**PYTHON PROGRAMMING**

**UNIT-1**

**COMPUTATIONAL THINKING AND PROBLEM SOLVING**

#### **1.1 Python Introduction**

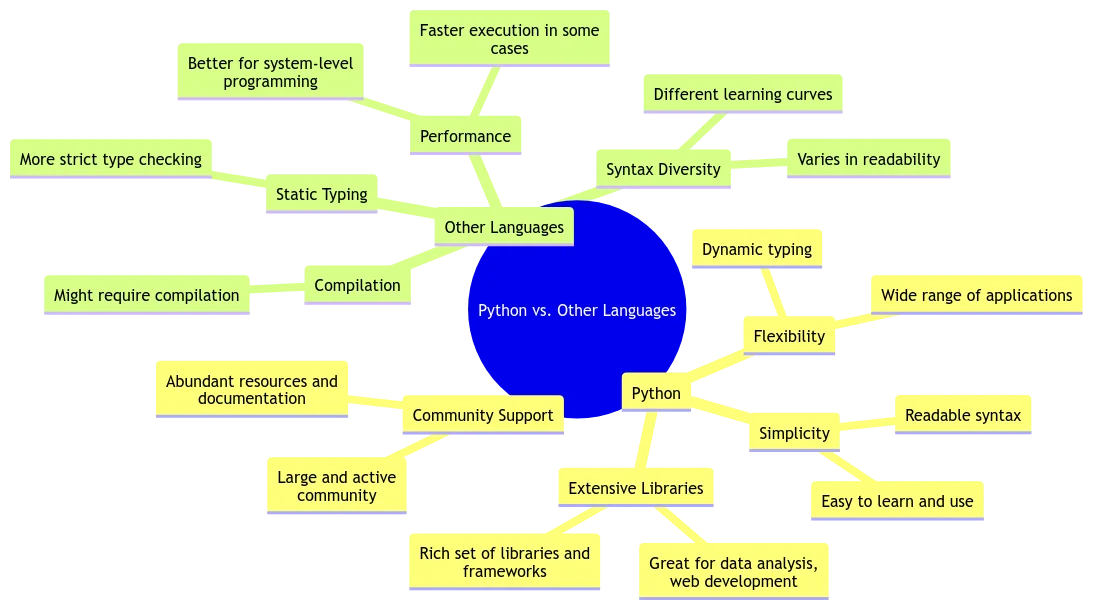
**1.1.1 Definition:** Python is a high-level, interpreted programming language known for its simplicity and readability. It allows developers to focus on solving problems rather than worrying about the technical details of computer hardware. Python supports multiple programming paradigms, including object-oriented, procedural, and functional programming. Its versatility and ease of learning have made it one of the most popular programming languages today, widely used in fields such as web development, data analysis, machine learning, and more.

**1.1.2 History of Python:**

* **Guido van Rossum** created Python in the late 1980s, and it was officially released in 1991.
* **Python 1.0**: First version of Python released on February 20, 1991.
* **Python 2.0**: Released on October 16, 2000, with significant features like garbage collection, list comprehensions, and more.
* **Python 3.0**: Released on December 3, 2008. It is not backward compatible with Python 2, which resulted in a shift toward Python 3 adoption.

**1.1.3 Why Learn Python:** Python is highly recommended for beginners and experts alike due to the following reasons:

1. **Ease of Learning**: Python has a simple syntax, making it easy to learn.
2. **Versatility**: Python is used for a wide range of applications—web development, data science, machine learning, etc.
3. **Strong Community and Resources**: A vibrant community and rich documentation make learning and problem-solving easier.
4. **Job Market Demand**: There is a high demand for Python developers, especially in fields like machine learning, data science, and web development.
5. **Cross-Platform**: Python can run on any major operating system, making it portable across platforms.
6. **Extensive Libraries/Frameworks**: Python offers frameworks like Django, Flask, TensorFlow, pandas, etc., which save development time.
7. **Supports Multiple Paradigms**: Python supports object-oriented, procedural, and functional programming.

****

**1.1.4 Python vs Other Programming Languages:** Python stands out due to its clean and simple syntax, dynamic typing, and wide range of applications compared to other languages like Java or C. While Java is more performance-focused, Python’s ease of use makes it highly popular for rapid development. Unlike C, which requires manual memory management, Python automatically handles memory through garbage collection.

**1.1.5 Features of Python:**

* **High-Level Language**: Python abstracts away complex hardware management tasks.
* **Object-Oriented**: It supports OOP principles like inheritance, polymorphism, and encapsulation.
* **Scalable**: Python can be used for both small scripts and large applications.
* **Extensible**: Python code can be integrated with languages like C or C++.
* **Portable**: Python can run across multiple platforms with minimal changes.
* **Easy to Learn and Read**: Simple, clean, and readable syntax makes Python easy to understand.
* **Memory Management**: Python uses automatic memory management (garbage collection).
* **Extensive Libraries**: Python offers a vast standard library with pre-built modules and functions.

**1.1.6 Applications of Python:** Python is used in various fields, including:

* **Web Development**: Building websites and web apps using frameworks like Django and Flask.
* **Software Development**: Python is used to build both simple and complex applications.
* **Data Science**: Python is essential for data analysis, manipulation, and visualization.
* **Machine Learning and AI**: Python, with libraries like TensorFlow, PyTorch, and scikit-learn, is the go-to language for AI and ML.
* **Automation**: Python scripts can automate repetitive tasks.
* **Scientific Computing**: Python is used in scientific research for simulations, modeling, and data analysis.
* **Game Development**: Python is used to develop games (e.g., using libraries like Pygame).

**1.1.7 Careers in Python:** Python developers have diverse career paths, such as:

* Python Developer
* Data Analyst
* Machine Learning Engineer
* Full-Stack Developer
* DevOps Engineer
* Cybersecurity Specialist

#### **1.2 Installing Python and Basic Syntax**

**1.2.1 Download and Install Python:** To install Python on Windows:

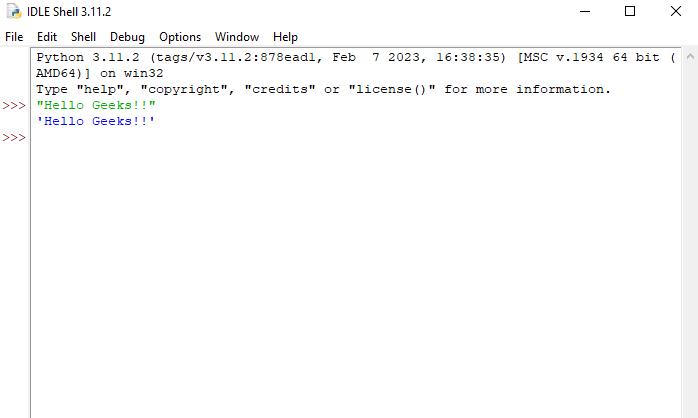
1. Visit the official website [python.org](https://www.python.org/downloads/).

****

1. Download the installer for the desired version (e.g., Python 3.12.3).
2. Run the installer and ensure you check "Add Python to PATH."



1. After installation, you can open the Python shell (interpreter) from the command prompt to start coding.



**1.2.2 Basic Syntax in Python:**

* **Comments**: Comments in Python start with the hash symbol #. The interpreter ignores everything after # on that line.
  + Example:

# This is a comment

print("Hello, Python!") # This prints a message



* **Continuation**: The backslash \ is used to split a statement into multiple lines.
  + Example:

myString = "Welcome to \ the world of \ Python programming"

print(myString)



* **Indentation**: In Python, indentation (spaces or tabs) is crucial for defining code blocks (unlike other languages that use curly braces {}).
  + Example:

if 5 > 2:

print("Five is greater than two!")



* **Multiple Statements on One Line**: You can use semicolons ; to write multiple statements on one line.
  + Example:

a = 10; b = 20; print(a); print(b)



* **Newline**: \n is used for line breaks within strings.
  + Example:

print("Hello\nWorld!")



* **Multiple Statement Groups**: Complex statements like if, while, def, and class require a header and a body (suite), where the body is indented.
  + Example:

if x > 10:

print("x is greater than 10")

elif x == 10:

print("x is exactly 10")

else:

print("x is less than 10")



* **Modules**: Python modules allow you to reuse code. You can create a Python module by saving your code in a .py file.
  + Example:

# mymodule.py

def greeting(name):

print(f"Hello, {name}!")

# main.py

import mymodule

mymodule.greeting("Alice")



#### **1.2.3 Python Identifiers**

**Identifiers** in Python are names used for variables, functions, classes, and other objects. These are rules for creating identifiers:

1. An identifier must begin with a letter (a-z, A-Z) or an underscore (\_).
2. It can only contain letters, digits (0-9), and underscores.
3. Identifiers are **case-sensitive** (myVariable and myvariable are different).
4. Identifiers **cannot start with a number**.

**Keywords**: These are reserved words that Python uses for its syntax. Examples include if, else, import, while, and more.

| False | Await | else | import | pass |
| --- | --- | --- | --- | --- |
| None | Break | except | in | raise |
| True | Class | finally | is | return |
| And | Continue | for | lambda | try |
| As | Def | from | nonlocal | while |
| Assert | Del | global | not | with |
| Async | Elif | if | or | yield |

**Built-in Names**: Python has several built-in names that are automatically imported and reserved, like print(), input(), list(), etc. These should not be used as identifiers.

#### **1.2.4 Variable Assignment**

In Python, you don't need to explicitly declare a variable before using it. Variables are assigned a value via the assignment operator (=). Python also supports dynamic typing, so a variable can be reassigned to any type of data.

* **Assignment Operator**: The equal sign (=) is used for assignment.
  + Example:

a = 12 # Integer

str = 'Hello' # String

f1 = -5.345 # Float



* **Augmented Assignment**: Python allows shorthand for updating variable values. For instance:
  + x += 1 is equivalent to x = x + 1.
  + Example:

x = 10

x += 5 # x becomes 15



* **Multiple Assignments**: Python allows you to assign values to multiple variables in a single statement.
  + Example:

x = y = z = 5



* **Multiple Assignment with Different Values**: You can also assign different values to multiple variables.
  + Example:

x, y, z = 1, 2, 'Hello'



* **Swapping Values**: Python allows swapping variables easily without the need for a temporary variable.
  + Example:

x, y = 1, 2

x, y = y, x # Now x = 2 and y = 1



# 

# **Computational Problems and Algorithms**

## **1. Identification of Computational Problems**

### **1.1 What is a Computational Problem?**

A **computational problem** is any problem that can be solved through a series of well-defined steps or instructions. These steps can be implemented using an **algorithm**, which is a precise method for solving the problem. The key characteristic of a computational problem is that it can be broken down into smaller, manageable tasks that can be solved systematically, often using a programming language like Python.

### **1.2 Examples of Computational Problems**

#### **1. Sorting**

Sorting is the process of arranging items in a particular order (ascending or descending). It’s one of the most common computational problems, and there are different algorithms to solve it, such as Bubble Sort, Merge Sort, Quick Sort, etc. However, Python provides built-in functions that make sorting straightforward.

**Sorting in Python**: Python has a built-in sorted() function and a method .sort() for lists that allow you to easily sort data.

* **sorted()**: This function returns a new sorted list from the given iterable.
* **.sort()**: This method sorts a list in place, meaning it modifies the original list.

**Example of Sorting (Ascending Order)**:

numbers = [10, 4, 2, 8, 15, 7]

sorted\_numbers = sorted(numbers)

print(sorted\_numbers)

# Output: [2, 4, 7, 8, 10, 15]

**Example of Sorting in Descending Order**:

numbers = [10, 4, 2, 8, 15, 7]

numbers.sort(reverse=True)

print(numbers)

# Output: [15, 10, 8, 7, 4, 2]



#### **2. Searching**

Searching is the task of finding an element in a collection of data. One of the simplest searching algorithms is **linear search**, where each element in the collection is checked one by one until the desired element is found. Python also provides a faster **binary search** method, but for that, the list must be sorted first.

##### **Linear Search:**

Linear search checks each element in the list until it finds the target or finishes checking all elements.

**Example of Linear Search in Python**:

def linear\_search(arr, target):

for index, value in enumerate(arr):

if value == target:

return index # returns the index where the target is found

return -1 # returns -1 if the target is not found

# Example usage:

numbers = [10, 4, 2, 8, 15, 7]

target = 8

result = linear\_search(numbers, target)

if result != -1:

print(f"Element found at index {result}")

else:

print("Element not found")

# Output: Element found at index 3

##### **Binary Search (Requires Sorted List):**

Binary search works by repeatedly dividing the list in half. It only works on sorted lists. It is much faster than linear search with a time complexity of O(log n).

**Example of Binary Search in Python**:

def binary\_search(arr, target):

left, right = 0, len(arr) - 1

while left <= right:

mid = (left + right) // 2

if arr[mid] == target:

return mid

elif arr[mid] < target:

left = mid + 1

else:

right = mid - 1

return -1 # return -1 if the target is not found

# Example usage:

numbers = [2, 4, 7, 8, 10, 15] # List must be sorted

target = 8

result = binary\_search(numbers, target)

if result != -1:

print(f"Element found at index {result}")

else:

print("Element not found")

# Output: Element found at index 3



#### **3. Finding Minimum/Maximum**

Another common computational problem is finding the smallest or largest number in a list. Python provides built-in functions min() and max() to solve this problem.

**Example of Finding the Minimum Value**:

numbers = [10, 4, 2, 8, 15, 7]

minimum = min(numbers)

print(f"The minimum value is {minimum}")

# Output: The minimum value is 2

**Example of Finding the Maximum Value**:

numbers = [10, 4, 2, 8, 15, 7]

maximum = max(numbers)

print(f"The maximum value is {maximum}")

# Output: The maximum value is 15

You can also find the minimum or maximum value with additional conditions. For example, finding the smallest number greater than a certain value:

**Finding the Minimum Greater Than a Given Number**:

numbers = [10, 4, 2, 8, 15, 7]

min\_greater\_than\_5 = min(x for x in numbers if x > 5)

print(f"The smallest number greater than 5 is {min\_greater\_than\_5}")

# Output: The smallest number greater than 5 is 7



### **Complexity of Sorting, Searching, and Finding Minimum/Maximum**

* **Sorting Complexity**:
  + **Timsort (used by Python’s sorted() and .sort() functions)**: O(n log n) in average and worst-case scenarios.
* **Searching Complexity**:
  + **Linear Search**: O(n) in the worst case, where n is the number of elements in the list.
  + **Binary Search**: O(log n), but it requires the list to be sorted beforehand.
* **Finding Minimum/Maximum Complexity**:
  + Both the min() and max() functions in Python run in O(n) time, where n is the number of elements in the list. This is because Python needs to examine every element to determine the minimum or maximum.

### **1.3 Identifying Computational Problems in Daily Life**

Think about some of the everyday tasks that could be described as computational problems:

* **Sorting tasks or files**: Organizing your to-do list or files in a folder could be formulated as a computational problem.
* **Finding an item**: If you're looking for your favorite shirt in a closet, this can be a searching problem.

### **1.4 How to Approach a Computational Problem**

When solving computational problems, you should:

1. **Understand the problem**: What is being asked?
2. **Break the problem into smaller steps**: How can you break the problem into smaller tasks?
3. **Identify the method (algorithm)**: How can you solve each smaller task step by step?

In Python, we can use built-in functions to solve these problems quickly. For instance:

* **Sorting** a list: sorted() or list.sort()
* **Searching** in a list: Use in or index()

### **1.5 Thought-Provoking Questions**

* Can you think of a real-world problem that you could break down into smaller steps?
* How do you use sorting or searching algorithms daily?

### **1.6 Python Example**

To find the smallest number in a list:

numbers = [3, 1, 4, 1, 5, 9]

smallest = min(numbers)

print(smallest) # Output: 1



## **2. Algorithms**

### **2.1 What is an Algorithm?**

An **algorithm** is a set of defined steps or instructions that you follow to solve a specific problem. Algorithms are the backbone of computational problem-solving.

### **2.2 Characteristics of an Algorithm**

* **Definite**: The steps should be clear and unambiguous.
* **Input**: The algorithm takes inputs (data) and processes them.
* **Output**: After processing, the algorithm produces an output (solution).
* **Finiteness**: It must terminate after a finite number of steps.
* **Effectiveness**: The steps should be simple and achievable.

### 

### 

### **2.3 Example Algorithms in Python**

* **Searching Algorithm**: Looking for an item in a list (e.g., linear search).
* **Sorting Algorithm**: Arranging items in ascending or descending order (e.g., bubble sort, quicksort).

### **2.4 Efficiency of Algorithms**

* **Time complexity**: How fast an algorithm solves the problem (measured in terms of the number of operations).
* **Space complexity**: How much memory an algorithm requires.

### **2.5 Python Example: Sorting a List**

Using Python's built-in sorted() method:

numbers = [3, 1, 4, 1, 5, 9]

sorted\_numbers = sorted(numbers)

print(sorted\_numbers) # Output: [1, 1, 3, 4, 5, 9]



## **3. Building Blocks of Algorithms**

### **3.1 Components of an Algorithm**

An algorithm is composed of the following components:

* **Statements**: Instructions that perform actions.
* **State**: The current condition or status of the system.
* **Control Flow**: The order in which statements are executed, which includes decision-making (e.g., if statements) and looping (e.g., for and while loops).
* **Functions**: Subroutines that help modularize code by breaking the algorithm into smaller, reusable tasks.

### **3.2 Python Control Flow**

* **If Statements**: These allow the program to make decisions.
* **Loops (For/While)**: These allow repetition of a block of code.
* **Functions**: Functions are used to modularize code and improve its reusability.

### **3.3 Python Example: Control Flow and Functions**

To determine if a number is prime:

def is\_prime(n):

if n <= 1:

return False

for i in range(2, n):

if n % i == 0:

return False

return True

print(is\_prime(7)) # Output: True

### **3.4 Thought-Provoking Questions**

* What happens when a loop is omitted from an algorithm?
* When is it appropriate to use a function?

## **4. Notation**

### **4.1 Representing Algorithms**

To effectively communicate algorithms, we use the following notations:

* **Pseudocode**: A simplified, human-readable version of an algorithm.
* **Flowcharts**: Diagrams that represent the flow of logic in an algorithm.
* **Programming Language**: The actual code that implements the algorithm.

### **4.2 Pseudocode Example**

For summing numbers, the pseudocode would be:

START

Initialize sum to 0

For each number in the list

Add the number to sum

Print the sum

END

### **4.3 Flowchart Example**

Flowcharts use shapes like ovals (start/end), diamonds (decisions), and rectangles (processes) to represent the steps of an algorithm.

### **4.4 Python Code Example**

For the above pseudocode, the Python implementation would look like:

numbers = [1, 2, 3, 4]

total = sum(numbers)

print(total) # Output: 10



## **5. Algorithmic Problem Solving**

### **5.1 Problem-Solving Strategies**

In Python, we can solve problems using two primary strategies:

1. **Iteration**: Repeating a process or set of instructions.
2. **Recursion**: A function calling itself to solve smaller instances of the problem.

### **5.2 Iteration vs. Recursion**

* **Iteration**: Good for problems where you need to repeatedly apply the same process.
* **Recursion**: Ideal for problems that can naturally be broken into smaller subproblems, like tree traversal or factorial calculation.

### **5.3 Example of Iteration: Sum of Numbers**

****total = 0

for number in [1, 2, 3, 4]:

total += number

print(total) # Output: 10

### **5.4 Example of Recursion: Factorial Calculation**

****def factorial(n):

if n == 0 or n == 1:

return 1

else:

return n \* factorial(n - 1)

print(factorial(5)) # Output: 120



## **6. Illustrative Problems**

### **6.1 Example Problems in Algorithmic Problem Solving**

* **Minimum in a List**: Find the smallest number in a list.
* **Insertion Sort**: Sort a list of numbers using the insertion sort algorithm.
* **Guessing Game**: Build a number guessing game using loops and conditions.
* **Towers of Hanoi**: Solve the classic puzzle using recursion.

### **6.2 Solving the Towers of Hanoi with Recursion**

The Towers of Hanoi is a classic problem where you move disks from one peg to another, following specific rules. Recursion is ideal here because the problem can be broken down into smaller subproblems.

### **6.3 Example Problem: Insertion Sort**

****def insertion\_sort(numbers):

for i in range(1, len(numbers)):

key = numbers[i]

j = i - 1

while j >= 0 and key < numbers[j]:

numbers[j + 1] = numbers[j]

j -= 1

numbers[j + 1] = key

return numbers

print(insertion\_sort([3, 1, 4, 1, 5, 9]))

# Output: [1, 1, 3, 4, 5, 9]



**Homework/Assignments**

### 

* Solve the Towers of Hanoi problem using recursion in Python.
* Write a Python program to solve a guessing game where the computer randomly selects a number, and the user tries to guess it.

**Quiz**

### **1. What is the key feature that makes Python a popular programming language for beginners?**

a) Its speed of execution  
 b) Its complex syntax  
 c) Its simplicity and readability  
 d) Its ability to execute code directly on hardware

**Answer: c) Its simplicity and readability**

### **2. What is the output of the following Python code?**

****numbers = [10, 4, 2, 8, 15, 7]

numbers.sort(reverse=True)

print(numbers)

a) [10, 4, 2, 8, 15, 7]  
 b) [15, 10, 8, 7, 4, 2]  
 c) [2, 4, 7, 8, 10, 15]  
 d) [4, 7, 8, 10, 15, 2]

**Answer: b) [15, 10, 8, 7, 4, 2]**

### **3. Which of the following is the correct syntax for defining a function in Python?**

a) function myFunction():  
 b) def myFunction[]:  
 c) def myFunction():  
 d) function myFunction[]:

**Answer: c) def myFunction():**

### **4. Which of the following searching algorithms is faster but requires the list to be sorted first?**

a) Linear Search  
 b) Binary Search  
 c) Bubble Sort  
 d) Quick Sort

**Answer: b) Binary Search**

### **5. What is the correct way to swap the values of two variables x and y in Python?**

a) x = y; y = x  
 b) x, y = y, x  
 c) swap(x, y)  
 d) x = y; x = x + y

**Answer: b) x, y = y, x**

**END OF UNIT-1**