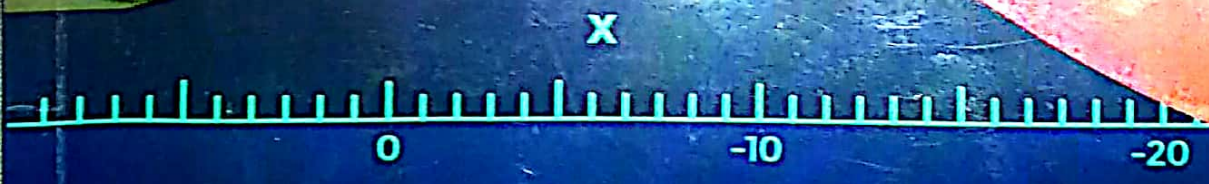
A Manual for

S.Y.B.Sc (Computer Science)
Mathematics

As per CBCS : Credit 2

PYTHON PROGRAMMING LANGUAGE

PAPER-III : MTC-233



Authors

Dr. Kalyanrao Takale | Dr. Amjad Shaikh
Dr. Mrs. Nivedita Mahajan | Dr. Veena Kshirsagar
Prof. Krishna Ghode | Dr. Shrikisan Gaikwad | Prof. S. R. Patil

GOLDEN SERIES by, NIRA

Specimen copy.

A Manual for B.Sc. (Computer Science) Mathematics

PAPER - III : MTC-233

PYTHON PROGRAMMING LANGUAGE - I

Dr. Kalyanrao Takale
M.Sc., B.Ed., Ph.D.
RNC Arts, JDB Commerce and NSC Science College
Nashik Road, Nashik

Dr. Amjad Shaikh
M.Sc., Ph.D
AKI's, Poona College of Arts, Science and Commerce
Pune

Dr. Mrs. Nivedita Mahajan
M.Sc., M.Phil., Ph.D.
Modern College of Arts, Science and Commerce
Shivajinagar, Pune

Dr. Veena P. Kshirsagar
M.Sc., M.Phil., Ph.D.
MIT World Peace University
Kothrud, Pune

Prof. Krishna Ghode
M.Sc., NET, SET
B. K. Birla College of Arts, Science and Commerce (Autonomous)
Kalyan (Thane)

Dr. Shrikisan Gaikwad
M.Sc., B.Ed., M.Phil., Ph.D.
New Arts, Commerce and Science College
Ahmednagar

Prof. S. R. Patil
M.Sc.
Ex. HOD., S.M. Joshi College
Hadapsar, Pune

Price ₹ 115.00

GOLDEN SERIES by, NIRALI

N5396

Syllabus: PAPER-III

MTC-233: Mathematics Practical: Python Programming Language-I

1. Unit 1: Introduction to Python [08]

   1.1 Installation of Python

   1.2 Values and types: int, float and str,

   1.3 Variables: assignment statements, printing variable values, types of variables.

   1.4 Operators, operands and precedence:+, −, /, *, **, %/0 PEMDAS(Rules of precedence)

   1.5 String operations: + : Concatenation, * : Repetition

   1.6 Boolean operator:

      1.6.1 Comparison operators: ==, ! =, >, =, <=

      1.6.2 Logical operators: and, or, not

   1.7 Mathematical functions from math, cmath modules.

   1.8 Keyboard input: input() statement

2. Unit 2: String, list, tuple [06]

   2.1 Strings:

      2.1.1 Length (Len function)

      2.1.2 String traversal: Using while statement, Using for statement

      2.1.3 String slice

      2.1.4 Comparison operators (>, <, ==)

   2.2 Lists:

      2.2.1 List operations

      2.2.2 Use of range function

      2.2.3 Accessing list elements

      2.2.4 List membership and for loop

      2.2.5 List operations

      2.2.6 Updating list: addition, removal or updating of elements of a list

   2.3 Tuples:

      2.3.1 Defining a tuple,

      2.3.2 Index operator,

      2.3.3 Slice operator,

      2.3.4 Tuple assignment,

      2.3.5 Tuple as a return value

3. Unit 3: Iterations and Conditional statements [10]

   3.1 Conditional and alternative statements, Chained and Nested Conditionals: if, if-else, if-elif-else, nested if, nested if-else

   3.2 Looping statements such as while, for etc, Tables using while.

   3.3 Functions:

      3.3.1 Calling functions: type, id

      3.3.2 Type conversion: int, float, str

      3.3.3 Composition of functions

      3.3.4 User defined functions, Parameters and arguments

4. Unit 4: Linear Algebra [06]

   4.1 Matrix construct, eye(n), zeros(n,m) matrices

   4.2 Addition, Subtraction, Multiplication of matrices, powers and invers of a matrix.

   4.3 Accessing Rows and Columns, Deleting and Inserting Rows and Columns

   4.4 Determinant, reduced row echelon form, nullspace, columnspace, Rank

   4.5 Solving systems of linear equations (Gauss Elimination Method, Gauss Jordan Method, LU- decomposition Method)

   4.6 Eigenvalues, Eigenvectors, and Diagonalization

5. Unit 5: Numerical methods in Python [06]

   5.1 Roots of Equations

   5.1 Newton-Raphson Method

   5.1 False Position (Regula Falsi) Mehtod

   5.1 Numerical Integration:

      5.1.1 Trapezoidal Rule,

      5.1.2 Simpson's 1/3rd Rule,

      5.1.3 Simpson's 3/8th Rule

Contents

Chapter 1

Introduction to Python

1.1 Getting started with Python

There are literally thousands of computer languages. There is no single computer language that can be considered the best. A particular language may be excellent for tackling problems of a certain type but be horribly ill-suited for solving problems outside the domain for which it was designed. Nevertheless, the language we will study and use, Python, is unusual in that it does so many things and does them so well. It is relatively simple to learn, it has a rich set of features, and it is quite expressive. Python is used throughout academia and industry. It is very much a real computer language used to address problems on the cutting edge of science and technology. Although it was not designed as a language for teaching computer programming or algorithmic design, Pythons syntax and idioms are much easier to learn than those of most other full-featured languages.
Python is an object-oriented language that was developed in late 1980s as a scripting language (the name is derived from the British television show Monty Pythons Flying Circus). Python is a living language. Python is a widely used high-level programming language for general-purpose programming, created by Guido van Rossum and first released in 1991. Python programs are not compiled into machine code, but are run by an interpreter. The great advantage of an interpreted language is that programs can be tested and debugged quickly, allowing the user to concentrate more on the principles behind the program and less on programming itself. Since there is no need to compile, link and execute after each correction, Python programs can be developed in a much shorter time than equivalent Fortran or C programs.

(i) Python is open-source software, which means that it is free; it is included in most Linux distributions.

(ii) Python is available for all major operating systems (Linux,Unix,Windows,MacOS etc.).. A program written on one system runs without modification on all systems.

(iii) Python is easier to learn and produces more readable code than other languages.

(iv) Python and its extensions are easy to install.

(v) Development of Python was clearly influenced by Java and C++, but there is also a remarkable similarity to MATLAB.

(vi) Python interpreter can be downloaded from the Python Language Website www.python.org. It normally comes with a nice code editor called Idle that allows you to run programs directly from the editor.

Python 3.x

Version Release	Date
3.8	2019-10-14
3.7	2018-06-27
3.6	2016-12-23
3.5	2015-09-13
3.4	2014-03-17
3.3	2012-09-29
3.2	2011-02-20
3.1	2009-06-26
3.0	2008-12-03

Also, Python features a dynamic type system and automatic memory management and supports multiple programming paradigms, including object-oriented, imperative, functional programming, and procedural styles. It has a large and comprehensive standard library.

Two major versions of Python are currently in active use:

Python 2.x is the legacy version and will receive only security updates until 2020, no new features will be implemented. Now a days, Python 3.x is the current version and is under active development. We can download and install earlier version of Python 3.x. In addition, some third-parties offer re-packaged versions of Python that add commonly used libraries and other features to ease setup for common use cases, such as math, data analysis or scientific use.

IDLE is Pythons Integrated Development and Learning Environment.

IDLE is a simple integrated development environment (IDE) that comes with Python. Its a program that allows you to type in your programs and run them. There are other IDEs for Python. When you first start IDLE, it starts up in the shell, which is an interactive window where you can type in Python code and see the output in the same window. I often use the shell in place of my calculator or to try out small pieces of code. But most of the time you will want to open up a new window and type the program in there.

IDLE has the following features:

- coded in 100 percent pure Python, using the tkinter GUI toolkit
- cross-platform: works mostly the same on Windows, Unix, and Mac OS X
- Python shell window (interactive interpreter) with colorizing of code input, output, and error messages

- multi-window text editor with multiple undo, Python colorizing, smart indent, call tips, auto completion, and other features
- search within any window, replace within editor windows, and search through multiple files (grep)
- debugger with persistent breakpoints, stepping, and viewing of global and local namespaces
- configuration, browsers, and other dialogs.

Keyboard shortcuts:

The following keystrokes work in IDLE and can really speed up your work.

Keystroke	Result
CTRL+C	Copy selected text
CTRL+X	Cut selected text
CTRL+V	Paste
CTRL+Z	Undo the last keystroke or group of keystrokes
CTRL+SHIFT+Z	Redo the last keystroke or group of keystrokes F5 Run module

1.2 The Basic Elements of Python

A Python program, sometimes called a script, is a sequence of definitions and commands. These definitions are evaluated and the commands are executed by the Python interpreter in something called the shell. Typically, a new shell is created whenever execution of a program begins. In most cases, a window is associated with the shell. A command, often called a statement, instructs the interpreter to do something. **Python interactive:** We use Python as a calculator, start Python (or IDLE, the Python IDE).

The symbol >>> is a shell prompt indicating that the interpreter is expecting the user to type some Python code into the shell. The line below the line with the prompt is produced when the interpreter evaluates the Python code entered at the prompt, as illustrated by the following interaction with the interpreter.

A prompt is showing up:

```
>>>
```

Display version:

```
>>> help()
Welcome to Python 3.8's help utility!.
...
```

Take care in Python 2.x if you divide two numbers:

Isn't this strange:

>>> 35/6

Output: 5

Obviously the result is wrong!

But, if we type >>> 35.0/6

Output: 5.833333333333333

>>> 35/6.0

Output: 5.833333333333333

Note: In the first example, 35 and 6 are interpreted as integer numbers, so integer division is used and the result is an integer. This uncanny behavior has been abolished in Python 3, where 35/6 gives 5.833333333333333. In Python 2.x, use floating point numbers (like 3.14, 3.0 etc....) to force floating point division!.

Another work around would be to import the Python 3.x like division at the beginning:

>>> 3/4

Output: 0.75

1.2.1 Print () Function

The **print()** function prints the specified message to the screen. The **print()** function writes the value of the argument(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple arguments, floating point quantities, and strings.

The print function requires parenthesis around its arguments. In the program above, its only argument is the string 'Hello'.

Any thing inside quotes will(with a few exceptions) be printed exactly as it appears that is the strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this.

Example 1.1. *A simple Hello-World program in Python.*

```
>>> print ("Hello","World!")
Hello World!
>>> print ("Good Morning to Every One")
Good Morning to Every One
>>> i=10
>>> print('The value of i is', i)
The value of i is 10
>>> x=7
>>> print("Rainbow has",x,"colors")
```

```
Rainbow has 7 colors
>>> x=('apple', 'banana', 'cherry')
>>> print(x)
('apple', 'banana', 'cherry')
>>> print("Pune","University")
Pune University
```

Note: Optional arguments

There are two optional arguments to the print function. They are not overly important at this stage of the game, so you can safely skip over this section, but they are useful for making your output look nice.

(i) **sep** Python will insert a space between each of the arguments of the print function.

There is an optional argument called sep, short for separator, that you can use to change that space to something else.

For example, using sep =':' would separate the arguments by a colon and sep =" would separate the arguments by two pound signs.

This says to put no separation between the arguments. Here is an example where **sep** is useful for getting the output to look nice:

Example:

>>> print ('The value of 3+4 is', 3+4, '.')

Output: The value of 3+4 is 7 .

>>> print ('The value of 3+4 is ', 3+4, '.', sep='')

Output: The value of 3+4 is 7.

(ii) **end** The print function will automatically advance to the next line. For instance, the following will print on two lines:

Example:

>>> print('On the first line')

Output: On the first line

>>> print('On the second line')

Output: On the second line

There is an optional argument called **end** that you can use to keep the print function from advancing to the next line.

Example:

>>> print('On the first line', end='')

print('On the second line')

Output: On the first lineOn the second line

1.2.2 Getting input

The **input** function is a simple way for your program to get information from people using your program. The basic structure is

variable name = input(message to user)

Example(i):

```
>>> name = input('Enter your name: ')
# Output: Enter your name: Dr. Geetanjali
>>> print('Hello, ', name)
# Output: Hello, Dr. Geetanjali
```

The above works for getting text from the user. To get numbers from the user to use in calculations, we need to do something extra.

Example(ii):

```
>>> num = eval(input('Enter a number: '))
Enter a number: 5
>>> print('Your number squared:', num*num)
# Output: Your number squared: 25
>>> num = eval(input('Enter a number: '))
Enter a number: 625
>>> print('Your number squared:', num*num)
# Output: Your number squared: 390625
```

Example(iii):

```
>>> num = eval(input('Enter a number: '))
Enter a number: 7
>>> print('Your number:', num+num)
# Output: Your number: 14
```

Remark 1.1. *The* **eval** *function converts the text entered by the user into a number. One nice feature of this is you can enter expressions, like* $4 * 9 + 25$*, and* **eval** *will compute them for you.*

Remark 1.2. *If you run your program and nothing seems to be happening, try pressing enter. There is a bit of a glitch in IDLE that occasionally happens with input statements.*

1.3 Variables

In most computer languages the name of a variable represents a value of a given type stored in a fixed memory location. The value may be changed, but not the type. This it not so in Python, where

variables are typed dynamically. To create a variable in Python, we need to specify the variable name and then assign a value to it.

<variable name> = <value>

Python uses = to assign values to variables. There's no need to declare a variable in advance (or to assign a data type to it), assigning a value to a variable itself declares and initializes the variable with that value. There's no way to declare a variable without assigning it an initial value.
The following interactive session with the Python interpreter illustrates.

```
a = 5                                  # Integer
print(a)
# Output: 5
b = 9999999999999999999                # Integer
print(b)
# Output: 9999999999999999999
pi = 3.14                              # Floating point
print(pi)
# Output: 3.14
# String
c =' A'
print(c)
# Output: A
# String
name = 'Aishwarya Takale'
print(name)
# Output: Aishwarya Takale
# Boolean
q = True
print(q)
# Output: True
```

Example:

```
x = 9
y = X * 5
=>NameError: name 'X' is not defined
```

Remark 1.3. *Even though there's no need to specify a data type when declaring a variable in Python, while allocating the necessary area in memory for the variable, the Python interpreter automatically picks the most suitable built-in type for it:*

```
a = 5
print(type(a))
# Output: < type 'int' >
b = 9999999999999999
print(type(b))
# Output: < type 'int' >
pi = 3.14
print(type(pi))
# Output: < type 'float' >
c =' A'
print(type(c))
# Output: < type 'str' >
name = 'Aishwarya Takale'
print(type(name))
# Output: < type 'str' >
q = True
print(type(q))
# Output: < type 'bool' >
```

To simplify calculations, values can be stored in variables, and and these can be used as in normal mathematics.

```
>>> a=2.0
>>> b = 3.36
>>> a+b
5.359999999999999
>>> a-b
-1.3599999999999999
>>> a**2 + b**2
15.289599999999998
>>> a > b
False
```

Note (i): We can assign multiple values to multiple variables in one line. Note that there must be the same number of arguments on the right and left sides of the = operator:

Example(i): We can also assign a several values to same number of variables simultaneously.

```
>>> a, b, c = 4,7, 9
print(a, b, c)
# Output: 4 7 9
```

Example(ii): We cannot assign a values to different numbers of variables.

```
>>> a, b, c = 4, 5
Traceback (most recent call last):
File " <pyshell#2> ", line 1, in <module>
a, b, c = 4, 5
ValueError: not enough values to unpack (expected 3, got 2)
```

Example(iii): We cannot assign a values to different numbers of variables.

```
>>> a, b = 1, 2, 3
Traceback (most recent call last):
File " <pyshell#3> ", line 1, in <module>
a, b = 1, 2, 3
ValueError: too many values to unpack (expected 2)
```

Example(iv): We can also assign a single value to several variables simultaneously.

```
>>> a = b = c = 1
print(a, b, c)
# Output: 1 1 1
```

Note (ii): When using such cascading assignment, it is important to note that all three variables *a*, *b* and *c* refer to the same object in memory, an int object with the value of 1. In other words, *a*, *b* and *c* are three different names given to the same int object. Assigning a different object to one of them afterwards doesn't change the others, just as expected:

Example(i): We assign a single value to several variables simultaneously.

```
>>> a = b = c = 1          # all three names a, b and c refer to same int object with value 1
print(a, b, c)
# Output: 1 1 1
b = 2                      # b now refers to another int object, one with a value of 2
print(a, b, c)
# Output: 1 2 1            # so output is as expected.
```

Example(ii): The above is also true for mutable types (like list, dict, etc.) just as it is true for immutable types (like int, string, tuple, etc.):

```
>>> x = y = [4, 5, 7]      # x and y refer to the same list object just created, [4, 5, 7]
x = [15, 5, 7]             # x now refers to a different list object just created, [15, 5, 7]
print(y)                   # y still refers to the list it was first assigned
# Output: [4, 5, 7]
```

Example(iii): Now, the nested lists are also valid in python that is a list can contain another list as an element.

```
>>> x = [1, 2, [3, 4, 5], [6, 7,8],9]      # this is nested list
```

>>> print(x[2])
# Output: [3, 4, 5]
>>> print(x[3])
# Output: [6, 7, 8]
>>> print(x[3][3])
Traceback (most recent call last):
File " <pyshell#20> ", line 1, in <module>
print(x[3][3])
IndexError: list index out of range
>>> print(x[3][1])
# Output: 7
>>> print(x[3][2])
# Output: 8

Example(iv): The variables in Python do not have to stay the same type as which they were first defined. We can simply use = to assign a new value to a variable, even if that value is of a different type. >>> a = 9
print(a)
# Output: 9 a = "New value"
print(a)
# Output: New value

The name of a variable must not be a Python keywor!

Keywords are:

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

1.4 Mathematical functions

We use **dir()** to check the built in function in python.
Also, we use built in function help.
>>> help(max)
Help on built-in function max in module builtins:

max(...)
max(iterable, *[, default=obj, key=func]) −> value

max(arg1, arg2, *args, *[, key=func]) − > value with a single iterable argument, return its biggest item. The default keyword-only argument specifies an object to return if the provided iterable is empty.

With two or more arguments, return the largest argument.

1.4.1 Math Operators

Python supports the usual arithmetic operators:

Supported operators:

Operator	Expression
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo (remainder)
**	Exponent
//	Floor Division (or integer division)

Consider the following examples:

```
>>> 2+5
7
>>> 2+5*4
22
>>> (2/5)+4/5-(7*5)/5
-5.8
>>> 17%5    #Modulo
2
>>> 2**5   #Exponent
32
>>> 2**5+5**2
57
>>> 56//8    #Floor Division
7
>>> 57//8
7
>>> 25//4
6
>>> (35/7*5-7)+4**2//5
21.0
```

```
>>> (35%7*5-7)+4*2//5
-6
>>> ((35/7*5-7)+4**2)//5
6.0
```

We can use variables for calculation.

```
>>> a=5
>>> a=a*2
>>> print (a)
10
>>> a=7
>>> b=5
>>> a+b
12
>>> c=a+b
>>> print (c)
12
>>> a**2+b**2
74
>>> a>b
True
>>> b>a
False
```

1.4.2 Mathematics Modules

math Module:

Most mathematical functions are not built into core Python, but are available by loading the math module. Also, built in modules contains extra functionalities. There are three ways of accessing the functions in a module.

The statement

from math import *

loads all the function definitions in the math module into the current function or module.

The use of this method is discouraged because it is not only wasteful, but can also lead to conflicts with definitions loaded from other modules.

You can load selected definitions by

from math import func1, func2,. . .

as illustrated below.

1. ```
   >>> from math import sin
   >>> sin(2.3)
   0.7457052121767203
   >>> x=1.03
   >>> sin(x)
   0.8572989891886034
   ```

2. ```
   >>> from math import sin,cos
   >>> sin(3.14)
   0.0015926529164868282
   >>> cos(-3.14)
   -0.9999987317275395
   ```

1.4.3 Random numbers

We can make an interesting computer game, its good to introduce some randomness into it. Python comes with a module, called random, that allows us to use random numbers in our programs. Now, there is only one function, called **randint**, that we will need from the random module.
To load this function, we use the following statement:

from random import randint

We can use **randint** as follows:

randint(a, b)

will return a random integer between a and b including both a and b.
Examples:

1. ```
   from random import randint
   x = randint(1,10)
   print('A random number between 1 and 10: ', x)
   #Output: A random number between 1 and 10: 3
   ```

2. ```
   from random import randint
   x = randint(1,10)
   print('A random number between 1 and 10: ', x)
   #Output: A random number between 1 and 10: 10
   ```

3. from random import randint
   x = randint(1,100)
   print('A random number between 1 and 100: ', x)
   #Output: A random number between 1 and 100: 88

Note that the random number will be different every time we run the program.

1.4.4 Math functions

The math module Python has a module called math that contains familiar math functions, including sin, cos, tan, exp, log, log 10, factorial, sqrt, floor, and ceil.
There are also the inverse trig functions, hyperbolic functions, and the constants *pi* and *e*.
The following are the examples:

1.
```
>>> from math import sin, pi
>>> print('Pi is roughly', pi)
Pi is roughly 3.141592653589793
>>> print('sin(0) =', sin(0))
sin(0) = 0.0
```

2.
```
>>> from math import log, sin
>>> print (log(sin(0.5)))
-0.7351666863853142
```

3.
```
>>> from math import *
>>> print(gamma(6))
120.0
```

4.
```
>>> from math import *
>>> sqrt(2)
1.4142135623730951
```

5.
```
>>> #Calculate the perimeter of a circle
>>> from math import *
>>> diameter = 5
>>> perimeter = 2 * pi * diameter
>>> print(perimeter)
31.41592653589793
```

6.
```
>>> #Calculate the amplitude of a sine wave:
>>> from math import *
```

```
>>> Ue=230
>>> amplitude = Ue* sqrt(2)
>>> print(amplitude)
325.2691193458119
```

7.
```
>>> from numpy import *
>>> print (sin(pi/4))
0.7071067811865475
```

The third method, which is used by the majority of programmers, is to make the module available by

import math and **from numpy import ***

The functions in the module can then be accessed by using the module name as a prefix:

Example(i):

```
import math
print (math.log(math.sin(0.5)))
#Output: -0.7351666863853142
```

Example(ii):

```
import math
import numpy as np
a, b, c, d, e = 3, 2, 2.0, -3, 10
print (a/b)
print (b/d)
print (c/d)
print (d/e)
print (a*b)
print (b*c)
#The ' // ' operator in Python 2 forces floored division regardless of type.
print (a//b)
print (b//c)
print (c//d)
print (d//e)
#Output: 1.5
-0.6666666666666666
-0.6666666666666666
-0.3
```

```
6
4.0
1
1.0
-1.0
-1
```

Example(ii):

```
from numpy import *
print (sqrt(2))
print (cos(pi/4))
print (tan(pi/3))
#Output: 1.4142135623730951
0.7071067811865476
1.7320508075688767
```

Example(iii):

```
from numpy import linspace,sin,cos,tan,exp,pi,log
print (sin(pi/4))
print (exp(5))
print (sin(pi/2))
print (cos(pi/4))
print (tan(pi/3))
print (pi)
print (log(2))
#Output: 0.7071067811865476
148.4131591025766
1.0
0.7071067811865476
1.7320508075688767
3.141592653589793
0.6931471805599453
```

Example(iv):

```
#Trigonometric Functions
import math
a, b = 1, 2
x=math.sin(a)                    # returns the sine of 'a' in radians
print (x)
```

```
#Output: 0.8414709848078965
y=math.cosh(b)              # returns the inverse hyperbolic cosine of 'b' in radians
print (y)
#Output: 3.7621956910836314
z=math.atan(math.pi)        # returns the arc tangent of 'pi' in radians
print (z)
#Output: 1.2626272556789115
u=math.hypot(a, b)          # returns the Euclidean norm, same as math.sqrt(a*a + b*b)
print (u)
#Output:2.23606797749979
v=math.degrees(a)
print (v)
#Output: 57.29577951308232
print (math.radians(57.29577951308232))
#Output: 1.0
```

Built-in math functions:

There are two built in math functions, abs (absolute value) and round that are available without importing the math module.

Here are some examples:

```
print(abs(-4.3))
print(round(3.336, 2))
print(round(345.2, -1))
#Output: 4.3
3.34
350.0
```

The **round** function takes two arguments: the first is the number to be rounded and the second is the number of decimal places to round to. The second argument can be negative.

1.4.5 cmath Module

The **cmath module** provides many of the functions found in the math module, but these accept complex numbers.

Complex Numbers

The numbers we have seen so far are the so-called real numbers. Python also supports complex numbers with the imaginary part identified by the letter j or J (as opposed to the letter i used in mathematical notation).

For example, the complex number $2 + 3i$ would be written in Python as $2 + 3j$:

```
>>> a = 2 + 3j
>>> type(a)
<class 'complex'>
```

As you can see, when we use the type() function on a complex number, Python tells us that this is an object of type complex. You can also define complex numbers using the complex() function:

```
>>> a = complex(2, 3)
>>> print (a)
(2 + 3j)
```

Here are examples of complex arithmetic:

```
>>> from cmath import sin
>>> x = 3.0 - 4.5j
>>> y = 1.2 + 0.8j
>>> z = 0.8
>>> print x/y
(-2.56205313375e - 016 - 3.75j)
>>> print sin(x)
(6.35239299817 + 44.5526433649j)
>>> print sin(z)
(0.7173560909 + 0j)
```

1.4.6 Keyboard input: input() statement

The following are the examples on keyboard input.

Example(i):

```
""" Row Input"""
from math import *
x = int(input("Input an integer: "))
y = float(input("Input a float: "))
print (x, y)
#Output: Input an integer: 7
Input a float: 2.5
7 2.5
```

Example(ii):

```
""" Row Input"""
from math import *
x = int(input("Input an integer: "))
print (x)
y = float(input("Input a float: "))
print (y)
print (x+y)
print (x/y)
print (x%y)
print (x**y)
print (x**2+y**2)
print (x//y)
#Output: Input an integer: 9
9
Input a float: 1.5
1.5
10.5
6.0
0.0
27.0
83.25
6.0
```

1.4.7 User Defined Functions

We have only been using the functions that come with Python, but it is also possible to add new functions. A function definition specifies the name of a new function and the sequence of statements that execute when the function is called. A function is a block of code which only runs when it is called. We can pass data, known as parameters, into a function. In Python a function is defined using the def keyword.

The syntax is given as follows:

```
def f_name(parameters):
    statement 1
    statement 2
    statement 3
⋮
```

For example:

```
>>> def f(x):
    r=x*5
print(r)
```

The first line tells Python that we are defining a new function and we are naming it f. The following lines are indented to show that they are part of the f function. To call a function, use the function name followed by parameter:

```
>>> f(2)
#Output: 10
>>> f('Pune')
#Output: PunePunePunePunePune
>>> f("Pune")
#Output: PunePunePunePunePune
```

Note that the parameters in function is not necessary.

For example:

```
>>> def hello() :
    print ("Hello")
    print("Computers are fun!")

>>> hello()
Hello
Computers are fun!
```

Consider the following simple programmes:

Example 1.2. *Write a function that calculate square of a given number.*

```
>>> def square(x):
    s = x * x
    print(s)
```

Or

```
>>> def square(x):
    print(x * x)

>>> square(5)
25
>>> square(5.2)
27.040000000000003
```

Example 1.3. *Write a function that gives* sin, cos *and* log *of a given number.*

```
>>> import math
>>> def f(x):
    a=math.sin(x)
    b=math.cos(x)
    c=math.log(x)
    print('sin',x,'=',a)
    print('cos',x,'=',b)
    print('log',x,'=',c)

>>> f(1.2)
sin 1.2 = 0.9320390859672263
cos 1.2 = 0.3623577544766736
log 1.2 = 0.1823215567939546

>>> f(math.pi)
sin 3.141592653589793 = 1.2246467991473532e-16        # Python calculate using numerical
iterations
cos 3.141592653589793 = -1.0
log 3.141592653589793 = 1.1447298858494002
```

Example 1.4. *Write a function that calculates sum and products of three number.*

```
>>> def sp(x,y,z):
    sum = x + y + z
    prod = x * y * z
    print('sum=',sum)
    print('product=',prod)

>>> sp(1,2,3)
sum = 6
product=6

>>> sp(1.1,2.2,3.3)
sum=6.6
product=7.986000000000001
```

Example 1.5. *Write a function that calculates quotient and remainder when 'a' divided by 'b'.*

```
>>> def division(a,b):
    q = a//b
    r = a%b     print('Quotient is:',q)
    print('Remainder is:',r)

>>> division(7,2)
Quotient is: 3
Remainder is: 1

>>> division(-7,2)
Quotient is: -4
Remainder is: 1          #Remainder assume to be positive.

>>> division(1000,7)
Quotient is: 142
Remainder is: 6
```

Example 1.6. *Write a function that calculates area and circumference of a circle if radius is given.*

```
>>> import math
>>> def circle(radius):
    area = math.pi* radius**2
    circumference = 2* math.pi * radius
    print('area is',area)
    print('circumference is', circumference)

>>> circle(2)
area is 12.566370614359172
circumference is 12.566370614359172

>>> circle(7)
area is 153.93804002589985
circumference is 43.982297150257104
```

Example 1.7. *Write a function that calculates roots of the quadratic equation* $ax^2 + bx + c = 0$.

```
>>> import cmath
>>> def f(a,b,c):
    r = -b/(2*a)
    i=cmath.sqrt(b*b-4*a*c)/(2*a)
    print('Roots=',r+i,',',r-i)

>>> f(1,1,4)
Roots= (-0.5+1.9364916731037085j) , (-0.5-1.9364916731037085j)

>>> f(1,2,1)
Roots = (-1+0j) , (-1+0j)

>>> f(1,1,1)
Roots= (-0.5+0.8660254037844386j) , (-0.5-0.8660254037844386j)
>>> f(1,3,2)
Roots = (-1+0j) , (-2+0j)

>>> f(1,-1,1)
Roots = (0.5+0.8660254037844386j) , (0.5-0.8660254037844386j)
```

1.5 Boolean expressions

A boolean expression is an expression that is either true or false. Python supports the usual logical conditions from mathematics.

The comparison operators are ==, >, <, >=, <=, and ! =.

Expression	Description
$x == y$	x is equal to y
$x! = y$	x is not equal to y
$x > y$	x is greater than y
$x < y$	x is less than y
$x >= y$	x is greater than or equal to y
$x <= y$	x is less than or equal to y

Note that == operator compares two numbers and produces True if they are equal, otherwise False.

For example

```
>>> 4 == 4
True
>>> 5 == 6
```

False

Similarly,

>>> 4 < 9

True

>>> 10 >= 48

False

1.5.1 Logical operators

There are three logical operators: and, or, not.

This operators works as usual.

>>> 8 == 8 and 5! = 8

True

>>> 8 == 9 and 5! = 8

False

>>> 4 > 7 or 7 < 10

True

>>> x = 5

>>> not x == 10

True

1.5.2 and opeator

Evaluates to the second argument if and only if both of the arguments are truthy. Otherwise evaluates to the first falsey argument.

The following are the examples.

Example(i):

x = True

y = True

z = x and y

print (z)

#Output: True

Example(ii):

x = True

y = False

z = x and y

print (z)

#Output: False

Example(iii):

```
x = False
y = True
z = x and y
print (z)
#Output: False
```

Example(iv):

```
x = False
y = False
z = x and y
print (z)
#Output: False
```

Example(v):

```
x = 1
y = 1
z = x and y
print (z)
#Output: z = 1
```

Note that 'and' and 'or' are not guaranteed to be a boolean.

```
x = 0
y = 1
z = x and y
print (z)
#Output: z = x, so z = 0
x = 1
y = 0
z = x and y
print (z)
# Output: z = y, so z = 0
x = 0
y = 0
z = x and y
print (z)
#Output: z = x, so z = 0
```

The $1's$ in the above example can be changed to any truthy value, and the $0's$ can be changed to any falsey value.

1.5.3 or opeator

Evaluates to the first truthy argument if either one of the arguments is truthy. If both arguments are falsey, evaluates to the second argument.

The following are the examples.

Example(i):

```
x = True
y = True
z = x or y
print (z)
#Output: True
```

Example(ii):

```
x = True
y = False
z = x or y
print (z)
#Output: True
```

Example(iii):

```
x = False
y = True
z = x or y
print (z)
#Output: True
```

Example(iv):

```
x = False
y = False
z = x or y
print (z)
#Output: False
```

Example(v):

```
x = 1
y = 1
z = x or y
print (z)
#Output: z = 1
```

Example(vi):

```
x = 0
y = 0
z = x or y
print (z)
#Output: z = 0
```

Here, the $1's$ in the above example can be changed to any truthy value, and the $0's$ can be changed to any falsey value.

1.5.4 not opeator

It returns the opposite of the following statement:

```
x = True
y = not x
#Output: False
x = False
y = not x
#Output: True
```

Example 1.8. *If x and y are false then find the value of "(x **or** y) **and (not** x **or not** y)".*

Solution: Here, x and y are false.

```
x = False
y = False
p = x or y
q = not x or not q
r = p and q
print (r)
#Output: False
```

Example 1.9. *Find the value of the following expressions if x and y are true and z is false.*

(i) (x **or** y) **and** z

(ii) (x **and** y) **or not** z

(iii) (x **and not** y) **or** (x **and** z)

Solution: Here, x and y are true and z is false.

(i)
```
x = True
y = True
z = False
```

```
p = x or y
print (p)
#Output: True
q = p and z
print (q)
#Output: False
```

The value of the expression (x **or** y) **and** z is False.

(ii) x = True

```
y = True
z = False
p = x and y
print (p)
#Output: True
q = p or not z
print (q)
#Output: True
```

The value of the expression (x **and** y) **or not** z is True.

(iii) x = True

```
y = True
z = False
p = x and not y
q = x and z
r = p or q
print (r)
#Output: False
```

The value of the expression (x **and not** y) **or** (x **and** z) is False.

1.6 Excercise

1. Start up an interactive Python session and try typing in each of the following commands. Write down the results you see.

   a) print ("Hello, world! ")

   b) print("Hello", "world!")

   c) print (3)

   d) print(3.0)

   e) print (2 + 3)

f) print(2. 0 + 3. 0)

g) print (" 2" + "3 ")

h) print ("2 + 3 =", 2 + 3)

i) print(2 * 3)

j) print (2 ** 3)

k) print (7 / 3)

1) print(7// 3)

2. Write a program that converts temperatures from Fahrenheit to Celsius.

3. Write a program that converts distances measured in kilometers to miles.
One kilometer is approximately 0.62 miles.

4. Show the result of evaluating each expression. Be sure that the value is in the proper form to indicate its type (int or float). If the expression is illegal, explain why.

a) 4.0 / 10.0 + 3.5* 2

b) 10 % 4 + 6 / 2

c) abs (4 - 20 // 3)** 3

d) sqrt (4.5 - 5.0) + 7 *3

e) 3 * 10 // 3 + 10 % 3

f) 3 ** 3

5. Translate each of the following mathematical expressions into an equivalent Python expression. You may assume that the math library has been imported (via import math).

a) $(3+4)(5)$

b) $\frac{n(n-1)}{2}$

c) $4\pi r^2$

d) $\sqrt{r(\cos a)^2 + r(\sin b)^2}$

e) $\frac{y_2-y_1}{x_2-x_1}$

6. Write a program to calculate the volume and surface area of a sphere from its radius, given as input. Here are some formulas that might be useful:

$$V = 4/3\pi r^3$$

$$A = 4\pi r^2$$

7. Write a program that calculates the cost per square inch of a circular pizza, given its diameter and price. The formula for area is πr^2.

8. Two points in a plane are specified using the coordinates (x_1, y_1) and (x_2, y_2). Write a program that calculates the slope of a line through two (non-vertical) points entered by the user.

9. Write a program that accepts two points and determines the distance between them.
10. Write a program to calculate the area of a triangle given the length of its three sides-a, b, and c.
11. Write a program to find the sum of the cubes of the first n natural numbers where the value of n is provided by the user.
12. Write a Python program to find the median of three values.

Chapter 2

String, List and Tuple

2.1 Strings

A string is a sequence of characters enclosed in single or double quotes. Strings are a data type in Python for dealing with text. Also, Python has a number of powerful features for manipulating strings. **Creating a string-** A string is created by enclosing text in quotes. We can use either single quotes,′, or double quotes ". A triple-quote can be used for multi-line strings.

Here are some examples:

s = 'Hello'

print (s)

#Output: Hello

t = "Hello"

print (t)

#Output: Hello

m = """This is a long string that is spread across two lines."""

print (m)

#Output: This is a long string that is spread across two lines.

Input Recall: For numerical input we use an eval statement with the input statement, but when getting text, we do not use eval. The difference is illustrated below.

Example:

num = eval(input('Enter a number: '))

string = input('Enter a string: ')

#Output:

Enter a number: 9

9

Enter a string: Hello Python

Hello Python

Note that the empty string '' is the string equivalent of the number 0.

It is a string with nothing in it. We have seen it before, in the print statement's optional argument, sep=''.

2.1.1 Length

To obtain the length of a string (how many characters it has), we use the built-in function **len("...")**

For example: Length of the str: "Hello"

```
L=len("Hello")
print (L)
#Output: 5
```

Consider the following examples:

Example (i):

```
L=len("Python")
print (L)
#Output: 6
```

Example (ii):

```
L=len("Mathematics")
print (L)
#Output: 11
```

Example (iii):

```
x=len('Good Morning')
print (x)
#Output: 12
y=len("KALYANRAO")
print(y)
#Output: 9
```

Remark 2.1. *The operators + and * can be used on strings. The + operator combines two strings and this operation is called* ***concatenation.***

The * repeats a string a certain number of times.

Example 2.1. *Using the operators + and * find "XY" + "pq", "X" + "9" + "Y" and "Hello" * 5.*

Solution: We use the + operator combines two strings and * repeats a string a certain number of times.

```
a = "XY" + "pq"
print (a)
#Output: XYpq
```

```
s = "X" + "9" + "Y"
print (s)
#Output: X9Y

t = "Hello"*5
print (t)
#Output:HelloHelloHelloHelloHello
```

Example 2.2. *To print a long row of dashes.*

```
print ("-"*25)
#Output: -------------------------
```

2.1.2 Indexing

We will often want to pick out individual characters from a string, Python uses square brackets to do this.

The table below gives some examples of indexing the string s='Python'.

Statement	Result	Description
s[0]	P	first character of s
s[1]	y	second character of s
s[-1]	n	last character of s
s[-2]	o	second-to-last character of s

A common error Suppose s='Python' and we try to do s[12].

```
s='Python'
print (s[10])
#Output: Traceback (most recent call last):
File "D:\ python 1_1_2020\ Seminar on Sfotware \ P222.py", line 48, in <module>
print (s[10])
IndexError: string index out of range
```

There are only six characters in the string and Python will raise the above error message.

Example 2.3. *Pick out individual characters from a string s='Python' to form 'PythonnohtyP'.*

s='Python'	Output
print (s[0])	P
print (s[1])	y
print (s[2])	t
print (s[3])	h
print (s[4])	o

```
print (s[5])        n
print (s[-1])       n
print (s[-2])       o
print (s[-3])       h
print (s[-4])       t
print (s[-5])       y
print (s[-6])       P
```

To become familiar with them, you may write simple programs performing arithmetic and logical operations using them.

```
#Example: oper.py
import math
import numpy as np
x = 2
y = 4
print (x + y * 2)
s = 'Hello '
print (s + s)
print (3 * s)
print (x == y)
print (y == 2 * x)
#Output: 10
Hello Hello
Hello Hello Hello
False
True
#Example: string.py
import math
import numpy as np
s = 'hello world'
print (s[0])            # print first element, h
print (s[1])            # print e
print (s[-1])           # will print the last character
#Output:
h
e
d
```

```
#Example: string2.py
import math
import numpy as np
a = 'hello'+'world'
print (a)
b = 'ha' * 3
print (b)
print (a[-1] + b[0])
#Output:
helloworld
hahaha
dh
```

Example 2.4. *Write a program in Python that repeatedly asks the user to enter a letter and builds up a string consisting of only the vowels that the user entered.*

Solution:

```
s =' '
for i in range(5):
t = input('Enter a letter: ')
if t=='a' or t=='e' or t=='i' or t=='o' or t=='u':
s = s + t
print(s)
#Output:
Enter a letter: a
a
Enter a letter: e
ae
Enter a letter: i
aei
Enter a letter: o
aeio
Enter a letter: u
aeiou
>>> s
'aeiou'
```

2.1.3 Slices

The part of a String can be extracted using the slicing operation. Indexing using $s[a : b]$ extracts elements $s[a]$ to $s[b..1]$. A slice is used to pick out part of a string. It behaves like a combination of indexing and the range function. The following are some examples.

Example 2.5. *Given the string s='abcdefghij'.*
index: 0 1 2 3 4 5 6 7 8 9
letters: *a b c d e f g h i j*
Calculate $s[2 : 5]$, $s[: 5]$, $s[5 :]$, $s[-2 : 5]$, $s[:]$, $s[1 : 7 : 2]$, $s[:: -1]$.

Solution: We have a sting s='abcdefghij'.

```
>>> s='abcdefghij'
>>> s[2:5]
#Output: 'cde'              # characters at indices 2, 3, 4
>>> s[:5]
#Output: 'abcde'            #first five characters
>>> s[5:]
#Output: 'fghij'
>>> s[-2:5]
#Output: ''
>>> s[: ]
#Output: 'abcdefghij'
>>> s[1:7:2]
#Output: 'bdf'
>>> s[ : :-1]
#Output: 'jihgfedcba'       # a negative step reverses the string
```

Example 2.6. *Write a program to pick out part of a string 'hello world'.*

```
#Example: slice.py
import math
import numpy as np
a = 'hello world'
print (a[3:5])
print (a[6:])
print (a[:5])
print (a[2:8])
```

#Output:lo
world
hello
llo wo

Example 2.7. *To extract a portion of the string by using strings are concatenated with the plus (+) operator, where as slicing (:).*

Solution: Here is an example:

```
>>> string1 = Press return to exit
>>> string2 = the program
>>> print string1 + " " + string2        # Concatenation
Press return to exit the program
>>> print string1[0 : 12]                # Slicing
Press return
```

Example 2.8. *Give an examples of repetition, append elements and Concatenation of strings and sequences using some operators.*

There are some of these operators are also defined for strings and sequences as illustrated below.

```
>>> s = 'Hello'
>>> t = 'to you'
>>> a = [1, 2, 3]
>>> print (3 * s)                        # Repetition
Hello Hello Hello
>>> print (3 * a)                        # Repetition
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> print (a + [4, 5])                   # Append elements
[1, 2, 3, 4, 5]
>>> print (s + t)                        # Concatenation
Hello to you
>>> print (3 + s)                        # This addition makes no sense
Traceback (most recent call last):
File " < pyshell#9 > ", line 1, in ?
print (n + s)
TypeError: unsupported operand types for +: int and str
```

Example 2.9. *Give and example of the comparison between integer, floating point and strings.*

Numbers of different type (integer, floating point etc.) are converted to a common type before the comparison is made. Otherwise, objects of different type are considered to be unequal.
Here are a few examples:

```
>>> a = 2                    # Integer
>>> b = 1.99                 # Floating point
>>> c =' 2'                  # String
>>> print (a > b)
1
>>> print (a == c)
0
>>> print (a > b) and (a! = c)
1
>>> print (a > b) or (a == b)
1
```

2.2 Lists

The Python List is a general data structure widely used in Python programs. They are found in other languages, often referred to as dynamic arrays. They are both mutable and a sequence data type that allows them to be indexed and sliced. The list can contain different types of objects, including other list objects. A list is similar to a tuple, but it is mutable, so that its elements and length can be changed. A list is identified by enclosing it in brackets.Here is a sampling of operations that can be performed on lists:

```
>>> a = [1.0, 2.0, 3.0]      # Create a list
>>> a.append(4.0)            # Append 4.0 to list
>>> print (a)
[1.0, 2.0, 3.0, 4.0]
>>> a.insert(0,0.0)          # Insert 0.0 in position 0
>>> print (a)
[0.0, 1.0, 2.0, 3.0, 4.0]
>>> print (len(a))           # Determine length of list
5
>>> a[2 : 4] = [1.0, 1.0]    # Modify selected elements
>>> print (a)
[0.0, 1.0, 1.0, 1.0, 1.0, 4.0]
```

Example 2.10. *Write a program to print list and obtain their length.*

```
L = [1,2,3,4,5,6,7,8,9]
print(L)
print (len(L))
#Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
9
```

Input:

We can use **eval(input())** to allow the user to enter a list.

Example 2.11. *Write a program enter 9 digits and pick the their elements.*

```
L = eval(input('Enter a list: '))
print('The first element is ', L[0])
print('The last element is ', L[8])
print('The fifth element is ', L[4])
#Output:
Enter a list: 1,2,3,4,5,6,7,8,9
The first element is 1
The last element is 9
The fifth element is 5
```

Data types:

Lists can contain all kinds of things, even other lists.

Example 2.12. *Write a program to print valid list and obtain their length.*

```
L=[5, 23.14, 'xyz', [1,2,3]]
print(L)
print (len(L))
#Output:[5, 23.14, 'xyz', [1, 2, 3]]
4
```

Corollary 2.1. ***(i) Indexing and slicing:-*** *These work exactly as with strings.*

Example 2.13. *Write a program to print valid list and obtain the first item of the list and the first three items etc.*

```
>>> L = [1,2,3,4,5,6,7]
>>> print(L[0])
1
>>> print( L[:3])
```

```
[1, 2, 3]
>>> print(L[4:2])
[]
>>> print (L[:-1])
[1, 2, 3, 4, 5, 6]
```

Corollary 2.2. ***Use of operators + and * in a list:***
*The + operator adds one list to the end of another. The * operator repeats a list.*

Example 2.14. *Write a program to combine and repeat a list.*

```
>>> s=[1,2,3,4]
>>> t=[5,6,7,8,9]
>>> print (s+t)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print (s*4)
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
>>> print (t*3)
[5, 6, 7, 8, 9, 5, 6, 7, 8, 9, 5, 6, 7, 8, 9]
>>> print (s+t*2)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 5, 6, 7, 8, 9]
```

2.3 Tuple

2.3.1 Defination

A tuple is a sequence of values. The values can be any type viz., numbers, floats, string etc., Members in tuple are indexed by integers. In simple word, a tuple is a comma-separated list of values. That list is enclose by parentheses.

For Examples:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> # Parentheses are not compulsory.
>>> t1 = 'a', 'b', 'c', 'd', 'e'
>>> # We can display the type of 't1'.
>>> type(t1)
<class 'tuple'>
```

tuple() is a inbuilt function that converts any sequence(string, list etc.) to tuple.

For Examples:

```
>>> t = tuple('lupins')
>>> print(t)
('l', 'u', 'p', 'i', 'n', 's')
>>> t1 = tuple([1, 2, 3, 'Python', 'Panda'])
>>> print(t1)
(1, 2, 3, 'Python', 'Panda')
```

Most of the list operations are also work on tuples.

2.3.2 Index operator

As similar to other sequence types, the index operator $\mathbf{t}(i)$ display the ith member of a tuple(indexing first member to 0).

Consider tuple,

>>> t = ('a', 'b', 'c', 'x', 'y', 'z')

Indexing as follows:

t(i)	'a'	'b'	'c'	'x'	'y'	'z'
i	0	1	2	3	4	5
	−6	−5	−4	−3	−2	−1

Consider tuple:

```
>>> t1=('I', 'n', 'd', 'i', 'a')
>>> t1[0]
'I'
>>> t1[-1]
'a'
>>> t1[2]
'd'
>>> t1[-4]
'n'
```

2.3.3 Slice operator

As similar to string and list, the part of a tuple can be extracted using the slicing operation. Indexing using $t[a:b]$ extracts elements $t[a]$ to $t[b..1]$. A slice is used to pick out part of a string.

Consider tuple,

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[2:4]
```

```
('c', 'd')
>>> t[1:3]
('b', 'c')
>>> t[-1]
'e'
>>> t[:3]
('a', 'b', 'c')
```

The tuples are immutable, if you try to modify one of the elements of the tuple, you get an error:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[0] = 'A'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Example 2.15. *Copy element 44 and 55 from the following tuple into a new tuple*

```
>>> tuple1 = (11, 22, 33, 44, 55, 66)
```

We define new tuple using slice operator,

```
>>> tuple1 = (11, 22, 33, 44, 55, 66)
>>> tuple2=tuple1[3:5]
>>> tuple2
(44, 55)
```

Example 2.16. *Sort a tuple of tuples by ascending order.*

```
>>> tuple1 = (44, 11, 66, 22, 33, 55)
```

First we convert tuple into list, then apply **sorted()** function then again convert into tuple.

```
>>> tuple1 = (44, 11, 66, 22, 33, 55)
>>> tuple(sorted(list(tuple1)))
(11, 22, 33, 44, 55, 66)
```

2.3.4 Tuple assignment

Tuples are usefull to assign values to multiple variables at instance.

For examples to assign $x = 2$, $y = 4$ and $z = 9$:

>>> (x, y, z) = (2, 4, 9)

Or

>>> x, y, z = 2, 4, 9

The left side is a tuple of variables; the right side is a tuple of expressions. Each value is assigned to its respective variable.

All the expressions on the right side are evaluated before any of the assignments.

For example:

```
>>> x, y = 3, 8
>>> x,y = x+y,x-y
>>> print(x,y)
11 -5
```

The number of variables on the left and the number of values on the right have to be the same, otherwise you will get an error:

```
>>> a, b = 1, 2, 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
```

Example 2.17. *Swap the following two tuples*

```
>>> tuple1 = (11, 22)
>>> tuple2 = (99, 88)

>>> tuple1 = (11, 22)
>>> tuple2 = (99, 88)
>>> tuple1, tuple2 = tuple2, tuple1
>>> tuple1
(99, 88)
>>> tuple2
(11, 22)
```

Example 2.18. *Unpack the following tuple into 4 variables*

```
>>> aTuple = (10, 20, 30, 40)

>>> aTuple = (10, 20, 30, 40)
>>> (x,y,z,w) = aTuple
>>> x
10
>>> y
```

```
20
>>> z
30
>>> w
40
```

2.3.5 Tuple as a return value

Any function can only return one value, but if the value is a tuple, then it can return multiple values. The built-in function **divmod()** takes two arguments and returns a tuple of two values, the quotient and remainder.

```
>>> divmod(7, 3)
(2, 1)
```

We can use tuple assignment to store the elements separately:

```
>>> quot, rem = divmod(7, 3)
>>> print quot
2
>>> print rem
1
```

Here is an example of a function that returns a tuple:

```
>>> def min_max(t):
return min(t), max(t)
```

In above programme **max** and **min** are built-in functions that find the largest and smallest elements of a sequence. **min_max** computes both and returns a tuple of two values.

Example 2.19. *Write aPython programm using tuple that swap the values of two variable.*

```
>>> def swap(x,y):
...     x,y = y,x
...     return x, y
...
>>>
>>> x=4
>>> y=7
>>> swap(x,y)
(7, 4)
```

2.4 Exercise

1. Given the initial statements:

s1 - " spam"

s2 - "ni ! "

Show the result of evaluating each of the following string expressions.

a) `"The Knights who say,"+s2`
b) `3*s1+2*s2`
c) `s1[1]`
d) `s1[1:3]`
e) `s1[2]+s2[:2]`
f) `s1+s2[-1]`
g) `s1.upper()`
h) `s2.upper().ljust(4)*3`

2. Two lists are given:

```
l1=[1,2,3,4,5]
l2=['a','b','c','d','e']
```

Show the result of evaluating each of the following string expressions.

a) `l1[4]`
b) `l2[:3]`
c) `l1[2:]+l2[:3]`
d) `l1+l2[-1]`
e) `l1+l2`
f) `l2*3`

3. Two tuples are given:

```
t1=('m','u','m','b','a','i')
t2=('p','u','n','e')
```

Show the result of evaluating each of the following string expressions.

a) `t1[3]`
b) `t2[:4]`
c) `t1[1:]+t2[:1]`

d) `t1+t2[-1]`

e) `t1+t2`

f) `t2*2`

4. Write a Python program to count the number of characters (character frequency) in a string.
5. Write a Python program to get a string made of the first 2 and the last 2 chars from a given a string.
6. Write a Python function that takes a list of words and returns the length of the longest one.
7. Write a Python program to change a given string to a new string where the first and last chars have been exchanged.
8. Write a Python program to add 'ing' at the end of a given string.
9. Write a Python program to reverse a tuple.
10. Write a Python program to find the length of a tuple.
11. Write a Python program to check whether an element exists within a tuple.
12. Write a Python program to sum all the items in a list.
13. Write a Python program to multiplies all the items in a list.
14. Write a Python program to get the largest number from a list.

Chapter 3

Iterations and Conditional statements

3.1 Conditional and alternative statements, Chained and Nested Conditionals: if, if-else, if-elif-else, nested if, nested if-else

To write useful programs, we always need the ability to check conditions and change the behavior of the program accordingly. Conditional statements give us this ability.

3.1.1 Conditional execution

It is the simplest form of conditional execution.

Syntax:

```
if (condition):
    statements
```

The boolean expression after if is called the condition. If it is true, then the followed statement gets executed. If not, nothing happens. **if** statements have the same structure as function definitions: a header followed by an indented body. The lines that are indented will be executed only if the condition is true. Once the indentation is done with, the if block is concluded. There is no limit on the number of statements that can appear in the body, but at least one should be there.

For example:

```
>>> x=5
>>> if x<10:
...     print(x,' is less that 10.')
...
5  is less that 10.
```

Consider the following simple programmes:

Example 3.1. *Write a function that gives number is positive if it is.*

```
>>> def p(x):
...     if x > 0:
```

```
...         print (x,'is positive.')
...
>>> p(5)
5 is positive.
>>> p(-8)  # no outputs
```

Example 3.2. *Write a function that prints whether number is divisible by another number.*

```
>>> def isdivisible(x,y):
...     if x%y==0:
...         print(x,'is divisible by',y)
...
>>> isdivisible(15,5)
15 is divisible by 5
>>> isdivisible(21,4)   # no outputs
```

Example 3.3. *Write a function that tests whether given number is multiple of 5 or not.*

```
>>> def multi5(a):
...     if a%5==0:
...         print(a,'is multiple of 5.')
...
>>> multi5(26)   # no outputs
>>> multi5(45)
45 is multiple of 5.
```

Example 3.4. *Write a function that tests whether given number is divisible by 2,3 and 5.*

```
>>> def f(x):
...     if x%2==0:
...         if x%3==0:
...             if x%5==0:
...                 print(x,'is divisible by 2, 3 and 5.')
...
...


>>> f(12)    # no output
>>> f(20)    # no output
>>> f(120)
120 is divisible by 2, 3 and 5.
```

There are three additional operators used to construct more complicated conditions: **and, or, not.** Here are some examples:

1.
```
>>> def f(x):
...     if x>=100 and x%5==0:
...         print(x,'is divisible by 5 and greater than 100.')
...
>>> f(12)  # no outputs
>>> f(140)
140 is divisible by 5 and greater than 100.
```

2.
```
>>> def f(x,y):
...     if x%2==0 or y%2==0:
...         print(x,'or',y,'is even.')
...
>>> f(3,9)  # no outputs
>>> f(7,-4)
7 or -4 is even.
```

Remark: In case of order of operations, **and** is done before **or**, so if you have a condition that contains both, you have to apply parentheses around the **or** condition. **and** work like multiplication and **or** work like addition.

3.1.2 Alternative execution(if___ else___)

A second form of the if statement is alternative execution, in which there are two possibilities and the condition determines which one gets executed.

Syntax:

```
if (condition):
    statements
else:
    statements
```

If the condition is true then first set of statements are executed, otherwise the second set of statements is executed. Since the condition must be true or false, exactly one of the alternatives will be executed. The alternatives are called branches, because they are branches in the flow of execution.

For example:

```
>>> x=11
>>> if x%2==0:
```

```
...     print(x,'is even.')
... else:
...     print(x,'is odd.')
...
11 is odd.
```

Consider the following simple programmes:

Example 3.5. *Write a function that gives absolute value of a given real number.*

```
>>> def f(x):
...     if x<0:
...         print(-x)
...     else:
...         print(x)
...


>>> f(10)
10
>>> f(-14.14)
14.14
>>> f(0.123)
0.123
```

Example 3.6. *Write a function that gives given number is positive or negative.*

```
>>> def p(x):
...     if x>0:
...         print(x,'is positive.')
...     else:
...         print(x,'is negative.')
...


>>> p(5)
5 is positive.
>>> p(-19)
-19 is negative.
>>> p(0)      # Mathematically it's not correct.
0 is negative.
```

Now we create **nested alternative execution.**

Example 3.7. *Write a function that gives given number is positive or negative or zero.*

```
>>> def p(x):
...     if x>0:
...         print(x,'is positive.')
...     else:
...         if x<0:
...             print(x,'is negative.')
...         else:
...             print('Given number is zero.')
...
...
>>>
>>> p(15)
15 is positive.
>>> p(-9)
-9 is negative.
>>> p(0)
Given number is zero.
```

3.1.3 Chained conditionals(If___elif___else___)

If there are more than two possibilities and we need more than two branches. One way to express a computation like that is a chained conditional:

```
if x<y:
    print(x,'is less than',y)
elif x>y:
    print(x,'is greater than',y)
else:
    print(x,'and',y,'are equal')
```

Here **elif** is an abbreviation of "else if". There is no limit on the number of elif statements. If there is an else clause, it has to be at the end. Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

Syntax:

```
if (condition1):
    print(value condition1 true)
elif (condition2):
    print(value condition2 true)
elif (condition3):
    print(value condition3 true)
        :
        :
else:
    print(value condition-n true)
```

Consider the following simple programmes:

Example 3.8. *Write a function that estimate the class for a given marks according to table.*

Marks range	*Class*
0 - 39	*Fail*
40 - 49	*Second Class*
50 - 59	*Higher Second Class*
60 - 75	*First class*
76 - 100	*First class with distinction*

```
>>> def cl(x):
...     if x <= 39:
...         print('Fail')
...     elif x <= 49:
...         print('Second Class')
...     elif x <= 59:
...         print('Higher Second Class')
...     elif x <= 75:
...         print('First Class')
...     else:
...         print('First class with distinction')
...

>>>
>>> cl(55)
Higher Second Class
```

```
>>> cl(79)
First class with distinction
>>> cl(36)
Fail
```

Example 3.9. *Write a program that asks the user how many credits they have taken. If they have taken 23 or less, print that the student is a freshman. If they have taken between 24 and 53, print that they are a sophomore. The range for juniors is 54 to 83, and for seniors it is 84 and over.*

```
>>> def credits(x):
...     if x <= 23:
...         print('Student is freshman')
...     elif x <= 53:
...         print('Student is sophomore')
...     elif x <= 83:
...         print('Student is juniors')
...     else:
...         print('Student is seniors')
...
>>>
>>> credits(55)
Student is juniors
>>> credits(89)
Student is seniors
>>> credits(16)
Student is freshman
```

3.2 Looping statements such as while, for etc, Tables using while.

3.2.1 while loop

The while loop can be found in most programming languages. It is mostly used to perform an action provided certain conditions are met.

Syntax:

```
while (condition):
    loop statements
```

Flow of execution for a while statement:

1. Evaluate the condition, yielding True or False.

2. If the condition is false, exit the while statement and continue execution at the next statement.
3. If the condition is true, execute the body and then go back to step 1.

This type of flow is called a loop because the third step loops back around to the top. The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an infinite loop. As per the Python syntax, the while statement ends with a colon and the code inside the while loop is indented. Indentation can be done using tab or few spaces. In this example, we have demonstrated a simple algorithm.

Here are some simple examples:

Example 3.10. *Print the first 5 natural numbers.*

```
>>> i=1
>>> while i<=5:
...     print(i)
...     i=i+1
...
1
2
3
4
5
```

Example 3.11. *Find the sum of first 10 natural numbers using while loop.*

```
>>> sum=0
>>> n=1
>>> while n <= 10:
...     sum=sum+n
...     n=n+1
...
>>> print('sum=',sum)
sum= 55
```

The **while loop** executes as long as the condition (here: $n < 10$) remains true.

Example 3.12. *Find the product of first n natural numbers using while loop.(factorial)*

```
>>> product=1
>>> n=1
>>> while n <= 10:
...     product=product*n
...     n=n+1
...
>>> print('product =', product)
product = 3628800
```

Example 3.13. *Print Fibonacci numbers less than 1000.*

```
>>> a, b = 0, 1
>>> while a < 1000 :
...     print(a,end=',' )
...     a, b = b, a + b
...
...
0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

Remark: Command **end** =',' in print gives outputs in comma seperated sequence.

3.2.2 Tables using while

We can generate the multiplication table for any number using while loop. We define a function **multitable(n)** that takes number **n** as argument and print **i*n**. The value of **i** starts with 1 incremented by 1 in each cycle. This process will be repeated until the value of **i** becomes greater than 10.

```
>>> def multitable(n):
...     i=1
...     while i<=10:
...         print(i*n)
...         i=i+1
...
>>> multitable(7)
7
14
21
```

```
28
35
42
49
56
63
70
>>> multitable(23)
23
46
69
92
115
138
161
184
207
230
```

We can display multiplication table as follows:

```
>>> def multitable(n):
...     i=1
...     while i<=10:
...         print(n,'x',i,'=',i*n)
...         i=i+1
...
...
>>> multitable(24)
24 x 1 = 24
24 x 2 = 48
24 x 3 = 72
24 x 4 = 96
24 x 5 = 120
24 x 6 = 144
24 x 7 = 168
24 x 8 = 192
```

```
24 x 9 = 216
24 x 10 = 240
```

3.2.3 range() function

We can generate a sequence of numbers using range() function. The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

range(10) will generate numbers from 0 to 9 (10 numbers).

This function does not store all the values in memory, it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go. To force this function to prints all the items, we can use the function **list()**.

We can also define the start, stop and step size as:

>>> range(start , stop , step size)

step size defaults to 1 if not provided. The following example will clarify this.

```
>>> print(list(range(10)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print(list(range(2, 8)))
[2, 3, 4, 5, 6, 7]
>>> print(list(range(20, 8)))
[]
>>> print(list(range(2, 20, 3)))
[2, 5, 8, 11, 14, 17]
>>> print(list(range(25, 10, -3)))
[25, 22, 19, 16, 13]
```

Example 3.14. *Find the numbers of integers between 1 to 1000, which are multiple of 7.*

```
>>> n=range(7,1000,7)
>>> len(n)
142
```

We can use the range() function in for loops to iterate through a sequence of numbers. It can be combined with the len() function to iterate though a sequence using indexing.

3.2.4 for loop

Loop means we proceed same set of statements again and again. We use loops to execute a sequence of statements multiple times in succession. The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects.

Syntax:

```
for (variable) in (sequence):
    (body of for)
```

The word **for** should be in lowercase, the first line must end with a colon, and the lines to be repeated must be indented. Indentation is used to tell Python which lines will be repeated.

Here are some simple examples:

Example 3.15. *Print four times massage 'Hello!' using for loop.*

```
>>> for i in range(4):
...     print('Hello!')
...
...
Hello!
Hello!
Hello!
Hello!
```

Example 3.16. *Print square of numbers in the list [0, 2, 7, 9].*

```
>>> for i  in [0, 2, 7, 9] :
...     print(i*i)
...
...
0
4
49
81
```

Example 3.17. *Print each member of the list ['a','f','r'].*

```
>>> for i  in ['a','f','r']:
...     print(i)
...
...
a
f
r
```

Example 3.18. *Print each letter twice from the str 'PUNE'.*

```
>>> for i  in 'PUNE':
...     print(i*2)
...
...
PP
UU
NN
EE
```

Example 3.19. *For each name in the list*

['Avinash','Rajesh','Aniket',Shankar']

print massage:

Hello, <name>, How are you?

```
>>> for i in ['Avinash','Rajesh','Aniket','Shankar']:
...     print('Hello', i,'How are you?')
...
...
Hello Avinash How are you?
Hello Rajesh How are you?
Hello Aniket How are you?
Hello Shankar How are you?
```

Example 3.20. *Create a sequence of numbers from 3 to 8, and print each item in the sequence.*

```
>>> x = range(3, 8)
>>> for n in x:
...     print(n)
...
...
3
4
5
6
7
```

Example 3.21. *Print the table of 7.*

```
>>> for i in range(1,11):
...     print(7*i,end=' ')
...
...
7 14 21 28 35 42 49 56 63 70
```

Example 3.22. *Write a python programme that print table of given number.*

```
>>> def table(n):
...     for i in range(1,11):
...         print(n*i,end=' ')
...


>>> table(14)
14 28 42 56 70 84 98 112 126 140
>>> table(29)
29 58 87 116 145 174 203 232 261 290
```

Example 3.23. *Print all integers between 1 to 100 that multiple of 3 and 7.*

```
>>> for i in range(1,100):
...     if i%3 == 0 and i%7 == 0:
...         print(i,end=' ')
...
...
21 42 63 84
```

Example 3.24. *Print all positive divisors of given number n.*

```
>>> def divisors(n):
...     for i in range(1, n + 1):
...         if n%i == 0:
...             print(i,end=' ')
...
...
>>> divisors(60)
1 2 3 4 5 6 10 12 15 20 30 60
>>> divisors(51)
```

```
1 3 17 51
>>> divisors(101)
1 101
```

Example 3.25. *Print first 10 terms of Fibonacci sequence using for loop.*

```
>>> a = 0
>>> b = 1
>>> for i in range(10):
...     a,b = b, a + b
...     print(a,end=' ')
...
...
1 1 2 3 5 8 13 21 34 55
```

Example 3.26. *Define a function that print all the integers between 1 to n, that are relatively prime to n.*

```
>>> import math
>>> def phi(n):
...     for x in range(1,n):
...         if math.gcd(n, x)== 1:
...             print(x)
...
>>>
>>> phi(10)
1
3
7
9

>>> phi(20)
1
3
7
9
11
13
```

```
17
19
```

Example 3.27. *Define Euler's phi function in python.*

```
>>> import math
>>> def phi(n):
...     i = 0
...     for i in range(1,n):
...         if math.gcd(n,i) == 1:
...             i = i + 1
...             print(i,end=' ')
...
...
>>>
>>> phi(10)
2 4 8 10
>>> phi(50)
2 4 8 10 12 14 18 20 22 24 28 30 32 34 38 40 42 44 48 50
>>> phi(31)
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
>>> phi(8)
2 4 6 8
>>> phi(20)
2 4 8 10 12 14 18 20
```

3.3 Functions

3.3.1 Calling functions: type, id

The **type()** is a inbuilt function in Python that returns type of the argument. The expression in parentheses is called the argument of the function.

For example:

```
>>> type(32)
<class 'int'>
>>> type(3.14)
<class 'float'>
>>> type('Pune')
```

```
<class 'str'>
>>> type([2,5,8])
<class 'list'>
>>> type((1,5,6,-9 ))
<class 'tuple'>
>>> type( {'a':1,'b':2,'c':3} )    #dictionary data type
<class 'dict'>
>>> type({'a', 'c', 'r', 'b', 'z', 'd'} )
<class 'set'>
```

Another inbuilt function is **id()** function that accept any object(String, Number, List etc) and is used to return the identity(unique integer) of an object. This is an integer which is unique for the given object and remains constant during its lifetime. All objects in Python has its own unique id. The id is assigned to the object when it is created.

```
>>> id(2)
1853552576
>>> # or
>>> a=2
>>> id(2)
1853552576
```

Remark 3.1. *The id is the object's memory address, and will be different for each time you run the program.*

3.3.2 Type conversion: int, float, str

Python provides inbuilt functions that convert values from one type to another. The **int** function takes any value and converts it to an integer, if it can, or criticizes otherwise:
For example:

```
>>> int('142')
142
```

int convert floating-point values to integers, it removes the fraction part:

```
>>> int(9.99999)
9
>>> int(-4.3)
-4
```

int can't convert string values like below to integers,

```
>>> int('Hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'Hello'
```

The **float** converts integers and strings to floating-point numbers, if it can:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
>>> float('Algebra')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: 'Algebra'
```

Finally, str converts its argument to a string:

```
>>> str(12)
'12'
>>> str(3.14159)
'3.14159'
```

3.3.3 Composition of functions

We have looked at the elements of a program like variables, expressions, and statements, all are in isolation. One of the most useful features of programming languages is their ability to take small building blocks and compose them.

For example, the argument of a function can be any kind of expression, including arithmetic operators:

```
>>> from math import *
>>> degrees=180
>>> x = sin(degrees/360.0 * 2 * pi)
>>> y = tan(sin(pi))
```

Almost anywhere you can put a value, you can put an arbitrary expression, with one exception: the left side of an assignment statement has to be a variable name. Any other expression on the left side is a syntax error.

```
>>> hours=36
>>> minutes = hours * 60 # right
```

```
>>> hours * 60 = minutes # wrong!
  File "<stdin>", line 1
SyntaxError: cannot assign to operator
```

3.3.4 Parameters and arguments

Some of the built-in functions we have seen require **arguments**. For example, when you call **math.sin** you pass a number as an argument. Some functions take more than one argument: **math.pow** takes two, the base and the exponent. Inside the function, the arguments are assigned to variables called **parameters**. Here is an example of a user-defined function that takes an argument:

```
>>> def print_twice(bruce):
...     print(bruce)
...     print(bruce)
...
```

This function assigns the argument to a parameter named **bruce**. When the function is called, it prints the value of the parameter (whatever it is) twice. This function works with any value that can be printed.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> print_twice(3.14)
3.14
3.14
```

Rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for **print_twice**:

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(17*4)
68
68
```

3.4 Exercise

1. Show the output from the following fragments:

   (a)

```
for i in range(5) :
    print (i * i)
```

   (b)

```
for d in [3,1,4,1,5]:
    print (d, end=" ")
```

   (c)

```
for i in range ( 4) :
    print ("Hello")
```

   (d)

```
for i in range ( 5) :
    print (i, 2 **i)
```

   (e)

```
print ("start")
for i in range (0) :
    print ("Hello")
    print ("end")
```

2. Show the sequence of numbers that would be generated by each of the following range expressions.

   a) range (5)

   b) range (3, 10)

   c) range (4, 13, 3)

   d) range (15, 5, -2)

   e) range (5, 3)

3. Show the output that would be generated by each of the following program fragments.

   a)

```
for i in range (1, 11) :
    print (i*i)
```

b)

```
for i in [1,3,5,7,9]:
    print (i, ":", i**3)
    print (i)
```

c)

```
x = 2
y = 10
for j in range (0, y, x) :
    print (j, end="")
    print (x + y)
    print ("done")
```

d)

```
ans = 0
for i in range (1, 11)
    ans = ans + i*i
    print (i)
    print (ans)
```

4. certain CS professor gives 5-point quizzes that are graded on the scale 5-A, 4-B, 3-C, 2-D, 1-F, 0-F. Write a program that accepts a quiz score as an input and prints out the corresponding grade.
5. Write a Python program to find those numbers which are divisible by 7 and multiple of 5, between 1500 and 2700 (both included).
6. Write a Python program that accepts a word from the user and reverse it.
7. Write a Python program to count the number of even and odd numbers from a series of numbers.
8. Write a Python program that prints all the numbers from 0 to 6 except 3 and 6.
9. Write a Python program which iterates the integers from 1 to 50. For multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".
10. Write a Python program that accepts a string and calculate the number of digits and letters.
11. Write a Python program to check whether an alphabet is a vowel or consonant.

12. Write a Python program to convert month name to a number of days.

13. Write a Python program to check a triangle is equilateral, isosceles or scalene.

14. Print First 10 natural numbers using while loop

15. Given a number count the total number of digits in a number.

16. Display -10 to -1 using for loop.

Chapter 4

Linear Algebra

Linear algebra is a branch of mathematics, which deals with the study of lines and planes, vector spaces and mappings that are required for linear transforms. **The numpy package** (module) is used in almost all numerical computation using Python. It is a package that provide high-performance vector, matrix and higher-dimensional data structures for Python.

The another module for matrix computation is **SymPy**. SymPy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS) while keeping the code as simple as possible in order to be comprehensible and easily extensible.

To load this SymPy module, we use the following statement:

```
>>> from sympy import *
```

To create new vectors and matrices we can use the ***matrix*** function.

Example 4.1. *Create a row vector* $u = [1, 2, 3]$

```
>>> from sympy import *
>>> Matrix([[1,2,3]])
Matrix([[1, 2, 3]])
```

Example 4.2. *Create a column vector* $v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$

```
>>> from sympy import *
>>> Matrix([[1],[2],[3]])
Matrix([
[1],
[2],
[3]])
```

Example 4.3. *For vectors* $u = \begin{bmatrix} 2 \\ 5 \\ -3 \end{bmatrix}$ *and* $v = \begin{bmatrix} 1 \\ 0 \\ -2 \end{bmatrix}$ *find,*

a) $u + v$

b) $u - v$

c) $3.u$

d) $2.u + 3.v$

```
>>> from sympy import *
>>> u=Matrix([[2],[5],[-3]])
>>> v=Matrix([[1],[0],[-2]])
>>> u+v
Matrix([
[ 3],
[ 5],
[-5]])
>>> u-v
Matrix([
[ 1],
[ 5],
[-1]])
>>> 3*u
Matrix([
[ 6],
[15],
[-9]])
>>> 2*u+3*v
Matrix([
[  7],
[ 10],
[-12]])
```

4.1 Matrix construct, eye(n), zeros(n,m) matrices

We can construct matrices in Python using SymPy module. To create matrix, the argument to the **Matrix** function.

```
>>> from sympy import *
>>> Matrix([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
Matrix([
[1, 2, 3],
```

```
[4, 5, 6],
[7, 8, 9]])
```

We can get information about the shape of an array by using the **shape**. We can display size of matrices as follows:

```
>>> from sympy import *
>>> M=Matrix([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
>>> M.shape
(3, 3)
```

The matrix M objects is of the type **sympy.matrices.** that the SymPy module provides.

```
>>> type(M)
<class 'sympy.matrices.dense.MutableDenseMatrix'>
```

For larger arrays it is inpractical to initialize the data manually, using explicit python lists. Instead we can use one of the many functions in numpy that generate arrays of different forms. Some of the more common are given below :

1. To create an identity matrix, use **eye**. eye(n) will create an $n \times n$ identity matrix.

```
>>> from sympy import *
>>> eye(3)
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> eye(4)
Matrix([
[1, 0, 0, 0],
[0, 1, 0, 0],
[0, 0, 1, 0],
[0, 0, 0, 1]])
```

2. To create a matrix of all zeros, use zeros. **zeros(n, m)** creates an $n \times m$ matrix of 0s.

```
>>> from sympy import *
>>> zeros(2,3)
Matrix([
[0, 0, 0],
```

```
[0, 0, 0]])
>>> zeros(4,3)
Matrix([
[0, 0, 0],
[0, 0, 0],
[0, 0, 0],
[0, 0, 0]])
```

3. Similarly, **ones** creates a matrix of ones.

```
>>> from sympy import *
>>> ones(2,3)
Matrix([
[1, 1, 1],
[1, 1, 1]])
>>> ones(4,3)
Matrix([
[1, 1, 1],
[1, 1, 1],
[1, 1, 1],
[1, 1, 1]])
```

4. To create diagonal matrices, use **diag**. The arguments to diag can be either numbers or matrices. A number is interpreted as a 1×1 matrix. The matrices are stacked diagonally. The remaining elements are filled with 0s.

```
>>> from sympy import *
>>> diag(1, 2, 3)
Matrix([
[1, 0, 0],
[0, 2, 0],
[0, 0, 3]])
>>> diag(-1, ones(2, 2), Matrix([5, 7, 5]))
Matrix([
[-1, 0, 0, 0],
[ 0, 1, 1, 0],
[ 0, 1, 1, 0],
[ 0, 0, 0, 5],
```

```
    [ 0, 0, 0, 7],
    [ 0, 0, 0, 5]])
```

4.2 Addition, Subtraction, Multiplication of matrices, powers and invers of a matrix

With the usual standard arithmetic operators +, -, * we can add, substract, multiply matrices respectively.

```
>>> from sympy import *
>>> # Consider matrices A and B,
>>> A=Matrix([[1,2,3],[4,5,6],[7,8,9]])
>>> B=Matrix([[4,6,-3],[-4,1,6],[0,8,-9]])
>>> v=Matrix([[1],[5],[6]])
>>> # Matrix addition
>>> A+B
Matrix([
[5,  8,  0],
[0,  6, 12],
[7, 16,  0]])
>>> # Matrix addition
>>> A-B
Matrix([
[-3, -4,  6],
[ 8,  4,  0],
[ 7,  0, 18]])
>>> # Matrix multiplication
>>> A*A
Matrix([
[ 30,  36,  42],
[ 66,  81,  96],
[102, 126, 150]])
>>> A*B
Matrix([
[-4,  32, -18],
[-4,  77, -36],
[-4, 122, -54]])
```

```
>>> A*v
Matrix([
[ 29],
[ 65],
[101]])
>>> A**3
Matrix([
[ 468,  576,  684],
[1062, 1305, 1548],
[1656, 2034, 2412]])
```

Note: If we try v*A the it will gives error as Matrix size mismatch: (3,1) * (3,3)

Inverse: Let A be any non singular matrix then there exits a matrix B such that

$$AB = BA = Identity\ matrix.$$

Matrix B is called inverse of matrix A. It is denoted by A^{-1}.

In python we can find inverse by **matrix.inv()** function.

For example

```
>>> from sympy import *
>>> # Consider matrices A and B,
>>> A=Matrix([[2,1,1],[1,2,1],[1,1,2]])
>>> B=Matrix([[1,1,1],[0,1,1],[0,0,1]])
>>> A.inv()
Matrix([
[ 3/4, -1/4, -1/4],
[-1/4,  3/4, -1/4],
[-1/4, -1/4,  3/4]])
>>> B.inv()
Matrix([
[1, -1,  0],
[0,  1, -1],
[0,  0,  1]])
```

4.3 Accessing Rows and Columns, Deleting and Inserting Rows and Columns

To get an individual row or column of a matrix, use **row** or **col**. For example, **M.row(0)** will get the first row. **M.col(-1)** will get the last column.

```
>>> from sympy import *
>>> # Consider matrices A and B,
>>> A=Matrix([[2,1,1],[1,2,1],[1,1,2]])
>>> B=Matrix([[1,1,1],[0,1,1],[0,0,1]])
>>> A.row(0)
Matrix([[2, 1, 1]])
>>> B.row(2)
Matrix([[0, 0, 1]])
>>> A.col(-1)
Matrix([
[1],
[1],
[2]])
```

We can add or delete particular row in matrix. To delete a row or column, use **row_del** or **col_del**. These operations will modify the Matrix in place.

```
>>> from sympy import *
>>> # Consider matrix A
>>> A=Matrix([[2,1,1],[1,2,1],[1,1,2]])
>>> A.row_del(1)
>>> A
Matrix([
[2, 1, 1],
[1, 1, 2]])
>>> A.col_del(-1)
>>> A
Matrix([
[2, 1],
[1, 1]])
```

To insert rows or columns, use **row_insert** or **col_insert**. These operations do not operate in place.

```
>>> from sympy import *
>>> # Consider matrix A
>>> A=Matrix([[2,1,1],[1,2,1],[1,1,2]])
>>> A=A.row_insert(1, Matrix([[0,5, 4]]))
>>> A
Matrix([
[2, 1, 1],
[0, 5, 4],
[1, 2, 1],
[1, 1, 2]])
>>> A=A.col_insert(0, Matrix([[0],[5],[6],[4]]))
>>> A
Matrix([
[0, 2, 1, 1],
[5, 0, 5, 4],
[6, 1, 2, 1],
[4, 1, 1, 2]])
```

4.4 Determinant, reduced row echelon form, nullspace, columnspace, Rank

Transpose of a matrix:

Let $A = [a_{ij}]$ be a matrix then matrix then transpose of matrix A is $A^t = [a_{ji}]$.

```
>>> from sympy import *
>>> # Consider matrix A,
>>> A=Matrix([[4,6,-3],[-4,1,6],[0,8,-9]])
>>> A
Matrix([
[ 4, 6, -3],
[-4, 1,  6],
[ 0, 8, -9]])
>>> # transpose
>>> A.T
Matrix([
```

```
[ 4, -4,  0],
[ 6,  1,  8],
[-3,  6, -9]])
```

Determinant:

We can find determinant of matrix. To compute the determinant of a matrix, use **det.**

```
>>> from sympy import *
>>> # Consider matrices A and B,
>>> A=Matrix([[1,2,3],[4,5,6],[7,8,9]])
>>> B=Matrix([[4,6,-3],[-4,1,6],[0,8,-9]])
>>> A.det()
0
>>> B.det()
-348
```

Reduced row echelon form:

To put a matrix into reduced row echelon form, use **rref.** rref returns a tuple of two elements. The first is the reduced row echelon form, and the second is a tuple of indices of the pivot columns.

```
>>> from sympy import *
>>> A=Matrix([[1,2,3],[4,5,6],[7,8,9]])
>>> B=Matrix([[4,6,-3],[-4,1,6],[0,8,-9]])
>>> A.rref()
(Matrix([
[1, 0, -1],
[0, 1,  2],
[0, 0,  0]]), (0, 1))
>>> B.rref()
(Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]]), (0, 1, 2))
```

Nullspace:

To find the nullspace of a matrix, use **nullspace.** nullspace returns a list of column vectors that span the nullspace of the matrix.

```
>>> from sympy import *
>>> A=Matrix([[1,2,3],[4,5,6],[7,8,9]])
```

```
>>> B=Matrix([[4,6,-3],[-4,1,6],[0,8,-9]])
>>> A.nullspace()
[Matrix([
[ 1],
[-2],
[ 1]])]
>>> B.nullspace()
[]
```

Columnspace:

To find the columnspace of a matrix, use **columnspace**. columnspace returns a list of column vectors that span the columnspace of the matrix.

```
>>> from sympy import *
>>> A=Matrix([[1,2,3],[4,5,6],[7,8,9]])
>>> B=Matrix([[4,6,-3],[-4,1,6],[0,8,-9]])
>>> A.columnspace()
[Matrix([
[1],
[4],
[7]]), Matrix([
[2],
[5],
[8]])]
>>> B.columnspace()
[Matrix([
[ 4],
[-4],
[ 0]]), Matrix([
[6],
[1],
[8]]), Matrix([
[-3],
[ 6],
[-9]])]
```

Rank:

Rank function returns the rank of a matrix.

```
>>> from sympy import *
>>> A=Matrix([[1,2,3],[4,5,6],[7,8,9]])
>>> B=Matrix([[4,6,-3],[-4,1,6],[0,8,-9]])
>>> A.rank()
2
>>> B.rank()
3
```

4.5 Solving systems of linear equations (Gauss Elimination Method, Gauss Jordan Method, LU- decomposition Method)

4.5.1 Gauss Elimination Method

To solve system of N linear equations with M variables, here we use python built-in function

linsolve()

The number of solutions may be zero, unique or infinitly many. If there is zero solutions then linsolve() gives a ValueError, whereas infinite solutions are represented parametrically in terms of the given symbols. For unique solution a FiniteSet of ordered tuples is returned. Consider the following examples:

Example 4.4. *Solve the following system of equations:*

$$3x + 2y - z = 3 \quad 2x - 2y + 4z = 6 \quad 2x - y + 2z = 9$$

```
>>> from sympy import *
>>> x, y, z = symbols("x, y, z")
>>> A = Matrix([[3, 2, -1], [2, -2, 4], [2, -1, 2]])
>>> b = Matrix([3, 6, 9])
>>> linsolve((A, b), [x, y, z])
FiniteSet((6, -11, -7))
```

Example 4.5. *Solve the following system of equations:*

$$7x + 6y - 8z = 3 \quad 7x - 2y + 2z = 0 \quad 6x - y - 2z = 9$$

```
>>> from sympy import *
>>> x, y, z = symbols("x, y, z")
>>> A = Matrix([[7, 6, -8], [7, -2,2 ], [6, -1,-2]])
```

```
>>> b = Matrix([3, 0, 9])
>>> linsolve((A, b), [x, y, z])
FiniteSet((-9/79, -276/79, -489/158))
```

Example 4.6. *Solve the following system of equations:*

$$x + 2y + 3z = 3 \quad 4x + 5y + 6z = 6 \quad 7x + 8y + 9z = 9$$

```
>>> from sympy import *
>>> x, y, z = symbols("x, y, z")
>>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> b = Matrix([3, 6, 9])
>>> linsolve((A, b), [x, y, z])
FiniteSet((z - 1, 2 - 2*z, z))
```

4.5.2 Gauss Jordan Method

We can solve sysytem $Ax = B$ using Gauss Jordan elimination. There may be zero, one, or infinite solutions. If one solution exists, it will be returned. If infinite solutions exist, it will be returned parametrically. If no solutions exist, It will throw ValueError.

Syntax:

$$A.gauss\_jordan\_solve(B, freevar = False)$$

Inputs:

B : Matrix

The right hand side of the equation to be solved for. Must have the same number of rows as matrix A.

freevar : List

If the system is underdetermined (e.g. A has more columns than rows), infinite solutions are possible, in terms of arbitrary values of free variables. Then the index of the free variables in the solutions (column Matrix) will be returned by freevar, if the flag freevar is set to True.

This function retuens:

x : Matrix

The matrix that will satisfy Ax = B. Will have as many rows as matrix A has columns, and as many columns as matrix B.

params : Matrix

If the system is underdetermined (e.g. A has more columns than rows), infinite solutions are possible, in terms of arbitrary parameters. These arbitrary parameters are returned as params Matrix.

Here are some examples:

Example 4.7. *Solve the following system:*

$$2x + y = 5; \quad x + 2y = 7$$

```
>>> from sympy import *
>>> A = Matrix([[2, 1], [1, 2]])
>>> B = Matrix([5, 7])
>>> A.gauss_jordan_solve(B)
(Matrix([
[1],
[3]]), Matrix(0, 1, []))
```

Example 4.8. *Solve the following system:*

$$x + 2y + 3z = 3; \quad 4x + 5y + 6z = 6 \quad 7x + 8y + 10z = 9$$

```
>>> from sympy import *
>>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 10]])
>>> B = Matrix([3, 6, 9])
>>> sol, params = A.gauss_jordan_solve(B)
>>> sol
Matrix([
[-1],
[ 2],
[ 0]])
>>> params
Matrix(0, 1, [])
```

4.5.3 LU- decomposition Method

In this section, we will see how to solve system $AX = B$ usinf LU Decomposition. Here we convert matrix A as the product of two simpler matrices. We will write

$$A = LU$$

where L is lower triangular matrix and U is upper triangular matrix.
$A = LU$ is called an LU decomposition of A.
We can solve sysytem $Ax = B$ using LU Decomposition. Here one is very important requrement is that coefficiant matrix A should be invertible.

Syntax:

solve_linear_system_LU(matrix, syms)

It accept the augmented matrix as **matrix** and return a solution that keyed to the symbols order given in of **syms**. Cosider the following examples:

Example 4.9. *Solve the following sysytem using LU Decomposition method.*

$$\begin{pmatrix} 6 & 18 & 3 \\ 2 & 12 & 1 \\ 4 & 15 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 3 \\ 19 \\ 0 \end{pmatrix}$$

```
>>> # First we import sympy.
>>> from sympy import *
>>> # We define variables £ x,y,z. £
>>> from sympy.abc import x, y, z
>>> # We need to give agumented matrix AB .
>>> AB = Matrix([[6,18, 3,3], [2, 12, 1,19], [4, 15, 1,0]])
>>> solve_linear_system_LU(AB,[x,y,z])
{x: -14, y: 3, z: 11}
```

Example 4.10. *Solve the following sysytem using LU Decomposition method.*

$$\begin{pmatrix} 1 & 2 & 2 \\ 2 & 1 & 2 \\ 2 & 2 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 7 \\ 8 \\ 6 \end{pmatrix}$$

```
>>> # First we import sympy.
>>> from sympy import *
>>> # We define variables £ x,y,z. £
>>> from sympy.abc import x, y, z
>>> # We need to give agumented matrix AB .
>>> AB = Matrix([[1,2, 2,7], [2, 1, 2,8], [2, 2, 1,6]])
>>> solve_linear_system_LU(AB,[x,y,z])
{x: 7/5, y: 2/5, z: 12/5}
```

4.6 Eigenvalues, Eigenvectors, and Diagonalization

4.6.1 Eigenvalues

Definition:

Let A be a square matrix. A non-zero vector v is an **eigenvector** for A with **eigenvalue** λ if

$$Ax = \lambda v$$

To find the eigenvalues of a matrix, we use **eigenvals**. It returns output as eigenvalue:algebraic multiplicity pairs.
For Example:

Example 4.11. *Find the eigenvalues for the matrix*

$$A = \begin{pmatrix} 1 & 2 & 2 \\ 2 & 1 & 2 \\ 2 & 2 & 1 \end{pmatrix}$$

```
>>> from sympy import *
>>> A = Matrix([[1, 2, 2], [2, 1, 2], [2, 2, 1]])
>>> A.eigenvals()
{5: 1, -1: 2}
```

Example 4.12. *Find the eigenvalues for the matrix*

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

```
>>> from sympy import *
>>> A = Matrix([[1, 1, 1], [0, 1, 1], [0, 0, 1]])
>>> A.eigenvals()
{1: 3}
```

4.6.2 Eigenvectors

To find the eigenvectors of a matrix, we use **eigenvects**. It returns a list of tuples of the form (eigenvalue:algebraic multiplicity, [eigenvectors])
For Example:

Example 4.13. *Find the eigenvectors for the matrix*

$$A = \begin{pmatrix} 1 & 2 & 2 \\ 2 & 1 & 2 \\ 2 & 2 & 1 \end{pmatrix}$$

```
>>> from sympy import *
>>> A = Matrix([[1, 2, 2], [2, 1, 2], [2, 2, 1]])
>>> A.eigenvects()
[(-1, 2, [Matrix([
[-1],
[ 1],
```

```
[ 0]]), Matrix([
[-1],
[ 0],
[ 1]])]), (5, 1, [Matrix([
[1],
[1],
[1]])])]
```

Example 4.14. *Find the eigenvectors for the matrix*

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

```
>>> from sympy import *
>>> A = Matrix([[1, 1, 1], [0, 1, 1], [0, 0, 1]])
>>> A.eigenvects()
[(1, 3, [Matrix([
[1],
[0],
[0]])])]
```

4.6.3 Diagonalization

Defination:

An $n \times n$ matrix A is **diagonalizable** if there exists invertible matrix P such that

$$P^{-1}AP = Diagonal\ matrix$$

In Python we use **diagonalize()** for diagonalizatiuon. It return matrix P and D if matrix A is diagonalizable, otherwise it will return rerror.

Example 4.15. *Find the P and D for the matrix*

$$A = \begin{pmatrix} 1 & 2 & 2 \\ 2 & 1 & 2 \\ 2 & 2 & 1 \end{pmatrix}$$

if A is diagonalization.

```
>>> from sympy import *
>>> A = Matrix([[1, 2, 2], [2, 1, 2], [2, 2, 1]])
>>> P, D= A.diagonalize()
```

```
>>> P
Matrix([
[-1, -1, 1],
[ 1,  0, 1],
[ 0,  1, 1]])
>>> D
Matrix([
[-1,  0, 0],
[ 0, -1, 0],
[ 0,  0, 5]])
```

Example 4.16. *Find the matrices P and D for the matrix*

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

if matrix A is diagonalizable.

```
>>> from sympy import *
>>> A = Matrix([[1, 1, 1], [0, 1, 1], [0, 0, 1]])
>>> A.diagonalize()
# It will display error as Matrix is not diagonalizable.
```

Note: There is one more way to decide matrix is diagonalizable or not. We can use **is_diagonalizable** function.

Example 4.17. *Using Python decide whether matrix*

$$A = \begin{pmatrix} 2 & 1 & 1 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{pmatrix}$$

is diagonalizable or not.

To dicide we use **is_diagonalizable** function. First import **sympy**.

```
>>> from sympy import Matrix
>>> M = Matrix([[1, 2, 0], [0, 3, 0], [ 2, -4, 2 ]])
>>> M.is_diagonalizable()
True
```

Example 4.18. *Using Python decide whether matrix*

$$A = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$$

is diagonalizable or not.

To dicide we use **is_diagonalizable** function. First import **sympy**.

```
>>> from sympy import Matrix
>>> A = Matrix([[0, 1], [0, 0]])
>>> A.is_diagonalizable()
False
```

4.7 Excersise:

1. Construct the following matrices:
   (a) Identity matrix of order 5
   (b) Zero matrix of order 5×6
   (c) ones matrix of order 5×4

2. For the folowing matrices

$$A = \begin{pmatrix} 4 & 2 & 4 \\ 4 & -1 & 1 \\ 2 & 4 & 2 \end{pmatrix} \quad B = \begin{pmatrix} 5 & 2 & 3 \\ 3 & -7 & 5 \\ 3 & 1 & -1 \end{pmatrix}$$

   find,
   (A) $A + B$
   (B)$A - B$
   (c) A^{-1}
   (d) $B * A$
   (e) $B^{-1}AB$

3. For matrix

$$C = \begin{pmatrix} -5 & 2 & 3 \\ 3 & -7 & 5 \\ -3 & 10 & -11 \end{pmatrix}$$

   (a) Delete 2nd column.
   (b) Add Row $R = [2, 3]$
   (c) Delet last row

4. For matrix

$$D = \begin{pmatrix} 10 & 2 & 3 \\ 12 & -7 & 15 \\ -15 & 10 & -11 \end{pmatrix}$$

   (a) Find its transpose
   (b) Find its Determinant
   (c) Find its invers if exists
   (d) Reduce the matrix to row echolon form.

(e) Find its rank

(f) Find nullspace

(g) Find columnspace

5. Solve the following system of linear equations using Gauss Elimination Method, Gauss Jordan Method, LU- decomposition Method.

(a)

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 4 \\ 2 & 5 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 6 \\ 7 \\ 8 \end{pmatrix}$$

(b)

$$\begin{pmatrix} -8 & 2 & -3 \\ 2 & 10 & 14 \\ -2 & -5 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 8 \\ 7 \\ -2 \end{pmatrix}$$

(c)

$$\begin{pmatrix} 5 & 2 & -2 \\ 7 & -3 & 0 \\ 0 & 6 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 8 \\ 17 \\ -12 \end{pmatrix}$$

6. Find the eigenvectors and eigenvectors for the matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 3 & -1 & 1 \\ 1 & 1 & -1 \end{pmatrix}$$

7. Find the eigenvectors and eigenvectors for the matrix

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

8. Find the matrices P and D for the matrix

$$A = \begin{pmatrix} 1 & -6 & 3 \\ 3 & -1 & 1 \\ -1 & 9 & -1 \end{pmatrix}$$

if matrix A is diagonalizable.

9. Find the matrices P and D for the matrix

$$A = \begin{pmatrix} 1 & -1 & 1 \\ -1 & 1 & -1 \\ 1 & -1 & 1 \end{pmatrix}$$

if matrix A is diagonalizable.

Chapter 5

Numerical methods in Python

A polynomial equation of the form

$$f(x) = Pn(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + ... + a_{n-1}x + a_n = 0$$

is called an **algebraic equation**.

An equation which contains polynomials, exponential functions, logarithmic functions, trigonometric functions etc. is called a **transcendental equation**.

For example,

$$5x^3 + 5x^2 - 4x - 1 = 0, \quad x^3 - 3x^2 + 5 = 0, \quad x^2 - 3x + 1 = 0$$

are algebraic (polynomial) equations, and

$$xe^{2x} - 1 = 0, \quad cosx - xe^x = 0, \quad tanx = x$$

are transcendental equations.

5.1 Roots of Equations

For analyse function $f(x)$, we need to find the roots of $f(x)$. i.e. determine the values of x for which $f(x) = 0$. A number α, for which $f(\alpha) = 0$ is called a root of the equation $f(x) = 0$, or a **zero** of $f(x)$. The roots of equations may be real or complex. The complex roots are rarely computed, because they have less significance. We will concentrate on finding the real roots of equations.

In general, an equation may have any number of (real) roots, or no roots at all.

For example:

(1) Function $f(x) = \sin x - x$ has a single root $x = 0$.

(2) Function $g(x) = x^2 + 1$ has no (real) root.

(3) Function $h(x) = cos(x) - 1$ has an infinite number of roots i.e. $x = 2n\pi, \quad n \in \mathbb{Z}$.

Simple root:. A number α is a simple root of $f(x) = 0$, if $f(\alpha) = 0$ and $f'(\alpha) \neq 0$. Then, we can write $f(x)$ as

$$f(x) = (x - \alpha)g(x), g(\alpha) \neq 0$$

For example, $x = 2$ is simple root for $f(x) \equiv x^2 - 5x + 6$.

Multiple root A number α is a multiple root, of multiplicity m, of $f(x) = 0$, if

$$f(\alpha) = 0, f'(\alpha) = 0, ..., f^{(m-1)}(\alpha) = 0, and f^{(m)}(\alpha) \neq 0.$$

Then, we can write $f(x)$ as

$$f(x) = (x-\alpha)^m g(x), g(\alpha) \neq 0.$$

For example, $x = 2$ is multiple root of order 2 of an equation $f(x) = (x-2)^2(x+2)$.

Finding root becomes easier if we restrict the domain of the root. It is convenient that first we fix the boundaries, where root lies. There are many methods of finding appropriate root. i.e. Bisection method, Nowton Raphson method, Secent method , Regula Falsi method (False position method), Muller's method etc. Some of them are discussed with Python in this chapter.

A fundamental idea of solving nonlinear equations is to construct a series of linear equations and the solutions of these linear equations bring us closer and closer to the solution of the nonlinear equation. This construction is used in Newton's method and the secant method.

5.2 Newton-Raphson Method

Newton-Raphson's method is a very Famous, fastest and widely used method for solving nonlinear algebraic equations. The only drawback of this method is that it involves the derivative of f. Therefore, the Newton–Raphson method is usable only in problems where $f'(x)$ can be computed. This methods require us to guess at a solution first. Here, this guess is called x_0.

x_0 should be close to the actual root. Being close to the root helps, because in general that is enough to make sure that the method converges to a good result.

Newton–Raphson formula:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}. \quad i = 0, 1, 2, ... \tag{5.1}$$

We start with an initial approximation x_0, then one by one estimate $x_1, x_2, ...$ until the convergence criterion is reached. The computation in (1.1) is repeated until $f(x_n)$ is close enough to zero. More precisely, we test if $|f(x_n)| < \epsilon$, with ϵ being a small number. A smaller value of eps will produce a more accurate solution.

The algorithm for Newton's method for numerically approximating a root of a function can be summarized as follows:

1. Given a function $f(x)$ of a variable x and a way to compute both $f(x_i)$ and its derivative $f'(x_i)$ at a given point x_i. Let x_0 be a guess for the root of $f(x) = 0$.

2. Compute next approximation x_1 for the root as $x_1 = -\frac{f(x_0)}{f'(x_0)}$.

3. Repeat step 2 until a sufficiently precise value is reached.

Before applying main python code for NR Method, we need to implement functions f and its derivative g.

Let $f(x) = x^3 + 3x - 2$

so $f'(x) = g = 3x^2 + 3$ then implimentation as follows:

```
def f(x):
    return x**3+3*x-2
def g(x):
    return 3x**2+3
```

In this python program, x_0 is initial guess, e is tolerable error, $f(x)$ is non-linear function whose root is being obtained using Newton Raphson method, $g(x)$ is the derivative of $f(x)$ and N is the number of maximum steps for the formula. After N steps, if tolerable error doesn't attend then will get output *not convergent* and then we can increase the value of N.

Python Code:

```
def newtonRaphson(f,g,x0,e,N):
    x0 = float(x0)
    e = float(e)
    N = int(N)
    step = 1
    flag = 1
    condition = True
    while condition:
        if g(x0) == 0.0:
            print('Divide by zero error!')
            break
        x1 = x0 - f(x0)/g(x0)
        print('Iteration-%d, x1 = %0.6f and f(x1) = %0.6f' % (step, x1, f(x1)))
        x0 = x1
        step = step + 1
        if step > N:
            flag = 0
            break
        condition = abs(f(x1)) > e
```

```
    if flag==1:
        print('\nRequired root is: %0.8f' % x1)
    else:
        print('\nNot Convergent.')
```

Let us apply above programme to find the approximat root of $f(x) = x^3 - 5x + 1 = 0$ with $x_0 = 0.5$ and tolerable error $e = 0.00001$.

Impliment the function $f(x)$ and its derivative $g(x)$ as follows:

```
def f(x):
    return x**3-5*x+1
def g(x):
    return 3*x**2-5
```

Now apply **newtonRaphson** as follows:

```
newtonRaphson(f,g,0.5,0.00001,100)
Iteration-1, x1 = 0.176471 and f(x1) = 0.123143
Iteration-2, x1 = 0.201568 and f(x1) = 0.000349
Iteration-3, x1 = 0.201640 and f(x1) = 0.000000

Required root is: 0.20163968
```

Consider another example:

Example 5.1. *A root of $f(x) = x^3 - 10x^2 + 5 = 0$ lies close to $x = 0.7$. Compute this root with the Newton–Raphson method.*

For $f(x) = x^3 - 10x^2 + 5 = 0$ its derivative will be $g(x) = 3x^2 - 20x$. First impliments the $f(x)$ and its derivative $g(x)$ as follows:

```
def f(x):
    return x**3-10*x**2+5
def g(x):
    return 3*x**2-20*x
```

Complete code for **newtonRaphson** as follows:

```
>>> def newtonRaphson(f,g,x0,e,N):
...     x0 = float(x0)
...     e = float(e)
...     N = int(N)
```

```
...     step = 1
...     flag = 1
...     condition = True
...     while condition:
...         if g(x0) == 0.0:
...             print('Divide by zero error!')
...             break
...         x1 = x0 - f(x0)/g(x0)
...         print('Iteration-%d, x1 = %0.6f and f(x1) = %0.6f' % (step, x1, f(x1)))
...         x0 = x1
...         step = step + 1
...         if step > N:
...             flag = 0
...             break
...         condition = abs(f(x1)) > e
...     if flag==1:
...         print('\nRequired root is: %0.8f' % x1)
...     else:
...         print('\nNot Convergent.')
...
>>> def f(x):
...     return x**3 - 10*x**2 + 5
...
>>> # Defining derivative of function
>>> def g(x):
...     return 3*x**2 - 20*x
...
>>> newtonRaphson(f,g,0.7,0.00001,100)
Iteration-1, x1 = 0.735355 and f(x1) = -0.009831
Iteration-2, x1 = 0.734604 and f(x1) = -0.000004

Required root is: 0.73460384
```

5.3 False Position (Regula Falsi) Mehtod

In Newton's method, finding the derivative $f'(x)$ is problematic. The idea of the False Position (Regula Falsi) Mehtod is to think as in Newton's method, but instead of using $f'(x)$, we approximate this derivative by a finite difference or the secant, i.e., the slope of the straight line that goes through the two most recent approximations x_i and x_{i-1}. This slope will be

$$\frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}$$

Inserting this expression for $f'(x)$ in Newton's method simply gives us

$$x_{i+1} = x_i - \frac{f(x_i)}{\frac{f(x_i)-f(x_{i-1})}{x_i - x_{i-1}}}$$

This implies,

$$x_{i+1} = x_i - f(x_i)\frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})}$$

we choose starting points x_0 and x_1 so that,

$$f(x_0)f(x_1) < 0$$

and used x_0 and x_1 to compute x_2. Once we have x_2, we similarly use x_2 and one of x_0 or x_1 so that

$$f(x_2)f(x_k) < 0$$

Where k is 1 or 0, so that above criteria holds, and used x_2 and x_k to compute x_3. Continue this procedure until desire accuracy.

In this python program, x_0 and x_1 are boundaries of domain, e is tolerable error, $f(x)$ is non-linear function whose root is being obtained using Newton Raphson method.

Python Code:

```
def falsePosition(f,x0,x1,e):
    x0 = float(x0)
    x1 = float(x1)
    e = float(e)
    if f(x0) * f(x1) > 0.0:
        print('Given guess values do not bracket the root.')
        print('Try Again with different guess values.')
    else:
        step = 1
        condition = True
        while condition:
            x2 = x0 - (x1-x0) * f(x0)/( f(x1) - f(x0) )
```

```
        print('Iteration %d, x2 = %0.6f and f(x2) = %0.6f' % (step, x2, f(x2)))
        if f(x0) * f(x2) < 0:
            x1 = x2
        else:
            x0 = x2
        step = step + 1
        condition = abs(f(x2)) > e
    print('\nRequired root is: %0.8f' % x2)
```

Let us try above code for $f(x) = x^3 - 5x - 9$ in interval $[2, 4]$ within $e = 0.00001$.
First impliment function $f(x) = x^3 - 5x - 9$ as follows:

```
def f(x):
    return x**3-5*x-9
```

Apply **falsePosition**,

```
>>> falsePosition(f,2,3,0.00001)
```

Iteration 1, x2 = 2.785714 and f(x2) = -1.310860 Iteration 2, x2 = 2.850875 and f(x2) = -0.083923
Iteration 3, x2 = 2.854933 and f(x2) = -0.005125 Iteration 4, x2 = 2.855180 and f(x2) = -0.000312
Iteration 5, x2 = 2.855196 and f(x2) = -0.000019 Iteration 6, x2 = 2.855196 and f(x2) = -0.000001
Required root is: 2.85519648

Example 5.2. *Obtain these roots correct to three decimal places of the equation* $x^3 - 3x + 1 = 0$ *using the method of false position in interval* $[0, 1]$ *with tolerable error* $e = 0.00001$.

Let $f(x) = x^3 - 3x + 1 = 0$ be a given equation.
Complete code for **falsePosition** as follows:

```
>>> def falsePosition(f,x0,x1,e):
...     x0 = float(x0)
...     x1 = float(x1)
...     e = float(e)
...     if f(x0) * f(x1) > 0.0:
...         print('Given guess values do not bracket the root.')
...         print('Try Again with different guess values.')
...     else:
...         step = 1
...         condition = True
...         while condition:
```

```
...             x2 = x0 - (x1-x0) * f(x0)/( f(x1) - f(x0) )
...             print('Iteration %d, x2 = %0.6f and f(x2) = %0.6f' % (step, x2, f(x2)))
...             if f(x0) * f(x2) < 0:
...                 x1 = x2
...             else:
...                 x0 = x2
...                 step = step + 1
...         condition = abs(f(x2)) > e
...     print('\nRequired root is: %0.8f' % x2)
...
>>> def f(x):
...     return x**3-3*x+1
...

>>> falsePosition(f,0,1,0.00001)
Iteration 1, x2 = 0.500000 and f(x2) = -0.375000
Iteration 2, x2 = 0.363636 and f(x2) = -0.042825
Iteration 3, x2 = 0.348703 and f(x2) = -0.003709
Iteration 4, x2 = 0.347414 and f(x2) = -0.000312
Iteration 5, x2 = 0.347306 and f(x2) = -0.000026
Iteration 6, x2 = 0.347297 and f(x2) = -0.000002

Required root is: 0.34729718
```

5.4 Numerical Integration

The problem of numerical integration is to find an approximate value of the integral

$$I = \int_a^b f(x)dx$$

Many integrals are difficult to solve by pen and paper, so we computerized the solution approach. Integration also demonstrates the difference between exact mathematics by pen and paper and numerical mathematics on a computer. We will focus on how to write Python code for numerical integrations. Calculating an integral is traditionally done by

$$\int_a^b f(x)dx = F(b) - F(a)$$

where

$$f(x) = \frac{dF}{dx}$$

Most numerical methods for computing this integral split up the original integral into a sum of several integrals that covers the original integration in the interval [a, b]. We will discuss trapezoidal rule and Simpson's rules in Python.

5.4.1 Trapezoidal Rule

Trapezoidal Rule is used for approximating the definite integral. it uses the linear approximations of the functions $f(x)$. Let f be continuous on $[a, b]$, then trapezoidal rule is given by,

$$\int_a^b f(x) = (b-a)\frac{f(a)+f(b)}{2}$$

This gives approximate integral of $f(x)$ on $[a, b]$.
Dividing interval $[a, b]$ into n equal subintervals gives more accurate integral. Let h be a size of each subinterval. Then $h = (b-a)/n$ Combining integral over $[a, b]$ gives composite trapezoidal rule. Composite trapezoidal rule is given by,

$$\int_a^b f(x) = \frac{h}{2}(f(a) + 2f(a+h) + 2f(a+2h) + ... + 2f(a+(n-1)h) + f(b))$$

We write a Python function **trapezoidal** that estimate definite integral by trapezoidal rule.
Define a function that takes an arguments function as **f** , endpoints of interval of integration as **a** and **b**, number of subintervals as **n** etc.

Python Code:

```
def trapezoidal(f, a, b, n):
    h = float(b-a)/n
    result = 0.5*f(a) + 0.5*f(b)
    for i in range(1, n):
        result += f(a + i*h)
    result *= h
    return result
```

Here are some examples:

Example 5.3. *Approximate the area beneath $y = \sin x$ on the interval $[o, \pi]$ using the Trapezoidal Rule with $n = 4$ trapezoids.*

First we need to define function in python as follows:

```
def f(x):
    return sin(x)
```

That results in whenever we call $f(a)$, it returns $\sin(a)$.
Complete code as follows:

```
def trapezoidal(f, a, b, n):
    h = float(b-a)/n
    result = 0.5*f(a) + 0.5*f(b)
    for i in range(1, n):
        result += f(a + i*h)
    result *= h
    return result
def f(x):
    return sin(x)
```

Call the function:

```
from math import *  # pi, sin are involves
trapezoidal( f , 0, pi, 4 )
1.89611889793704
```

Note: Actual value of integration is 2.
If we increase the number of subintervals, then we get more accurate answers.
For example,

```
trapezoidal( f , 0, pi, 100 )
1.9998355038874436

trapezoidal( f , 0, pi, 10000 )
1.9999999835506606
```

The approximate area between the curve and the $X-axis$ is the sum of the four trapezoids. This is a trapezoidal approximation, not a Reimann sum approximation. Reimann sum refers only to an approximation with rectangles.

Example 5.4. *Approximate the area beneath* $y = x^3 - 5x - 9$ *on the interval* $[0, 3]$ *using the Trapezoidal Rule with* $n = 5,100$ *trapezoids.*

Complete Code as follows:

```
>>> def trapezoidal(f, a, b, n):
...     h = float(b-a)/n
...     result = 0.5*f(a) + 0.5*f(b)
...     for i in range(1, n):
...         result += f(a + i*h)
...     result *= h
```

```
...     return result
...


>>> from math import *
>>> def f(x):
...     return sin(x)
...
>>> trapezoidal( f, 0, 3, 5 )
1.9299814248929247
>>> trapezoidal( f, 0, 3, 100 )
1.989843244924411
```

5.4.2 Simpson's 1/3rd Rule

Another area approximating tool is Simpson's Rule. Geometrically, it creates tiny parabolas to wrap closer around the function we're approximating. The formula is similar to the Trapezoidal Rule, with a small difference i.e., in this formula we can only use an even number of subintervals.
Let f be continuous on $[a, b]$ and $h = \frac{b-a}{2}$ then Simpson's Rule is given by

$$\int_a^b f(x) = \frac{h}{3}(f(a) + 4f(a+h) + f(b))$$

This gives approximate integral of $f(x)$ on $[a, b]$.
Dividing interval $[a, b]$ into n even equal subintervals gives more accurate integral. Let h be a size of each subinterval. Then $h = (b-a)/n$ Combining integral over $[a, b]$ gives composite trapezoidal rule. Composite Simpson's 1/3rd rule is given by,

$$\int_a^b f(x) = \frac{h}{3}(f(a) + 4f(a+h) + 2f(a+2h) + 4f(a+3h) + 2f(a+4h) + ... + f(b))$$

We write a Python function **simpson** that estimate definite integral by Simpson's rule.
Define a function that takes an arguments function as **f** , endpoints of interval of integration as **a** and **b**, number of subintervals as **n** etc.

Python Code:

```
def simpson13(f, a, b, n):
    h = float(b-a)/n
    result = f(a) + f(b)
    for i in range(1,n):
        k = a + i*h
        if i%2 == 0:
```

```
            result = result + 2 * f(k)
        else:
            result = result + 4 * f(k)
    result *= h/3
    return result
```

Here are some examples:

Example 5.5. *Approximate the area beneath $y = \sin x$ on the interval $[o, \pi]$ using the Simpson's Rule with $n = 6$.*

Complete code as follows:

```
>>> def simpson13(f, a, b, n):
...     h = float(b-a)/n
...     result = f(a) + f(b)
...     for i in range(1,n):
...         k = a + i*h
...         if i%2 == 0:
...             result = result + 2 * f(k)
...         else:
...             result = result + 4 * f(k)
...     result *= h/3
...     return result
...
>>>
>>> def f(x):
...     return sin(x)
...
>>> from math import *   # pi, sin are involves
>>> simpson13( f, 0, pi, 6 )
2.0008631896735363
```

Note: Actual value of integration is 2.

If we increase the number of subintervals, then we get more accurate answers.

Example 5.6. *Find the approximate value of*

$$I = \int_0^1 \frac{dx}{1+x}$$

using the Simplson's 1/3 rd rule with n=6.

Complete code as follows:

```
>>> def simpson13(f, a, b, n):
...     h = float(b-a)/n
...     result = f(a) + f(b)
...     for i in range(1,n):
...         k = a + i*h
...         if i%2 == 0:
...             result = result + 2 * f(k)
...         else:
...             result = result + 4 * f(k)
...     result *= h/3
...     return result
...
...
>>> def f(x):
...     return 1/(1+x)
...
...
>>> simpson13( f, 0, 1, 6 )
0.6931697931697931
```

5.4.3 Simpson's 3/8th Rule

To derive the Simpson's 1/3 rule, we approximated $f(x)$ by a quadratic polynomial. To derive the Simpson's 3/8 rule, we approximate $f(x)$ by a cubic polynomial. For interpolating by a cubic polynomial, we require four nodal points. Hence, we subdivide the given interval $[a, b]$ into 3 equal parts so that we obtain four nodal points. Let $h = (b–a)/3$. In this formula we can only use three times number of subintervals.

Let f be continuous on $[a, b]$ and $h = \frac{b-a}{3}$ then Simpson's Rule is given by

$$\int_a^b f(x) = \frac{3h}{8}\left(f(a) + 3f(a+h) + 3f(a+2h) + f(b)\right)$$

This gives approximate integral of $f(x)$ on $[a, b]$.

Dividing interval $[a, b]$ into n equal subintervals gives more accurate integral. Here n should be multiple of three. Let h be a size of each subinterval. Then $h = (b - a)/n$ Combining integral over $[a, b]$ gives composite Simpson's 3/8th rule. Composite Simpson's 3/8th rule is given by,

$$\int_a^b f(x) = \frac{3h}{8}(f(a) + 3f(a+h) + 3f(a+2h) + 2f(a+3h) + 3f(a+4h)... + f(b))$$

We write a Python function **simpson** that estimate definite integral by Simpson's3/8th rule. Define a function that takes an arguments function as **f** , endpoints of interval of integration as **a** and **b**, number of subintervals as **n** etc.

Python Code:

```
def simpson38(f, a, b, n):
    h = float(b-a)/n
    result = f(a) + f(b)
    for i in range(1,n):
        k = a + i*h
        if i%2 == 0:
            result = result + 2 * f(k)
        else:
            result = result + 3 * f(k)
    result *= (3*h)/8
    return result
```

Here are some examples:

Example 5.7. *Approximate the area beneath $y = \sin x$ on the interval $[o, \pi]$ using the Simpson's 3/8th Rule with $n = 6$.*

Complete code as follows:

```
>>> def simpson38(f, a, b, n):
...     h = float(b-a)/n
...     result = f(a) + f(b)
...     for i in range(1,n):
...         k = a + i*h
...         if i%2 == 0:
...             result = result + 2 * f(k)
...         else:
...             result = result + 3 * f(k)
...     result *= (3*h)/8
...     return result
...
...
>>>
>>> def f(x):
```

```
...     return sin(x)
...
>>>
>>> from math import *   # pi, sin are involves
>>> simpson38( f, 0, pi, 6 )
1.8582720066840042
>>>
```

Example 5.8. *Find the approximate value of*

$$I = \int_0^1 \frac{dx}{1+x^2}$$

using the Simplson's 3/8th rule with n=6.

Complete code as follows:

```
>>> def simpson38(f, a, b, n):
...     h = float(b-a)/n
...     result = f(a) + f(b)
...     for i in range(1,n):
...         k = a + i*h
...         if i%2 == 0:
...             result = result + 2 * f(k)
...         else:
...             result = result + 3 * f(k)
...     result *= (3*h)/8
...     return result
...
>>>
>>>
>>> def f(x):
...     return 1/(1+x**2)
...
>>>
>>> simpson38( f, 0, 1, 6 )
0.7358766316758121
```

5.5 Excercise

1. Using Newton-Raphson method solve $xlog_{10}x = 12.34$ with $x_0 = 10$.
2. Using Newton-Raphson method find an approximation value of $\sqrt{5}$ to ten decimal places.
3. Using Newton-Raphson method approximate the solution to the equation

$$x = \cos(x)$$

4. Find approximate root of $f(x) = xe^x - cosx$ in the interval (0, 1) by false position method.
5. Find approximate root of $f(x) = tanx - 2x$ in the interval (1.1, 1.2) by false position method.
6. Approximate the area beneath $f(x) = x^2 + 1$ on the interval $[0, 3]$ using the Trapezoidal Rule with $n = 5$ trapezoids.
7. Evaluate

$$\int_{1/2}^{1} \frac{dx}{x}$$

by trapozoidal rule, dividing the range into four equal parts.

8. Evaluate

$$\int_0^1 x^2 dx$$

by trapozoidal rule, dividing the range into four equal parts.

9. Using the trapozoidal rule, evaluate

$$\int_1^6 sin(x)dx$$

with $h = 0.5$.

10. Using the Simpson's 1/3 rule, subevaluate

$$\int_0^1 \sin^2(\pi x)dx$$

taking six subintervals.

11. Using the Simpson's 1/3 rule, evaluate

$$\int_0^3 \sqrt{4 + x^3}dx$$

taking six subintervals.

12. Using the Simpson's 1/3 rule, evaluate

$$\int_0^1 xe^x dx$$

taking four subintervals.

13. Evaluate

$$\int_0^3 \frac{1}{1+x^3} dx$$

using the Simpson's 3/8th rule with taking six subintervals..

14. Evaluate

$$\int_{0.1}^{0.5} \frac{\cos x}{x} dx$$

using the Simpson's 3/8th rule with taking six subintervals.

15. Evaluate

$$\int_0^2 e^{x^2} dx$$

using the Simpson's 3/8th rule with taking six subintervals.

Salient Features

This manual of **"PYTHON PROGRAMMING LANGUAGE-I"** using Python software is very appropriate and helpful for the second year BSc (Computer Science).

It contains the step by step instructions and procedure to solve problems in subject Groups and Coding Theory, Numerical Techniques which includes Introduction to Python, String, list, tuple, Iterations, Conditional statements, Linear Algebra and Numerical methods in Python using set of various commands.

A large number of problems are solved using the software.

It also demonstrates how this open source mathematical software is useful to understand, evaluate and visualize abstract concept in mathematics in a simplified manner.

We strongly believe that this manual will be enormously beneficial to teachers as well as students to analyse and practice numerous problems in text books according to new syllabus of second year BSc (Computer Science).

ISBN 939022510-8
9 789390 225101
N5396

GOLDEN SERIES by NIRALI