Lab Manual for Numerical Analysis

Lab No. 1

INTRODUCTION TO PYTHON

BAHRIA UNIVERSITY KARACHI CAMPUS

Department of Software Engineering

NUMERICAL ANALYSIS

LAB EXPERIMENT # 1

Introduction to Python

OBJECTIVE:

This lab aims to familiarize students with Python's fundamentals, syntax, and basic operations, enabling them to confidently write and execute simple programs, building a strong foundation for programming and problem-solving.

What is PYTHON?

Python is a versatile and widely-used programming language known for **its simplicity, readability**, and **flexibility**. Developed by Guido van Rossum over the period from 1985 to 1990, Python has become a popular choice among programmers and developers. It offers a range of features, including an **interactive** and **interpreted nature**, support for **object-oriented programming**, and a **high-level syntax** that emphasizes code simplicity and clarity.

Python's syntax is designed to resemble the **English language**, making it accessible and readable for both beginners and experienced programmers. It is renowned for its ability to work across various platforms, including **Windows, Mac, Linux,** and even on **embedded systems** like Raspberry Pi.

Why Python?

Python's popularity is driven by a multitude of factors that make it an excellent choice for a wide range of applications. Its **simplicity** and **readability** reduce development time and make maintenance easier. Python's **extensive standard** library provides ready-to-use modules and functions for common tasks, saving developers from reinventing the wheel.

Additionally, Python is **open-source,** fostering a collaborative and dynamic community that continuously enhances the language. It is particularly well-suited for **web development, data analysis, scientific computing, artificial intelligence, automation**, and more. Its adaptability to procedural, object-oriented, or functional programming approaches offers flexibility and versatility, accommodating developers at all skill levels and a wide array of programming needs.

Python Syntax and Basics

a. Variables in Python:

In Python, **variables** are **indispensable components** used for storing and managing data. They are created by assigning a value to a name, and Python's dynamic typing allows them to adapt to various data types, such as integers, floating-point numbers, strings, lists, and more.

Variable names must follow specific rules, commencing with a letter or an underscore and comprising letters, numbers, and underscores.

Python's **case-sensitive** nature treats variable names distinctly based on their letter casing. For instance, "myVariable" and "myvariable" are considered separate variables.

Here's an **example** showcasing the assignment and use of variables in Python:

```
age = 30
name = "Alice"
height = 5.8
fruits = ["apple", "banana", "cherry"]
```

In this example, we've declared and assigned values to variables for **age, name, height**, and a **list of fruits**, showcasing the versatility and simplicity of Python's variable usage.

b. User Input & Output in Python:

User input and **displaying output** are fundamental aspects of Python programming that enable interaction between the program and the user. To receive input from the user, Python provides the **input()** function, which prompts the user for text input and stores it as a string. On the other hand, to display output to the user, the **print()** function is employed, allowing you to showcase information, variables, or results in the console.

The syntax for obtaining user input is as follows: **user_input = input("Prompt: ")**, where "Prompt" is a message displayed to user.

For output, use **print("Message or variable")**, where "Message" can be a text message or a variable to display.

Here's an illustrative example:

```
user_name = input("Enter your name: ")
print("Hello, " + user_name + "! Welcome to Python.")
```

In this code snippet, the **input()** function gathers the user's name, and the **print()** function greets them by displaying a personalized message, creating a basic interaction in a Python program.

c. Operators in Python

Operators in Python are special symbols or keywords that perform various operations on one or more operands, which can be values or variables. Python supports a wide range of operators, each designed for specific tasks. Here are some common types of operators in Python with examples:

i. **Arithmetic Operators:** Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, division, modulus, and exponentiation.

```
a = 5
b = 2

addition = a + b  # 7
subtraction = a - b  # 3
multiplication = a * b  # 10
division = a / b  # 2.5
modulus = a % b  # 1
exponentiation = a ** b  # 25
```

ii. **Comparison Operators:** Comparison operators compare two values and return a Boolean result (True or False).

```
x = 5
y = 10

is_equal = x == y  # False
is_not_equal = x != y  # True
is_greater = x > y  # False
is_less = x < y  # True
```

iii. **Logical Operators:** Logical operators perform logical operations on Boolean values.

```
has_apple = True
has_banana = False

both_fruits = has_apple and has_banana  # False
either_fruit = has_apple or has_banana  # True
no_banana = not has_banana  # True
```

iv. **Assignment Operators:** Assignment operators are used to assign values to variables.

```
x = 5
x += 3  # x is now 8
```

d. Loops in Python

Loops are fundamental control structures in Python (and in programming in general) that allow you to repeatedly execute a block of code. Python provides **two main** types of loops: the **"for"** loop and the **"while"** loop. These loops are used to automate repetitive tasks and iterate through sequences like lists, strings, or ranges.

i. **For Loop:** A "for" loop is typically used when you know beforehand how many times you want to execute a block of code. It iterates over a sequence (such as a list or a range) or any iterable object.

```
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print(fruit)
```

In this example, the "for" loop iterates through the list of fruits, and the code inside the loop (in this case, printing each fruit) is executed for each item in the list.

ii. **While Loop:** A "while" loop is used when you want to execute a block of code as long as a certain condition is true. It repeatedly checks the condition before each iteration.

```
count = 0

while count < 5:
    print(count)
    count += 1
```

Here, the "while" loop continues executing as long as the "count" is less than 5. The count is incremented inside the loop, and the loop terminates when the condition becomes false.

iii. **Loop Control Statements:** Python provides loop control statements like "break" and "continue" to modify the flow of loops. "Break" is used to exit a loop prematurely, and "continue" is used to skip the current iteration and move to the next.

```
for number in range(1, 11):
    if number == 5:
        break  # Exit the loop when number is 5
    print(number)
```

In this example, the "break" statement is used to exit the loop when the number is 5.

iv. **Nested Loops:** Python allows you to nest loops inside each other, creating a loop within a loop. This is useful for working with multidimensional data or solving complex problems.

```
for i in range(3):
    for j in range(3):
        print(i, j)
```

This nested loop prints combinations of "i" and "j" values, resulting in nine pairs.

Loops are invaluable tools in Python programming, enabling you to automate repetitive tasks, process data efficiently, and create dynamic programs that respond to changing conditions. Understanding how to use loops effectively is a crucial skill for any Python developer.

e. Conditional Statements in Python

Conditional statements, also known as decision-making or branching statements, are essential in programming as they allow you to execute different blocks of code based on specified conditions. Python provides several conditional statements, with the most common ones being "if," "elif" (short for "else if"), and "else."

i. **If Statement:** The "if" statement allows you to execute a block of code only if a specified condition is true.

```
x = 10

if x > 5:
   print("x is greater than 5")
```

In this example, the code inside the "if" block is executed because the condition **x > 5** is true.

ii. **If-Else Statement:** The "if-else" statement provides an alternative block of code to execute if the condition in the "if" statement is false.

```
x = 3

if x > 5:
   print("x is greater than 5")
else:
   print("x is not greater than 5")
```

Here, since the condition **x > 5** is false, the code inside the "else" block is executed.

iii. **Elif Statement:** The "elif" statement is used when you have multiple conditions to check. It allows you to test additional conditions if the previous ones are false.

```
x = 5

if x > 10:
   print("x is greater than 10")
elif x > 5:
   print("x is greater than 5 but not 10")
else:
   print("x is not greater than 5")
```

In this example, the "elif" statement is used to test a second condition when the first one is false.

4. Nested Conditional Statements: You can nest conditional statements inside each other to handle more complex scenarios.

```
age = 18
has_id = True

if age >= 18:
    if has_id:
        print("You are eligible to enter the club.")
    else:
        print("You need an ID to enter.")
else:
    print("You are too young to enter.")
```

This nested example first checks if the person is old enough (age >= 18) and then, if they have an ID.

Conditional statements are fundamental to making decisions and controlling the flow of your Python programs. They allow your code to react dynamically to different situations, making your programs more versatile and powerful.

f. Arrays in Python

In Python, arrays are often implemented using lists or the more specialized **array** module. While Python doesn't have a built-in array data type like some other programming languages, lists are flexible enough to serve as arrays for most purposes. Here's an overview of arrays in Python:

i. **Lists as Arrays:** Lists in Python are versatile and can be used as arrays to store collections of elements. You can create lists of any data type, including numbers, strings, or even other lists.

```
numbers = [1, 2, 3, 4, 5]
names = ["Alice", "Bob", "Charlie"]
```

ii. **Accessing Elements:** You can access elements in a list using zero-based indexing, just like with arrays in many other programming languages.

```
first_number = numbers[0]  # Accessing the first element
second_name = names[1]  # Accessing the second element
```

iii. **Modifying Elements:** Lists are mutable, so you can change their content by assigning new values to specific elements.

```
numbers[0] = 10  # Modifying the first element
names[1] = "Bobert"  # Modifying the second element
```

iv. **Working with Arrays:** You can perform various operations on arrays/lists, including adding elements, removing elements, slicing, and more.

```
numbers.append(6)  # Adding an element to the end
names.pop(0)  # Removing the first element
sliced_numbers = numbers[1:4]  # Slicing a portion of the array
```

Arrays/lists in Python are flexible and adaptable, making them suitable for a wide range of data storage and manipulation tasks. Whether you need to work with simple lists or more complex array structures, Python offers the tools and libraries to handle your data efficiently.

g. Function in Python

Functions are a fundamental concept in Python (and in programming in general). They are reusable blocks of code that perform a specific task or set of tasks. Functions help in modularizing code, improving code readability, and enabling code reuse. Here's a comprehensive overview of functions in Python:

i. **Defining Functions:** You define a function in Python using the **def** keyword, followed by the function name and parentheses. Any arguments (input) the function accepts are placed within the parentheses, and a colon **:** marks the beginning of the function block.

```
def greet(name): def greet(name):
   print("Hello, " + name)
```

ii. **Calling Functions:** To use a function, you call it by its name and provide any required arguments.

```
greet("Alice")  # Calls the greet function with "Alice" as the argument
```

iii. **Function Arguments:** Functions can accept zero or more arguments. Arguments are used to pass data into the function for processing.

```
def add(x, y):
   result = x + y
   return result
```

4. Return Statement: Functions can return values using the **return** statement. The returned value can be stored in a variable or used directly.

```
sum_result = add(5, 3)  # Calls the add function and stores the result in sum_result
```

Functions are a crucial building block in Python programming. They promote code organization, reusability, and modularity, making complex tasks more manageable. Understanding how to create, use, and work with functions is essential for writing efficient and maintainable Python code.

Lab Tasks:

1. Write a Python program to calculate the roots of a quadratic equation of the form ax2 + bx + c = 0, where a, b, and c are coefficients provided by the user.

2. Write a Python program that calculates the factorial of a non-negative integer n provided by the user.

3. Write a python program to find the maximum and minimum number in the following array {12,56,34,2,56,98,6,54,6,54}

4. Write a Python program that checks whether a given positive integer n provided by the user is a prime number or not.

5. Write a python program that takes two matrix (2 by 2) from user, and perform addition, subtraction, multiplication and division on them.