from set
5. Write a Python program to remove an item from a set if it is present in the set.

6. Write a Python program to create an intersection of sets.
7. Write a Python program to create a union of sets.
8. Write a Python program to create set difference.
9. Write a Python program to create a symmetric difference.
10. Write a Python program to check if a set is a subset of another set.
11. Write a Python program to create a shallow copy of sets.
12. Write a Python program to clear a set.
13. Write a Python program to use of frozensets.
14. Write a Python program to find maximum and the minimum value in a set.
15. Write a Python program to find the length of a set.
16. Write a Python program to check if a given value is present in a set or not.
17. Write a Python program to check if two given sets have no elements in common.
18. Write a Python program to check if a given set is superset of itself and superset of another given set.
19. Write a Python program to find the elements in a given set that are not in another set.
20. Write a Python program to check a given set has no elements in common with other given set.
21. Write a Python program to remove the intersection of a 2nd set from the 1st set.

2.10.4 Strings

1. Write a program that counts the occurrences of a character in a string. Do not use use built-in functions.
2. Write a program to reverse the string.
3. Write a program to parse an email id such that it has only the date and time.
4. Write a program to print the first word of a string.
5. Write a program that prints only words that starts with a vowel.
6. Write a program that accepts a comma separated sequence of words and print only unique words
7. Write a function to insert a string in the middle of the string.
8. Write a program to remove the n^{th} index character from a non-empty string.
9. Write a program to get the information as First name and Last name and print the initial with first name.

10. Write a program to count the number of digits, upper case letters, lower case letters and special characters in the given string.
11. Write a program to print the first n characters in the given string.
12. Write a program with two functions to delete the first and last characters of a string.
13. Write a python program to find the bigger length of two strings
14. Write a python program to find common characters in two strings
15. Write a python program to remove the repeated characters.
16. Write a python program to remove extra spaces within string.
17. Write a python program to create a password with different types such that the entered value should be not less than six characters long and should have an integer and special character.

2.11 References

- Reema Thareja, Python Programming using Problem Solving Approach, Oxford University Press, 2019
- Charles Dierbach, Introduction to Computer Science using Python A Computational Problem-Solving Focus, Wiley Publications, 2015
- https://www.w3schools.com/python/python_datatypes.asp
- https://openbookproject.net/thinkcs/python/english3e/variables_expressions_statements.html
- https://openbookproject.net/thinkcs/python/english3e/functions.html
- https://openbookproject.net/thinkcs/python/english3e/iteration.html
- https://openbookproject.net/thinkcs/python/english3e/strings.html
- https://openbookproject.net/thinkcs/python/english3e/lists.html
- https://openbookproject.net/thinkcs/python/english3e/tuples.html
- https://openbookproject.net/thinkcs/python/english3e/recursion.html

  Following links give some references of day to day applications/ modules that use list, tuples, sets and strings
- https://www.programiz.com/python-programming/directory
- https://www.tensorflow.org/
- https://scikit-learn.org/stable/
- https://www.nltk.org/

Chapter 3

FILE HANDLING AND EXCEPTION HANDLING

Learning Objectives

- To read and write external files through Python Program
- To perform different string operations in text files through Python Program
- To introduce exception error handling
- To write efficient programs without errors

In this Chapter, you will see how external files can be accessed with Python in Section 3.1. Since the language supports string extensively, you will see many examples with respect to text files. Apart from files, you will study about the class of errors that can be caught while programming. These are called Exceptions, which are provided in Section 3.2.

3.1 Files

To open a file from your folder, you will probably follow these steps.

- First navigate to that folder
- Select the file with mouse and click

Alternately, you can also open the file with a command by 'cd' to the specific folder and then type the name of the file with an extension. Assuming the file is a text file, you can edit the text and save it for future use. All these operations can be done using Python.

3.1.1 File path

A file path is the location of the file and it generally constitutes three parts:

- Folder path, that provides the location of the folder or directory.
- File name
- Extension of the file

The path in Windows and Linux Environment folder path differ. Suppose you have a text file named 'test.txt' in Desktop, then the path for this file through Windows will be similar to '`C:/Users/tom/Desktop/test.txt`', assuming 'tom' is the user.
In Linux, the file path is '`/home/tom/Desktop/test.txt`'.

This path is important and should be specified when working with files in Python. When talking about files, two types of files can be classified.

1. Textfiles containing characters with a common structure of a single line (sentence). When you read the text you understand that one sentence through the mark of full-stop. In Python, the mark is through a newline character `\n`.

2. Binaryfiles contain binary data, which can be recognized only by machine.

3.1.2 Opening and Closing files

To open a text file, use the keyword 'open'. The syntax of opening the file is given below.

```
file_variable = open(file_name [, access_mode][, buffering])
```

Here is the explanation for this syntax.

- 'file_variable' indicates the specific file which can be referred in the program. This variable can be linked to the file object.
- 'file_name' indicates the name of the file
- 'access_mode' denotes the mode the file is to be accessed such as read, write, append etc.
- 'buffering' is optional. If set to 0 no buffering occurs. Default is -1, which is the system buffer size.

Different access methods are:

1. 'r' - opens a file for reading only.

2. 'rb' - opens a file for reading only in binary format.

3. 'r+' - opens a file for both reading and writing.

4. 'rb+' - opens a file for both reading and writing in binary format.

5. 'w' - opens a file for writing.

6. 'wb' - opens a file for writing in binary format.

7. 'w+' - opens a file for reading and writing. It overwrites the existing file and creates a new file.

8. 'wb+' - opens a file for reading and writing in binary format. Like 'w+', it creates a new file if it does not exist. If exists, the file overwrites with the information.

9. 'a' - Appends the file.

10. 'ab' - Appends the file in binary format.

11. 'a+' - Opens the file for appending and reading.

12. 'ab+' - Opens the file for appending and reading in binary format.

Once the file is opened through the 'file_name' variable, it can be used with different methods such as:

1. file.closed - Returns the value of 'True' if file is closed, else 'False'.

2. file.mode - Returns the access mode of the file.

3. file.name - Returns the name of the file.

These methods are illustrated in the following example. The file 'course.txt' is located in the same folder where this python code also resides. The first line specifies the opening of the file with the access mode of 'a+' and different objects are called. Try these with different access methods.

```
f = open("course.txt",'a+')
print("Check whether the file is open or closed: ", f.closed)
print("Access mode of the file: ", f.mode)
print("Name of the file: ", f.name)
```

```
OUTPUT
Check whether the file is open or closed:  False
Access mode of the file:  a+
Name of the file:  course.txt
```

To close a file, `filename.close()` can be used as shown in the following example.

```
f = open("course.txt",'a+')
f.close()
print ("Check whether the file is closed ", f.closed)
```

```
OUTPUT
Check whether the file is closed  True
```

The files can be opened using `with` keyword also. If this is used, you need not explicitly close the file. An example is given below.

```
with open('course.txt','r') as file:
   for line in file:
       print(line)
```

3.1.3 Reading and Writing files

Continuing with the above Section, here you will see how to read the contents of the file and to write. We will use the same example text file named 'course.txt' with the content 'This is a file to experiment with python handling.'. First, let us read the contents of this statement from python program. To do so, you have to use `filename.read()`. The following program provides an illustration of reading the contents from the file. Here, 'str' is the variable where the file contents are read and then printed.

```
f = open("course.txt",'r+')
str = f.read()
print (str)
f.close()
```

```
OUTPUT
This is a file to experiment with python handling.
```

If there are multiple lines, use `filename.readlines()` to read line by line.

To write the content to the file, use `filename.write()` with appropriate access method. An example is shown below, which write the statement to the file 'course.txt'.

```
f = open("course.txt",'a+')
str = "Python supports text files extensively. "
f.write(str)
print ("The content is written to the file")
f.close()
```

```
 OUTPUT from python
The content is written to the file
OUTPUT from course.txt
This is a file to experiment with python handling.
Python supports text files extensively.
```

3.1.4 File Position

When reading files, a specific pointer to the file position can be created using the methods `tell` and `seek`. The `tell` method gives the current position within the file. The `seek` method has the syntax of `(offset,from)`, where 'offset' indicates the position in terms of bytes to be moved and 'from' indicates the reference position from where the bytes are moved. Consider the example below. Here, `read(10)` indicates 10 bytes to be read. Now, the variable of position1 is assigned to this number through 'f.tell()'. The position1 is given as offset value to 'seek' with 'from' value as 0. According to these positions, the text is displayed as shown in the output. Try varying the values instead of 10, 'offset' and 'from' and check for yourself.

```
 # opening file
f = open("course.txt",'r+')
str = f.read(10)
print ("Reading 10 characters from file....")
print(str)
print ("-----------------")

# check the current position
position1 = f.tell()
print ("The current file position: ", position1)
print ("-----------------")
#reposition pointer to a particular offset
position2 = f.seek(position1,0)
str = f.read(30)
print ("Reading again.....")
print (str)

#close the file
f.close()
```

```
 Reading 10 characters from file....
This is a
```

```
The current file position:  10
```

```
Reading again.....
file to experiment with python
```

3.2 Errors and Exception

You might have come across some errors while you program. These can basically be classified into syntax errors, logical errors and exceptions. You might get a syntax error when you forget an indent or did not specify brackets in print statement. These are printed while executing the program. See the example below.

```
>>> for i in (1,5):
print (i)
SyntaxError: expected an indented block
```

Logical error is due to the understanding of the programmer in the wrong sense. In this case, the program executes but gives a different answer. For example, the requirement may be to find the square of the number, but the code is written for multiplying the two numbers. These cases of programs are difficult to debug.

```
#program demonstrating logical error
n = int(input("enter any integer to find the square of the number "))
sqr  = n*2
print ("The square of the number is ", sqr)
```

```
OUTPUT
enter any integer to find the square of the number 6
The square of the number is  12
```

Another class of programs pass syntax error but throw errors like 'type error' which include 'divide by zero'. These types of errors are already built in so that these can be easily surpassed in a program. For example, the math module has a factorial function that calculates the factorial of integers. But when -1 is entered, Value error arises as shown below.

```
 #program demonstrating Value error
import math
n = int(input("enter any integer to find the factorial
                              of a number "))
print ("The factorial of the number is ", math.factorial(n))
```

```
 OUTPUT
 enter any integer to find the factorial of a number -1
Traceback (most recent call last):
  File "exception_error1.py", line 4, in <module>
    print ("The factorial of the number is ", math.factorial(n))
ValueError: factorial() not defined for negative values
```

These types of exception can be well handled within a program.

3.2.1 Exception handling

Exception can be handled using the blocks **try**, **except** and **finally**. The exceptions can be defined either by the programmer or can be caught from the errors from Python such as Value error. To start with, see what happens when you type **print**(n) and execute. You will see this error.

```
 OUTPUT
 Traceback (most recent call last):
  File "exception_error2a.py", line 1, in <module>
    print(n)
NameError: name 'n' is not defined
```

This error can be handled within Program using **try** and **except**.

```
 try:
    print(n)
except:
    print("An exception occured because the variable is not assigned")
```

```
 OUTPUT
 An exception occured because the variable is not assigned
```

3.2.2 Multiple Exceptions

You can define multiple blocks to find the particular error.

try-except-except

Continuing with the same example, here two blocks are provided with exception. First, the **try** block executes and if this fails, the next **except** block executes and if successful, prints the message. Else the next **except** message gets executed.

```
 try:
    print(n)
except NameError:
    print("Variable not defined")
except:
    print("Something else is wrong")
```

```
 OUTPUT
 Variable not defined
```

try-except*-else

If nothing went wrong and you want to check that, you can include **else** as shown below.

```
 try:
    n = 5
    print(n)
```

```
except NameError:
    print("Variable not defined")
except:
    print("Something else is wrong")
else:
    print("Nothing is wrong")
```

```
 OUTPUT
 5
Nothing is wrong
```

try-except*-else-finally

finally is executed regardless of other blocks being executed. See the example below. Here, the blocks **try**, **except** nameError and **finally** are executed.

```
 try:
    #n = 5
    print(n)
except NameError:
    print("Variable not defined")
except:
    print("Something else is wrong")
else:
    print("Nothing is wrong")
finally:
    print("relevant blocks are executed")
```

```
 OUTPUT
 Variable not defined
relevant blocks are executed
```

raise exception

A keyword of **raise** can be used in the program to indicate that the error will occur. In the following example, the exception is raised if string is used instead of integers.

```
 n = "abc"
if not type(n) is int:
    raise TypeError("Enter only integers")
```

```
 Traceback (most recent call last):
  File "exception_error5.py", line 3, in <module>
    raise TypeError("Enter only integers")
TypeError: Enter only integers
```

Here, you have seen an example of NameError. This is a built-in exception available in Python. A few other exceptions are given in Table 3.1.

Table 3.1: A few built-in exceptions in Python

Sl. No	Exception	Description
1	StandardError	Base class for all built-in exceptions
2	ArithmeticError	Base class errors due to mathematical calculations
3	OverflowError	Raised when the maximum limit of a numeric type is exceeded
4	FloatingPointError	Raised when the floating point calculation could not be performed
5	ZeroDivisionError	Raised when a number is divided by zero
6	AssertionError	Raised when the assert statement fails
7	EOFError	Raised when end of file is reached or there is no input for input() function
8	KeyError	Raised when a key is not found in the dictionary
9	NameError	Raised when a variable is not found in local or global namespace
10	TypeError	Raised when two or more data types are mixed without coercion

3.3 Worked out Programming Problems

Example 3.1. Write a Python program to display the contents of the file in the form of list.

```
  #program to display the contents of file in list.
f = open('course.txt','r')
print (list(f))
f.close()
```

```
  OUTPUT
  ['This is a file to experiment with python handling.\n',
  'Python supports text files extensively. ']
```

Example 3.2. Write a program to split the words and count the number of words in each sentence in the file.

```
  #program to display the count of words in each sentence
with open('course.txt','r') as file:
    for line in file:
        print ("Actual Sentence: ", line)
        words = line.split()
        print("Number of words = ", len(words))
```

```
 OUTPUT
 Actual Sentence:  This is a file to experiment with python handling.

Number of words =  9
Actual Sentence:  Python supports text files extensively.
Number of words =  5
```

Example 3.3. Write a Python program to copy the file to another file.

```
  #program to copy the contents of one file to another
with open('course.txt','rb') as file1:
    with open ('copied.txt','wb') as file2:
        text = file1.read()
        file2.write(text)
    print ("File is copied to file2")
```

```
OUTPUT
File is copied to file2
```

In this example, the contents of course.txt is copied to copied.txt.

Example 3.4. Write a Python program to rename a file.

```
  #renaming a file
import os # importing the os system command
os.rename('course.txt','course1.txt')
print('course.txt is renamed to course1.txt')
```

```
course.txt is renamed to course1.txt
```

This can be checked in the folder where the textfiles and codes are residing.

Example 3.5. Write a Python program to create a directory named as 'tenfiles'. Within that 10 files, create 10 textfiles with names file0, file1, ...,file10. In each of these files, write the text content - 'This is file 1', 'This is file 2' appropriately.

```
  #create 10 text files in a new directory and enter the string
# "This is file 1, This is file 2,.. " in each of the files
import os
directory = "tenfiles"
os.mkdir(directory)
for i in range(11):
    fname = "tenfiles/file"+str(i)
    with open(fname,'w+') as f:
        content = "This is file " + str(i)
        f.write(content)
```

Output of this code is shown in Figures 3.1 and 3.2.

OUTPUT

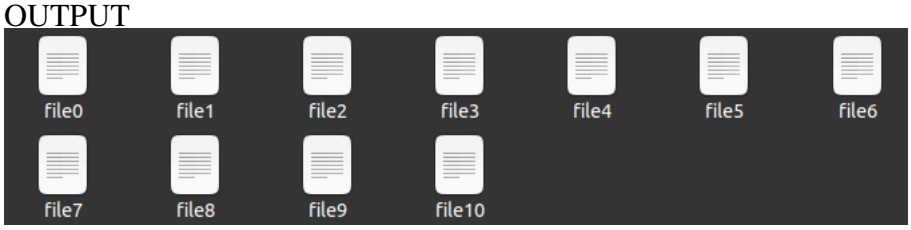


Figure 3.1: creation of ten files in the directory tenfiles

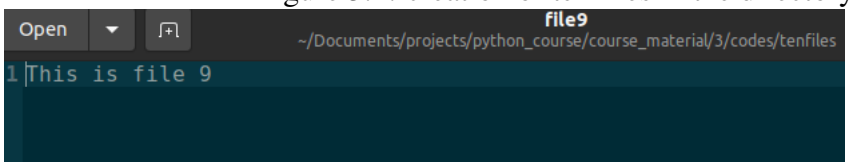


Figure 3.2: Content in file9

Example 3.6. Write a Python program to delete a file.

```
  #delete a file
import os
os.remove("copied.txt")
print ("The text file named copied.txt is removed")
```

OUTPUT
The folder can be checked for the file 'copied.txt' which no longer exists after the execution of this code.

Example 3.7. Write a Python program to count the number of tabs, spaces and newline characters in a string. The poem.txt has these contents.

```
The song I came to sing
remains unsung to this day.
I have spent my days in stringing
and in unstringing my instrument.

The time has not come true,
the words have not been rightly set;
only there is the agony
of wishing in my heart.....
         A poem by Rabindranath Tagore
```

```
  #count number of tabs, spaces and newline characters in a text
with open ('poem.txt','r') as file:
    text = file.read()
    count_tab = 0
    count_space = 0
    count_newline = 0
    for char in text:
        if char == '\t':
            count_tab += 1
        if char == ' ':
```

```
            count_space +=1
        if char =='\n':
            count_newline += 1
print ("Tabspace = ", count_tab)
print ("Spaces = ", count_space)
print ("newlines = ", count_newline)
```

```
 OUTPUT
Tabspace =  1
Spaces =  42
newlines =  10
```

Example 3.8. Write a Python program to take a text file from URL and save those to a text file.

```
  #copy the text content from url to a file
import urllib.request

url = "http://textfiles.com/food/aphrodis.txt"
text = urllib.request.urlopen(url)
data = text.read()
#print (text)
file = open('urltext','w+')
file.write(str(data))
file.close()
print ("The content is written")
```

```
OUTPUT
 The content is written
```

Example 3.9. Write a Python program to copy the file in such a way that the words in each sentence is reversed.

```
  #program to copy the contents of one file to another with
  #reversal words
with open('course1.txt','r') as file1:
    with open ('copied.txt','a+') as file2:
        text = file1.readlines()
        for line in text:
            words = line.split()
            words.reverse()
            reverse_sent = " ".join(words)
            print (reverse_sent)
            file2.write(reverse_sent)
            file2.write('\n')
```

```
  OUTPUT
line. first the is This
```

```
line. second the is This
line. third the is This
```

Example 3.10. Write a Python program to print a random line from a text file.

```
  #program to print a random line from the file

import random
with open ('poem.txt','r') as file:
    lines = file.readlines()
    print (random.choice(lines))
```

```
 OUTPUT
 and in unstringing my instrument.
```

Example 3.11. Write a Python program to demonstrate divide by zero error through exception.

```
  def divide(nr,dr):
    try:
        quotient = nr/dr
    except ZeroDivisionError:
        print("You cannot divide a number by 0")
n = 10
divide(n,0)
```

```
  OUTPUT
  You cannot divide a number by 0
```

Example 3.12. Write a Python program to demonstrate index error in a list. (Index error arises when the index value is out of range).

```
 #demonstration of index error
def count(list1):
    print("The number of elements in list ", len(list1))
    try:
        last_element = list1[3]
    except IndexError:
        print("Index value should be less than 3")

n = [10,20,30]
count(n)
```

```
 OUTPUT
 The number of elements in list  3
Index value should be less than 3
```

Example 3.13. Write a Python program to demonstrate Value error.

```
  #demonstration of value error
def gettype(num):
    print("The actual type is of ", num, "is", type(num))
    print ("Now, try convert this type to int")
    try:
        int(num)
    except ValueError:
        print("Throwing exception......")
        print("String cannot be converted  to integer value")

n = '15.64'
gettype(n)
```

```
 OUTPUT
The actual type is of  15.64 is <class 'str'>
Now, try convert this type to int
Throwing exception......
String cannot be converted  to integer value
```

Example 3.14. Write a Python program to raise an exception for the values of month. In other words, if the user enters the value other than 1 to 12, the program should raise an exception.

```
  #get the input of 1 to 12 for month. Else raise exception
month = int(input("Enter the month in values from 1 to 12 "))
if month < 1 or month > 12:
    raise ValueError("Enter only values between 1 to 12")
else:
    print("The value of month is ", month)
```

```
 OUTPUT
 Enter the month in values from 1 to 12 -2
Traceback (most recent call last):
  File "month.py", line 4, in <module>
    raise ValueError("Enter only values between 1 to 12")
ValueError: Enter only values between 1 to 12
```

Example 3.15. Write a Python program to demonstrate KeyboardInterrupt exception. This exception is used when the user presses Ctrl + C to terminate the execution of the program.

```
  #Example for keyboard interrupt error
try:
    inp = input()
    print ('Press Ctrl+C or any other key')
except KeyboardInterrupt:
    print ('Caused by KeyboardInterrupt')
else:
    print ('No exception occurred')
```

```
OUTPUT
^CCaused by KeyboardInterrupt
```

Example 3.16. Write a Python program to write to file and raises an IOError exception.

```
  #file input output error
try:
    #The access mode of the file is read,
    #but the content is to be written.
    #So, exception of IOerror block gets executed
    with open("eg.txt",'r') as file:
        file.write("Adding a new statement. ")
except IOError:
    print ("Error working with file ")
else:
    print ("Written the statement to file")
```

```
  OUTPUT
  Error working with file
```

Example 3.17. Write a Python program to print the values infinitely. Then call an exception of StopIteration to print only upto 10 natural numbers.

```
  #illustration of StopIteration
def add(num):
    while True:
        try:
            num = num + 1
            if num == 11:
                raise StopIteration
        except StopIteration:
            break
        else:
            print(num)

x = 0
add(x)
```

```
1
2
3
4
5
6
7
8
9
10
```

Example 3.18. Write a Python program to illustrate overflow exception. This exception occurs when the value is out of range.

```
  #illustration of overflow exception
import math
try:
    n = math.exp(863)
except OverflowError:
    print("Overflow exception is raised. ")
else:
    print ("Success, the value of e^(863) is ", n)
```

```
OUTPUT
 Overflow exception is raised.
```

If the value is changed from 863 to 5, the output `Success, the value of e^(5) is 7.225973768125749e+86` occurs.

Example 3.19. Write a Python program to illustrate assertion error.

```
  #illustration of assertion error
n = -5
try:
    assert n > 5
    print("number is positive")
except AssertionError:
    print("Assertion error exception raised")
```

```
OUTPUT
Assertion error exception raised
```

Example 3.20. Write a Python program to deliberately raise the exception.

```
  #raise an exception
try:
    n = 5
    print(n)
    raise ValueError # deiberately raising an exception
except:
    print("Exception raised")
```

```
  OUTPUT
  5
Exception raised
```

3.4 Summary

- Text files and binary files can be handled with Python extensively.

- Files can be accessed through the file path in any operating system by importing `os` module.
- Each files ends with end-of-file marker, which can be used for programming.
- open() method creates file object
- close() method closes the file. However if `with` keyword is used, close() is not necessary.
- Read, Write and Append access methods are available for both text and binary files.
- tell() and seek() methods provide the location of file pointer information
- Exceptions can be handled using **`try`**−**`except`** block.
- Several built-in methods of exception can be integrated while programming.
- User defined exceptions are also possible to catch exception.
- Multiple blocks of exception can be used to find the exact error.

3.5 Learning Outcomes

After reading and working out the examples in this chapter, you should be able to:

1. Write programs to read and write to text file.
2. Analyse the text and binary files.
3. Work with a large set of files and automate the requirements.
4. Add built-in and user-defined exceptions to catch errors

3.6 Chapter Exercises

3.6.1 Multiple Choice Questions

1. To access files, this function is used.
   (a) access()
   (b) open()
   (c) read()
   (d) write()
2. To close a file, you use
   (a) close(file)
   (b) file.close()
   (c) cl.file
   (d) file.close
3. To get the entire contents of the file, you use the method
   (a) file.read()
   (b) file.access()
   (c) file.read

(d) read(file)

4. The default access mode is

   (a) r

   (b) rw

   (c) w+

   (d) b

5. Suppose a file named 'course.txt' has 8 lines of text. The code `len(open('course.txt').readlines())` will return.

   (a) 1

   (b) 8

   (c) 0

   (d) 5

6. When you open a file for reading, the file pointer is located at location

   (a) 0

   (b) 1

   (c) 2

   (d) k

7. What is the correct way to write the string to a file.

   (a) file.write("This is the text message")

   (b) write(file,"This is the text message")

   (c) "This is the text message". write(file)

   (d) file("This is text message"). write

8. In the seek method, the reference postion of **from** at the current position of the file is at

   (a) 0

   (b) 1

   (c) 2

   (d) -1

9. An exception is a

   (a) module

   (b) object

   (c) procedure

   (d) method

10. Which of the following is not a standard exception in Python?

(a) IOError

(b) NameError

(c) AssignmentError

(d) ValueError

11. Which of the following errors executes the program but gives wrong results?

(a) Exception

(b) Logical Error

(c) Syntax Error

(d) None of these

12. Which block is executed when no exception occurs?

(a) try

(b) except

(c) catch

(d) raise

13. Which keyword is used to generate an exception?

(a) raise

(b) throw

(c) assert

(d) print

14. Which of the following blocks handle exception?

(a) try

(b) except

(c) finally

(d) def

15. '5' == 5 will result in

(a) TypeError

(b) ValueError

(c) True

(d) False

16. Which of the following blocks are optional to catch an exception?

(a) try

(b) except

(c) finally

(d) except-else

3.6.2 Descriptive Questions

1. What are files? How are those handled in Python?
2. Differentiate text and binary files.
3. What are the different access modes in Python? Explain any five with examples.
4. With examples, explain the attribute of file object.
5. Explain the syntax of read and readlines in Python.
6. Give the significance of `with` method.
7. With examples, explain how you will write to a file.
8. Differentiate error and exception.
9. Differentiate syntax, logical errors and exception.
10. What happens when an exception is raised in a program.
11. Explain the syntax of try-except block.
12. How will you handle multiple exceptions in Python?
13. What is the significance of **`finally`** block?
14. Explain any three built-in exceptions with examples.
15. How can you create your own exceptions in Python.

3.6.3 Programming Questions

1. Write a Python program to read a text file and count the number of particular words in the file.
2. Write a Python function to read a text file and display the words starting with the letter 'R'.
3. Write a Python Program to compare two text files.
4. Write a Python Program to combine two text files and place in the third file.
5. Write a Python program to read all the contents from one file. Then copy into another file with the conversion of lower-case to upper-case.
6. Write a Python program to enter details into two files separately and merge those.
7. Write a Python program to read the file. From the content, count the number of articles. Copy the file to another file, remove the articles.
8. Write a Python program to append the content to an already available text file and display those.
9. Write a Python program to read first n lines of text file.

10. Write a Python program to find the longest word in the text file.
11. Write a Python program to find the frequency of words in a file.
12. write a Python program to remove a newline character from the file.
13. Write a Python program to extract the characters from the text file and include those in a list.
14. Write a program to find the square root of a number. If the number is negative, raise an assertion error.
15. Write a program to get the input of a number from the user. If the user enters a string value, the program should catch the error through an exception.
16. Write a program that validates the user's name with an initial as fullstop. If 'fullstop' is not entered by the user, the program should raise an exception.
17. Write a program that has multiple except blocks.
18. Write a program to add the even numbers to the list infinitely. Raise an exception such that the list should store only 20 even numbers.
19. Consider the list `[20,30,40,80]`. Ask the user to get the index value and display the corresponding element. Add an exception such that the user can only give the index values from this list.
20. Write a program to raise exception while working with files.

3.7 References for External Learning

- Reema Thareja, Python Programming using Problem Solving Approach, Oxford University Press, 2019
- Charles Dierbach, Introduction to Computer Science using Python A Computational Problem-Solving Focus, Wiley Publications, 2015
- `https://docs.python.org/3/c-api/exceptions.html`
- `https://www.programiz.com/python-programming/exception-handling`
- `https://realpython.com/python-exceptions/`
- `https://www.w3schools.com/python/python_try_except.asp`

Chapter 4

MODULES AND PACKAGES

Learning Objectives

- To understand how external modules can be imported.
- To create a customized module.
- To develop a package with Python programs.
- To use numpy, pandas and plotting libraries in Python.

4.1 Modules

Modules are small pieces of code, which can be collectively put together for a common purpose. To give an analogy, assume how the construction of a simple chair takes place. The legs, handles and body of the chair have to be built separately and then assembled together. In a similar way, when you are writing a program it is always a good practice to separate into smaller chunks of code that does a specific job. These smaller chunks of code constitute a module. Here are a few advantages to use modules in programming.

1. Portability: Each module constitute a specific function or functions, which are smaller in size.

2. Maintenance: Debugging and updating a single module is easier.

3. Re-usability: The modules can be imported to other program constructs.

Python provides an efficient way of importing and create modules. A Python module constitutes definition and statements. By default, a python program that is executed is considered to be a main module of the program. These are given a special name `__main__`. These provide the basis for the entire program and include other modules too.

The standard library of Python has a set of pre-defined standard modules, some of which you have used already like `math`, `random` and `os`.

4.1.1 Creating your own module

You can create a module by writing a Python program and giving it an extension of `.py`. As a first example, let us create a module two functions, function1 and function2. Within each function, a print statement can be given as shown below. Save this program as `simple.py`. The name of this file is the name of module, which can be used in other programs.

```
#module simple
print ("module simple loaded")

def function1():
    print ("function1 called")

def function2():
    print ("function2 called")
```

Now, write two other python scripts which calls the module (the program simple.py) similar to the one shown below. Save these two programs as prog1.py and prog2.py in the same folder where you have saved the above program `simple.py`. The line `import simple` specifies the module named `simple` to be imported to the program. The next line `print(simple.function1())` specifies the module name followed by a dot with particular function name. In this case, function1 from simple module is used.

```
#prog1.py
import simple
print(simple.function1())
```

```
#prog2.py
import simple
print(simple.function2())
```

On execution of these programs, you will notice that the print statement from simple.py is printed as shown below.

```
OUTPUT from prog1.py
module simple loaded
function1 called
OUTPUT from prog2.py
module simple loaded
function2 called
```

To see all functions in the module, **dir(modulename)** is used. For example, prog1.py is modified as shown below to include **dir(simple)**.

```
#prog1.py
import simple
print (dir(simple))
print(simple.function1())
```

The output of (dir(simple)) lists all the functions as listed below, which include both in-built and user defined functions.

```
 OUTPUT
 module simple loaded
['__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__',
'function1', 'function2']
function1 called
```

Python provides an alternate import constructs in the form
`from modulename import something` and `import modulename as somename`. The first construct instructs the particular function or variable to be imported and the second specifies the name of module as something else. An example of the first construct is explained below.

Consider the example of finding a total of three numbers in a list. The function of total is stored in 'totavg.py' as:

```
#module
def total(l1):
    tot = l1[0]+l1[1]+l1[2]
    return tot
```

This file acts as a module and can be used by other programs. Here, the following program imports the above program using `from totavg import total` line. The values of 5,6 and 7 are passed to the function located at totavg. In the line `print (total(list1))`, note that total(list1) is used, which indicates the function from line 2.

```
1.#total.py
2.from totavg import total
3.list1 = [5,6,7]
4.print("Sum of the numbers of ", list1)
5.print (total(list1))
```

```
OUTPUT
Sum of the numbers of  [5, 6, 7]
18
```

Like functions, the variables from the module can also be called. Consider the module named student.py which stores the details of student as:

```
#student.py
person = {"Rollnumber": 250123, "Name": "Tamil Selvan", "Course":"MCA"}
num = 50
```

From another program, the name of this person can be accessed as:

```
#prog3.py
from student import person
print ("Printing the name of the person from Student module ")
print (person["Name"])
```

```
OUTPUT
Printing the name of the person from Student module
Tamil Selvan
```

Note here that 'person' is the variable from module student.

Change the import statement to `from student import person as st` and accordingly change the last line to `print(st[''Name''])`. See the result for yourself.

Other alternatives of import statements in Python are
`from modulename import function1, function2`, which imports two functions from the module and `from modulename import *`, which imports all functions from module.

Namespace in Python provides an efficient way to address the identifier. Using namespace avoids the nameclash incase the same identifier is used. For example, consider two modules module1.py and module2.py which have the same function double, but different functionality.

```
#module1
def double(num):
    print (num*2)
```

```
#module2
def double(num):
    print (num,num)
```

The main program hence will have a nameclash. In such cases to identify the specific function from a module, the namespace is specified through a module in the form of `modulename.function`. For this example, the namespaces are `module1.double()` and `module2.double()`.

```
import module1
import module2
print (module1.double(5))
print (module2.double(5))
```

```
OUTPUT
10
None
5 5
```

4.1.2 Module Loading and Execution

The modules that you have seen above is located in the same place where you are writing programs to import those. But these can also be placed in the `PYTHONPATH` directory - that is where Python is stored.

By import statement, the module is loaded and a compiled version of `.pyc` is automatically generated. It should be located in __pycache__ folder. An example of `.pyc` files used for the above set of programs are shown in Figure 4.1. This compiled file is used for loading in

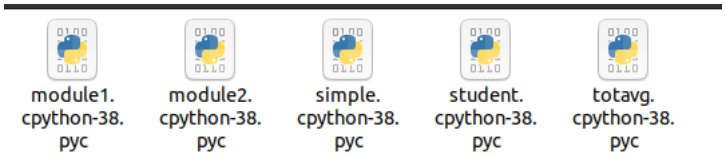


Figure 4.1: Compiled ptyhon modules with .pyc extension

the subsequent execution which saves computational time in loading again. If the module is modified, accordingly the compiled files also gets updated.

4.1.3 Packages

When you are developing a software, you will have to create many programs as modules. To bundle up these, a package is required so that the relevant modules are in one place. Python provides an efficient way of packaging and distributing across different platforms.

Here, you will understand how to develop a package with multiple modules. For simplicity, two modules namely, add.py and subtract.py can be considered as follows.

```
#addition module
def ad(n1,n2):
    print("Addition of two numbers ", n1, n2, " ", n1+n2)
```

```
#subtraction module
def subtraction(n1,n2):
    print("Subtraction of two numbers ", n1, n2, " ", n1-n2)
```

To execute, one way is to call these modules from another program such as calc.py, which is located where the two modules add.py and subtract.py are located.

```
#calc.py
from add import ad as total
from subtract import subtraction
a = 10
b = 5
total(a,b)
subtraction(a,b)
```

On executing this program, you will get the following output.

```
Addition of two numbers  10 5   15
Subtraction of two numbers  10 5   5
```

To bundle up all the modules and other information, you can use packaging mechanism. To create a package in Python, first a directory should be created where the modules are to be placed. Along with the modules, an empty file with `__init__.py` should be included. This file indicates that the package is in place. The structure of the example files looks like:

```
package
   |
   |-----__init__.py
   |
   |-----add.py
   |
   |-----subtract.py
```

First create a directory named ‘package’ and within that place the modules ‘add.py’ and ‘subtract.py’. Along with that, create an empty file called `__init__.py`. Now, outside the directory have a python file with these information.

```
1. from package import add,subtract
2. a = 100
3. b = 80
```

```
4. add.ad(a,b)
5. subtract.subtraction(a,b)
```

The first line denotes the package directory to be considered as package and import the modules add and subtract. There are other variants of import statements which can also be included. Line 3 and Line 4 indicate the modules to be executed from the package. The output of running this program is shown below.

```
Addition of two numbers  100 80   180
Subtraction of two numbers  100 80   20
```

4.2 The Python Libraries for data processing, Data Mining and Visualization

Python is one of the popular languages used by scientists, economists and software developers owing to the range of libraries. In this section, the prevalent three libraries namely Numpy, Pandas and Matplotlib are discussed.

4.2.1 Numpy

Numpy is a library to perform numerical computations with ready made functionalities, particularly in the form of matrices. It stands for Numerical Python. This library is also optimized to work with various CPU architectures. First you have to install `numpy`. Refer `https://pip.pypa.io/en/stable/installing/` to install `pip` and through that you have to install `numpy` with the command `pip install numpy`.

With `numpy`, following topics are discussed.

1. Creating Array
2. Indexing and Slicing Arrays
3. Shape and Reshape
4. Iterating Arrays

Creating Array

An array can be created using an `array` keyword. In the following example, an array of integer numbers $[2, 4, 6, 8]$ is created.

```
 >>> import numpy as np
>>> a = np.array([2,4,6,8])
>>> a
array([2, 4, 6, 8])
```

The first line indicates the module `numpy` imported as `np`. This is then used in the second line to convert to an array. Note that the array is represented with a tuple and then square bracket. If only square bracket is given, it will lead to an error. Arrays are declared using `dtype`, which are different from the data structure of list. Manipulations with arrays are faster than that of looping within lists.

```
>>> a = np.array([200,400,600])
>>> a
array([200, 400, 600])
>>> type(a)
<class 'numpy.ndarray'>
```

1-D, 2-D and higher dimension arrays can be created with `numpy`. Example of creating 2-D and 3-D arrays are shown below.

```
#Example of 2–D and 3–D array
import numpy as np
#creation of 2–d array
arr = np.array([[1,2,3],[4,6,8]])
print(arr)
#shape
print(arr.shape)
#creation  of 3–d array
arr2 = np.array([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]])
print (arr2)
print (arr2.shape)
print (arr2.ndim)
```

```
The array is [[1 2 3]
 [4 6 8]]
The shape of array is (2, 3)
The dimension of array is 2
The array is [[[ 1  2  3]
  [ 4  5  6]]

 [[ 7  8  9]
  [10 11 12]]]
The shape of an array is  (2, 2, 3)
The dimension of array is 3
```

You can create an array with zeros or 1 instantly. See these examples below.

```
>>> m2 = np.zeros(4)
>>> m2
array([0., 0., 0., 0.])
>>> m3 = np.ones(4)
>>> m3
array([1., 1., 1., 1.])
>>> m5 = np.zeros((3,5))#Creating a 3X5 matrix
>>> m5
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

To have values within a range `arange` can be used. In the following example, an array of numbers in the range 2 to 20 in the steps of 2 is shown below.

```
>>> m6 = np.arange(2,20,2)
>>> m6
array([ 2,  4,  6,  8, 10, 12, 14, 16, 18])
```

Indexing and Slicing

Similar to lists, arrays can be indexed and sliced as shown below.

```
 >>> m6 = np.arange(2,20,2)
>>> m6
array([ 2,  4,  6,  8, 10, 12, 14, 16, 18])
>>> m6[2]#Get the element of index 2
6
>>> m6[1:5]# Get elements in the range 1 to 5
array([ 4,  6,  8, 10])
>>> m6[-2:]
array([16, 18])
>>> m6[0]+m6[4]# Add elements 0 and 4
12
>>> m7 = np.array([[1,2,3],[4,5,6]])#slicing from two dimension array
>>> m7[1,0:2]
array([4, 5])
```

Shape and Reshape

The keyword of `shape` is used to indicate the dimension. To convert to a different dimension, you can use the keyword `reshape`. In the following example, 9X1 is the dimension of m1, which is displayed with the keyword `shape`. The same array can be converted to 3X3 by specifying `reshape`.

```
 >>> import numpy as np
>>> m1 = np.array([1,2,3,4,5,6,7,8,9])
>>> m1.shape
(9,)
>>> m1.reshape(3,3)
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Here is another example of converting a 3X2 array to 6X1.

```
 >>> import numpy as np
>>> m2 = np.array([[1,2,3],[4,5,6]])
>>> m2
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.reshape(m2,6)
array([1, 2, 3, 4, 5, 6])
```

Iterating Arrays

Like lists and tuples, the array can be iterated. An example is shown below.

```
 >>> import numpy as np
>>> m2 = np.array([[1,2,3],[4,5,6]])
>>> for i in m2:
        print(i)
[1 2 3]
[4 5 6]
```

`numpy` provides a way to enumerate through indexes. The keyword to indicate the index is `ndenumerate` as shown below.

```
 >>> import numpy as np
>>> m2 = np.array([[1,2,3],[4,5,6]])
>>> for idx,x in np.ndenumerate(m2):
        print(idx,x)
(0, 0) 1
(0, 1) 2
(0, 2) 3
(1, 0) 4
(1, 1) 5
(1, 2) 6
```

4.2.2 Pandas

Pandas is a python library for working with large datasets. It is used widely to analyse data. It can be installed using the command of `pip3 install pandas`. In this section, you will see some introductory material of how data are read through the structure of dataframe, getting the data through `csv` format and performing standard operations like maximum, minimum and average values of data.

A dataframe in Pandas is a two dimensional structure. Suppose, we have two columns of data with food and calories. Each column can represented as a list and the whole data as a dictionary. These are converted to a dataframe through Pandas as shown below.

```
import pandas as pd #importing the module pandas
data = {"fruits": ["apple", "apricot","grapes", "lemon"],
                                    "calories":[95,17,62,17]}
df = pd.DataFrame(data)
print(df)
```

```
OUTPUT
    fruits  calories
0    apple        95
1  apricot        17
2   grapes        62
3    lemon        17
```

To select the row, you can use `loc[0]`. Add an extra line in the above program as **print**(df.loc[0]) and check for yourself. To read csv files, you can use `pd.read_csv(<filename>)`. An example is shown below, assuming the same data available with name 'fruits.csv'.

```
import pandas as pd
data = pd.read_csv('fruits.csv')
df = pd.DataFrame(data)
print(df)
```

```
 OUTPUT
     fruits  calories
0    apple         95
1  apricot         17
2   grapes         62
3    lemon         17
```

Determination of sum, average, maximum and minimum of data is straight forward indicated by the respective column title. For the same example, the code is given below.

```
 import pandas as pd
data = pd.read_csv('fruits.csv')
df = pd.DataFrame(data)
print(df)
print ("————————————————————————")
print("Sum: ",df["calories"].sum())
print("Mean: ",df["calories"].mean())
print("Maximum: ",df["calories"].max())
print("Minimum: ",df["calories"].min())
```

```
OUTPUT
     fruits  calories
0    apple         95
1  apricot         17
2   grapes         62
3    lemon         17
————————————————————————
Sum:  191
Mean:  47.75
Maximum:  95
Minimum:  17
```

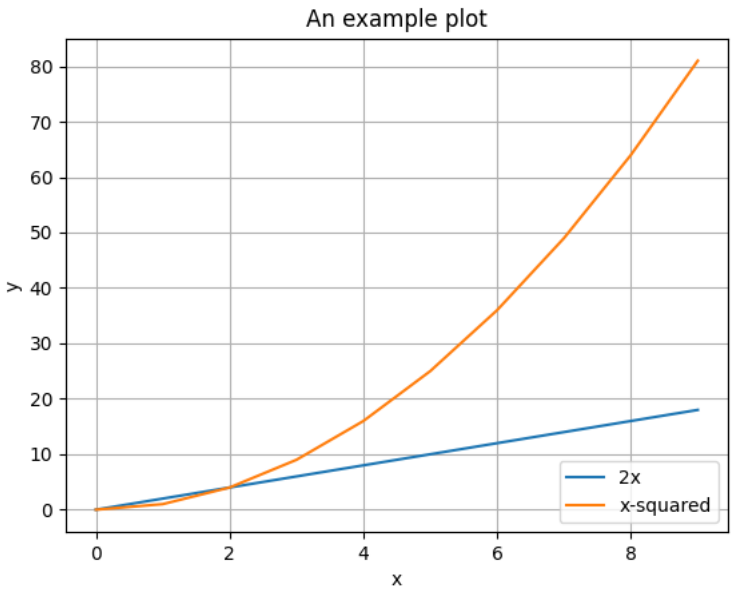
4.2.3 Matplotlib

Matplotlib is a plotting library that is predominantly used. You can install this library by the command `pip3 install matplotlib`. Here, an example is shown below to plot data with two functions $y = 2 * x$ and $y = x^2$.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
x = np.arange(0,10) # values in x axis
y = 2*x # double x
z = x*x # x squared
print("x values are", x)
print("y values are", y)
print("z values are", z)
#Give the title
plt.title("An example plot")
# Name x and y axis
plt.xlabel("x")
plt.ylabel("y")
#plot x with y and x with z
plt.plot(x,y)
plt.plot(x,z)
# add legend to lower right corner
plt.legend(["2x", "x-squared"], loc ="lower right")
plt.grid()
plt.show()
```

```
OUTPUT
x values are [0 1 2 3 4 5 6 7 8 9]
y values are [ 0  2  4  6  8 10 12 14 16 18]
z values are [ 0  1  4  9 16 25 36 49 64 81]
```



4.3 Worked Out Programming Problems

Example 4.1. Use the module of glob to print all files in the current directory

```
#use module glob and print the files
import glob
for i in glob.iglob('*.*',recursive=True):
```

```
    print(i)
```

Example 4.2. Use the module of random to print random words from the dictionary.

```
#get random words from dictionary
#https://www.w3resource.com/python-exercises/
#math/python-math-exercise-54.php
import random
with open('/usr/share/dict/words', 'rt') as f:
#change this directory if using windows
    words = f.readlines()
words = [w.rstrip() for w in words]
for w in random.sample(words, 7): # printing 7 random words
    print(w)
```

```
OUTPUT
Isabelle's
fanboys
trussed
Bishkek's
councilmen
Cognac
sureness's
```

Example 4.3. Use the module of time to print the execution of program to find the sum of first 10,000 numbers.

```
#get execution time of a program
import time
start_time = time.time()
s = 0
for i in range(1,10000):
    s = s+i
print("sum of 10,000 numbers: ", s)
end_time = time.time()
duration = end_time - start_time
print ("time for execution of adding 10000 numbers is ",
                                    duration, "seconds")
```

```
OUTPUT
sum of 10,000 numbers:  49995000
time for execution of adding 10000 numbers
                            is  0.0009145736694335938 seconds
```

Example 4.4. Use the module of urllib and read the data from internet.

```
#read the data from url and print it
import urllib.request
with urllib.request.urlopen('https://wiki.python.org/moin/')
```

```
                                                    as response:
    html = response.read()
    print (html)
```

Example 4.5. Write a python program with a main program having a number. Have two modules for calculating the number two times and square of the number.

```
#main program calling sqr and twice function
from double import twice
from sq import sqr
n = 6
print("Entered number is ", n)
print("Twice the number is", twice(n))
print("Square of the number is", sqr(n))
```

```
#double.py
def twice (n):
    return 2*n
```

```
#square function sq.py
def sqr (n):
    return n*n
```

```
OUTPUT
Entered number is  6
Twice the number is 12
Square of the number is 36
```

Example 4.6. Create a Vector with first 6 numbers and print the dimension. Convert it to an array of 2X3.

```
#creation of vector with first 6 values and print the dimension.
#Convert it to matrix of 2X3
import numpy as np
m1 = np.arange(1,7)
print("The vector is", m1)
print("The shape is ", m1.shape)
m2 = m1.reshape(2,3)
print("Converted to 2X6: ",  m2)
print("The shape of m2 is ", m2.shape)
```

```
 OUTPUT
The vector is [1 2 3 4 5 6]
The shape is  (6,)
Converted to 2X6:  [[1 2 3]
 [4 5 6]]
The shape of m2 is  (2, 3)
```

Example 4.7. Create a 3X3 identity matrix

```
#create a 3X3 identity matrix
import numpy as np
z = np.eye(3)
print(z)
```

```
OUTPUT
[[1. 0. 0.]
[0. 1. 0.]
[0. 0. 1.]]
```

Example 4.8. Create a 3X3 matrix with random values.

```
#create a 3X3 matrix with random values
import numpy as np
z = np.random.random((3,3))
print(z)
```

```
OUTPUT (\textit{Answer may vary for you})
[[0.60459325 0.62779635 0.08762768]
[0.22702361 0.47654269 0.78861996]
[0.95579087 0.85556748 0.99093165]]
```

Example 4.9. Create a 5X5 random array and find minimum, maximum and mean.

```
  #5X5 array with random values and find mean, min and max
import numpy as np
m = np.random.random((5,5))
minimum = m.min()
maximum = m.max()
mean = m.mean()
print("The array is ", m)
print("The minimum number is ", minimum)
print("The maximum number is ", maximum)
print("The mean is ", mean)
```

```
OUTPUT (\textit{Answer may vary})
The array is  [[0.02421837 0.61330188 0.92770418 0.41407668 0.30829906]
[0.05412621 0.66277552 0.68904742 0.63535547 0.65422681]
[0.73913225 0.22415544 0.01008474 0.4048251  0.72680656]
[0.91716115 0.95909384 0.95427565 0.06296586 0.8367862 ]
[0.97248135 0.58209004 0.90995593 0.23829975 0.68707503]]
The minimum number is  0.010084744287053082
The maximum number is  0.9724813457378829
The mean is  0.568332818588544
```

Example 4.10. Write a Python Program to add and subtract two matrices.

```
#Add and subtract two arrays
import numpy as np
m1 = np.array([[13,45,67],[2,4,89],[45,67,89]])
m2 = np.ones((3,3))
print ("matrix 1 \n", m1)
print ("matrix 2 \n", m2)
m3 = m1+m2
print("Addition of two matrices \n", m3)
m4 = m1 - m2
print("Subtraction of two matrices \n", m4)
```

```
OUTPUT
matrix 1
 [[13 45 67]
 [ 2  4 89]
 [45 67 89]]
matrix 2
 [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
Addition of two matrices
 [[14. 46. 68.]
 [ 3.  5. 90.]
 [46. 68. 90.]]
Subtraction of two matrices
 [[12. 44. 66.]
 [ 1.  3. 88.]
 [44. 66. 88.]]
```

Example 4.11. Write a Python program to multiply two matrices.

```
#Multiplication of two matrices
import numpy as np
m1 = np.array([[13,45,67],[2,4,89],[45,67,89]])
m2 = np.ones((3,3))
m3 = np.dot(m1,m2)
print ("matrix 1 \n", m1)
print ("matrix 2 \n", m2)
print ("Matrix multiplication \n", m3)
```

```
 OUTPUT
 matrix 1
 [[13 45 67]
 [ 2  4 89]
 [45 67 89]]
matrix 2
 [[1. 1. 1.]
 [1. 1. 1.]
```

```
 [1. 1. 1.]]
Matrix multiplication
 [[125. 125. 125.]
 [ 95.  95.  95.]
 [201. 201. 201.]]
```

Example 4.12. Write a Python program to multiply the corresponding elements of two matrices.

```
  #Multiplication element wise
import numpy as np
m1 = np.array([[1,2,3],[4,5,6],[7,8,9]])
m2 = np.array([[2,2,2],[1,1,1],[3,3,3]])
m3 = m1*m2
print ("matrix 1 \n", m1)
print ("matrix 2 \n", m2)
print("Multiplication of corresponding elements \n", m3)
```

```
 OUTPUT
 matrix 1
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
matrix 2
 [[2 2 2]
 [1 1 1]
 [3 3 3]]
Multiplication of corresponding elements
 [[ 2  4  6]
 [ 4  5  6]
 [21 24 27]]
```

Example 4.13. Write a Python program to multiply the scalar value of 5 to a matrix.

```
#scalar operation
import numpy as np
m1 = np.array([[1,2,3],[4,5,6],[7,8,9]])
m2 = 5
m3 = m1*5
print ("matrix 1 \n", m1)
print ("scalar \n", m2)
print("Multiplication of matrix by scalar \n", m3)
```

```
 OUTPUT
 matrix 1
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
scalar
```

```
 5
Multiplication of matrix by scalar
 [[ 5 10 15]
 [20 25 30]
 [35 40 45]]
```

Example 4.14. Write a Python program to compute row wise and column wise sum of a matrix.

```
#column wise sum and row wise sum
import numpy as np
m1 = np.array([[13,45,67],[2,4,89],[45,67,89]])
columnsum = np.sum(m1,axis=0)
print("Column wise sum", columnsum)
rowsum = np.sum(m1,axis = 1)
print("Row wise sum", rowsum)
```

```
OUTPUT
Column wise sum [ 60 116 245]
Row wise sum [125  95 201]
```

Example 4.15. Create a random sequence of 6X4 array using numpy. With these, create a dataframe and add labels as A, B, C, D

```
  #create a random sequence of 6X4 array with numpy
# Create a dataframe and add labels as A,B,C,D
import numpy as np
import pandas as pd
m = np.random.random((6,4))
df = pd.DataFrame(m,columns = list('ABCD'))
print (df)
```

```
OUTPUT (\textit{Answer may vary}
          A         B         C         D
0  0.519864  0.566056  0.434699  0.790753
1  0.516732  0.101248  0.536113  0.832022
2  0.185247  0.395699  0.638829  0.325936
3  0.600849  0.835752  0.286540  0.730958
4  0.539179  0.203676  0.747126  0.408591
5  0.895899  0.629390  0.106228  0.738018
```

Example 4.16. With the same dataframe as above, get first three rows of column A and C

```
#With same dataframe as above, get columns A,C first three rows
import numpy as np
import pandas as pd
m = np.random.random((6,4))
df = pd.DataFrame(m,columns = list('ABCD'))
splitdf = df[['A','C']]
print(splitdf[0:3])
```

```
 OUTPUT
          A         C
0  0.826541  0.247127
1  0.027224  0.116862
2  0.649350  0.844085
```

Example 4.17. Change the dataseries to a different datatype.

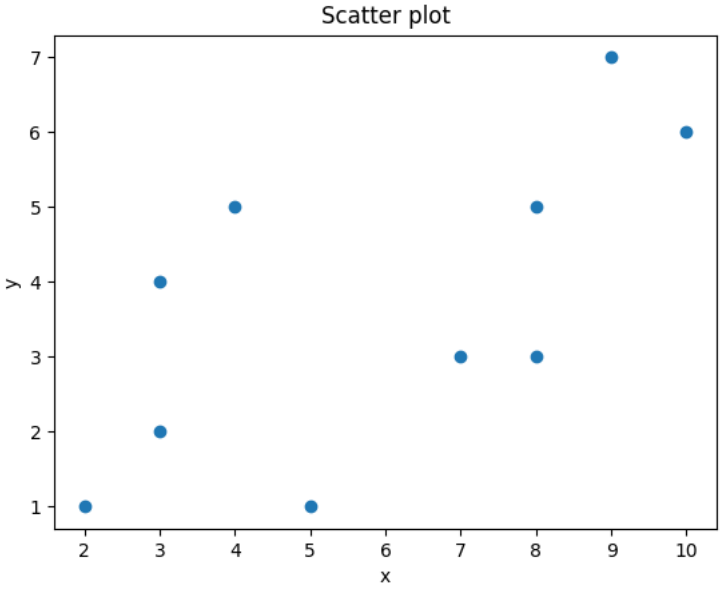
```
#Change the column to a different datatype
import pandas as pd
ser1 = pd.Series(['10','20','30','40','50'])
print ("The series is \n ", ser1)
print("Now changing to numeric ")
ser2 = pd.to_numeric(ser1)
print(ser2)
```

```
  OUTPUT
  The series is
  0    10
1    20
2    30
3    40
4    50
dtype: object
Now changing to numeric
0    10
1    20
2    30
3    40
4    50
dtype: int64
```

Example 4.18. With matplotlib, write a Python program for scatter plot.

```
#scatterplot with matplotlib
import matplotlib.pyplot as plt
import numpy as np
x = [2,3,5,8,3,4,10,9,7,8]
y = [1,2,1,3,4,5,6,7,3,5]
print("x values are", x)
print("y values are", y)
#Give the title
plt.title("Scatter plot")
# Name x and y axis
plt.xlabel("x")
plt.ylabel("y")
plt.scatter(x,y)
plt.show()
```

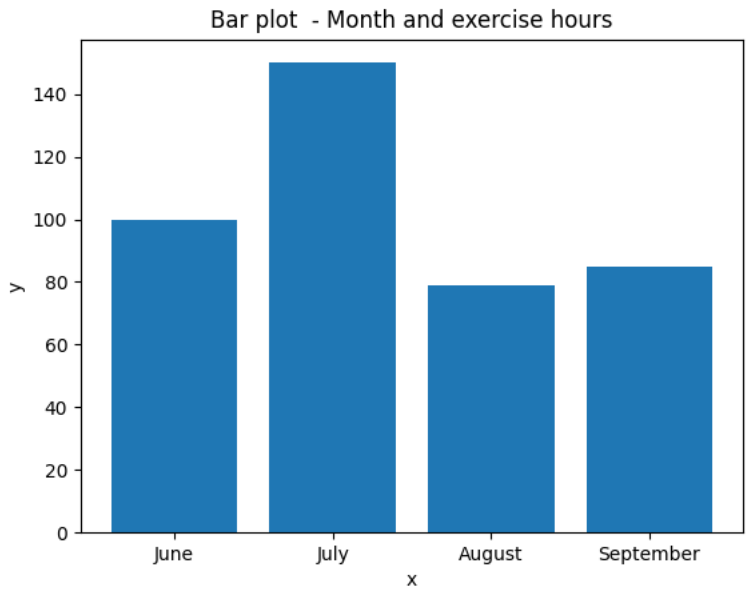
OUTPUT



Example 4.19. With matplotlib, write a program for bar plot.

```
#barplot with matplotlib
import matplotlib.pyplot as plt
import numpy as np
x = np.array(["June","July","August","September"])
y = np.array([100,150,79,85])
print("x values are", x)
print("y values are", y)
#Give the title
plt.title("Bar plot  – Month and exercise hours")
# Name x and y axis
plt.xlabel("x")
plt.ylabel("y")
plt.bar(x,y)
plt.show()
```

OUTPUT



4.4 Summary

- Python supports modularity. A simple python program with .py can be imported as a module.
- A function within a program can be called from an external program.
- Python libraries can be installed and can be imported as modules.
- A number of Python programs can be stacked together and can be built as a package.
- Modules of numpy and pandas can be used for mathematical calculations and these can scale well.
- Plotting libraries such as plotly and matplotlib can be incorporated within a Python program to draw graph.

4.5 Learning Outcomes

After reading and working out examples from this chapter, you should be able to:

1. Create modules of Python programs and use them.
2. Create a software package bundled with Python programming modules.
3. Import external modules to Python Program.
4. Work with modules of numpy, pandas and matplotlib.

4.6 Chapter Exercises

4.6.1 Multiple Choice Questions

1. A Python module is a file with ____ extension

   (a) .pym

(b) .py

(c) .pt

(d) .ht

2. To add modules in your program you use ____ statement

   (a) import

   (b) add

   (c) insert

   (d) update

3. A function named add is used in the program adder. In the main program, the correct statement is

   (a) import add

   (b) import addition

   (c) from add import adder

   (d) from adder import add

4. Compiled version of module is located in

   (a) root directory

   (b) __pycache__ directory

   (c) MyDocuments or /Home

   (d) current directory

5. Pandas key datastructure is called

   (a) data

   (b) dataframe

   (c) framest

   (d) series

6. Dataframe in Pandas is

   (a) 1-D

   (b) 2-D

   (c) 3-D

   (d) 0-D

7. Series in Pandas is

   (a) 1-D

   (b) 2-D

   (c) 3-D

   (d) 4-D

8. How do you change the shape of numpy array?

   (a) shape
   (b) reshape
   (c) shape.reshape
   (d) reshape.shape

9. How do you convert a numpy array to a list?

   (a) list(array)
   (b) array(list)
   (c) list
   (d) array[list]

10. What is the datatype of numpy array?

    (a) ndarray
    (b) int
    (c) float
    (d) str

4.6.2 Descriptive Questions

1. Why do we need modules?
2. How do you create your own module and add to main program. Explain with an example.
3. How are modules loaded and executed?
4. What is the difference between package and module? Explain with an example.
5. With an example, explain how arrays are created using numpy.
6. Explain shape and reshape features available in numpy.
7. How will you create a dataseries using Pandas?
8. With examples, explain any five operations possible with Pandas.
9. How will you find the largest size file in the current directory with Python?
10. With an example explain how will you plot a 2-D graph with Python.

4.6.3 Programming Questions

1. Write a Python program to get the input from the user as starting date and ending date. Display the number of hours, minutes and seconds between the two dates. Between these two dates, write modules to convert days to hours, hours to minutes and minutes to seconds.

2. For Question 1, create a Python package.

3. Using numpy, find the inverse of a matrix.

4. Write a Python program to reverse a numpy array.

5. Find the number of rows and number of columns in a matrix. Transpose the matrix and then find the number of rows and number of columns (Use numpy module).

6. Write a Python program to read columns of data and find the unique value (Use pandas).

7. Write a Python program to give the statistical measure of all column data in terms of mean, standard deviation, percentile (Use pandas).

8. Create a new column of data by adding the existing two columns of data.

9. Write a Python program to create multiple types of charts (a simple curve and plot some quantities) on a single set of axes.

10. Write a Python programming to create a pie chart of the popularity of programming Languages.

4.7 References and External Learning

- Reema Thareja, Python Programming using Problem Solving Approach, Oxford University Press, 2019
- Charles Dierbach, Introduction to Computer Science using Python A Computational Problem-Solving Focus, Wiley Publications, 2015

To understand modules, refer to these websites.

- `https://docs.python.org/3/tutorial/modules.html`
- `https://www.w3schools.com/python/python_modules.asp`
- `https://www.learnpython.org/en/Modules_and_Packages`

To get more details of the entire package mechanism these sites will help.

- `https://www.python-course.eu/python3_packages.php`
- `https://packaging.python.org/tutorials/packaging-projects/`

Case study and use of external modules for datascience

- `https://runestone.academy/runestone/books/published/httlads/Statistics/cs1_exploring_happiness.html`
- `https://runestone.academy/runestone/books/published/httlads/Statistics/cs2_new_business_data.html`
- `https://runestone.academy/runestone/books/published/httlads/MovieData/toctree.html`

Links on numpy, pandas and matplotlib

- https://www.python-course.eu/numpy.php
- https://www.datacamp.com/community/blog/python-numpy-cheat-sheet
- https://pandas.pydata.org/
- https://www.kaggle.com/kashnitsky/topic-1-exploratory-data-analysis-with-pandas
- https://matplotlib.org/stable/tutorials/index.html
- https://s3.amazonaws.com/assets.datacamp.com/blog_assets/Python_Matplotlib_Cheat_Sheet.pdf

Chapter 5

Object Oriented Programming in Python

Learning Objectives

- To understand the creation of classes and methods.
- To know the creation of instances associated with classes.
- To understand the concept of inheritance and encapsulation.
- To work with different methods.
- To explore the permanent storage of objects.

Python supports object oriented programming extensively. In this Chapter, you will learn the basics of structuring the program by creating objects with the associated properties and behaviors.

5.1 Creating a Class

An object is like a component which has some features and has certain functionalities. For example, assume a motorbike, which is an object with features of model, colour and speed. One of the functionality is to make noise through horn.

A class is more like a blueprint, which helps to create a mode of template associated with features and functionalities. In Python, the keyword of **`class`** is used to create a class with a property. The name of the class is usually capitalized. In the following example, the class of Circle is created and the property of the class is $radius = 8$.

```
class Circle:
    radius = 8
```

Now, assume there are two instances of objects $c1$ and $c2$ as two circles. These can be written as:

```
c1 = Circle()
c2 = Circle()
```

With $c1$ and $c2$, you can access the property as:

```
c1.radius
c2.radius
```

Although this works, in real life example objects must be defined with the method of **`__init__`**() with other associated properties. This is called as constructor. The first paramenter specified

should be **self**. When a new instance is created, the instance is automatically passed to this self parameter. Properties are otherwise known as attributes. Attributes that are in **__init__()** are instance attributes. For example, consider the class `Bike` with attributes model, color and speed. These can be written as:

```
class Bike:
    def __init__(self, model,color,speed):
        self.model = model
        self.color = color
        self.speed = speed
```

Here, **self**.`model = model` creates an attribute of name through self. The model value is assigned to this attribute. A similar explanation is applicable to `self.color` and `self.speed`. These attributes are called as instance attributes and are specific to that particular class. All bikes have model, color and speed specification. Now, we will create two instances A and B as follows:

```
A = Bike("Hero","Red","70Kmph")
B = Bike("TVS","Blue","60Kmph")
```

Each instance specifies the class name with parameters. For the instance of A, "Hero", "Red" and "70Kmph" are passed as parameters to the class as model, color and speed, respectively. To access the instance attributes, the notation of `instance.attribute` can be used. In the following example, A.model, A.color and A.speed represent the instance attributes.

```
print (A.model, A.color, A.speed)
print (B.model, B.color, B.speed)
```

The whole code and the output is given below.

```
class Bike:
    type = "two wheeler" #class attribute
    def __init__(self, model,color,speed):
        self.model = model
        self.color = color
        self.speed = speed

A = Bike("Hero","Red","70Kmph")
B = Bike("TVS","Blue","60Kmph")
print (A.model, A.color, A.speed)
print (B.model, B.color, B.speed)
```

```
OUTPUT
Hero Red 70Kmph
TVS Blue 60Kmph
```

The values of instance attributes can be changed. You can change the speed of B as `B.speed = 75Kmph`. Other than instance attributes, class attributes can also be defined. In the above program, the second line indicates the type as a class attribute of two wheeler. This type is also mutable. As a continuation of the above program, enter these lines and check.

```
print(A.type)
```

```
A.type = "three wheeler"
print(A.type)
```

You will see that, first the class is two wheeler and then three wheeler.

5.2 Class Methods

With classes and instances, you can create some functionality. These are called as methods. In the same example with Bike class, two methods are added. The methods for a specific class are defined through the statement `def`. The first method gives a description with model, color and speed. The second method indicates the nature of noise it creates.

```
class Bike:
    type = "two wheeler"
    def __init__(self, model,color,speed):
        self.model = model
        self.color = color
        self.speed = speed

    def description(self):
        return f"{self.model} has {self.color} color
                    with speed of {self.speed}"
    def sound(self,snd):
        return f"{self.model} makes noise through {snd}"
A = Bike("Hero","Red","70Kmph")
print(A.description())
print(A.sound("horn"))
```

```
Hero has Red color with speed of 70Kmph
Hero makes noise through horn
```

5.3 Class Inheritance

A class can belong to another class. For example, Bike can have two classes PedalBike and MotorBike. These can be included as `class PedalBike(Bike)` and `class MotorBike(Bike)`. The new class that is created is called the child class and the one already available, the parent class. In this example, `MotorBike` and `PedalBike` are child classes and `Bike` is the parent class. Always the childclass should have a unique name and the parent class should be included within the round brackets. See the example below.

```
class Bike:
  //same as above code//
class PedalBike(Bike):
    pass
class MotorBike(Bike):
    pass
```

We can instantiate the childclass with the same attributes as that of parent class. The child class inherits the parent class. See the example below, where A and C are instances of MotorBike and PedalBike, respectively. When these are instantiated, the parent class attributes are invoked.

```
class Bike:
  //same as above code//
class PedalBike(Bike):
    pass
class MotorBike(Bike):
    pass
A = MotorBike("Hero","Red","70Kmph")
C = PedalBike("Atlas","Blue","30Kmph")
print(A.description())
print(C.sound("Ringer"))
```

```
Hero has Red color with speed of 70Kmph
Atlas makes noise through Ringer
```

To check which class the instance belongs to, you can use the keyword **isinstance**. Check what happens when you add these two lines to the above program. You should get True value for the first line and False for the second line.

```
print("Check whether C is an instance of PedalBike: -",
                                    isinstance(C,PedalBike))
print("Check whether C is an instance of MotorBike: -",
                                    isinstance(C,MotorBike))
```

Sometimes, the method from the parent class may not be applicable. In such cases, a separate method can be defined in the child class. In the example, the PedalBike has ringer, which can be stated in the method of PedalBike class. When the object is instantiated, the method is called from the child class rather than the parent class. This is known as overriding. You will be able to understand this concept in the following example, where the class of PedalBike overrides the parent class of Bike with the sound of ringer.

```
class Bike:
    type = "two wheeler"
    def __init__(self, model,color,speed):
        self.model = model
        self.color = color
        self.speed = speed

    def description(self):
        return f"{self.model} has {self.color}
                    color with speed of {self.speed}"
    #parent class method
    def sound(self,snd):
        return f"{self.model} makes
                noise through {snd}"

class PedalBike(Bike):
```

```
    #child class method -overrides the parent class
    def sound(self,snd= "ringer"):
        return f"{self.model} has {snd}"
class MotorBike(Bike):
    pass
A = MotorBike("Hero","Red","70Kmph")
C = PedalBike("Atlas","Blue","30Kmph")
print(C.description())
print(C.sound())
```

5.4 Encapsulation

Encapsulation wraps variables and methods, so that it cannot be changed accidentally. This can be accomplished using underscore before the variable as '__<variablename>'. These are private instance variables belonging to that particular class and are **protected** - meaning that the assignment of the variable cannot be changed. Consider the following example of class Banana. Here, the variable **self.__length** has the assignment of '$6cms$' in the 4^{th} line. From outside the class at 10^{th} line the value is changed to $12cm$. Now if these are displayed, you will notice that the value of $6cms$ is not changed. This is because of the private variable indicated by **self.__length**.

```
1. # Encapsulation example
2. class Banana:
3.     def __init__(self):
4.         self.__length = "6 cms"#private variable
5.
6.      def long(self):
7.          print("The maximum length: {}".format(self.__length))

8. b = Banana()
9. b.long()
10.b.__length = "12 cms"
11.print("The length is changed to 12 cms in the program, but...")
12.b.long()
```

```
OUTPUT
The maximum length: 6 cms
The length is changed to 12 cms in the program, but...
The maximum length: 6 cms
```

An underscore before the variable name indicates the private variable. Any change in value to this variable is not permitted.

Now, what if you want to change this value? You can always change the value by creating a function inside the class and pass the value as an argument to this function. Let us take the same example. Add the function as `setlength(self,length)` with the argument as `length`. The

value for the variable is passed through `b.setlength=''12cms''`. After these changes, if you run the program you will notice that the value changes to $12cms$.

```
# Encapsulation example
class Banana:
    def __init__(self):
        self.__length = "6 cms"#private variable

    def long(self):
        print("The maximum length: {}".format(self.__length))
    #create a function with length as argument
    def setlength(self,length):
        #assign the value of length to the variable
        self.__length=length

b = Banana()
b.long()
b.__length = "12 cms"
print("The length is changed to 12 cms in the program, but...")
b.long()
print("Pass the value of length as 12 cms to function.")
b.setlength("12 cms")
print("The value should change now.")
b.long()
```

```
 OUTPUT
The maximum length: 6 cms
The length is changed to 12 cms in the program, but...
The maximum length: 6 cms
Pass the value of length as 12 cms to function.
The value should change now.
The maximum length: 12 cms
```

5.5 Polymorphism

Polymorphism as the name suggests takes many forms (Poly - many, morphism - forms). It means that the same function can take different forms. For example, consider banana and lemon. Here, both are yellow in color but one is a fruit and the other is a vegetable. We can indicate such cases through the concept of polymorphism. In the following code, there are two classes Banana and Lemon with two similar functions `type` and `col`. But the functionality of each of class is different. When instantiating the object, you can see that one belongs to Banana class and the other to Lemon. These attributes pass to the respecive class and methods. Note that there is no common parent class here and therefore no inheritance.

```
class Banana:
    def __init__(self,typ,color):
        self.typ = typ
        self.color = color
```

```
    def type (self):
        print(f"Banana is {self.typ}")
    def col(self):
        print(f"Banana is {self.color} in color.")
class Lemon:
    def __init__(self,typ,color):
        self.typ = typ
        self.color = color
    def type (self):
        print(f"Lemon is {self.typ}")
    def col(self):
        print(f"Lemon is {self.color} in color.")


obj_banana = Banana("fruit","yellow")
obj_lemon = Lemon("vegetable","yellow")
for obj in (obj_banana,obj_lemon):
    obj.type()
    obj.col()
```

```
OUTPUT
Banana is fruit
Banana is yellow in color.
Lemon is vegetable
Lemon is yellow in color.
```

In the above program, since the function is common to both classes, a for loop is used to access the respective methods.

5.6 Class methods vs Static methods

Generally three types of methods are available in Object Oriented approach. These are Instance Methods, Class Methods and Static Methods.

In instance type, the method acts on the instance of the particular class. The access to this method is through `self` operator. The object state can be modified through this method. The methods that you saw in the above programs belong to this category.

Class methods are indicated through the decorator `@classmethod`. This method is applicable to the class and not to the object. Like `self`, it can be accessed through `cls` (*Note here that `cls` is used for conveience. Other notations can also be used*).The state of the class can be changed with this method, which implies that object states could also be changed.

Static methods are denoted through the decorator `@staticmethod`. This method neither takes a `self` or `cls` parameter. But it can take other parameters. This method cannot access instance or class methods.

To explain these concepts, take a look at the following example (Adapted from `https://realpython.com/instance-class-and-static-methods-demystified/`.)

```
class MyClass:
    def method(self):
```

```
        return 'instance method called'
    @classmethod
    def classmethod(cls):
        return 'class method called'

    @staticmethod
    def staticmethod():
        return 'static method called'

obj = MyClass()#instance
print(obj.method())#object instance
print("------------------------")
print(obj.classmethod())#class instance
print("------------------------")
print(obj.staticmethod())#static method
print("--------------------------------")
```

```
OUTPUT
instance method called
------------------------
class method called
------------------------
static method called
------------------------
```

The function `method`, `classmethod` and `staticmethod` indicate instance, class and static method, respectively. An object `obj` is instantiated to `MyClass` and on calling `obj.method()`, you will notice that the `Instance method is called`. Similarly on calling the class method and static methods, the particular functions are executed.

Now, extend the program to access the function directly without an object. You will see that the methods of class and static are executed.

```
\\same as above program \\
print(MyClass.classmethod())
print("--------------------------------")
print(MyClass.staticmethod())
print("--------------------------------")
```

```
 OUTPUT
 class method called
------------------------------
static method called
------------------------------
```

But what happens when you call `Myclass.method()`?. Add the following extra line to the program.

```
\\same as above program\\
print(MyClass.objectmethod())
```

```
 OUTPUT
 Traceback (most recent call last):
  File "method1.py", line 23, in <module>
    print(MyClass.method())
TypeError: method() missing 1 required positional argument: 'self'
```

You will notice that there is a type error since the object method is called without creation.

5.7 Python Object Persistence

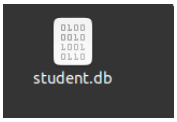
In the above Sections, you have created objects. You would have noticed that objects that you created are alive only in the run time. The created object ceases to exist after the execution of the program. What if you have to store the data (object) for future use? One solution is to use relational database. But for some tasks, using the database may be overkill. For such cases, persistence mechanisms can be used.

One of the mechanisms you have seen earlier in Chapter 3 is the use of Pickle files, where the content are loaded as binary format in a file. Another common mechanism is the use of `'Shelf'`, a persistent dictionary like object.

The `shelf` module uses the data structure of dictionary to store the data to the disk. As an example, consider the following program containing the data of a student with roll number, name and course. First a name is defined and opened as shelf, which is similar to that opening a file. Here, the name of `student.db` is created and assigned to the variable s. This variable with the index of `'key1'` assigns the dictionary of student information with roll number, name and course. The shelf record after necessary operation should close as indicated by `s.close`.

```
import shelve
s = shelve.open('student.db')
s['key1'] = {'Roll number':202111520, 'Name':"Geiko",'Course':"MCA"}
s.close()
print("Stored as a persistent data"
```

After execution of this program, you can see that a file with name 'student.db' is available in the same directory, from where you run the program. It should be visible like this image.



Having stored the information, you can retrieve it from another file using the same shelf name as shown in the following example.

```
import shelve
t = shelve.open('student.db')
print(t['key1'])
t.close()
```

```
 OUTPUT
 {'Roll number': 202111520, 'Name': 'Geiko', 'Course': 'MCA'}
```

Here, the same shelf `student.db` is used and the program prints `key1`.

Now, what if the values need to be updated? To modify the value, `writeback=True` is to be used as shown below.

```
#update with writeback
import shelve
s = shelve.open('student.db',writeback=True)
s['key1']['Course'] = "MCA Distance Education"
s.close()
print("Updated with Course as MCA Distance Education")
```

Now if you retreive the information from `student.db`, you should see this output.

```
OUTPUT
{'Roll number': 202111520, 'Name': 'Geiko',
                'Course': 'MCA Distance Education'}
```

5.8 Worked out Programming Problems

Example 5.1. Create a class named number with the variable `num`.

```
  class Number:
    def __init__(self,num):
        self.num = num
        print(f"The number is {self.num}")
```

```
  OUTPUT
  The number is 6
```

Example 5.2. Add a method to Example1 and find the square of two numbers.

```
  #Add a method to find square
class Number:
    def __init__(self,num):
        self.num = num
    def sq(self):
        self.square = self.num *self.num
        print (f"The square of {self.num} is {self.square}")

n = Number(6)
n.sq()
n = Number(7)
n.sq()
```

```
  OUTPUT
The square of 6 is 36
The square of 7 is 49
```

Example 5.3. Extend the above program to find the square of first ten numbers.

```
  #Extend the program to find the square of 1st 10 integers
class Number:
    def __init__(self,num):
        self.num = num
    def sq(self):
        self.square = self.num *self.num
        print (f"The square of {self.num} is {self.square}")
    def FirstTen():
        for i in range(1,11):
            n = Number(i)
            n.sq()

Number.FirstTen()
```

```
  OUTPUT
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
The square of 6 is 36
The square of 7 is 49
The square of 8 is 64
The square of 9 is 81
The square of 10 is 100
```

Example 5.4. With a parent class as Number, create two child classes Integers and Floating-Point.

```
#Create two child classes within numbers as
#integers and floating point
class Number:
    def __init__(self,num):
        self.num = num
class Integers(Number):
    pass
class FloatingPoint(Number):
    pass
```

Example 5.5. Extend the above program and add methods. The program should identify the type of the number entered outside the class and print accordingly.

```
class Number:
    def __init__(self,num):
        self.num = num
class Integers(Number):
    def description(self):
        return f"{self.num} is integer"
```

```
class FloatingPoint(Number):
    def description(self):
        return f"{self.num} is floating point"

n = 45
if type(n) == int:
    n1 = Integers(n)
    print(n1.description())
elif type(n) == float:
    n1 = FloatingPoint(n)
    print(n1.description())
```

```
OUTPUT
45 is integer
```

Example 5.6. In the above example, check the class of `n1` and check whether it is an instance of FloatingPoint.

```
  \\Same as above program \\
n = 45
if type(n) == int:
    n1 = Integers(n)
    print("Check the class of n1 ", type(n1))
print ("check whether n1 is an instance of
        FloatingPoint ", isinstance(n1,FloatingPoint))
```

```
Check the class of n1  <class '__main__.Integers'>
check whether n1 is an instance of FloatingPoint  False
```

Example 5.7. Create a class with Number and a child class named Integers. Demonstrate inheritance.

```
#Create a child classes within number as integers
#demonstrate inheritance
class Number:
    def __init__(self,num):
        self.num = num
class Integers(Number):
    def double(self):
        self.twice = self.num*2
        #self.num is inherited from parent class
        return f"{self.twice} is double of {self.num}"
n = Integers(45)
print(n.double())
```

```
  90 is double of 45
```

Example 5.8. Create a class with Number and a child class named Integers. Demonstrate overriding.

```
  #Create two child classes within numbers as integers
#demonstrate overriding
class Number:
    def __init__(self,num):
        self.num = num
    def description(self):
        return f"called from parent class - the number is {self.num}"
class Integers(Number):
    def description(self,num = 1000009): # overrides the parent class
        return f"called from child class - the number is {num}"

n1 = Number(35)
print(n1.description())
n2 = Integers(35)
print(n2.description())
```

```
OUTPUT
called from parent class - the number is 35
called from child class - the number is 1000009
```

Example 5.9. Create a child class named Integers with Number as parent class. Demonstrate the use of super()

```
  #Create a child class within number as integers
#demonstrate super()
class Number:
    def __init__(self,num):
        self.num = num
    def description(self):
        return f"called from parent class
            - the number is {self.num}"
class Integers(Number):
    def description(self,num = 1000009):
        return super().description()
        return f"called from child class
            - the number is {num}"#this has no effect

n1 = Number(35)
print(n1.description())
n2 = Integers(35)
print(n2.description())
```

```
OUTPUT
called from parent class - the number is 35
called from parent class - the number is 35
```

Example 5.10. Create a class named DaysInYear and assign 365 to the variable of self.days. Change it to 366 from the instance of the class.

```
# set the number of days in a year as 365 in class DaysInYear.
#Change it to 366
class DaysInYear:
    def __init__(self):
        self.days = 365
    def describe(self):
        return f"days in year = {self.days}"

y = DaysInYear()
print(y.describe())
y.days = 366
print(y.describe())
```

```
 OUTPUT
days in year = 365
days in year = 366
```

Example 5.11. Consider the above example. Use private variable such that 365 is not changed in the above program.

```
# set the number of days in a year as 365 in class DaysInYear.
# Change it to 366.
#But the output should still remain at 365

class DaysInYear:
    def __init__(self):
        self.__days = 365
    def describe(self):
        return f"days in year = {self.__days}"

y = DaysInYear()
print(y.describe())
y.__days = 366
print(y.describe())
```

```
 OUTPUT
days in year = 365
days in year = 365
```

Example 5.12. Use encapsulation in the above program and see what different it makes.

```
# set the number of days in a year as 365 in class DaysInYear.
#Change it to 366.
#But the output should still remain at 365
#Use Encapsulation and change the value to 366

class DaysInYear:
    def __init__(self):
```

```
        self.__days = 365
    def describe(self):
        return f"days in year = {self.__days}"
    def LeapYear(self,days):
        self.__days = days

y = DaysInYear()
print(y.describe())
y.LeapYear(366)
print(y.describe())
```

```
 OUTPUT
days in year = 365
days in year = 366
```

Example 5.13. Create two classes Integers and FloatingPoint. Use the same function to find the cube of that number and demonstrate polymorphism.

```
#Have two classes integers and floats.
#Have a same function of cube and demonstrate
#through polymorphism

class Integers:
    def __init__(self,num):
        self.num = num

    def cube(self):
        self.cb = (self.num)**3
        return f"Cube of {self.num} is {self.cb}"

class FloatingPoint:
    def __init__(self,num):
        self.num = num

    def cube(self):
        self.cb = (self.num)**3
        return f"Cube of {self.num} is {self.cb}"

n1 = Integers(5)
n2 = FloatingPoint(5.5)
print(n1.cube())
print(n2.cube())
```

```
 OUTPUT
Cube of 5 is 125
Cube of 5.5 is 166.375
```

Example 5.14. Create a class with name Bird and add a class method.

```
#Have a class bird and add a class method
class Bird:
    def __init__(self, name,color):
        self.name = name
        self.color = color
    @classmethod
    def description(cls,name,color):
        return cls(name, color)

parrot = Bird.description('Parrot','Green')
print(parrot.name,"is",parrot.color)
```

```
  OUTPUT
  Parrot is Green
```

Example 5.15. Extend the above program with a static method and see the difference.

```
#Have a class bird and add a class method
class Bird:
    def __init__(self, name,color):
        self.name = name
        self.color = color
    @classmethod
    def description(cls,name,color):
        return cls(name, color)

    @staticmethod
    def isParrot(name):
        if name == "Parrot":
            return f"{name} can fly"

parrot = Bird.description('Parrot','Green')
print(parrot.name,"is",parrot.color)
print(Bird.isParrot("Parrot"))
```

```
 OUTPUT
Parrot is Green
Parrot can fly
```

5.9 Summary

- Python is an Object Oriented Programming Language.
- Objects and the corresponding properties can be well aligned using Python.
- The class is initiated using __init__ function.
- A number of methods can be written within a class.
- Child classes can be created within a Parent class, thereby supporting inheritance.

- If there is a specific property associated only to child class, the program can be designed as per the child's specific property. However, using super(), the parent property can also be called.
- Python supports polymorphism.
- With encapsulation, the private variables in a class cannot be changed, thereby preventing the user to change the parameter of a particular class.
- Python provides the flexibility of using three methods - instance method, class method and static methods.
- Python object persistance through the module of shelf provides a way to store the details of data in the disk.

5.10 Course Outcomes

Afer reading this Chapter and working out programs, you should be able to:

1. Develop Python program with Classes and methods.
2. Write Python programs supporting inheritance from Parent Class.
3. Construct multiple classes with polymorphism.
4. Develop effective programming constructs using encapsulation techniques.
5. Develop complex programs with instance method, class method and static method.
6. Provide a permanent storage of Objects with the module of Shelf.

5.11 Chapter Exercises

5.11.1 Multiple Choice Questions

1. To create a class called Flower, the syntax is
   (a) class Flower:
   (b) cls
   (c) cls <name>
   (d) class
2. Assume there is a class Flower. The instance of Flower can be represented as:
   (a) f = Flower()
   (b) f = Flower
   (c) Flower
   (d) f
3. The default method of a class is initiated with
   (a) init

(b) __init__

(c) default

(d) con()

4. Consider the following piece of code. What is the output?

```
>>> class Flower:
        def __init__(self,name):
                self.name = name
>>> f = Flower('sunflower')
>>> f.name
```

(a) name

(b) sunflower

(c) f

(d) error

5. For the code mentioned in previous question, which creation of object is not valid?

(a) f1 = Flower('fl')

(b) f = Flower('rose')

(c) f3 = Flower('jasmine')

(d) f4 = newFlower('rose')

6. In the block of code mentioned in Q.4, the instance is:

(a) name

(b) Flower

(c) f.name

(d) f

7. _____ is a convention that is used to represent the instance of a class and to access the attributes and methods of the class.

(a) method

(b) self

(c) class

(d) instance

8. In the following block of code, how many classes and methods are there?

```
class Number:
    def __init__(self,num):
        self.num = num
    def sq(self):
        self.square = self.num *self.num
        print (f"The square of {self.num} is {self.square}")
```

```
    def FirstTen():
        for i in range(1,111):
            n = Number(i)
            n.sq()
Number.FirstTen()
```

(a) classes =1, methods = 3

(b) classes = 1, methods =2

(c) class = 2, methods =2

(d) class = 3, methods =1

9. What is the output of the program mentioned in previous question?

(a) Square of first 10 numbers

(b) Square of first 100 numbers

(c) Square of first 99 numbers

(d) Square of first 111 numbers

10. Which among the following is the correct way to initialize the class?

(a) def __init__(self,name,age):

(b) def _init():

(c) def __init__():

(d) def __init__(name,age):

11. Which among the following is the correct representation of private variable?

(a) self.__length = 5

(b) self.length = 7

(c) __self.length = 7

(d) self__length = 5

12. The same method name with different functionalities are

(a) Polymorphism

(b) Encapsulation

(c) Inheritance

(d) Data hiding

13. Which among the following is not a method in Object Oriented Programming?

(a) class method

(b) dynamic method

(c) object method

(d) static method

14. Class method changes the state of
    (a) class and object
    (b) only class
    (c) all methods outside the class
    (d) only object
15. Static methods cannot access
    (a) class method
    (b) object method
    (c) both class and object methods
    (d) other parameters

5.11.2 Descriptive Questions

1. How do you create Parent Class and Child Class. Explain with an example.
2. What is an instance? How will you create an instance?
3. How will you create methods inside the class?
4. Explain the concept of inheritance with an example.
5. Explain how overriding takes place.
6. Give some instances of using super() in your program.
7. What is encapsulation. Explain with an example.
8. Explain polymorphism. Give some applications.
9. What is the difference between class method and static method? When will you use these methods?
10. Explain the need for Python object persistence.

5.11.3 Programming Questions

1. Create a class and defines the __init__ method with one parameter, "name". Add a method to it and return that name.
2. Construct a class named Dog. The __init__ method has parameters of "name" and "age". Also create the method updateAge, that increases the age by 1.
3. Extend the above program to create a class of Cat. The __init__ method has "name" and "age" as parameters. Then define the "make_sound" method, returning the cat's meow and another description to introduce itself.
4. Construct a class named 'Book' that has an __init__ method with the parameters title and author. Then create an instance of the 'Book' class named newbook.

5. In Example 5.3, add an other method to write the multiplication table of the number.

6. Create two classes Car and Bus as child classes with Parent class as Vehicle.

7. Extend the above example to demonstrate inheritance and overriding.

8. Create a parent class with two child classes Ostrich and Parrot. In the main class have a method called fly and return the value that "Birds can fly". Create a method in Ostrich class namely, fly and return the value that "Ostrich cannot fly". Create a method in Parrot class namely, fly and return the value that "Parrots can fly". Create an instance for Parrot and Ostrich. Demonstrate inheritance, overriding and the user of super() with this example.

9. Create a class called Person with variables name and age. Set the age as 2 as private variable. Add a method called adult and return the age as age+16. Use encapsulation for this program.

10. Create two classes UnderGraduates and PostGraduates. For each of the classes create the same function courses and return the values as Bachelors and Masters. Demonstrate the concept of polymorphism through this program.

11. Assume there is a list of books that needs to be stored as a persistent object. Write a program to store that using the module of shelf.

12. For the above program, update the value of a book and store back again. Retrieve the list of books from another program.

13. Create a parent class called Polygon and child classes Triangle, Rectangle, pentagon and Hexagon. For each of these, add methods to return the number of sides and the area.

14. Define a class called Bike that accepts a string and a float as input, and assigns those inputs respectively to two instance variables, color and price. Assign to the variable testOne an instance of Bike whose color is blue and whose price is 89.99. Assign to the variable testTwo an instance of Bike whose color is purple and whose price is 25.0.

15. Create a class named Vehicle. Its constructor should take three parameters, the year, the name of the make and the name of the model, which should be stored in variables named year, make, and model, respectively. Create an instance of the Vehicle class to represent a vehicle of your choice.

16. Use your existing Vehicle class as a starting point. Create a method named years_old(). It should return how many years old the vehicle is. You can find the current year using this code:

    ```
    from datetime import datetime
    current_year = datetime.now().year
    ```

    Create an instance of the Vehicle class to represent a vehicle of your choice. Call the years_old() function and print the return value.

(The last three questions are taken from `https://runestone.academy/runestone/books/published/fopp/Classes/Exercises.html`

5.12 References

- Reema Thareja, Python Programming using Problem Solving Approach, Oxford University Press, 2019
- Charles Dierbach, Introduction to Computer Science using Python A Computational Problem-Solving Focus, Wiley Publications, 2015
- https://realpython.com/python3-object-oriented-programming/
- https://docs.python.org/3/tutorial/classes.html
- https://www.python-course.eu/python3_object_oriented_programming.php
- https://www.greenteapress.com/thinkpython/thinkCSpy.pdf