Chapter 1

Supplement No. 1(F): Fundamentals of the PYTHON Programming Language

LEARNING OBJECTIVES

The objective of this supplement is to

- present a short summary of basic concepts of programming with MATHEMATICA;
- describe the implementation of basic programming operations such as loops, accumulators, conditional constructs in MATHEMATICA;
- explain how to apply pseudofunction or pseudocodes into functions or subprograms in MATHEMATICA programming.

The textbook focuses on computational engineering applications. Thus, supplementary course materials involving programs in C/C++, Fortran, Visual Basic, Python, Matlab® and Wolfram Mathematica® languages or software are used. This supplement is a reference document for PYTHON, and illustrates how pseudocode statements can be converted to actual programming languages. It is assumed that the reader is familiar with programming concepts in general and may also be familiar with the fortran programming language.

The algorithms presented in this book are presented in a form that requires very little time and effort to digest. The motivations for using pseudocodes have been stated in the preface and Chapter 1. The aim of this appendix is to present the basic syntax and constructions with simple and concise examples.

1.1 PYTHON BASICS

PYTHON is a high-level language which parses (decompose and analyse) the source code and *interprets* the instructions line by line at run-time. In this regard, PYTHON is an interpreted programming language where we the source code can be run interactively. As soon as a line of command is typed, the interpreter immediately processes it and allows the user to type in another line of a PYTHON code. On the other hand, a compiler takes the completed source code and translates it into the machine language.

Most computer languages delimate blocks of program statements using curly brackets { } or *Begin* and *End* keywords. In these languages, indentation of blocks which is optional is encouraged to make the programs readable. However, PYTHON only requires indentation of delimited blocks and is therefore mandatory to mark the code segments.

1.2 VARIABLES, CONSTANTS, AND INITIALIZATION

1.2.1 Identifiers and Data Types

Identifiers: Identifiers (i.e., symbolic names for variables, functions, and so on) are represented symbolically with combination of upper and/or lower case letters, or combinations of letters with numbers or an underscore (_); e.g., a, b, Ax, V_x, V_y, V_z, Name_1, a1, TOL, and so on. There is no restriction on the length of a variable name.

Variables: A *variable* is a named placeholder that holds any data that can be assigned or changed during program execution. Identifiers (variable names) are case-sensitive; that is, `tol`, `Tol`, and `TOL` denote three different variables. The first character of an identifier cannot be a digit (0 through 9), e.g., `1name` and `2numbers` are illegal. The PYTHON language keywords (such as `class`, `do`, `while`, `in`, `and`, `elif`, etc.) reserved for specific function or commands and cannot be assigned as identifiers.

Data Type. Variables can hold various data types: `integer`, `float`, and `complex` numbers, `strings`, `booleans`, `lists`, and `tuples`. The most common data types in a PYTHON program are integers, floats, strings, lists, and tuples.

Integer is whole number that can be either positive or negative; e.g., `5, -11, 99`, etc.

Floats are real numbers or numbers with a decimal point; e.g., `33.0`, `3.14159`, `-99.737`, etc.

Complex numbers are numbers internally stored using *Cartesian* coordinates, denoting the *real* and *imaginary* parts; e.g., `complex(3,4)` (i.e., $3+4j$), `complex(0,-1)` (i.e., $-j$), etc.

Strings, also called character variables, consists of symbols, letters, and numbers and treated as text. Strings are enclosed by matching single (') or double (") quote; e.g., 'Hello World!', "PO Box 123456", "123-45-6789", etc.

Boolean variables hold `True` or `False` values and used in comparison of two variables, e.g., `2>1` yields `True` , `99<=10` gives `False`, etc.

List, ordered, mutable collections of values, is used to store multiple items in a single variable; e.g., `fruits = ["apple","orange","peach"]`, `person=["name","lastname","age","adress"]`, etc.

Tuples are also used to store multiple items in a single variable; e.g., `fruits = ("apple", "orange", "peach")`, `person=("name","lastname","p_age","adress")`, etc. They are ordered, unchangeable, and allow duplicate values.

PYTHON is considered as a *dynamically-typed* language, meaning that the type of a variable can change during the execution of a program. This feature allows variables to be used without having to define their types one by one. The `type` function is used to query the `type` of a variable, e.g.,

```
ssn="123-45-6789"
pi=3.14
print(type(pi))     # output <class 'float'>
print(type(ssn))    # output <class 'str'>
```

1.3 ASSIGNMENT OPERATION

In PYTHON, an assignment to a variable is to assign a variable a *specific value* and then use the *variable name* to represent that value in subsequent operations. An *assignment* operation denoted by ← (a left-arrow) in pseudocode notation is replaced with `'='` sign.

An assignment operation is carried out as follows:

```
variable_name  =  < expression >    or
variable_name  =  < value >
```

Table 1.1: Compound assignment operators in PYTHON.

Operator	Description	Example
±=	Addition assignment	`p ±= q` is equivalent to `p=p±q`
`*=`	Multiplication assignment	`p *= q` is equivalent to `p=p*q`
`/=`	Division assignment	`p /= q` is equivalent to `p = p /q`
`**=`	Exponentiation assignment	`p **= q` is equivalent to `p=p**q`
`%=`	Assigns the remainder after a division to the lhs	`p%=q` is equivalent to `p=p%q`

Here, *variable_name* can be either initialized with a specified `<value>` or its pre-existing value can be modified with the value resulting from the `<expression>`. That is, a previous value of a *variable* is replaced by its most recently computed value.

In an *assignment* process, the `<expression>` on the *rhs* of the '=' sign is evaluated first, and then the resulting value is placed at the allocated memory location of the *variable* on the *lhs*. For example, in the following code segment, the variables `x` and `y` are initialized in lines 1 and 2. In line 3, the `<expression` (sum of the numerical values of `x` and `y`) on the rhs is evaluated (`x + y = 140.0`) and the result is assigned to the variable `z` on the lhs.

```
1    x = 99         # initialize
2    y = 41.0       # initialize
3    z = x + y      # modify
```

In PYTHON, a variable can be initialized (or assigned) without the need to type-declare it, and its *type* can change dynamically during the program execution. The user can assign a value to a variable without worrying about type-declarations.

1.3.0.1 COMPOUND ASSIGNMENT OPERATORS

In computer programming, the same variable is commonly observed on both sides of an '=' sign, such as `sums = sums + x`, `sums = sums - x`, or `sums = sums * x`, etc. In such expressions, *compound assignment operators* are used to avoid repeating the lhs variable on the rhs.

In PYTHON, compound assignment operators combine assignment operator (=) with another operator (+, -, *, /) with the = operator placed at the end of the first operator. In light of this, the expression `sums = sums + x` can be compressed as `x += y`, where the operator += now denotes an compound assignment operator. Each arithmetic operator (+, -, `*`, and `/`)) has a corresponding *compound assignment operator*: += ($x = x + dx$), -= ($x = x - dx$), `*=` ($x = x * dx$), and /= ($x = x/dx$)). *See* **Table 1.1** for listing.

In following expressions, `X` in line 1 is initialized with zero, which makes the memory value of `X` zero. In line 2, the memory value of `X` is substituted in the *rhs*, which updates the value of `X` as `4`. Finally, in line 3, the *rhs* is evaluated first (`4 + 6 = 10`), and the result is placed in the memory location of `X`.

```
1    X = 0      # X is initialized by zero
2    X+=1       # Equivalent to X = X +1, adding 1 to X, X becomes 1
3    X +=2      # Equivalent to X = X +2, add 2 to X, X becomes 3
4    X = X + 4  # Add 4 to X, X becomes 7
5    X-=1       # Equivalent to X = X -1, predecrement X by 1, X becomes 6
6    X -=3      # Equivalent to X = X -3, subtract from X by 3, X becomes 3
```

Python Code 1.1

```
 1  import sys               # Import any pertinent libraries
 2
 3  def main():
 4      """===================== example.py ===========================
 5    Description: An Python program to calculate the area of a circle.
 6    Written by : Z. Altac
 7      ==================================================================="""
 8      # Constants
 9      PI = 3.14159          # PI is defined as a float constant
10
11      # Execution section
12      radius = float(input("Enter radius: "))    # Prompt input instruction
13      area = PI * radius * radius                # Calculate the area
14      print("The area of the circle is:", area)  # Print output (area)
15
16  if __name__ == '__main__':                     # Program is terminated
17     main()                                      # The calls to run main()
```

Unlike C, C++, Mathematica, or Matlab, Python does not support for the ++ and - operators, which are commonly used for incrementing and decrementing variables by 1.

1.4 Python PROGRAM STRUCTURE

A Python program consists of `import` statements, comments, initialization of variables, `I/O` statements, control and conditional structures, functions, classes, and exceptions, etc.

THE `main()` FUNCTION: A simple Python script code (`example.py`) is illustrated in **Python Code 1.1**. A script file, having a `.py` extension, contains Python code that is executed by the Python interpreter. In this code, `main()` is a function that contains the body (between lines 4-14) of the main program. In line 16, the `if__name__ == "__main__":"` block ensures that the code inside the main body runs only when the script is executed directly, not when it is imported as a module in another script. In line 17, the Python code defined as `main` is executed upon properly indenting.

IMPORTING AND USING MODULES: In Python, standard arithmetic operations are directly available by default but more advanced mathematical functions or operations are not. Libraries for specialized tasks contain functions or *modules* to carry out predefined tasks. In this regard, Python has plenty of built-in functions and modules in libraries, or packages designed for various subjects, such as linear algebra, calculus, data handling and analysis, plotting, and so on (*see* **Table 1.2**). Programmer generally needs to *import* at least one module while preparing a program.

In line 1 of `example.py`, `import sys` statement imports the module `sys`. (It should be pointed out that this module, which provides system-specific parameters and functions, is essentially not required for the program but is added to demonstrate the use of the `import` statement.

Standard mathematical functions (trigonometric, hyperbolic, logarithmic, etc) can be found in a module named `math`. If, in a program, any one of these functions is required, then the `Import` statement is used to import a `module` into the program.

Table 1.2: Some commonly used PYTHON modules.

Library file	Purpose
`numpy`	Scientific Computing (vectors, matrices, math functions, etc.)
`math`	mathematical functions (trig., hyperbolic, log., etc.)
`pandas`	Data Analysis (data analysis and modelling, etc.)
`scipy`	Scientific Computing (igh-performance computing)
`keras`	Machine Learning/AI (enable fast experimentation with deep neural networks)
`pytorch`	Machine Learning/AI
`flask`	Web Development
`pygame`	Game Development
`sympy`	Symbolic Mathematics (provides symbolic mathematics)
`plotly`	Interactive Visualization (provides basic line, pie, scatter, heat maps, polar plots, etc.
`matplotlib`	Data Visualization (creates charts, graphs, pie charts, hhistograms, etc.

Below are three possible ways a module can be imported and used in any program. In line 1, every function from the `math` module is imported; however, every function must be used with a prefixed math (i.e., *module name*), as shown in line 2. In line 4, only the exponential function is imported, and it can be used without a prefix as in line 5. Finally, in line 7, everything inside module `math` is imported and used in the program without the prefix (in line 6). In line 7, only two functions are imported from `math` with the given statement, and then they can be used with `math` prefix.

```
1  import math                  # importing module 'math'
2  y = math.exp(x)              # evaluate $e^x$ by using the prefix math
3
4  from math import exp         # importing 'sqrt' from math
5  y = exp(x)                   # evaluate $e^x$ without a prefix
6
7  from math import *           # importing everythin in 'math'
8  y = expt(x)                  # evaluate $e^x$ without a prefix
9
10 from math import exp, sqrt   # importing $e^x$ and $\sqrt{x}$ from 'math'
```

It is common to import several modules into a program, some of which may contain functions with identical names. But, when importing modules, the programmer is given some control over the functions used in the program either by selecting only the functions needed from each module (as in line 4), or by prefixing all imported functions with the module name (as in line 2).

ADDING COMMENTS: Commenting is done to allow *human-readable* descriptions detailing the purpose of some of the expression(s) and/or to create *in situ* documentation. The PYTHON interpreter ignores the code comments.

A hash (#) symbol is used for *single-line-comments*; it comments out everything that follows #) on the same line. In lines 9, 12-14, and 16-17 of `example.py`, it is basically used to describe an expression (or statement) on the same line. It can also be used to commit an entire line to a comment, as illustrated in line 8 and 11. On the other hand, a *multi-line* comment, such as *Header Comments* (generally placed at the beginning of functions, as implemented in lines 4-7), is used to describe the purpose of the program, its variables, exceptions, other functions used, etc. To comment out a block lines, the beginning and end of a block are marked with triple-quoted string constants (''' ''' or `""" """`).

INPUT/OUTPUT DATA: Input/Output (`I/O`) operations in any programming language are fundamental for interacting with users and processing data.

To get an input data from a user, the `input()` function is used. By default, `input()` reads data as a *string*. However, we may need to convert the input string to other types, such as *integers* or *floats*. In line 12, the user is prompted to enter the radius (i.e., a float number). The input value is converted to `float` and the result is assigned to `radius`.

To submit an output data to the user, the `print()` function is used. In line 14, the value of `area` is printed with an explanatory string (i.e., `"The area of the circle is :"`). We will settle with this information for now, but this topic will be covered in detail in **Section 1.5**.

MULTIPLE STATEMENTS IN A LINE: Python allows multiple code statements per line by using semicolon (`;`) to separate sequential arranged statements, as illustrated in the code segment below:

```
a=10; b= 15; c= 25            # initializations
print(a); print(b); print(c)  # display variables
```

However, this is not advised as it reduces the readability of the code.

LINE CONTINUATION: Most statements in program will fit to a single line. The assignment statement (`area`$\leftarrow \pi R^2$) in line 13 fits a single line. Likewise, I/O statements in line 12 and 14 are single-line expressions.

Some statements may be too long and complex to fit in a single line. Python allows us to write a single statement in multiple lines, also known as *line continuation*. There are two types of line continuation: *implicit* and *explicit line continuation*. In *implicit line continuation*, a statement containing an opened parenthesis (`[`, `(`, or `{`) is assumed to be incomplete until a matching parenthesis (`]`, `)`, or `}`) is encountered. In the following example, the first bracket [in line 1 is matched in in line 3, which causes lines 1-3 to be perceived as a single line.

```
matA = [ [ 1, 2, 3 ],      # outermost '[' bracket is introduced
         [ 4, 5, 6 ],
         [ 7, 8, 9 ] ]     # corresponding bracket ']' is matched here
```

An *explicit line continuation* is used in situations where implicit line joining is not applicable. In this case, a backslash (\) at the end of a current line is used to mark that the current statement spans to the next line. The following example illustrates the use of '\' as continuation mark in lines 2-5.

```
a=0.5
summ = a      \   # indicates continuation of the next line
     + a**2 \   # indicates continuation of the next line
     + a**3 \   # indicates continuation of the next line
     + a**4     # summ now gives a + a^2 + a^3 + a^4.
```

THE OTHER ELEMENTS OF A PYTHON PROGRAM: A Python program also consistes of operators, control and repetition structures, functions, classes, exceptions, etc. *Operators* (arithmetic, comparison, logical, etc) are used to perform operations on variables and data. *Control and repetition structures* (`if-else`, `for`, `while`) provide control the flow of a program. *Functions* are reusable named code segments that perform a specific task. These elements are discussed in the following sections.

1.5 INPUT/OUTPUT (I/O) FUNCTIONS

An inevitable element in any program is the communication of the input and output data with the program. In Python programming, the `print()` and `input()` are the most important and useful functions to display and read data on or from input and output devices.

1.5.1 Displaying Output

The `print()` is a function to display the results of a set of intermediate, or final operations on the monitor, or other standard output device. The `print()` function is designed to convert its arguments into a string representation before printing them.

The `print()` function automatically converts integers, floats, lists, etc. to their string representations using the `str()` function, ensuring that anything passed to the `print` function is converted to a string. In the following example, in lines 2 and 3, the variables `x` and `y` are internally converted to strings before displaying. In line 3, string labels (`'x'` and `"y="`) are concatenated with numbers after converting them to strings.

```
1  x=5; y=1.2345
2  print(x,y)                         # displays : 5 1.2345
3  print('x ',x,"y=",y)               # displays : x  5 y= 1.2345
4  print("Hello World!")              # displays : Hello World!
5  print("Hello", "World!")           # displays : Hello World!
```

The general syntax for `Print` function is given as

```
print( objects, sep=separator, end=end, file=file, flush=flush)
```

where `objects` denotes one or more objects that will be converted to string before printed, *sep* (optional) specifies how objects are separated (in case of two or more objects, the default is a single blank space), *end* (optional) specifies what to print at the end (default is '`\n`', i.e., line feed), *file* denotes an object with a write method (default is `sys.stdout`, i.e., the console), and *flush* is an optional boolean, specifying if the output is flushed (`True`) or buffered (`False`). Default is `False`. Some examples are presented below:

```
print('Name','Bob','Age',35,sep= ';')  # displays objects separated with ';'
print('Hello', end= '!! ')             # displays !! at the end of print
print('Hello ' + ' World!')            # displays joined strings
```

1.5.2 Reading Input

The `input()` is a function used to supply an input data from the user *via* keyboard and has the following syntax:

```
        variable_name = input ( prompt )
```

where `prompt` is a string denoting a message to be displayed before the input. The prompt statement informs the user of the value that needs to be entered through the keyboard. The input data entered is passed as a `string`, which may be a problem if the input data is not a *string*. In such a case, the string should be converted to an appropriate number type using type conversion functions, i.e., `int()`, `float()`, etc.

The following code segment requires two numerical data: `age` (integer) and `height` (float). The inputs, as strings, are converted to integer and float types using `int` and `float` and stored in `age` and `height`, respectively.

```
age = int(input("Enter your age : " ))       # converted to integer type
height = float(input("Enter height in m: ")) # converted to float type
print ("Your age is",age)                    # displays age
print ("Your height is",height)              # displays height
```

The operators + and * can be used on strings for concatenating and repeating, respectively. The concatenation of two strings can be carried out by using a + operator. The * is used to generate repetition of a string a certain number of times. Here are some examples.

```
'Name'+'Last name'          # results in 'NameLast name'
'Name '+'Lastname'          # results in 'Name Lastname'
'Name'+'/'+'Last name'      # results in 'Name/Last name'
 '-'*10                     # results in '----------'
```

1.5.3 Output Formatting

During its development, Python has offered different ways of formatting numbers. The two major string formats are *f-strings* and *str.format*.

1.5.3.1 Formatting with strings

With Python 3.6, the formatted string literals, called f-strings, were introduced. This method requires a prefix `f` to create an *f-string*. The prefix `f` indicates that the string is used for formatting. This method is faster than other available string formatting methods.

Formatting begins with the string `f` or `F` before the opening quotation mark or triple quotation mark in a `print()` statement. In this string, an expression referring to variables or literal values are written between curly braces, i.e., `{variable-name-i}`. The parts of the f-string outside of the curly braces are literal strings. The syntax is shown below:

```
print(f" string-1 {variable-name-1} string-2 {variable-name-2} ... ")
```

where `string-1`, `string-2`, and so on are the strings that will appear in the output, `variable-name-1`, `variable-name-2`, and so on are the variables (whose values) to be displayed.

In the following example, the literal values of `state` and `name` are `"New York"` and `"Mary"`, respectively, while `graduated from` and `State University` are the literal strings.

```
state="New York"
name="Mary"
print(f"{name} graduated from {state} State University.")
```

The above code segment yields

```
Mary graduated from New York university.
```

Next, we consider a case with a string, integer, and float values.

```
age=25
height=1.65
name="Mary"
print(f"{name} is {height}m tall and {age}-years old.")
```

which displays the following output:

```
Mary is 1.65m tall and 25-years old.
```

F-strings may include expressions, function calls, and even conditional logic:

Table 1.3: F-strings for data types.

Type	Description
`s`	String format (default for strings)
`d`	Integer (decimal); Comma is used as a number separator character
`e`	Exponential notation (displays floats in scientific notation with the letter 'e' for the exponent; *default precision* is 6
`f`	Fixed-point notation (displays floats as a fixed-point number; *default precision* is 6

```
x = 3; y = 7
name="Mary"
print(f"x^2 + y^2 = {x*x + y*y}")  # displays x^2 + y^2 = 58
print(f"Hello {name.upper()}!")    # displays Hello, MARY!
```

where the built-in function `string.upper()` is used to convert lower case letters to uppercase.

In foregoing examples, the default formatting settings were used. Python also gives the user control over the display formats with advanced string formatting capabilities, such as specifying field width, alignment, precision, and so on.

The f-format supports a wide range of options for creating string representations of values. F-strings allow the programmer to embed expressions inside string literals with curly braces { }, where format specifiers can be placed to modify the formatting. A typical format specification is done as `f"var:format_spec"`, where `var` is the variable and `format_spec` is the format specification string.

Consider the following example:

```
age=25; height=1.65; name="Mary"
print(f"{name:s} is {height:f}m tall and {age:d}-years old.")
# Output :
# Mary is 1.650000m tall and 25-years old.
```

Here the default type-dependent format-widths have been implemented.

Numeric precision: The numeric precision of numbers can be very important when dealing with floating-point numbers. The user can control numeric precision through formatting options by using **:.nf** to specify the number of decimal places for a floating-point number, where **n** is an integer.

```
num1 = 12.34567
num2 = num1/100

# Format to three decimal places
print(f"Number 1: {num1:.3f}")  # Output: Number 1: 12.346
print(f"Number 2: {num2:.3f}")  # Output: Number 2: 0.123

# Format to two decimal places
print(f"Number 1: {num1:.2f}")  # Output: Number 1: 12.35
print(f"Number 2: {num2:.2f}")  # Output: Number 2: 0.12

# Format to no decimal places (integer rounding)
print(f"Number 1: {num1:.0f}")  # Output: Number 1: 12
print(f"Number 2: {num1:.0f}")  # Output: Number 2: 12
```

Table 1.4: Examples format specifying for a decimal integer (`num=12345678`) and a floating-point number (pi=3.141592653589793).

Format	Description	Example	Displayed
`d`	Default integer format	`f"{num:d}"`	`'12345678'`
`,d`	With comma separators	`f"{num:,d}"`	`'12,345,678'`
`10d`	At least 10-chr wide	`f"{num:10d}"`	`' 12345678'`
`010d`	At least 10-chr wide, with leading zeros	`f"{num:010d}"`	`'0012345678'`
`f`	Default 6 decimal places	`f"{pi:f}"`	`'3.141593'`
`.4f`	Rounded to 4 decimal places	`f"{pi:.4f}"`	`'3.1416'`
`8.4f`	Rounded to 4 decimal places, at least 8-chr wide	`f"{pi:8.4f}"`	`' 3.1416'`
`08.4f`	Rounded to 4 decimal places, at least 8-chr wide, with leading zeros.	`f"{pi:08.4f}"`	`'003.1416'`

Additional examples involving floats and integers are presented in **Table 1.4**.

String alignment and width: Aligning a bunch of data in tabular makes it easier for the analyst to follow. In this context, the user can use `:<w`, `:>w`, or `:^w` to align a string to the left, right, or center within a given width `w`, where `w` is an integer. In the following example, `f"var:>12"`, `f"var:<12"`, and `f"var:^12"` will left-, right-, and center-align the `var` within 12 spaces, respectively.

```
name = "Mary"                      # the data
# Left alignment
print(f"Her name is {name:<12}!") # Output: Her name is Mary        !
# Right alignment
print(f"Her name is {name:>12}!") # Output: Her name is         Mary!
# Center alignment
print(f"Her name is {name:^12}!") # Output: Her name is     Mary    !
```

Type-specific formatting: One may use `:t` to apply type-specific formatting to a value, where `t` is a character that represents the type. For example, :e for scientific notation, :% for percentage, etc.

```
num1 = 12.34567
num2 = num1/100

# Format to three decimal places using F
print(f"Number 1: {num1:<12.3f}")  # Output: Number 1: 12.346
print(f"Number 2: {num2:>12.5f}")  # Output: Number 2:      0.12346

# Format to three decimal places using E
print(f"Number 1: {num1:<12.3E}")  # Output: Number 1: 1.235E+01
print(f"Number 2: {num2:>12.5e}")  # Output: Number 2:  1.23457e-01
```

In Python, *escape sequences* are used to represent characters that cannot be easily typed or difficult to directly include in a format string. They begin with a backslash followed by a character or series of characters that form the escape sequence. The most commonly encountered escape sequences are `\n` (creates a new line), `\t` (creates a horizontal tab), `\r` (creates a carriage return), etc.

```
    print("Line-1\nLine-2")      # \n is used between two strings
    print("Line-1\tLine-2")      # \t is used between two strings
    print("Line-1\rLine-2")      # \r is used between two strings
    # Output:
    #  Line-1
    #  Line-2                    # Line 2 is displayed in a newline
    #  Line-1   Line-2           # Line 2 is displayed after tabbing
    #  Line-1                    # carriage return after line-1
    #  Line-2
```

F-strings support extensive modifiers that control the final appearance of an output string. For a complete list, consult the official web site for more information.

1.6 ARITHMETIC OPERATIONS

Arithmetic operations involve plus (+), minus (-), multiplication (*), division (/), exponentiation (**, integer (floor) division (//) and the modulus operators (%). These operations (excluding the modulus operator) can be used with integer or floating-point types. The modulus operator involving an integer division truncates any fractional part, e.g., 15/3, 16/3, and 17/3 all yield 5. The modulus operator(x%y) produces the remainder from the division x/y, e.g., 15%3=0, 16%3=1, and 17%3=2.

The module math contains the *basic math functions:*, such as trigonometric, hyperbolic, logarithmic (log, log10), exponential (exp), factorial, sqrt, abs, round, floor, and ceil. This module also contains inverse trig functions, hyperbolic functions, and the constants pi and e.

1.7 RELATIONAL AND LOGICAL OPERATORS

Branching in a computer program causes a computer to execute a different block of instructions, deviating from its default behavior of executing instructions sequentially.

Logical calculations are carried out with an assignment statement:

```
        Logical_variable = Logical_expression
```

Logical_expression can be a combination of logical constants, logical variables, and logical operators. A logical operator is defined as an operator on numeric, character, or logical data that yields a logical result. There are two basic types of logical operators: relational operators (<, >, <=, >=, ==, !=) and combinational (logical) operators (and, or, not). In **Table 1.5**, the arithmetic, relational, and logical operators are summarized.

Branching structures are controlled by *logical variables* and *logical operations*. Logical operators evaluate relational expressions to either (True) or 0 (False). Logical operators are typically used with Boolean operands. The logicaland operator and the logical or operator are both binary in nature (require two operands). The logical not operator negates the value of a Boolean operand, and it is a unary operator.

Logical operators are used in a program together with relational operators to control the flow of the program. The and and or operators connect pairs of conditional expressions. Let $\mathbf{L}_1$ and $\mathbf{L}_2$ be two logical prepositions. In order for logical ($\mathbf{L}_1$ and $\mathbf{L}_2$) expression to be True, both $\mathbf{L}_1$ and $\mathbf{L}_2$ must be True. In order for logical ($\mathbf{L}_1$ or $\mathbf{L}_2$) expression to be True, it is sufficient to have either $\mathbf{L}_1$ or $\mathbf{L}_2$ to be True. When using not, a unary logical operator, in any logical statement, the logic value is changed to True when it is False or changed to False when it is True. These operators can be used to combine multiple expressions. For given x=5, y=9, a=18, and b=3, we can construct following logical expressions:

Table 1.5: Arithmetic, relational, and logical in PYTHON.

Operator	Description	Example
`+, -`	Addition and subrations	`a + b or a - b`
`*`	Multiplication	`a * b`
`/`	Division	`a / b`
`%`	finding the remainder (modulo).	`5 % 2 = 1`
`//`	integer division.	`15 // 4 = 3`
`==`	compares the operands to determine equality	`a == b`
`!=`	compares the operands to determine unequality	`a != b`
`>`	determines if first operand greater	`a > b`
`<`	determines if first operand smaller	`a > b`
`<=`	determines if first operand smaller than or equal to	`a > b`
`>=`	determines if first operand greater and equal to	`a > b`
`and`	Logical AND operator	`a and b`
`or`	Logical OR operator	`a or b`
`not`	Logical NOT operator	`not (a)`

```
    (x < y and y < a and a > x)     # True
    (x < y and y > a and a >= b)    # False
    ((x < y and y < a) or a < b)    # True
    ((x > y or y > a) or a < b)     # False
```

In logical expressions, the order of evaluation of `and` and `or` is from left to right.

1.8 PROGRAM CONTROL OPERATIONS

Program control operations are viewed in four categories: *conditional control* (`if`, `if-elif-else`), *loop control* (`for`, `while`, `continue`, `break`), *error control* (`catch`), and *program termination* (`return`).

1.8.1 CONDITIONAL CONTROL: `if` STRUCTURES

PYTHON supports the following variants of `if`, `if-else`, and `if-elif-else` constructs. These structures allow the direction of the process to be changed or to make decisions. The flow path (code blocks to be executed) is based on whether a `<condition>` (boolean expression) is `True` or `False`.

An `if` construct, shown below, executes a block of statements *if and only if* the specified `condition` is `True`.

```
        if <condition> :
             STATEMENTS   #  if <condition> is True
```

An `if-else` construct, syntax shown below, is used to execute two separate blocks, based on whether a `<condition>` evaluates to `True` or `False`.

```
        if <condition> :
             STATEMENTS   #  if <condition> is True
        else:
             STATEMENTS   #  if <condition> is False
```

`elif`s can be chained with an `if-else` construct to allow a more complex decision-making procedure to be implemented. An general form of an `if-elif-else` construct is illustrated below:

```
        if <condition-1>:
             STATEMENTS   # <condition-1> is True
        elif <condition-2>:
             STATEMENTS   # <condition-2> is True
             . . .
        elif <condition-n>:
             STATEMENTS   # <condition-n> is True
        else:
             STATEMENTS   # <condition-n> is False
```

Here, `else` and `elif`'s are optional statements but allow the flexibility of handling many more conditions to be processed.

The body of statements in `if`, `while`, and `for` and so on structures should be grouped to have *one level of indentation*. PYTHON adopts indentation (four spaces per indentation level) for a block of statements. Improper indentation is interpreted as an error.

Some examples involving `if` constructs are illustrated below. The block of statements in the following `if` construct will be executed *if and only if* `x` is greater than `10`.

```
    if x>10 :
        print("x is greater than 10")           # executed when x > 10
```

Note that a course of action for `x<=10` has not been specified.

In the following `if-else` construct, the first block of statements is executed *if and only if* `x>10`; `else` (i.e., $x \leqslant 10$), the second block of statements is executed.

```
    if x>10 :
        print("x is greater than 10")           # executed when x > 10
    else :
        print("x is less than or equal to 10")  # executed when x <= 10
```

In the following example, an `if-elif-else` construct is used to handle multiple conditions.

```
    year = 3
    if year == 1:
        print('Freshman')
    elif year == 2:
        print('Sophomore')
    elif year == 3:
        print('Junior')  # Output is Junior for year=3
    elif year == 4:
        print('Senior')
    else:                 # cases of year>=5
        print('Graduated')
```

1.9 CONTROL CONSTRUCTIONS

Control (loop) constructions are used when a program needs to execute a block of instructions repeatedly until a <*condition*> is met, at which time the loop is terminated. There are three control statements in most programming languages that behave in the same way: `while` and `for`-constructs.

1.9.1 `while` CONSTRUCTS

A `while`-construct has the following general syntax:

```
while  <condition> :   # line should end with a colon
       STATEMENT(s)    # statements block is executed if condition=True
```

In `while` constructs, `<condition>` is evaluated before the statement block. If `<condition>` is `True`, then the block of *statement(s)* is executed. If `<condition>` is `False`, the statement block is skipped.

Consider the following `while`-construct example:

```
n = 0
while n <10 :            # as long as n <10 execute loop
    print('n=', n)
    n += 3               # increment n by 3
```

This code generates integer numbers starting from 0 to 10, skipping by 3. The code displays `n=`*, 3, 6, and 9.

Python does not have a Repeat-Until construct, as presented in the pseudocodes. However, this construct can be emulated using a `while` loop by placing a conditional test (`<condition>`) at the bottom of the loop. It is similar to `while`-construct in that the statement-block is executed as long as the `<condition>` is `False`.

The `do-while` construct has the form

```
while True:                # line should end with a colon
     STATEMENT(s)          # this block is executed at least once
     if <condition>:       # <condition> should end with colon
         break             # exits loop if <condition> is true
```

The loop will be executed when `<condition>` becomes `True`.

The following loop performs the same task by repeating the while-block until $n \geqslant 10$.

```
n = 0
while True:              # repeat until loop execution
    print('value of n:', n)
    n += 3               # increment n by 3
    if n >= 10:          # break out of loop if n>=10
        break
```

Note that the not only the location of the condition but also the condition itself has been changed; however, the output is the same.

Nested-`while` loops includes at least one (*inner*) loop within the body of another (i.e., *outer*) loop. The outer loop controls the number of the inner loop's full execution. Indentation of the inner loop is important to mark where the statement block ends.

1.9.2 for- CONSTRUCT

A `for` construct (or loop) is used for iteration and counting purposes; that is, it is used when a block of STATEMENT(S) is to be executed a specified number of times. A `for`-construct has the following syntax:

```
for  <loop-variable> in <sequence>: # line should end with a colon
      STATEMENT(s)    #statements in the block are executed
```

where `for` and `in` are keywords, the `<loop-variable>` specifies the iteration variable that takes the available values in the listgiven by `<sequence>`. For example,

```
seqn={1,5,6,-4,9}     # seqe is a list of irregular integers
for i in seqn:        # i sequentially takes the values in seqn
     print("i=",n)    # prints i=1, i=5, i=6, i=-4, and i=9

myList=[(1, 0), (2, 4), (3, 6)]
for i, j in myList:   # i,j sequentially takes the values in myList
   print(f" i={i}, j={j}")
# The output is
# i=1, j=0
# i=2, j=4
# i=3, j=6
```

1.9.2.1 range FUNCTION In for LOOPS

The `range()` function is a commonly used to implement counting in a `for` loop. It is used to generate a sequence of integers between two numbers with a specified step size. The syntax for the `range()` function is given as

```
range  ( [<start>], <stop>, [<step>] )
```

where it generates numbers starting from `<start>` (i.e., initial value of the loop-control variable) up to but not including the `<stop>` (i.e., the terminal value) with increments (or decrements if `start>stop`) of `<step>`. The default values are used when the options specified in square brackets above are omitted. If `<start>` is omitted, the control variable starts from *zero*. When `<step>` is omitted, the increments are +1.

Consider the following examples:

```
for i in range(3):         # runs for i=0, 1, 2,  step=+1
     <block>
for j in range(1,9,2):     # runs for j=1, 3, 5, 7, step=+2
     <block>
for k in range(5,2,-1):    # runs for k=5, 4, 3, step=-1
     <block>
for m in range(6, 0, -2):  # runs for k=6, 4, 2, step=-2
     <block>
```

Note that the control (loop) variables do not take the values at the `<stop>`.

Following is an example of nested `for` loops:

```
1  for i in range(2):
2      for j in range(2):
3          print(f" i={i}, j= {j}")
```

Lines 2-3 make up of the block of the outer loop, and line 3 is the block of inner loop. So the `print` statement will be executed for all legal `i` and `j` values. The code output is

```
i=0, j= 0
i=0, j= 1
i=1, j= 0
i=1, j= 1
```

Since `<start>` and `<step>` values of `i` and `j` are not specified, by default they are set `i=0`, `j=0`, and Δ`i=1`, Δ`j=1`, respectively. Also note that `i` and `j` do not take the values of 2.

The indentation is important in that it marks where a block starts and where it ends. To illustrate this consider the following code segment:

```
1  for i in range(2):
2      for j in range(5):
3          print(f" *** j = {j}")
4      print(f" ((( i={i}  j={j} ))) ")
```

Lines 2-4 make up of the block of the outer loop, while the block of the inner loop is a single line (line 3). The `print` statement in line 3 is executed for every `i` and `j`, but the `print` statement in line 4 is executed only for every `i`. The code output is

```
 *** j = 0
 *** j = 1
 *** j = 2
 *** j = 3
 *** j = 4
 ((( i=0  j=4 )))
 *** j = 0
 *** j = 1
 *** j = 2
 *** j = 3
 *** j = 4
 ((( i=1  j=4 )))
```

Consider the following code segment:

```
for i in range(2):
    for j in range(5):
        print(f" *** j = {j}")
        print(f" ((( i={i}  j={j} ))) ")
```

The output is

```
*** j = 0
((( i=0  j=0 )))
```

```
*** j = 1
((( i=0  j=1 )))
*** j = 2
((( i=0  j=2 )))
*** j = 3
((( i=0  j=3 )))
*** j = 4
((( i=0  j=4 )))
*** j = 0
((( i=1  j=0 )))
*** j = 1
((( i=1  j=1 )))
*** j = 2
((( i=1  j=2 )))
*** j = 3
((( i=1  j=3 )))
*** j = 4
((( i=1  j=4 )))
```

1.9.2.2 break, continue AND pass STATEMENTS

BREAK: A `break` is a control statement used to terminate or change the ongoing loops. `break` is mostly used with the looping statements, such as `while` or `for` loops. A `break` terminates the nearest enclosing loop and skips any (optional) `else` statements in the loop. If a `for` loop is terminated using a `break`, the loop control variable preserves its current value.

In the following code segment, the loop control variable `kount` runs up to 4, i.e., executes the loop for `kount < 4`. The condition for exiting the loop is given by an `if` structure within the loop. The first `print()` statement will be executed with each iteration until the `break` statement is encountered.

```python
for kount in range(100):
    if kount == 4:
        break       # break is placed here
    print('Number is ',kount)
print('Outside of the loop')
```

The output is

```
Number is 0
Number is 1
Number is 2
Number is 3
Outside of the loop
```

Note that the final `print()` statement has the same indentation with the `for` statement, while the first `print()` is indented to be a statement of `if`-construct.

CONTINUE: A `continue` statement, when triggered by an external condition, skips part of the loop and continues with the next cycle of the nearest enclosing loop. A typical use of `continue` statement is illustrated in the following code segment.

```
for kount in range(5):
    if kount == 3:
        continue         # skips following block for kount = 3
    # Place loop block here, indented in line with the print statement
    print('Number is ',kount)
print('Outside of the loop')
```

Here, the `for` loop skips the loop block for `kount=3` only. The last print statement is outside the loop so its content is printed only once. The code output becomes

```
Number is  0
Number is  1
Number is  2
Number is  4
Outside of the loop
```

The difference in between `continue` and `break` statements is that the `continue` statement disrupts the current loop iteration, but continues with the next iteration. On the other hand, `break` statement exits the loop completely and moves on to the code that follows the loop.

PASS: A `pass` statement is a null operation. When an external condition is triggered, it allows the condition to be processed without affecting the loop in any way. Consider the following code segment:

```
from math import sqrt
a=...; b=...; c=... # input a, b, and c
d=b*b-4*a*c
if d>0:
    x1=(-b-sqrt(d))/(2.*a); x2=(-b+sqrt(d))/(2.*a)
    print(f"x1={x1} x2={x2}")
else:
    pass            # the case of imaginary roots is not processed
```

1.10 VECTOR AND MATRIX OPERATIONS

Python, unlike most programming languages, does not have built-in support for arrays. But, Python is furnished with several data types, such as *lists* and *tuples* that are often used as *arrays*. Moreover, the items stored in list or tuple types of sequences need not be of the same type.

1.10.1 DEFINING ARRAYS

In Python, vectors can be represented with *lists*. Several modules and libraries support arrays and array operations. As a part of the standard library, `array` module provides a basic array type with support for efficient storage and manipulation of the same type data. Thus, to create an array in Python, the `array` module need to be imported and used along with the `array()` function.

```
import array as aname  # import array module
```

Using this function, it is possible to create an array of basic types, i.e., `integer`, `float`, or `characters`. `array()` function accepts *type code* and *initializer* as a parameter value and returns an object of array class.

The syntax for creating an array is

```
object = aname.array(typecode[, initializer])  # create an array
```

where `typecode` is a character used to specify the type of elements in the array (`'i'`, `'u'`, `'f'`, and `'d'` respectively denote integer, character, float, and double precision) and the `initializer` is an optional value from which array is initialized.

```
import array as arr

a = arr.array('i',[1, 2, 3])  # create an integer type array
print(type(a), a)  # gives <class 'array.array'> array('i', [1, 2, 3])

b = arr.array('u', 'aBcD')    # create a char type array
print(type(b), b)  # gives <class 'array.array'> array('u', 'aBcD')

c = arr.array('d', [pi, e, 3.]) # create a double type array
print(type(c), c) # gives <class 'array.array'> array('d', [3.14, 2.78, 0.1])
```

In Python, a matrix can be defined as a 2D list or 2D array. Additionally, `NumPy`, a fundamental package for scientific computing in Python, provides a powerful array object and many functions for array operations, including mathematical operations, statistical analysis, and linear algebra. For complete list of functions, go to official NumPy site.

The `NumPy` module must be imported, as shown below, before the arrays are defined and used in array operations.

```
import numpy as nname  # import array module
```

The syntax for creating an array is given as follows:

```
object = nname.array(initializer)     # create an array
```

Once an array is defined, the `NumPy` allows element by element operations as in the following example operations with matrices.

```
import numpy as np   # importing numpy for matrix operations

A = np.array([[8, 6], [4, 30]])  # initialize matrix A
B = np.array([[2, 3], [4, 15]]) # initialize matrix B

C = np.add(A,B)        # use add() to add matrices
print ("Matrix C is : ")
print (C)

D = np.subtract(A,B)   # use subtract() to subtract matrices
```

```
print ("Matrix D is : ")
print (D)

E = A/B    # use divide() to perform element by element division
print ("Matrix E is : ")
print (E)

F = np.multiply(A,B)   # use subtract() to subtract matrices
print ("Matrix F is : ")
print (F)
```

The output is

```
Matrix C is :
[[10  9]
 [ 8 45]]
Matrix D is :
[[ 6  3]
 [ 0 15]]
Matrix E is :
[[4. 2.]
 [1. 2.]]
Matrix F is :
[[ 16  18]
 [ 16 450]]
```

1.10.2 ALGEBRAIC OPERATIONS WITH ARRAYS

The NumPy linear algebra functions relying on BLAS and LAPACK routines provide efficient low level implementations of linear algebra algorithms. The SciPy library also contains a linalg submodule, and there is overlap in the functionality provided by the SciPy and NumPy submodules. SciPy contains some of the functions (pertinent to matrix decompositions, pseudo-inverses, etc) not found in NumPy.

NumPy allows for efficient operations on the data structures often used in vectors and matrices. Although NumPy is not the main focus of this material, it is frequently throughout the programs involving vector and matrix operations.

A vector (one-dimensional array) or a matrix can be created using numpy as follows:

```
import numpy as np           # importing numpy for matrix operations

ar = np.array([8, 6, 4, 3])          # create a row array, $ar_i \leftarrow ar_{1,i}$
ac = np.array([[2], [-3], [4], [2]]) # create a column array, $ac_i \leftarrow ac_{i,1}$
M  = np.array([[1, 2],               # create a $2 \times 2$ square matrix, M.
               [7,-3]])              # using 'array' structure
B  = np.mat([[1, 2],                 # create a $3 \times 2$ rectangular matrix
        [3, 4],
        [5, 6]])

print("row array=",ar)              # print row array
print("column array=\n",ac)         # print column array
print("Matrix [M]=\n",M)             # print matrix M
```

```
    print("Matrix [N]=\n",N)             # print matrix N
```

The output is

```
row array= [8 6 4 3]
column array=
 [[ 2]
 [-3]
 [ 4]
 [ 2]]
Matrix [M]=
 [[ 1  2]
 [ 7 -3]]
Matrix [B]=
 [[1 2]
 [3 4]
 [5 6]]
```

The matrix data structure (`numpy.matrix`) is not recommended for basically two reasons: (1) arrays are the de facto standard data structure of `NumPy`; (2) the majority of `NumPy` operations return arrays, not matrix objects.

Let `M`, `N` and `r` and `x` denote respectively matrices and vectors. Retrieving and using pieces of vectors or matrices can be accomplished using the following:

```
    multiply(M,N)         # element by element multiplication
    linalg.inv(M)         # finds inverse of matrix M
    lingalg.eig(M)        # finds eigenpairs of matrix M
    lingalg.solve(M,b)    # solves mx=b matrix equation
    lingalg.matmul(M,N)   # computes MN matrix product
    linalg.dot(r,x)       # computes dot product of two arrays
    linalg.norm(M)        # computes vector or matrix norm
```

`linalg` module provides many more built-in commands dealing with vector, matrix, and linear algebra operations. The reader is referred to `numpy.linalg`.

1.10.3 ACCESSING ARRAY ELEMENTS

Indexing in Python starts at `0`. With this in mind, accessing an element of an array is carried out by using the index of the element, using square brackets. Python allows accessing elements from the end of the array by using negative indices, as illustrated in the examples below:

```
    import numpy as np    # import numpy for matrix operations
    a= np.array([4, -2, -1, 2, -3, 1])      # create a row vector
    print(a[0])           # gives the 1st element, i.e.,  4
    print(a[2])           # gives the 3rd element, i.e., -1
    print(a[-1])          # gives the last element, i.e., 1
    print(a[-2])          # gives element 2nd to the last, i.e., -3
```

Sometimes a sublist of an array is required. Extracting a sublist from an array is done through so-called *slicing*. A *slice*, which has `[start: end+1]` structure, is used to select any part of an array. This notation acts like a `range` function in that the second argument does not include the `stop` (i.e., `=end + 1` value). For example, continuing with the definitions in the above code, we find

```
print(a[:3])          # gives first three elements, i.e., [ 4 -2 -1]
print(a[-2:])         # gives last two elements, i.e., [-3  1]
print(a[:])           # gives all elements, i.e., [ 4 -2 -1  2 -3  1]
print(a[1:6:2])       # gives elements from 1 to 5 by 2s, i.e., [-2  2  1]
```

In case of two-dimensional arrays, `NumPy` provides more direct and flexible slicing with the ability to slice both rows and columns simultaneously. Examples of slicing a matrix is given below:

```
import numpy as np     # importing numpy for matrix operations
M= np.array([[1, 2, 3, 4],
     [1, 3, 6, 9],    # create a square matrix, M
     [2, 4, 6, 8],
     [0,-1,-2,-3]])
print(M)
print(M[1,2])         # find 2nd row, 3rd column element of M
print(M[2:])          # gives 3rd and 4th rows of M
print(M[0:5,0:2])     # gives 1st two column of M
print(M[1][3])        # gives 2nd row 4th column element of M
```

The output is

```
[[ 1  2  3  4]
 [ 1  3  6  9]
 [ 2  4  6  8]
 [ 0 -1 -2 -3]]
6
[[ 2  4  6  8]
 [ 0 -1 -2 -3]]
[[ 1  2]
 [ 1  3]
 [ 2  4]
 [ 0 -1]]
 9
```

The following code performs `A = 2 * M + 3 * N` matrix operation, where the matrices are 4×4 square matrices.

```
import numpy as np     # importing numpy for matrix operations
M= np.array([[1, 2, 3, 4],
     [1, 3, 6, 9],      # create a square matrix, M
     [2, 4, 6, 8],
     [0,-1,-2,-3]])
N= np.array([[1, 0, -3, 2],
     [1,-1, 2, 6],      # create a square matrix, N
     [2, 2,-3, 4],
     [1, 1,-2, 3]])
print("A=\n",2*M+3*N)  # displays A, does not require initialization
```

Table 1.6: Some of the basic matrix operations provided by NumPy.

Function	Description
`array()`	creates a matrix
`dot()`	performs matrix multiplication
`inner()`	performs inner product (`@` operator, `np.matmul()`, and `np.dot()` also return the inner product when both arguments are one-dimensional arrays.)
`transpose()`	transposes a matrix
`linspace()`	creates an array of n-uniformly spaced points between `start` and `stop`
`linalg.inv()`	calculates the inverse of a matrix
`linalg.det()`	calculates the determinant of a matrix
`flatten()`	transforms a matrix into 1D array
`matmul()` or `@` operator	performs matrix multiplication, `matmul(A,B)=A@B`

```
A = np.zeros((4,4))    # initialized with zero before next operation
for i in range(4):
    for j in range(4):
        A[i][j] =2* M[i][j] + 3*N[i][j]
print("A=\n",A)        # prints all elements of A
```

This code evaluates the matrix operations using the Python functionality (i.e., `2*M+3*N`) and coding the mathematical procedure with `for` loops. The latter requires initialization (creating memory space and number type). The code output becomes

```
A=
 [[  5   4  -3  14]
 [  5   3  18  36]
 [ 10  14   3  28]
 [  3   1 -10   3]]
A=
 [[  5.   4.  -3.  14.]
 [  5.   3.  18.  36.]
 [ 10.  14.   3.  28.]
 [  3.   1. -10.   3.]]
```

In Python, matrices generally need to be initialized before performing operations, whether using basic *lists* or advanced libraries like NumPy. For matrix operations, NumPy is recommended owing to its efficiency and extensive functionality. Some of the functions required in basic matrix operations are presented in **Table 1.6**.

```
import numpy as np    # importing numpy for matrix operations
A = np.array([[1, -1, 1], [1, 0, 2], [-1, 1, -2]])
print("A=\n",A)               # prints all elements of A
print("A*A=\n",A*A)           # prints element-by-element A*A
print("A.A=\n",np.dot(A, A)) # prints A*A matrix multiplication
print("Det(A)=",np.linalg.det(A))  # prints determinant of A
print("A^-1=\n",np.linalg.inv(A))  # prints inverse of A
print("Norm(A)-f=",np.linalg.norm(A, 'fro')) # prints f-norm of A
b = np.array([3, -3, 2])      # define a row vector b
print("A.b=",np.dot(A, b))    # print A.b matrix-vector multiplication
```

Table 1.7: Some of the basic matrix operations provided by NumPy.

Function	**Description**
Instance variable	**Output**
`.size`	number of elements in array
`.shape`	number of rows, columns, etc.
`.ndim`	number of array dimensions
`.real`	real part of array
`.imag`	imaginary part of array
method	**Output**
`.mean()`	average value of array elements
`.std()`	standard deviation of array elements
`.min()`	return minimum value of array
`.max()`	return maximum value of array
`.sort()`	low-to-high sorted array (in place)
`.reshape(a, b)`	Returns an a×b array with same elements
`.conj()`	complex-conjugate all elements

```
    print("b.b=",np.dot(b, b))   # print b.b dot product
    print("Norm(b)-f=",np.linalg.norm(b))     # prints L2 norm of b
    print("Array of zeros=\n",np.zeros((4)))  # prints array of zeros
    print("Matrix of ones=\n",np.ones((2,4))) # prints array of ones
    print("Identity matrix=\n",np.eye((3)))   # prints 3x3 identity matrix
    print(np.linspace(3,4,5))                 # prints [3. 3.25 3.5 3.75 4. ]
```

Notice that `A*A` is element-by-element multiplications, not multiplication in matrix operation sense. Matrix multiplications carried out using `np.dot()` can be carried out using `np.matmul()` or using the operator `@`. However, `np.matmul()` and `np.dot()` behave differently for arrays with more than three dimensions.

1.10.4 OBJECT ATTRIBUTES

In NumPy, arrays (i.e. objects) come with a variety of attributes that can help user to understand the array properties and manipulate them effectively. A list of some of the key attributes that can be used with NumPy arrays is given in **Table 1.7**. Some examples are given below:

```
    import numpy as np    # importing numpy for matrix operations
    A = np.array([[1, -1, 1], [1, 0, 2]])
    b = np.array([2,-1,3,2,-4])
    print(A.ndim, b.ndim)       # gives 2  1
    print(A.shape, b.shape)     # gives (2, 3) (5,)
    print(A.size, b.size)       # gives 6 5
    print(A.mean(),b.mean())    # gives 0.666666666 0.4
    print(A.min(), b.min())     # gives -1 -4
    print(A.max(), b.max())     # gives 2 3
    b.sort()
    print(b)                    # gives -4 -1  2  2  3]
    print(A.reshape((3,2)))     # gives [[ 1 -1] [ 1  1] [ 0  2]]
```

1.11 FUNCTIONS IN PYTHON

1.11.1 Built-In Functions

PYTHON has built-in functions which we can use by simply suitably calling them with their names and arguments. The built-in functions need not be defined. PYTHON has some built-in functions, some of which are presented below. To see the full list of functions, click on the link.

`abs(x)` returns the absolute value of a number, which may be an integer, or a floating point, or a complex number. For a complex number, it returns its magnitude;
`bool(x)` returns a Boolean value (`True` or `False`);
`chr(code)` returns the string representing a character whose Unicode code value is `code`, e.g., `chr(65)` and `chr(97)` correspond to 'A' and 'a', respectively;
`divmod(a, b)` returns a pair of numbers consisting of their quotient and remainder of an integer division;
`max(iterable, *[, key, default])` returns the largest item in an iterable argument;
`max(arg1, arg2, *args[, key])` returns the largest of two or more arguments;
`minx(iterable, *[, key, default])` returns the smallest item in an iterable argument;
`min(arg1, arg2, *args[, key])` returns the smallest of two or more arguments;
`pow(arg1, arg2, *args[, key])` returns `x` to the power of `y`, optionally modulo `z`;
`range(start, stop, *[step])` returns an iterable range object from `start` to `stop` with `steps`;
`len(s)` returns the length of `s` (i.e., string, list, tuple);
`round(number[, ndigits])` returns number rounded to `ndigits` precision after the decimal point. If `ndigits` argument is omitted or is `None`, then the function returns the nearest integer to its input number.

The functions `input()`, `print()`, `dir`, `global`, `int()`, `float()`, `str()` and so on are built-in functions as well. The `int()`, `float()`, and `str()` functions are type conversion functions, which convert values from one type of to another type. For example,

```
print(int('13'))     # gives 13 as integer
print(int(13))       #    same as above
print(int(13.9))     #    same as above
print(int(-13))      # gives -13 as integer
print(float('13'))   # gives 13.0
print(float(13.9))   # gives 13.9
print(float(-12))    # gives -12.0
print(str(12))       # gives '12'
print(str(12.9))     # gives '12.9'
```

For mathematical operations, the `math` module can be used. This module provides functions on the number representations, power and logarithmic, trigonometric and inverse trigonometric, angular conversions, hyperbolic and inverse hyperbolic functions, and constants. However, the user should import the `math` module (*see* Section 1.3 on how to import modules) before using any one of these functions.

1.11.2 User-Defined Functions

Numerical algorithms often require performing a task numerous times to accomplish the intended job, which is not built in the PYTHON libraries. To simplify matters, it is generally desired to collect all the statements under one function, which also helps make large programs easier to manage.

In PYTHON, a specific task in a complicated program is often prepared as a function. A function in PYTHON is introduced with the keyword `def` statement. The function name (identifier) and the parenthesized list of formal parameters end with a colon. The next line(s) form the body of the intended function, which are expressed as an indented block of code.

A function in general have input and output variables, which are communicated with the calling program. The syntax for an n-parameter function is given as

```
def function_name (p1, p2, ..., pn):
    """ doc-string """
    STATEMENTS
    return
```

where `p1`, `p2`, ..., `pn` are the parameters (variables) of the function that are used to pass data into `function_-name`, `doc-string`, a triple-quoted string, (if required) is typed immediately after the function header. A doc-string may be short or long depending on the complexity of the function and its input and output parameters.

A `function` localizes the variable names to avoid the name conflicts between the *local* and *global names* used in a program. That is why, it is more suitable to prepare and use functions that define variables locally. In such *localized environments*, the variables are allowed to have different set of values (from the global definitions) and are used locally. These local environments are called functions.

Traditionally the function variable list is referred to as the *parameter* list. *Arguments* are the values of the variables sent to the function. As PYTHON is a dynamically typed language, and the types of the input nor output parameters need not be designated beforehand.

A `return` statement is used within the body of a function to stop execution and the *expressed value* (if any) should be returned to the caller. In other words, a `return` statement is required if you want to send a result back to the caller of the function. If an explicit argument with a `return` statement is not specified, then the `None` value is returned. The `return` statement usually is the final command in the body of the function. However, a function may have multiple `return` statements. For instance, consider the following piece-wise defined function:

$$saw(x) = \begin{cases} x, & 0 \leqslant x \leqslant 2 \\ 4 - x, & 2 \leqslant x \leqslant 4 \\ 0, & \text{otherwise.} \end{cases}$$

The below PYTHON function code illustrates the use of multiple returns. The code branches into three cases, depending on the input value `x` and, after performing the operations, returns the computed value at the end of the block.

```
def saw(x):                # define function 'saw'
    if 0 <= x <= 2:
        return x           # for case 0 ⩽ x ⩽ 2 returns x
    elif 2 <= x <= 4:
        return 4 - x       # for case 2 ⩽ x ⩽ 4 returns 4 - x
    else:
        return 0           # otherwise returns 0
```

A potential source of confusion in PYTHON is that the global variables can also be accessed from within a function as well as everywhere else in the code. Consider the following `function` example that returns the sum of the first n-terms of a geometric series with common ratio r:

```
r = 0.4                # r, a global variable, initialized
def geosum(n):         # computes S = 1 + r + r^2 + ... + r^(n-1) = (1 - r^n)/(1 - r)
```

```
    return (1-r**n)/(1-r)

print(geosum(10))   # prints 1.6664919
print(r)            # prints 0.4
```

where r, global variable, is assigned the value of `r = 0.4` outside the function, while n is the only input argument passed to `geosum`. In fact, we could assigned a value to n outside `geosum` before calling it (without arguments `geosum()`), the function would still carry out its task.

It is also possible to define local and global variables with the same name, as in the following modified function:

```
r = 0.4
def geosum(n):
    r = 0.5         # uses r = 0.5 to compute the return value
    return (1-r**n)/(1-r)

print(geosum(10))   # prints 1.998046875
print(r)            # prints 0.4
```

Here $r = 0.5$ is used but its value is confined to the function `geosum`; that is why, its global value ($r = 0.4$) is printed outside the function. In other words, when the function finishes its task, the local variables no longer exist (in programming terms, they go *out of scope*) upon exiting the function, whereas the global variables are still there and retain their the most current values.

It should be kept in mind that the local variable names inside a function always take precedence over the global names. PYTHON looks for the values of the variables with the given names (i.e., *local identifiers*) that appear in the the function. If the local variables are found, then these values are used. If some of the variables are not found in the local identifiers, PYTHON will search the global identifiers for matching names. If the variable is found among the global variables (i.e. defined in the main program), then the corresponding value is used.

If some of the global variables are to be changed inside a function, they must be explicitly stated by using the keyword `global`. Consider the case where r is made a global variable in the code above:

```
r = 0.4
def geosum(n):
    global r
    r = 0.5
    return (1-r**n)/(1-r)

print(geosum(10))   # prints 1.998046875
print(r)            # prints 0.5
```

In this case, the keyword `global` instructs PYTHON not to define `r` as a new local variable.

In general, you should avoid using global variables inside functions. Instead, define all variables used inside a function either as local variables or as arguments passed to the function.

MULTIPLE `returns`. Functions in practice may end up having multiple outputs (i.e., multiple `return` values). As an example for a such case, consider the determining the magnitude and direction of a 2D vector in the first quadrant ($\mathbf{a} = x\mathbf{i} + y\mathbf{j}$ where $x \geqslant 0$ and $y \geqslant 0$).

```
from math import pi, sqrt, atan  # importing required functions

x=1.0; y=1.0                    # supply vector components

def vector0(x, y):              # define a function called "vector0"
    mag = sqrt(x*x + y*y)       # find magnitude
    dir = atan(y/x)             # find direction in radians
    return mag, dir             # retrun both mag and dir (two outs)

m, d = vector0(x,y)             # call function & assign its outs to m and d
print("magnitude=",m)           # prints magnitude = 1.41421356
print("direction=",d*180/pi)    # prints direction = 45.0 (in degrees)
```

Here, the `return` arguments are separated by a comma; `mag, dir` which are local variables. With the `vector0(x,y)` call, these output variables are assigned to the global variables `m` and `n` in the order they are returned, also separated by a comma. When a function returns multiple values like this, it actually returns a *tuple*.

A Void Function A void function is a function without a `return`. In Python, there are exceptional cases where a function does not need to return any value, in which case a return statement is not required. For example, some functions only serve the purpose of printing information to the screen.

The following code involves the definition and the use of two functions: `warnin` and `fx`. The function `warnin` prints a message to the user informing him that the data to be entered must be complex type. That is, no computations, or evaluations ,or decision makings, etc. are performed. The second function `fx` carries out `b=a*a` operation and the result is stored on the local variable `b`, but this value of the local variable is not passed to a global variable with a `return` statement.

```
def warnin():     # function 'warnin' puts out a message
    print("Entered value must be a complex number")

def fx(a):        # function 'fx' computes b=a**2
    b=a*a

warnin()          # prints 'Entered value must be a complex number'
print(fx(a=3.))   # prints 'None'
```

When a function is not terminated with `return` statement, Python automatically return a variable with value `None`.

Functions as arguments to functions. Arguments to Python functions can be any Python object, including another function. This feature of functions is quite useful for many applications.

Consider the function *saw(x)* defined as follows:

$$saw(x) = \begin{cases} func(x), & 0 \leqslant x \leqslant 2 \\ func(4-x), & 2 \leqslant x \leqslant 4 \\ 0, & \text{otherwise.} \end{cases}$$

where $func(x)$ is an arbitrary real function. Note that $saw(x)$ uses $func(x)$, which may be left as a separate function or defined as a function argument to $func$; i.e., $saw(func, x)$. In this case, the code segment can be arranged as follows:

```
def func(x):                # define an arbitrary function 'func(x)'
    return x**2             # func is defined as x^2

def saw(func,x):            # define a function with a function argument
    if 0 <= x <= 2:
        return func(x)      # func is passed inside 'saw'
    elif 2 <= x <= 4:
        return func(2-x)    # func is passed inside 'saw'
    else:
        return 0
```

In case of simpler functions, PYTHON offers defining a small user-defined function, called the lambda function. A `lambda` function may have several, but only a single-line expression. The lambda function syntax is as follows:

```
function_name = lambda arg1, arg2, ..., argn : <expression>
```

where `function_name` is the name of the function, `arg1`, `arg2`, ..., `argn` are the arguments, and `<expression>` is the expression that can fit on a single line.

Some examples of lambda function are illustrated below:

```
func = lambda x : x*x        # defines $func(x) = x^2$
f = lambda x, y, z : x*x/( y*y + z*z)  # defines $f(x,y,z) = x^2/(y^2+z^2)$
print(func(3.0))             # gives 9.0
print(f(5,3,4))              # gives 1.0
```

Bibliography

[1] GOWRISHANKAR, S., VEENA, A. *Introduction to Python Programming*. CRC Press, 2018.

[2] LUTZ, M., *Learning Python*. O'Reilly Media, 2008.

[3] https://www.python.org/

[4] https://www.python.org/downloads/

[5] https://www.programiz.com/python-programming

[6] https://www.w3schools.com/python/