CHAPTER 1 NUMERICAL ALGORITHMS AND ERRORS

SUPPLEMENT No. 1f:

PYTHON TUTORIAL

prepared for

NUMERICAL METHODS FOR SCIENTISTS AND ENGINEERS With Pseudocodes

By Zekeriya ALTAÇ

October 2024

Supplement No. 1f: The Python Tutorial

LEARNING OBJECTIVES

The objective of this Python programming tutorial is to

- present a short summary of the basics of Python;
- describe the implementation of basic programming operations such as loops, accumulators, conditional constructs;
- explain how to prepare functions or subprograms.

The textbook "*Numerical Methods for Scientists and Engineers: With Pseudocodes*" focuses on implementing numerical methods in science and engineering applications. Supplemental course materials and resources, including C/C++, Fortran, Visual Basic, Python, Matlab, and Mathematica, are provided to assist the instructors in their teaching activities outside the class.

The aim of this short tutorial is to enable students to acquire the knowledge and skills to convert the pseudocodes given in the text into running Python programming language. It is not intended to be a "complete language reference document." The author assumes that the reader is familiar with programming concepts in general and may also be familiar with the Python programming language at the elementary level. In this regard, this tutorial illustrates the conversion and implementation of pseudocode statements (such as formatted/unformatted input/output statements, loops, accumulators, control and conditional constructs, creating and using functions, and subprograms, etc.) to the Python programming language.

1 Python BASICS

Python is a high-level language that parses (decomposes and analyzes) the source code and *interprets* the instructions line by line at runtime. In this regard, Python is an interpreted programming language where 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.

In most computer languages, blocks of program statements are delimited using curly brackets { }, parentheses (), or Begin and End keywords, etc. In these languages, indentation of blocks is optional, but it is encouraged to improve the readability of programs. However, Python requires indentation of delimited blocks and is mandatory for marking the code segments.

2 VARIABLES, CONSTANTS, AND INITIALIZATION

2.1 IDENTIFIERS AND DATA TYPES

Identifiers: Identifiers (i.e., symbolic names for variables, functions, and so on) are represented with combinations 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.) are reserved for specific functions or commands and cannot be assigned as identifiers.

Data Type. Variables can hold various data types: numeric types (`integer`, `float`, and `complex` numbers), text types (`strings`), Boolean types (`bool`), sequence types (`lists`, `ranges`, and `tuples`), set, map, binary, and none types. The most common data types used in a PYTHON program are integers, floats, strings, lists, and tuples.

Integer (`int`) is a whole number that can be either positive or negative; e.g., `5, -11, 99`, etc. The maximum integer value is not limited by PYTHON, but it is limited by the memory on the computer.

Floats (`float`) are real numbers or numbers with a decimal point; e.g., `33.0`, `3.14159`, `-99.737`, etc. The range of `floats` is limited; floats typically represent values between approximately -1.8×10^{308} and 1.8×10^{308}, but precision decreases with larger numbers. Operations that exceed these limits may result in overflow (`inf`) or underflow (values closer to zero than the minimum representable value).

Complex numbers (`complex`) are numbers internally stored using *Cartesian* coordinates, denoting the *real* and *imaginary* parts; e.g., $3 + 4j$ as `complex(3,4)`, $-j$ as `complex(0,-1)`, etc. The imaginary part is denoted as a `j` or `J`. The real and imaginary parts of a complex variable `zvar` can be separated as `zvar.real` and `zvar.imag`.

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

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

List (`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","ssn"]`, 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 *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, as shown below:

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

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`

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 ← (a left-arrow) in the pseudocode notation is replaced with `'='` sign.

An assignment operation is carried out as follows:

```
variable_name  =  expression   (or)
variable_name  =  value
```

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 x with 99 (int)
2   y = 41.0      # initialize y with 41.0 (float)
3   z = x + y     # assign the result of an expression
```

In PYTHON, a variable can be initialized (assigned values) 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.

3.1 COMPOUND ASSIGNMENT OPERATORS

In computer programming, it is common to use the same variable on both sides of the '=' sign (as in `x = x + val`, `x = x * val`, etc.), allowing the value of the variable to be updated, modified, or evaluated. In PYTHON, such expressions can be condensed using the so-called *compound assignment operators*, which combine assignment operator (=) with another operator (+, -, *, /) with the = operator placed at the end of the first operator. The expression `x = x + val` can then be compressed as `x += val`, where the operator `+=` now is the compound assignment operator.

Each arithmetic operator (`+`, `-`, `*`, and `/`)) has a corresponding compound assignment operator (*See* **Table 1.1**). In the 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`.

Py Code 1.1

```python
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()
```

```python
X = 0      # X is initialized by zero
X +=1      # Equivalent to X = X +1, adding 1 to X, X becomes 1
X +=2      # Equivalent to X = X +2, add 2 to X, X becomes 3
X = X + 4  # Add 4 to X, X becomes 7
X-=1       # Equivalent to X = X -1, predecrement X by 1, X becomes 6
X -=3      # Equivalent to X = X -3, subtract from X by 3, X becomes 3
X *=4      # Equivalent to X = X *3, multiplies X by 4, X becomes 12
X /=6      # Equivalent to X = X /6, divides X by 6, X becomes 2
```

Unlike C, C++, Java, etc., Python does not support the `++` and `--` operators, which are commonly used for incrementing and decrementing variables by 1.

4 A Python PROGRAM STRUCTURE

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

THE `main()` FUNCTION: A simple Python script code (`example.py`) is illustrated in **Py 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 (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

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, polar plots, etc.)
`matplotlib`	Data Visualization (creates charts, graphs, pie charts, hhistograms, etc.)

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**). A programmer generally needs to `import` at least one module to prepare a running program.

In line 1 of `example.py`, the `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. Below are three possible ways a module can be imported and used in any program.

```
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 only 'exp' from 'math'
5  y = exp(x)                   # evaluate e^x without a prefix
6
7  from math import *           # importing everything in 'math'
8  y = exp(x)                   # evaluate e^x without a prefix
9
10 from math import exp, sqrt   # importing e^x and √x from 'math'
```

Every function from the `math` module is imported with the `import` command in line 1; however, every function must be used with a prefixed `math` (i.e., module name), as shown in line 2. Using the command in line 4, only the exponential function is imported and can be used without a prefix, as shown in line 5. Everything inside `math` can be imported and used in the program without the prefix (as shown in lines 6-7). In line 10, only two functions are imported from `math` and used with the prefix `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 lines 4 and 10) 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 expressions and/or to create *in situ* documentation. The PYTHON interpreter ignores the code comments. A hash (**#**) symbol is used for *single line* or *partial line comments*; that is, it comments out everything that follows # on the same line.

In lines 9, 12-14, and 16-17 of `example.py`, the hash symbol reserved for partial commenting is basically used to describe an expression (or explain a statement) on the same line. It can also be used to commit an entire line to a comment, as illustrated in lines 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 of lines, the beginning and end of a block are marked with triple-quoted string constants (”’`comments`”’ or `"""comments"""`).

INPUT/OUTPUT DATA: Input/Output (`I/O`) operations in any programming language are fundamental for interacting with users and processing data. To get input data from a user, the `input()` function is used. By default, `input()` reads data as a *string*. However, the input string needs to be converted 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 a `float` by using the `float` function, and the result is assigned to `radius`. To submit 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 :"`.

MULTIPLE STATEMENTS IN A LINE: Python allows multiple code statements on a single line by using a semicolon (`;`) to separate sequentially placed statements (as shown in the code segment below); however, this practice is generally discouraged for readability.

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

LINE CONTINUATION: Most statements in a program will fit into a single line. The assignment statement (`area`$\leftarrow \pi R^2$) in line 13 fits a single line. Likewise, I/O statements in lines 12 and 14 are single-line expressions. Nonetheless, some statements may be too long and complex to fit in a single line. Python allows one to write a single statement in multiple lines, also known as *line continuation*. There are two methods of line continuation: *implicit* and *explicit line continuation*.

Method 1: 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.

```
1  out = ( 1 + 2 + 3 + 4 +   # leftmost '(' bracket is introduced
2        + 5 + 6 + 7 + 8  )  # corresponding bracket ')' is matched here
3  matA = [ [ 1, 2, 3 ],     # leftmost '[' bracket is introduced
4           [ 4, 5, 6 ],
5           [ 7, 8, 9 ] ]    # corresponding bracket ']' is matched here
6  a_dict = {                # leftmost '{' bracket is introduced
7    "brand": "Ford",
8    "model": "Fiesta",
9    "year": 1976
10 }                         # corresponding bracket '}' is matched here
```

Method 2: 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 a continuation mark in lines 2-5.

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

OTHER ELEMENTS OF A PYTHON PROGRAM: A PYTHON program also consists 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 over the flow of a program. *Functions* are reusable named code segments that perform a specific task. These elements are discussed in the following sections.

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, `print()` and `input()` are the most important and useful functions to display and write data on output or read from input devices.

5.1 DISPLAYING OUTPUT

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

5.2 READING INPUT

The `input()` is a function used to supply 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.

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

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 '----------'
```

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*.

5.3.1 FORMATTING WITH STRINGS

With Python 3.6, the formatted string literals, called f-strings, were introduced. This method requires the prefix **f** to create an *f-string* (*see* **Table 1.3** for the list of prefixes). 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 is 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:

```
x = 3; y = 7
name="Mary"
form1 = f"The sum of {x} and {y} is {x + y}."
print(form1)                       # displays 'The sum of 3 and 7 is 10.'
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 lowercase letters to uppercase. In the foregoing examples, the default formatting settings were used. Format specification may be prepared outside a print statement, as in `form1`.

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. In this regard, 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.")
print(f"{name:s} is an Adult" if age >= 18 else "Minor")
```

Here the default type-dependent format widths have been implemented. Also, conditional expressions (also known as the ternary operator) within the **print()** function are implemented using a simple `if-else` logic directly in the format. The output of this segment is

```
Mary is 1.650000m tall and 25-years old.
Mary is an Adult
```

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'`

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
```

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

String alignment and width: Aligning a bunch of data in a tabular form 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 are 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()                     # creates an empty line
print("Line-1\tLine-2")     # \t is used between two strings
Print(" ")                  # creates an line with one black character
print("Line-1\rLine-2")     # \r is used between two strings
```

The output is shown below:

```
Line-1
Line-2                # Line 2 is displayed in a newline

Line-1   Line-2       # Line 2 is displayed next to line 1 after tabbing

Line-1                # carriage return at the end of line-1
Line-2                # Line 2 is displayed after cr
```

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.

In many cases, using these escape sequences may depend on the environment (e.g., terminal, console, or text editor) and might not have the expected visual effect, so be cautious when using them for user interfaces.

6 ARITHMETIC OPERATIONS

Arithmetic operations involve the basic arithmetic operators plus (`+`), minus (`-`), multiplication (`*`), division (`/`), as well as 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. The modulus operator (**x%y**) produces the remainder from the division `x/y` (*see* **Table 1.5**).

```
a = 5; b = 3
print(a **b)            # displays 125
print(15%b," ",15//b)   # displays 0  5
print(16%b," ",16//b)   # displays 1  5
print(17%b," ",17//b)   # displays 2  5
```

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

Operator	Description	Example
ARITHMETIC OPERATORS		
`+, -`	Addition and subractions	`a + b` or `a - b`
`*`	Multiplication	`a * b`
`/`	Division	`a / b`
`%`	finding the remainder (modulo).	`5 % 2 = 1`
`//`	integer division.	`15 // 4 = 3`
RELATIONAL OPERATORS		
`==`	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`
LOGICAL OPERATORS		
`and`	Logical AND operator	`a and b`
`or`	Logical OR operator	`a or b`
`not`	Logical NOT operator	`not (a)`
IDENTITY OPERATORS		
`is`	returns `True` if both variables are the same object	`a is b`
`is not`	returns `True` if both variables are the same object	`a is not b`
COMPOUND OPERATORS		
±=	Addition assignment	`p ±= q` (gives `p = p ± q`)
`*=`	Multiplication assignment	`p *= q` (gives `p = p * q`)
`/=`	Division assignment	`p /= q` (gives `p = p /q`)
`**=`	Exponentiation assignment	`p **= q` (gives `p = p ** q`)
`%=`	Modulus assignment	`p %= q` (gives `p = p % q`)
`//=`	Modulus assignment	`p //= q` (gives `p = p // q`)

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`.

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`.

In **Table 1.5**, the arithmetic, relational, and logical operators are listed. *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**). 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 logical**and** 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 L_1 and L_2 be two logical prepositions. In order for L_1 **and** L_2 to be **True**, both L_1 and L_2 must be **True**. In order for L_1 **or** L_2 to be **True**, it is sufficient to have either L_1 or L_2 to be **True**. When using the unary **not** operator in any logical statement, the logic value is changed to **True** when it is **True** or changed to **True** when it is **True**. These operators can be used to combine multiple expressions.

For example, for given **x=5**, **y=9**, **a=18**, and **b=3**, we can construct the following logical expressions:

```
    (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
    (not x>6 and not y<5 and a>x)  # True
```

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

8 PROGRAM CONTROL OPERATIONS

In Python, program control operations allow you to change the flow of execution within your program. These operations are viewed in four categories: *conditional control* (**if**, **if-elif-else**, **match-case**), *loop control* (**for**, **while**, **continue**, **break**), *error control* (**catch**), and *program termination* (**return**).

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, whose syntax is 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
```

elifs 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.

> Recall that indentation is critical in Python syntax, used to define blocks of code in **if**, **while**, **for**, and so on constructs. This means that the amount of space at the beginning of a line determines whether the line belongs to a certain block. Thus, all lines in a block must be indented by the same number of spaces; a widely recommended standard is 4 spaces. *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., $x \leqslant 10$), and 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')
```

Note that `if` checks the first condition (`year=1`?). If it is `True`, the corresponding block executes. The `elif` statement, which stands for `else if`, allows checking additional conditions if the previous conditions were `False`. The `else` statement executes if none of the preceding conditions are `True`.

8.2 TERNARY CONDITIONAL OPERATORS

Python supports a shorthand way to write an `if-else` statement in a single line, known as the *ternary conditional operator*. The syntax is given as

```
result = val_if_True if condition else val_if_False
```

This statement evaluates *condition* first. If *condition* is *True*, *val_if_True* is executed; otherwise, *val_if_False* is evaluated. Note that *val_if_True* and *val_if_False* must be of the same type, and they must be simple expressions rather than full statements. The following example, which determines the maximum of a pair of integers, illustrates the use of a ternary operator:

```
a = 10; b = 20
c = a if a > b else b;     # c = max(a, b)
print(c);                  # c becomes 20
d = (13 if b<=25 else 25) if a>6 else 100
print(d);                  # d becomes 13
```

In evaluating `c`, the condition `(a > b)` is evaluated, and since `a<b`, the value of `c` is set to `b`, i.e., 20. In evaluating `d`, the condition `(b <= 25)` is evaluated as `(a > 6)`, which yields `13` since the condition is `True`.

8.3 THE `match-case` CONSTRUCTION

The `match-case` statement was introduced in Python 3.10 and is used for pattern matching. It is similar to `switch` (C/C++, Matlab), `select case` (Fortran 95), or `Which` (Mathematica) constructions, and it is an alternative to the `if-elif-else` ladder. A `match-case` construction allows a multi-decision case to be executed based on the value of a switch variable. It is a cleaner alternative to using multiple `if` constructions when you have many conditions based on a single variable.

Using `match-case` construction can improve the clarity and maintainability of the code when dealing with multiple conditions based on a single variable. The general form of the `match` statement is as follows:

```
match variable
    case value1:        # if variable = value1
       STATEMENTS-1
    case value2:        # if variable = value2
       STATEMENTS-2
    ....
    case _:             # if variable is not = value1, value2, ...
       STATEMENTS-n
```

where `value1`, `value2`, and so on depends on the type of `variable`. Each `case` is checked sequentially, and when a `value` matches, the corresponding block of code statements is executed. The underscore in the last case acts as a wildcard, matching anything if no other `value` matches (similar to a `default case` in a `switch`).

The following Python code segment uses `year` to execute `match-case` construct. For the case of `year=1`, `Freshman` is displayed; for `year=2`, `year=3`, and `year=4`, `Sophomore`, `Junior`, and `senior` are displayed, respectively. If `year` corresponds to none of the above, the message displayed is `Graduated`.

```
year = 3                        # year is initialized
match year:
    case 1:
        print("Freshman")
    case 2:
        print("Sophmore")
    case 3:
        print("Junior")         # Output is  Junior
    case 4:
        print("Senior")
    case _:
        print("Graduated")
```

This construction can be used to match complex data structures like tuples or lists. Also, additional conditions (guards) can be implemented to a `case` pattern using `if` condition. The case will only match if both the pattern and the guard's condition are satisfied. Guards make pattern matching more flexible and allow one to impose more specific constraints, as shown in the example below:

```
x=9
match x:
    case x if x > 0 and x <= 5:   # first interval 0 < x <= 5
        print("x is in the first interval")
    case x if x > 5 and x <=10 :  # second interval 5 < x <= 10
        print("x is in the second Interval")
    case x if x <0 :              # x < 0
        print("x is a negative number")
    case _:                       # x > 10
        print("x is a number greater than 10")
```

9 CONTROL (LOOP) 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. In PYTHON, there are basically two control constructions: `while` and `for` constructs.

9.1 while CONSTRUCTION

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 a `while` loop, *condition* is evaluated before the code block. The block of code is executed as long as the specified *condition* remains `True`. 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 executes following indented block
    print('n=', n)     # write n on output device
    n += 3             # increment n by 3
```

This code generates integer numbers starting from 0 to 10, skipping by 3, i.e., 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`.

A `while` construction, functioning as **Repeat-Until**, can have the following 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` is `False`. The following loop performs the same task using a `while` loop until $n \geqslant 10$.

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

Note that not only the location of the *condition* but also the *condition* itself has been changed; however, the output is the same. Nested-`while` loops can also be constructed as `if` constructions.

9.2 for CONSTRUCTION

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

```
for  loop-variable in sequence: # line ends with a colon
      STATEMENT(s)              # block of code to be executed
```

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

```
seqn={1,5,6,-4,9}    # seqn is a list of integer numbers
for i in seqn:       # i sequentially takes the values in seqn
     print("i=",i)   # displays every i in a new line with label 'i='

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

The out of the second loop is

```
i=1, j=0
i=2, j=4
i=3, j=6
i=4, j=3
```

9.2.1 `range` FUNCTION IN `for` LOOPS

The **`range()`** function is 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] )
```

which generates numbers starting from *start* (i.e., the 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 of *stop*.

Following is an example of nested **`for`** loops:

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

Lines 2-3 make up the block of the outer loop (that runs over *i*), and line 3 is the block of the inner loop (that runs over *j*). So the **`print`** statement will be executed for all valid of `i` and `j`. The code output is as follows:

```
i=0, j= 0
i=0, j= 1
i=0, j= 2
i=0, j= 3
i=1, j= 0
i=1, j= 1
i=1, j= 2
i=1, j= 3
```

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`.

As mentioned earlier, the indentation in PYTHON is important in that it marks where the block starts and where it ends. To illustrate this, consider the following code segment:

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

In this code, lines 2-4 make up the block of the outer loop (running over `i`), while the block of the inner loop (running over `i`) 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
((( i=0  j=3 )))
*** j = 0
*** j = 1
*** j = 2
*** j = 3
((( i=1  j=3 )))
*** j = 0
*** j = 1
*** j = 2
*** j = 3
((( i=2  j=3 )))
```

Consider the following code segment:

```python
for i in range(2):
    for j in range(4):
        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 = 0
((( i=1  j=0 )))
*** j = 1
((( i=1  j=1 )))
*** j = 2
((( i=1  j=2 )))
```

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.

```
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 as 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 the **continue** statement is illustrated in the following code segment.

```
for i in range(2):
    for kount in range(5):
        if kount == 3:
            continue         # skips following block for kount = 3
        # Code block is placed here, indented in line with the print statement
        print('inner loop for kount ',kount)
    print('Outer loop for i ',i)
```

Here, the inner **for** loop skips the loop block for **kount=3** only. The loop control is transferred to the outer loop. The code output becomes

```
inner loop for kount  0
inner loop for kount  1
inner loop for kount  2
inner loop for kount  4
Outer loop for i  0
inner loop for kount  0
inner loop for kount  1
inner loop for kount  2
inner loop for kount  4
Outer loop for i  1
```

PASS: A **pass** statement is a null operation. It allows you to write code constructions that are not yet implemented or require no action without causing syntax errors. In other words, when an external condition is triggered, it allows the condition to be processed without affecting the loop in any way.

The difference 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, the `break` statement exits the loop completely and moves on to the code that follows the loop.

In the following example for finding the roots of a quadratic equation, the `if` construct is executed only for `d>=0` (case of real roots). The code skips the case of imaginary roots and does not yield an *error* or *warning*.

```
from math import sqrt   # imports the square root function from math
a=...; b=...; c=...     # arbitrary values of a, b, and c are supplied
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
```

10 VECTOR AND MATRIX OPERATIONS

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

10.1 DEFINING ARRAYS

PYTHON lists are flexible and can hold elements of different data types, including numbers, strings, and other objects. They can also behave like arrays if used for numerical data; that is, vectors can be represented with *lists*.

LISTS: A list is defined using square brackets [], and its elements are separated by commas. Its elements can be accessed using zero-based indexing. Since lists are mutable, they can be modified as illustrated below:

```
arr_a = [7, "a", 2.7183]    # define a list (array) of mixed data
arr_b = [9, -2, 3, 11, 6]   # define a list (array) of integers
print(arr_a[1])             # display arr_a(0), which is a
print(arr_b[0])             # display arr_b(1), which is 9
arr_a[1] = 3.14             # modify 2nd element of arr_a
print(arr_a)                # displays arr_a, which is [7, 3.14, 2.7183]
```

MODULE array: Several modules and libraries support arrays and array operations. As a part of the standard library, the `array` module provides a basic array type with support for efficient storage and manipulation of the same type of data. Thus, to create an array in PYTHON using the `array` module, it needs to be imported and used 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`**. The **`array()`** function accepts *type code* and *initializer* as parameter values and returns an object of the 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 the 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])
```

MODULE NumPy: For numerical computations, in Python, a matrix can be defined as a 2D list or 2D array. It is more efficient to use arrays from the `NumPy` library. The `NumPy` arrays are homogeneous (i.e., they contain elements of the same data type), and they allow for more advanced operations. For the complete list of functions available, visit the official `NumPy` site.

Before the arrays are defined and used in array operations, the `NumPy` module must be imported as follows:

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

The syntax for creating an array is given as follows:

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

A vector (one-dimensional array) or a matrix can be created as shown below:

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

ar = np.array([8, 6, 4, 3])          # create a row array, ar_i <- ar_{1,i}
ac = np.array([[2], [-3], [4], [2]]) # create a column array, ac_i <- ac_{i,1}
M  = np.array([[1, 2],               # create a 2 x 2 square matrix, M.
               [7,-3]])              # using 'array' structure
B  = np.mat([[1, 2],                 # create a 3 x 2 rectangular matrix
        [3, 4],
        [5, 6]])
print("row array=",ar)               # displays row array
print("column array=\n",ac)          # displays column array
print("Matrix [M]=\n",M)             # displays matrix M
```

Table 1.6: Some of the common matrix creation functions available in **NumPy**.

Function	**Description**
`zeros()`	returns an array of specified shape and type, filled with zeros
`ones()`	returns an array of specified shape and type, filled with zeros
identity()	returns an identity matrix of specified size
`linspace()`	Return evenly spaced numbers over a specified interval
`shape`	return the shape of an array
`size`	Return the number of elements along a given axis

```
print("Matrix [N]=\n",N)              # displays matrix N
print("1st: ",ar[0]," last: ",ar[3]) # displays 1st and last elements
print("trace(M)= ",M[0,0]+M[1,1])    # displays trace of matrix M
```

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]]
1st:  8  last:  3
trace(M)=  -2
```

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.

There are some useful functions for creating arrays in `NumPy` module. Several of the commonly used functions are listed in **Table 1.6**. The properties of the arrays, such as the shape, data type, ordering of data, etc., can be specified while creating the arrays. The users are advised to refer to the official `NumPy` web site.

Several examples of using array creation functions for constructing special arrays are illustrated below:

```
import numpy as np   # importing numpy for matrix operations
arr_a = np.zeros(5,dtype=int) # 1D row array of length 5 filled with zeros
arr_b = np.ones((3,1))        # 1D column array of length 3 filled with zeros
arr_c = np.ones((3,3))        # 2D array of length 9 filled with zeros
arr_d = np.identity(2)        # set up a 2x2 identity matrix
print("Array a=",arr_a)
print("Array b=",arr_b)
print("Array c=",arr_c)
```

```
print("size of arr_a=", np.size(arr_a)) # display size of arr_a
print("size of arr_b=", np.size(arr_b)) # display size of arr_b
print(np.linspace(1,2,num=5)) # create linearly spaced array with 5 elements
print(np.linspace(1,2,5))     # between 1 and 2 ('num=' can be omitted)
print("Matrix I=\n",arr_d)
```

The output of the code is as follows:

```
Array a= [0 0 0 0 0]
Array b= [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
 Array c= [[1]
 [1]
 [1]]
size of arr_a= 5
size of arr_b= 9
[1.   1.25 1.5  1.75 2.  ]
[1.   1.25 1.5  1.75 2.  ]
Matrix I=
 [[1. 0.]
 [0. 1.]]
```

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

In the 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 are 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)             # displays matrix M
print(M[1,2])        # displays 2nd row, 3rd column element
print(M[2:])         # displays 3rd and 4th rows of M
print(M[0:5,0:2])    # displays 1st two column of M
print(M[1][3])       # displays 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
```

10.3 ALGEBRAIC OPERATIONS WITH ARRAYS

`NumPy` linear algebra functions relying on BLAS and LAPACK routines provide efficient low-level implementations of linear algebra algorithms. `SciPy` library also contains a `linalg` submodule, and there is overlap in the functionality provided by `SciPy` and `NumPy` submodules. `SciPy` contains some of the functions (related 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 used throughout the programs involving vector and matrix operations. Once an array is defined, `NumPy` functions allow *element-by-element* operations, as shown in the following array operations:

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

arr_a = np.array([9, -2, 3, 11, 6])
arr_a += 2             # add 2 to every element of arr_a
print("Array a=\r",arr_a,"\r")) % carriage returns '\r' are added

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 : \n",C)

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

Table 1.7: 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`

```
print ("Matrix D is :\n ",D)

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

F = A*B    # use multiply() to perform element-by-element multiplication
print ("Matrix F is :\n",F)
```

Here, we may use the `np.multiply(A,B)` function, which gives the same result as `A*B`. The output of the code is

```
Array a=
 [11  0  5 13  8]

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]]
```

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 1.7**. The following code performs the `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]])
A = np.zeros((4, 4), dtype=int)    # 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",2*M+3*N)  # displays A, does not require initialization
print("A=\n",A)        # displays all elements of A
```

This code evaluates the matrix operations using the Python functionality (i.e., `2*M+3*N`) and codes 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.]]
```

The following are some examples of implementations of `Numpy` array functions:

```
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)               # displays all elements of A
print("A*A=\n",A*A)           # displays element-by-element A*A
print("A.A=\n",np.dot(A, A)) # displays A*A matrix multiplication
b = np.array([3, -3, 2])      # define a row vector b
print("A.b=",np.dot(A, b))    # displays A.b matrix-vector multiplication
print("b.b=",np.dot(b, b))    # displays b.b dot product
print("Array of zeros=\n",np.zeros((4)))  # displays array of zerors
print("Matrix of ones=\n",np.ones((2,4))) # displays 2x4 array of ones
print("Identity matrix=\n",np.eye((3)))   # displays 3x3 identity matrix
print(np.linspace(3,4,5))                 # displays [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 are carried out using `np.dot()` or can also be calculated using `np.matmul()` or using the operator `@`. However, `np.matmul()` and `np.dot()` behave differently for arrays with more than three dimensions.

10.4 OBJECT ATTRIBUTES

In NumPy, arrays (i.e., objects) come with a variety of attributes that can help the 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 1.8**.

Here are some examples of applying object attributes:

```
import numpy as np          # importing numpy for matrix operations
A = np.array([[1, -1, 1], [1, 0, 2]])  # intialize 2x3 matrix 'A'
b = np.array([2,-1,3,2,-4])            # intialize a vector 'b'
print(A.ndim, b.ndim)       # displays 2  1
print(A.shape, b.shape)     # displays (2, 3) (5,)
print(A.size, b.size)       # displays 6 5
print(A.mean(),b.mean())    # displays 0.666666666 0.4
print(A.min(), b.min())     # displays -1 -4
print(A.max(), b.max())     # displays 2 3
print(A.T)                  # displays transpose of A (3x2)
print(b.sort())             # displays -4 -1  2  2  3]
print(A.reshape((3,2)))     # displays [[ 1 -1] [ 1  1] [ 0  2]]
```

The linalg module provides a comprehensive set of tools for linear algebra operations, making it a fundamental library for scientific computing in Python. Several most common **linalg** object attributes are listed in **Table 1.8**. For more and detailed information, the reader is referred to numpy.linalg. Let **A** and **B** be two matrices and **b** be a vector. Implementation of some of the **linalg** attributes is illustrated below:

```
import numpy as np      # importing numpy for matrix operations
A = np.array([[7, 2], [3, 1]])        # intialize A, a 2x2 matrix
B = np.array([[1, 2, 4], [5, 3, 1]])  # intialize B, a 2x3 matrix
b = np.array([8, 43])                  # intialize b, a 3x1 vector
print(np.linalg.inv(A))               # displays inverse of A
print(np.linalg.eig(A))               # displays eigenpairs of A
print(np.linalg.solve(A, b),'\r')     # solves Ax=b matrix equation
print(np.linalg.matmul(A,B),'\r')     # displays A*B product
print(np.linalg.norm(A,1))            # displays L1 norm of A
print(np.linalg.norm(A,np.inf))       # displays Linf norm of A
print(np.linalg.norm(A, 'fro'))       # displays Frobenius norm of A
print(np.linalg.norm(b),'\r')         # displays norm of vector b
print(np.linalg.matrix_power(A, 3))   # displays A*A*A=A^3
```

The code output is as follows:

```
[[ 1. -2.]
[-3.  7.]]
EigResult(eigenvalues=array([7.87298335, 0.12701665]),
eigenvectors=array([[ 0.91649636, -0.2794051 ],
   [ 0.40004303,  0.96017331]]))
    [ 2. -3.]

[[17 20 30]
 [ 8  9 13]]
```

Table 1.8: Some of the basic matrix operations provided by `NumPy` attributes.

Function	**Description**
Instance variable	**Output**
`.size`	number of elements in array
`.shape`	number of rows, columns, etc.
`.ndim`	number of array dimensions
`.dtype`	data type of array elements
`.T`	transposed version of the array
`.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

```
10.0
9.0
7.937253933193772
8.54400374531753

[[433 126]
 [189  55]]
```

11 FUNCTIONS IN PYTHON

Creating and using functions is a fundamental part of writing structured and modular code. Functions help organize code into smaller reusable blocks, making it easier to read, maintain, and debug. Functions also prevent code repetitions.

In PYTHON, functions are viewed in two categories: (i) *The Standard Built-in Functions*, those provided with the PYTHON 3.13.0 and other functions provided by specialized libraries such as `NumPy`, `Pandas`, `SciPy`, etc., and (ii) *user-defined functions*, those prepared by the user.

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. The `int()`, `float()`, and `str()` functions are type conversion functions, which convert values from one type to another. For example,

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

For mathematical operations, the `math` library can be used. This library provides functions on the number representations, power and logarithmic, trigonometric and inverse trigonometric, angular conversions, hyperbolic and inverse hyperbolic functions, constants, etc. However, the user should import the `math` library (*see* Section 3 on how to import modules) before using any one of its functions.

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 of an algorithm under one function, which also helps make large programs easier to manage.

General Structure: 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`. A *function header* (i.e., function definition line) includes the *function name* (identifier) and the parenthesized list of input and output parameters, and it ends with a colon (`:`). A function usually has one or more *input parameters*, which are supplied by the user, and *output parameters*, which are the returned results of the function once it has completed its task. As PYTHON is a dynamically typed language, the types of the input and output parameters need not be designated beforehand. Following lines of statements (*function body*) form the body of the intended task (i.e., function).

The syntax for an n-parameter function is given as

```
def function_name (p1, p2, ..., pn):
       """ description string """
       statements (code block)
    return
```

where `p1`, `p2`, ..., `pn` are the input parameters (communicated with the calling program) of the function that are used to pass data into `function_name`. Note that if the function block is not indented properly, it will yield an indentation error. Normally the function *variable* list is referred to as the *parameter* list. Sometimes the terms "argument" and "parameter" are used interchangeably in conversation and documentation. However, *parameters* are variables defined by a function receiving the values when the function is called. On the other hand, *arguments* are the values of the variables sent to the function.

A function body is an indented block indicating the main body of the function, which consists of three parts: (1) An *optional* `description string` (a triple-quoted string), describes the function, its arguments, exceptions, algorithm implemented, etc.; (2) A **`code block`** that includes step-by-step instructions that the function will carry out when it is called; (3) A `return` statement is used to stop execution, and the *expressed value* (if any) should be returned to the caller. `Return` statement(s), which contain output parameter(s) to be returned after the function is called. As we will discuss later, any data type may or may not be returned.

Following code illustrates creating and using a PYTHON function:

```
1  import math                         # import math library to use sqrt
2
3  def sumsq(a, b):                    # define function 'sumsq'
4  # function to calculate the square root of a^2 + b^2
5    dsqr = a * a + b * b;             # find sums of the squares of a and b
6    d    = math.sqrt(dsqr);           # find square root of dsqr
7    return d                          # return d to the caller
8
9  a = sumsq(4,3)
10 print(" a=",a)                      # displays =>  a= 5.0
```

In this code, the function **`sumsq`**, with input parameters (**`a`** and **`b`**), calculates the square root of the sums of the squares of the parameters, i.e., $d = \sqrt{a^2 + b^2}$. Note that the **`return`** statement (in line 7) is accompanied by the parameter **`d`** meaning that the function only returns the value of **`d`** (a single value), even though **`dsqr`** is also calculated in the function block.

Multiple Output (Return) Values: A PYTHON function may have multiple output parameters. If a function returns multiple values, the output parameters are automatically packed into a tuple. Thus, when such a function is called, the results (outputs) need to be unpacked, separated by commas. The following version of the code illustrates creating and using a PYTHON function with two output parameters:

```
import math                         # import math library to use sqrt

def sumsq(a, b):                    # define function 'sumsq'
# calculates the square root of a^2 + b^2
  dsqr = a * a + b * b;             # find sums of the squares of a and b
  d    = math.sqrt(dsqr);           # find square root of dsqr
  return dsqr, d                    # return both dsqr and d

a, b = sumsq(4,3)
print("dsqr=",a," d=",b)            # displays =>  dsqr= 25  d= 5.0
```

In this code, the **`return`** statement is accompanied by "**`dsqr, d`**", which means that the function returns the calculated values of **`dsq`** and **`d`**, respectively.

Use of Multiple Returns: In general, a **`return`** statement is required if the function is to send a result back to the caller module. If an output argument is not explicitly specified with a **`return`** statement, the `None` 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 piecewise defined function:

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

The following is the PYTHON function code for the function **saw(x)**, demonstrating the use of multiple returns. The code branches into three blocks, depending on the value of **x**, and the value calculated at the end of each block is returned after the operations are completed.

```
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
```

Local and Global Variables: In PYTHON, variables are classified into two main types based on their scope: *local* and *global variables*. Local variables are the ones defined and accessed within that function, and they exist only for the duration of the function's execution. On the other hand, global variables are the ones defined outside of the functions and can be accessed from any function within the same program. These variables exist for the duration of the program's execution. That is why it is more suitable to prepare and use functions that define variables locally. 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 program.

Consider the following example of a **function** 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))    # displays 1.6664919
print(r)             # displays 0.4
```

where the global variable is assigned the value **r=0.4** outside the function, while n is the only input argument passed to **geosum**. In fact, we could assign a value to n outside **geosum** before calling the function (without the argument, **geosum()**), and the function would still perform its intended task. It is also possible to define local and global variables with the same name, as in the modified function illustrated below:

```
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))    # displays 1.998046875
print(r)             # displays 0.4
```

Here, the value $r = 0.5$ defined in **geosum** is a local variable of the function (i.e., confined to the function only), and it is used in calculating the returned value. In other words, when a 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 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 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. On the other hand, 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:

```
1  r = 0.4
2  def geosum(n):
3      global r
4      r = 0.5
5      return (1-r**n)/(1-r)
6
7  print(geosum(10))   # displays 1.998046875
8  print(r)            # displays 0.5
```

In this case, the keyword **`global`** instructs PYTHON not to define **`r`** as a new local variable. The value in line 4 overrides the assignment statement for **`r`** in line 1. That is why the displayed result becomes **`r=0.5`**.

> 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.

A Void Function: A void function is a function without a **`return`** (i.e., a value). 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`** displays a message to the user informing him that the data to be entered must be of a complex type; that is, no computations, evaluations, decision makings, etc., are performed. The second function **`fx`** carries out the **`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()          # displays 'Entered value must be a complex number'
print(fx(a=3.))   # displays 'None'
```

When a function is not terminated with the **`return`** statement, PYTHON automatically returns a variable with the 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:

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

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

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

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

In the 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, **par1**, **par2**, ..., **parn** are the arguments, and *expression* is the expression that can fit on a single line.

Some examples of lambda functions 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/