UNIT – I INTRODUCTION TO PYTHON

Problem solving, Problem Analysis Chart, Developing an Algorithm, Flowchart and Pseudocode, Interactive and Script Mode, Indentation, Comments, Error messages, Variables, Reserved Words, Data Types, Arithmetic operators and expressions, Built-in-Functions, Importing from Packages.

1. Problem Solving and Analysis 💡

1.1 Problem Solving

1.1.1 Introduction

Problem solving is the process of identifying a computational or engineering problem, analyzing its requirements, and developing an **efficient solution** using systematic methods. In the context of programming, problem solving is the foundation for writing correct, efficient, and reusable code.

1.1.2 Definition

Problem solving in computer science can be defined as:

"A **step-by-step process** of defining a problem, designing a procedure (**algorithm**), and implementing it using a programming language to obtain the desired output."

1.1.3 Characteristics of a Good Problem-Solving Process

1. **Clarity** – The problem statement must be clearly understood.
2. **Systematic Approach** – Steps should follow a logical sequence.
3. **Efficiency** – The solution must minimize time and space complexity.
4. **Correctness** – The solution should produce accurate results for all valid inputs.
5. **Scalability** – The approach should adapt to larger or complex inputs.

1.1.4 General Steps in Problem Solving

1. **Problem Identification** – Clearly state the problem.
2. **Problem Analysis** – Understand input, process, and output requirements.

CS25C02

3. **Solution Design** – Develop an algorithm, flowchart, or pseudocode.
4. **Implementation** – Translate the design into a program (Python).
5. **Testing and Verification** – Run test cases to check correctness.
6. **Maintenance** – Modify or optimize the solution as requirements evolve.

1.1.5 Computational Problem Solving in Engineering

In engineering, problem solving often involves:

- Mathematical computations (e.g., stress analysis, area, volume calculations).
- Data processing (e.g., analyzing sensor data, simulation outputs).
- Automation (e.g., controlling devices, file handling, batch processing).
- Modeling and Simulation (e.g., using Python libraries for system modeling).

CS25C02

Example: Calculating Area of a Rectangle

- **Problem Statement:** Write a program to compute the area of a rectangle given its length and breadth.

Step	Detail
Step 1 – Problem Identification	Input $\rightarrow$ Length and Breadth, Process $\rightarrow$ Multiply Length $\times$ Breadth, Output $\rightarrow$ Area of the rectangle
Step 2 – Algorithm	1. Start; 2. Read length and breadth; 3. Compute area = length $\times$ breadth; 4. Display area; 5. Stop
Step 3 – Pseudocode	BEGIN, READ length, breadth, area $\leftarrow$ length $*$ breadth, PRINT area, END

1.1.6 Summary

- Problem solving is a systematic process that transforms a problem statement into a working solution.
- It involves understanding inputs/outputs, designing algorithms, and implementing solutions in Python.
- Engineering applications frequently rely on computational problem solving to automate and optimize tasks.

1.2 Problem Analysis Chart (PAC)

1.2.1 Definition

A **Problem Analysis Chart (PAC)** is a tabular representation that breaks down a problem into its fundamental components:

- **Input** – The data required for the problem.
- **Process** – The computational or logical steps needed to transform input into output.
- **Output** – The expected result of the problem.

CS25C02

1.2.2 Components of Problem Analysis Chart (PAC)

Component	Description	Example
Problem Statement	Clearly defines what needs to be solved	Find the area of a rectangle
Input	Data values required from the user	Length, Breadth
Process	Operations or calculations performed on input	Area = Length $\times$ Breadth
Output	Final result to be displayed	Area of the rectangle

1.2.3 Steps to Construct a PAC

1. Identify the problem statement.
2. Determine all necessary inputs.
3. List the process steps needed to solve the problem.
4. Define the expected output.
5. Tabulate the above information into a PAC.

1.2.4 Example: PAC for Area of Rectangle

Problem	**Find the Area of a Rectangle**
Input	Length, Breadth
Process	Area = Length $\times$ Breadth
Output	Area of the rectangle

1.2.5 Example: PAC for Simple Interest

Problem	**Calculate Simple Interest**
Input	Principal (P), Rate of Interest (R), Time (T)
Process	SI = (P $\times$ R $\times$ T) / 100
Output	Simple Interest amount

1.2.6 Benefits of PAC

- Provides **clarity** before coding.
- Reduces **logical errors** by defining all inputs and outputs.
- Acts as a bridge between problem statement and algorithm.
- Helps beginners systematically approach programming problems.

CS25C02

1.2.8 Summary

- The Problem Analysis Chart (PAC) is a tool to break a problem into input, process, and output.
- It simplifies problem-solving by organizing requirements before coding.
- PAC ensures that the transition from problem statement $\rightarrow$ algorithm $\rightarrow$ code is systematic.

---

3. Developing Algorithms, Flowcharts, and Pseudocode 📐

1.3 Developing an Algorithm

Definition of Algorithm:

An algorithm is a finite set of instructions for performing a particular task. The instructions are nothing but the statements in simple English language.

In computing, we focus on the type of problems categorically known as **algorithmic problems**, where their solutions are expressible in the form of algorithms. The term "algorithm" was derived from the name of **Mohammed al-Khowarizmi**, a Persian mathematician in the ninth century.

CS25C02

An algorithm is a well-defined computational procedure consisting of a set of instructions that takes some value or set of values, as **input**, and produces some value or set of values, as **output**.

$$\text{INPUT} \rightarrow \text{ALGORITHM} \rightarrow \text{OUTPUT}$$

1.3.1 Characteristics of a Good Algorithm

1. **Precision** – Every step in the algorithm must be **clearly and accurately** defined.
2. **Uniqueness** – Each step should yield a unique result, depending solely on the given input and the result of previous steps.
3. **Finiteness** – The algorithm must **terminate** after a limited number of steps.
4. **Effectiveness** – Among all possible approaches, the algorithm should provide the **most efficient** solution to the problem.

5. **Input** – It should accept **one or more inputs** to begin the process.
6. **Output** – It must generate **at least one output** as the result.
7. **Generality** – The algorithm should be applicable to a **wide range of input** values, not just specific cases.

Example 1.3.1: Algorithm to Check if a Number is Even or Odd

1. START
2. READ number N.
3. CALCULATE remainder $R = N \text{ mod } 2$.
4. IF $R = 0$ THEN PRINT "N is Even".
5. ELSE PRINT "N is Odd".
6. STOP.

Example 1.3.2: Algorithm to Find the Largest of Two Numbers

1. START
2. READ two numbers, A and B.
3. IF $A > B$ THEN PRINT "A is the largest".
4. ELSE PRINT "B is the largest".
5. STOP.

CS25C02

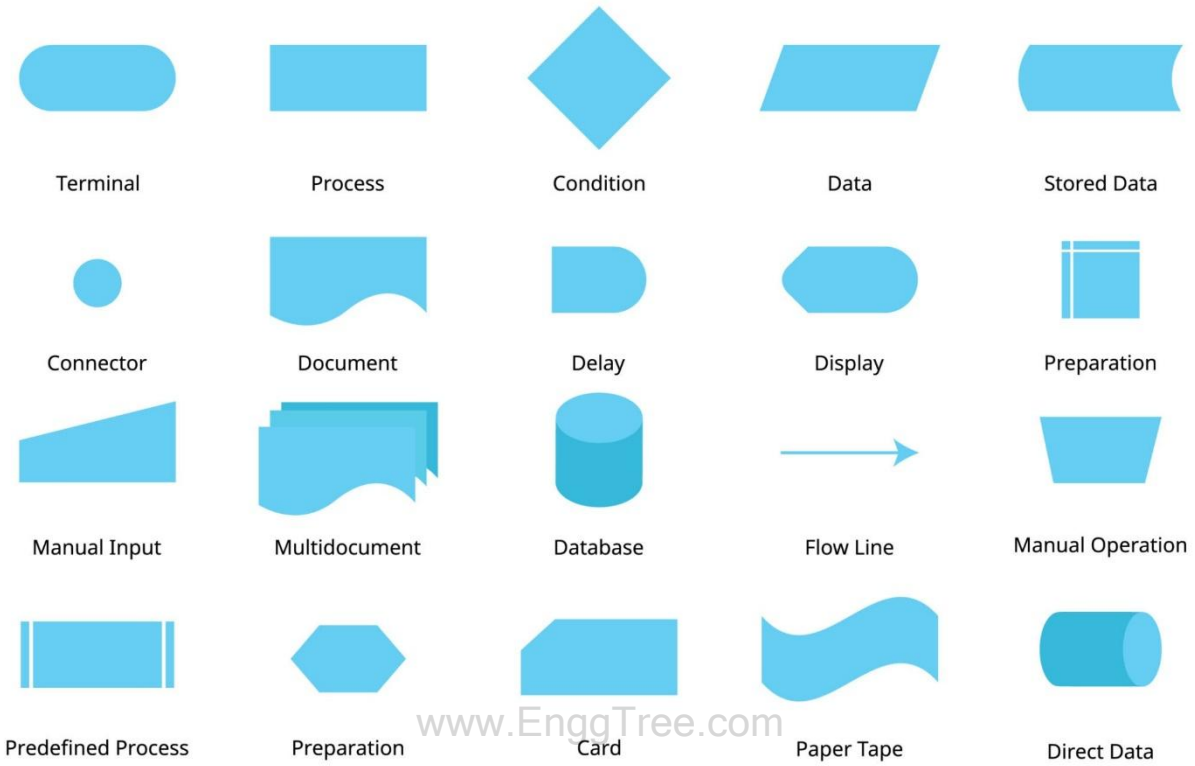
1.4 Flow Chart and Pseudocode

1.4.1 Flow Charts

- Flowcharts are the **graphical representation** of the algorithms.
- The algorithms and flowcharts are the final steps in organizing the solutions.
- Using the algorithms and flowcharts the programmers can find out the bugs in the programming logic and then can go for coding.

- Flowcharts can show errors in the logic and set of data can be easily tested using flowcharts.

1.4.1.1 Flow chart Symbols



S. No	Name of Symbol	Symbol	Type	Description
1.	**Terminal Symbol**			

| Oval | Represent the start and stop of the program. |

| 2. | Input/ Output symbol | | Parallelogram | Denotes either input or output operation. |

| 3. | Process symbol | | Rectangle | Denotes the process to be carried. |

| 4. | Decision symbol | | Diamond | Represents decision making and branching. |

| 5. | Flow lines | | Arrow lines | Represents the sequence of steps and direction of flow. Used to connect symbols. |

CS25C02

| 6. | Connector | | Circle | Used to Connect flow charts (e.g., across pages). |

1.4.1.2 Rules for drawing flow chart

1. In drawing a proper flowchart, all necessary requirements should be listed out in logical order.
2. The flow chart should be **clear, neat and easy to follow**. There should not be any room for ambiguity in understanding the flowchart.
3. The usual directions of the flow of a procedure or system are from **left to right** or **top to bottom**. **Only one flow line should come out from a process symbol.**
4. Only **one flow line should enter a decision symbol**, but two or three flow lines, one for each possible answer, can leave the decision symbol.
5. Only one flow line is used in conjunction with a terminal symbol.
6. If the flow chart becomes complex, it is better to use connector symbols to reduce the number of flow lines.
7. Ensure that the flow chart has logical **start and stop**.

CS25C02

1.4.1.3 Advantages of Flow chart

1. **Communication:** Flow charts are a better way of communicating the logic of the system.
2. **Effective Analysis:** Helps analyze a problem in a more effective way.
3. **Proper Documentation:** Used for good program documentation.
4. **Efficient Coding:** Flowcharts act as a guide or blue print during development.
5. **Systematic Testing and Debugging:** Helps in testing and debugging the program.
6. **Efficient Program Maintenance:** Makes the maintenance of operating programs easier.

1.4.1.4 Disadvantages of Flowchart

1. **Complex Logic:** Flow charts become complex and difficult to use for complicated program logic.
2. **Alteration and Modification:** If alterations are required, the flowchart may require re-drawing completely.
3. **Reproduction:** As the flowchart symbols cannot be typed, reproduction becomes problematic.

1.4.2 Pseudo Code

Pseudocode is an informal high-level description of the operating principle of a computer program or algorithm. It uses the basic structure of a normal programming language, but is intended for **human reading** rather than machine reading. It is a text-based detail design tool. **Pseudo** means false and code refers to instructions written in a programming language.

Pseudocode **cannot be compiled nor executed**, and there are no real formatting or syntax rules. The pseudocode is written in normal English language which cannot be understood by the computer.

CS25C02

Example 1.4.1: Pseudocode: To find sum of two numbers

READ num1, num2

sum ← num1 + num2

PRINT sum

Basic rules to write pseudocode:

1. **Only one statement per line.** Statements representing a single action are written on the same line. For example, to read the input, all the inputs must be read using a single statement.
2. **Capitalized initial keywords** The keywords should be written in capital letters. Eg: **READ, WRITE, IF, ELSE, ENDIF, WHILE, REPEAT, UNTIL**

Example 1.4.2: Pseudocode: Find the total and average of three subjects

READ subject1, subject2, subject3

total ← subject1 + subject2 + subject3

```
average ← total / 3

PRINT total, average
```

3. **Indent to show hierarchy** Indentation is a process of showing the boundaries of the structure.
4. **End multi-line structures** Each structure must be ended properly, which provides more clarity.

Example 1.4.3: Pseudocode: Find greatest of two numbers

```
READ A, B

IF A > B THEN

  PRINT "A is greater"

ELSE

  PRINT "B is greater"

ENDIF
```


5. Keep statements language-independent.

Pseudocode must never use the specific syntax of any programming language.

CS25C02

1.4.2.1 Advantages of Pseudocode

- Can be done easily on a word processor.
- Easily modified.
- Implements structured concepts well.
- It can be written easily.
- It can be read and understood easily.
- Converting pseudocode to a programming language is easy as compared with a flowchart.

1.4.2.2 Disadvantages of Pseudocode

- It is **not visual**.

- There is **no standardized** style or format.

---

4. Python Programming Environment and Syntax 💻

1.5 Python Programming Environment

1.5.1 Interactive Mode

- Python has two modes: normal mode and interactive mode.
- **Interactive Mode:** Also called the **Shell** or **Script Mode** (Note: this is often called Shell/Console mode in modern Python teaching, Script mode is the opposite). In this mode, the user can type Python commands directly in the interpreter. The interpreter immediately executes the command and displays the result. Each statement is executed as soon as it is typed. This is particularly useful for testing small code snippets.
- The prompt >>> indicates that the interpreter is ready for user input.
- When an expression is typed after the prompt, the interpreter displays the result immediately.
- Proper indentation is required while writing the code on the interpreter shell.

CS25C02

1.5.2 Script Mode

- **Script mode** is the standard mode.
- In this mode, Python commands are saved in a file with the extension **.py**.
- The saved file can then be executed repeatedly.

Steps to run Python in Script Mode:

1. **Step 1:** Open Python Shell by double-clicking the Python IDE.
2. **Step 2:** On the File Menu, click on **New File** option.
3. **Step 3:** Give some suitable file name with extension .py (e.g., example.py).
4. **Step 4:** A file will get opened and then type some programming code.

5. **Step 5:** Now run your code by clicking on **Run Menu $\rightarrow$ Run Module (F5)**.

- The output will be displayed on the python shell.

1.6 Python Syntax and Indentation

1.6.1 Python Syntax

- Python syntax refers to the set of rules that defines how Python programs are written and interpreted.
- Unlike many programming languages (such as C, C++ or Java), Python syntax is simple and emphasizes **readability**.
- Statements in Python are written line by line and executed sequentially.
- Each statement usually ends with a **newline**, not with a semicolon (;).
- **Indentation**, rather than braces {} or keywords, is used to define code blocks.

CS25C02

Example: A simple Python statement

Python

```
print("Welcome to Python")
```

Example: Multiple statements on one line

Python

```
a = 10; b = 20; print(a + b)
```

1.6.2 Indentation in Python

- **Indentation** means the space at the beginning of a line of code.
- In Python, indentation is **mandatory** and indicates a **block of code**.
- Other programming languages like C/Java use curly braces {} to define blocks, but Python relies on indentation only.
- The standard indentation is **4 spaces** (tabs may also be used, but mixing tabs and spaces should be avoided).

Example: Using indentation in if statement

Python

```
if True:
    print("This is a block of code")
    print("It runs because it's indented")
```

1.6.2.1 Indentation Errors

- If indentation is not used correctly, Python raises an **IndentationError**.
- This helps prevent ambiguity in program structure.

Example: Error due to incorrect indentation

Python

```
if True:
print("This line will cause an IndentationError")
```


CS25C02

1.7 Comments and Documentation

Comments and documentation are essential for writing readable, maintainable, and error-free code. They help programmers explain the logic, purpose, and functionality of code segments without affecting execution. Python ignores comments during program execution.

1.7.1 Comments in Python

Python provides two types of comments:

a) Single-Line Comments

- Begin with the # symbol.
- Everything after # on the same line is ignored by the interpreter.

Example:

Python

```
# This is a comment about the next line

result = 10 + 5 # Calculate sum
```

b) Multi-Line Comments

- Python does not have explicit multi-line comment syntax (like /* ... */ in C).
- Instead, multi-line comments are created using **triple quotes** (''' ... ''' or """ ... """).
- They can span across multiple lines.

Example:

Python

```
"""
This is a multi-line comment.
It explains the overall program logic.
"""
```


CS25C02

1.7.2 Documentation Strings (Docstrings)

- **Docstrings** are special string literals used for documenting modules, classes, functions, or methods.
- Declared using **triple quotes** (""" ... """ or ''' ... ''').
- Can be accessed at runtime using the **.__doc__** attribute.

Example: Function with Docstring

Python

```
def add_numbers(x, y):
  """
  This function takes two numbers and returns their sum.
  """
```

```
    return x + y


print(add_numbers.__doc__)
```

Output:

```
This function takes two numbers and returns their sum.
```

1.7.3 Importance of Comments and Documentation

- Increases **readability** of programs.
- Helps in **debugging and maintenance**.
- Acts as **reference material** for future developers.
- Essential for collaborative coding projects and engineering software systems.

CS25C02

1.8 Error Messages and Debugging Basics

Errors are inevitable in programming and occur when the Python interpreter fails to execute code correctly. Understanding error messages and learning debugging techniques are essential skills for every programmer. Debugging is the systematic process of detecting, analyzing, and fixing errors to ensure reliable software execution.

1.8.1 Types of Errors in Python

(a) Syntax Errors

- Occur when Python code **violates grammatical rules**.
- Detected during the **parsing stage**, before execution.
- Example: Missing colon, wrong indentation.

Python

```
# Syntax Error Example

if True
```

```
print("Hello") # Missing colon after if

# Error Message:

# SyntaxError: expected ':'
```

(b) Runtime Errors (Exceptions)

- Occur while the program is **running**.
- Examples: division by zero, accessing an invalid index, wrong data type.

```
Python

# Runtime Error Example

x = 10 / 0

# Error Message:

# ZeroDivisionError: division by zero
```

(c) Logical Errors

- The program runs without crashing but produces **incorrect results**.
- **Hardest to detect** since Python does not raise an error.

```
Python

# Logical Error Example

# Program to calculate average (wrong formula used)

marks = [80, 90, 70]

average = sum(marks) * len(marks) # Incorrect calculation

print("Average =", average)
```

1.8.2 Debugging Basics

- **Step 1: Reading Error Messages:** Identify the type of error (e.g., SyntaxError, ValueError) and locate the **line number** provided in the traceback.

CS25C02

- **Step 2: Using Print Statements:** Insert print() statements at different points to check **variable values**. Helps trace program flow and detect logic errors.
- **Step 3: Using Python Debugger (pdb):** Python has a built-in debugger module: pdb. Allows **step-by-step execution**, variable inspection, and breakpoints.
- **Step 4: Exception Handling:** Use **try-except** blocks to gracefully handle runtime errors.

Python

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
```


1.8.3 Importance of Debugging in Engineering

- Ensures **reliable** and **efficient** software.
- Reduces development time and cost by preventing failures.
- Improves system performance and accuracy.
- Essential for safety-critical engineering applications (e.g., healthcare, automotive, aerospace).

1.8.4 Best Practices for Debugging

- Read and interpret error messages carefully.
- Start debugging from the **topmost error line** in the traceback.
- Use modular programming to isolate and test components.
- Avoid excessive use of print(); rely on debugging tools.
- Write unit tests to catch errors early.

CS25C02

5. Variables, Keywords, and Data Types

1.9 Variables and Reserved Words

In Python programming, **variables** are symbolic names used to store data values, while **reserved words** (keywords) are predefined identifiers with special meaning.

1.9.1 Variables in Python

Definition

A variable is a name given to a memory location that holds a value. Python allows dynamic typing, meaning variables do not need explicit declaration of type; their type is determined at runtime.

Rules for Variable Naming

1. Must begin with a **letter or underscore** (_).
2. Cannot start with a **digit**.
3. Can contain letters, digits, and underscores.
4. Are **Case-sensitive** (Age and age are different).
5. Cannot use **reserved keywords** as variable names.

Valid Examples	Invalid Examples
name, _count, temp_C	1st_name (Starts with digit)
area2D, my_Variable	for (Reserved keyword)

Variable Assignment

- Python uses the = operator for assignment.
- Multiple assignments are allowed in one line.

Examples:

Python

CS25C02

temperature = 25.5

length, width = 10, 5

Dynamic Typing

- The type of variable is determined automatically.
- Variable type can change during execution.

1.9.2 Reserved Words in Python

Definition

Reserved words (or keywords) are predefined identifiers in Python with special meaning. They are part of the Python language syntax and cannot be used as variable names.

List of Python Reserved Words (Python 3.10+)

Keywords				
False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
async	except	in	raise	
await				

CS25C02

1.9.3 Difference Between Variables and Reserved Words

Aspect	Variables	Reserved Words
Definition	User-defined names storing data	Predefined identifiers with special meaning
Flexibility	Chosen by programmer	Fixed and cannot be changed
Example	student, marks, count	for, while, if, class
Usage	Store values, perform computations	Define control flow, logic, and structure

1.9.5 Applications

- **Applications of Variables:** Data storage and processing (sensor values, results), Simulation models (represent physical parameters), User input handling, Mathematical computations.
- **Applications of Reserved Words:** Control structures (if, while), Function/Class Definitions (def, class), Exception Handling (try, except), Resource Management (with).

CS25C02

1.10 Data Types

In Python, **data types** define the type of values that variables can store. Since Python is a **dynamically typed** language, the interpreter automatically determines the data type of a variable during program execution.

Understanding data types is fundamental because they determine:

- How data is stored in memory.
- What operations can be performed on the data.
- How the program interprets the stored values.

1.10.1 Data Types in Python

Python provides several built-in data types, broadly categorized as follows:

a) Numeric Types

- **int** – Integer values without a decimal.
- **float** – Real numbers with decimal points.
- **complex** – Numbers with real and imaginary parts ($a + bj$).

Example:

Python

```
num_int = 10      # int
num_float = 15.5  # float
num_complex = 2 + 3j # complex
```

b) Sequence Types

1. **str (String)** – Collection of characters enclosed in quotes (" " or ' ').
2. **list** – Ordered, **mutable** collection of items.
3. **tuple** – Ordered, **immutable** collection of items.
4. **range** – Represents an immutable sequence of numbers.

Example:

Python

```
name = "Python" # str
data_list = [1, 2, 'a'] # list
data_tuple = (10, 20) # tuple
```

c) Set Types

- **set** – Unordered collection of **unique** elements.
- **frozenset** – Immutable version of a set.

Example:

Python

unique_numbers = {1, 2, 3}

d) Mapping Type

- **dict (Dictionary)** – Collection of **key–value pairs**.

Example:

Python

student = {"name": "John", "age": 20, "marks": 95}

e) Boolean Type

- **bool** – Represents truth values: **True** or **False**.

Example:

Python

is_adult = True

f) None Type

- **None** – Special data type representing the **absence of a value** or null value.

Example:

Python

x = None

1.10.2 Type Conversion

Python supports type conversion in two forms:

1. **Implicit Conversion (Type Casting by Interpreter)** – Automatically converts smaller data type to larger data type to avoid data loss.
    - a = 10 (int), b = 5.5 (float), c = a + b (result is **float** (15.5)).
2. **Explicit Conversion (Type Casting by User)** – Programmer manually converts data type using built-in functions (int(), float(), str(), etc.).
    - x = "100", y = int(x) (converts string to **int**).

CS25C02

1.10.3 Applications of Data Types

1. **Engineering Computations** – Numeric types for calculations in simulations, measurements, and modeling.
2. **Data Handling** – Strings, lists, and dictionaries for handling user input, datasets, and structured information.
3. **Logical Decision-Making** – Boolean values help in control flow, testing conditions, and validations.

1.11 Arithmetic Operators and Expressions ➗

In Python, **arithmetic operators** are used to perform mathematical operations on numerical data types. An **expression** is a combination of variables, constants, and operators that evaluates to a value.

1.11.1 Arithmetic Operators in Python:

Operator	Description	Example	Result
+	**Addition** (Sum of two operands)	$10 + 5$	15
-	**Subtraction** (Difference)	$10 - 5$	5
*	**Multiplication** (Product)	$10 * 5$	50
/	**Division** (Returns a floating-point result)	$10 / 3$	$3.333...$
//	**Floor Division** (Returns the largest integer less than or equal to the result, discarding decimal)	$10 // 3$	3

CS25C02

Operator	Description	Example	Result
%	**Modulus** (Returns the **remainder** of a division operation)	$10 % 3$	1
**	**Exponentiation** (Raises first operand to the power of the second)	$2 ** 3$	8

Arithmetic Expressions

An arithmetic expression is a combination of numbers, variables, and operators that produces a numerical result.

Examples:

Python

a = 10

b = 3

expression1 = (a + b) * 2 # 13 * 2 = 26

expression2 = a % b # 10 % 3 = 1

CS25C02

1.11.2 Operator Precedence and Associativity

When multiple operators are used in a single expression, Python follows **operator precedence** rules.

Precedence Order (Highest to Lowest):

1. Parentheses ()
2. Exponentiation **
3. Multiplication, Division, Floor Division, Modulus *, /, //, %
4. Addition, Subtraction +, -

Associativity:

- Most operators are **left-to-right** associative.

- Exponentiation (**) is **right-to-left** associative.

Example:

- result = 2 + 3 * 4 $\rightarrow 2 + (12) = 14$ (Multiplication before Addition)
- power = 2 ** 3 ** 2 $\rightarrow 2 ** (9) = 512$ (Right-to-left associativity for **)

1.11.3 Applications of Arithmetic Operators and Expressions

1. **Engineering Calculations** – Mathematical modeling, solving equations, and scientific simulations.
2. **Data Analysis** – Statistical computations like mean, variance, and regression analysis.
3. **Control Systems** – Expressions are applied in feedback loop calculations and signal processing.
4. **Financial Applications** – Interest calculation, budgeting, and forecasting.

CS25C02

1.12 Built-in Functions

Python provides a rich collection of **built-in functions** that can be used directly **without importing any external library**. These functions simplify programming by performing commonly needed tasks.

Characteristics of Built-in Functions

1. Available by **default** in Python (no need to import).
2. Can be used across different data types and applications.
3. Improve efficiency by reducing the need for manual implementation.

Commonly Used Built-in Functions

Function	Description	Example	Output
print()	Displays output on the screen.	print("Hello")	Hello
input()	Reads user input as a **string**.	x = input("Enter: ")	User input
len()	Returns length of a sequence.	len("Python")	6
type()	Displays the data type of a variable.	type(10)	<class 'int'>
int()	Converts a value to **integer**.	int("5")	5
float()	Converts a value to **float**.	float("3.2")	3.2
str()	Converts a value to **string**.	str(25)	"25"
max()	Returns maximum value from a sequence.	max(4, 9, 2)	9
min()	Returns minimum value from a sequence.	min(4, 9, 2)	2
abs()	Returns **absolute value**.	abs(-7)	7
round()	Rounds a number to nearest integer or given decimals.	round(3.75, 1)	3.8
sum()	Returns sum of elements in a sequence.	sum([1, 2, 3])	6

Example Program

CS25C02

Python

```
# Demonstrating built-in functions
a = -15
data = [5, 8, 7]
print("Absolute value of a:", abs(a))
print("Maximum value in list:", max(data))
print("Sum of list:", sum(data))
print("Type of a:", type(a))
```

Output:

```
Absolute value of a: 15
Maximum value in list: 8
Sum of list: 20
Type of a: <class 'int'>
```


CS25C02

1.13 Importing from Packages

In Python, a **package** is a collection of modules organized in a directory structure. To use the functions, classes, or variables defined in these modules, they must be **imported** into the program.

Syntax for Importing

Python provides multiple ways to import from packages:

1. **Import Entire Module**

Python

```
import math
print(math.sqrt(16)) # Using function with module name
```

2. **Import Specific Functions/Classes**

Python

```
from math import sqrt, pi

print(sqrt(25))

print(pi)
```

3. **Import Module with Alias**

Python

```
import numpy as np

print(np.array([1, 2, 3]))
```

4. **Import All Contents (Not Recommended)**

Python

```
from math import * print(sin(0)) # Function used directly, leading to namespace pollution
```


Commonly Used Standard Packages

- **math** – Provides mathematical functions like sqrt(), sin(), cos().
- **random** – Used for generating random numbers.
- **datetime** – Deals with date and time.
- **os** – Provides operating system-related functions (file handling).
- **sys** – Provides system-specific parameters and functions.

CS25C02

Example Program

Python

```
# Importing from math and random packages

import math

import random

num = 49

print("Square root of num:", math.sqrt(num))
```

```
print("Random number between 1 and 10:", random.randint(1, 10))
```

Output:

Square root of num: 7.0

Random number between 1 and 10: 6 (Note: output will vary)

Applications of Importing from Packages

1. **Scientific Computation** – Using math, numpy, scipy for engineering and research calculations.
2. **Data Science and AI** – pandas, matplotlib, scikit-learn for data processing and machine learning.
3. **System Programming** – os and sys for file operations and process management.

CS25C02

Why Programming Language?

1. **Time Management.**
2. **Project Implementation.**
3. **Communication Process.**
4. **Calculation.**
5. **S/W and H/W** (Software and Hardware integration).
6. **Web Developing.**

UNIT -2 CONTROL STRUCTURES

If, if-else, nested if, multi-way if-elif statements, while loop, for loop, nested loops, pass statements.

2. Control Flow and Functions in Python ⚙

2.1 Introduction to Control Flow

Control flow refers to the order in which individual instructions, statements, or function calls are executed within a program. In Python, control flow is determined by decision-making statements, loops, and function calls, which allow programmers to alter the natural top-to-bottom execution sequence.

1. **Sequential Execution:** By default, program statements execute one after another in the order they appear. *Example: Assigning variables and printing values.*
2. **Decision Making (Selection):** Control flow can branch based on conditions using if, if-else, and if-elif-else. *Example: Checking if a number is positive or negative.*
3. **Repetition (Iteration):** Loops (for, while) allow repeating a block of code until a condition is met. *Example: Printing numbers from 1 to 10.*
4. **Transfer of Control:** Special keywords like break, continue, and pass can alter the normal flow within loops.
5. **Function Calls:** Execution can jump to a function block and return to the caller after completion.

CS25C02

2.1.1 Functions

A **Function** is a **sub program** which consists of sets of instructions used to perform a specific task. A large program is divided into basic building blocks called functions.

Need for Function:

- When the program is too complex and large, they are divided into parts. Each part is separately coded and combined into a single program. Each subprogram is called a function.

- **Debugging, Testing, and maintenance** becomes easy when the program is divided into subprograms.
- Functions are used to avoid **rewriting the same code** again and again in a program.
- Function provides **code re-usability**.
- The length of the program is reduced.

Types of Functions:

1. **User-defined function** (Programmer creates)
2. **Built-in function** (Already created & Stored)

Function Definition: (Sub program)

- def keyword is used to define a function.
- Give the function name after def keyword, followed by which arguments are given.
- End with a **colon** (:).
- Inside the function, add the program statements to be executed.
- End with or without a **return** statement.

CS25C02

Syntax:

Python

```
def function_name(arguments):
    # function body/statements
    # return statement (optional)
```

Example:

Python

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

Function Prototype:

Functions can be categorized based on arguments and return type:

- Function **without Argument** & **without return** type.
- Function **with Argument** & **without return** type.
- Function **without Argument** & **with return** type.
- Function **with Argument** & **with return** type.

Type	**Definition**	**Example Usage**
Without Return Type, Without Argument	def add (): a = int(input("enter a :")) b = int(input("enter b:")) c = a + b print (c)	add ()
Without Return Type, With Argument	def add (a, b):	add (a, b)

CS25C02

Type	Definition	Example Usage
	c = a + b print (c) a = int(input(“enter a:”)) b = int(input(“enter b:”))	
With Return Type, Without Argument	def add (): a = int(input(“enter a:”)) b = int(input(“enter b:”))	add ()

Type	Definition	Example Usage
	c = a + b **Return c** Print (c)	
With Return Type, With Argument	def add (a, b): c = a + b **Return c** Print (c) a = int(input(“enter a:”))	add (a, b)

CS25C02

Type	Definition	Example Usage
	b = int(input(“enter b:”))	

---

3. Decision-Making Statements (Selection) ⚖️

Decision-making statements allow a program to execute specific blocks of code based on whether a condition is true or false.

2.2.1 Conditional (if):

if is used to test a condition. If the condition is **true**, the statements inside the if block will be executed.

Syntax:

Python

```
if condition:
    Statement
```

Example:

Python

```
a = 10
if a > 5:
    print("a is greater than 5")
```

Flowchart:

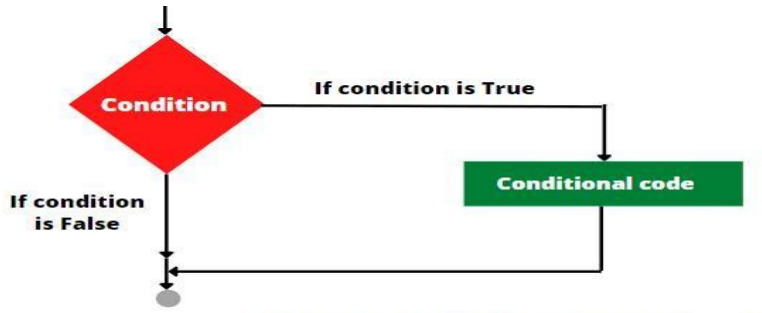


Fig: Flowchart of single selection if statement

Program Example:

Program to provide bonus mark if the category is sports	**Output**
m=eval(input("enter ur mark out of 100"))	enter your mark out of 100
c=input("enter ur categery G/S")	85
if(c=="S"):	enter your categery G/S
m=m+5	S
print("mark is",m)	mark is 90

2.2.2 if-else condition (Alternative Statement)

CS25C02

The else statement is combined with if. If the condition is **true**, statements inside the if execute; otherwise (if the condition is **false**), the code inside the else part executes. The alternatives are called **branches**.

Syntax:

Python

```
if condition:
    Statement 1 # Executed if condition is TRUE
else:
    Statement 2 # Executed if condition is FALSE
```

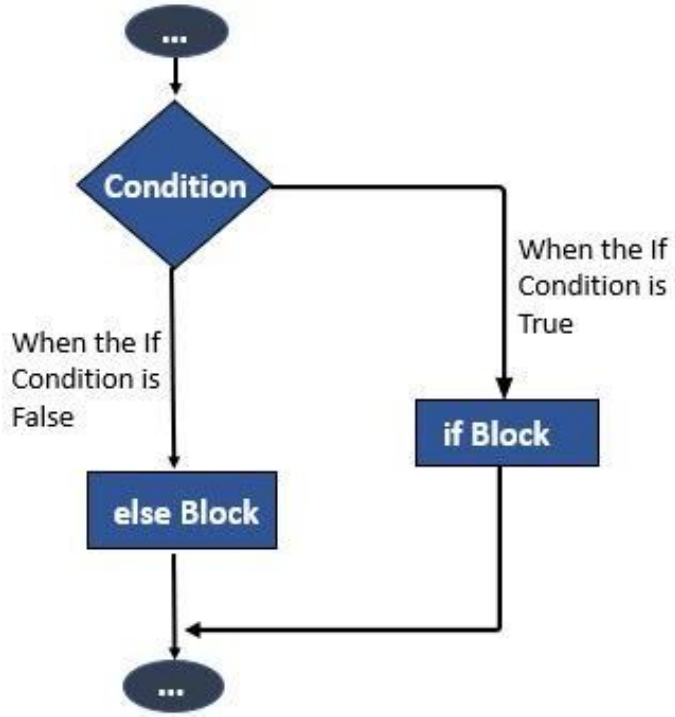
Example:

Python

```
if a < 10:
    print("The number is less than 10")
else:
    print("The number is not less than 10")
```


Flowchart:

CS25C02



2.2.3 Chained Condition: if-elif-else

- The elif is short for **else if**.
- This is used to check **more than one condition** sequentially.
- If condition1 is **False**, it checks condition2 of the elif block, and so on.
- If all the conditions are **False**, then the else part is executed.
- Among the several if-elif-else parts, **only one** part is executed.
- The if block can have only one else block, but it can have **multiple elif blocks**.

Syntax:

Python

```
if condition1:
    # statements
                                elif condition2: # statements
```


CS25C02

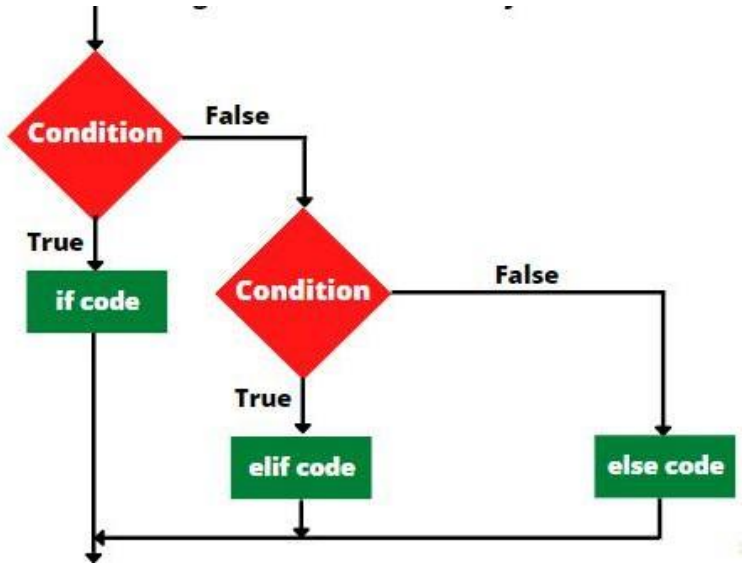
```
else:
    # statements
```

Example:

Python

```
def datatypes():
    # Assume 'a' is defined outside or passed in
    # This example demonstrates the structure, not functional data type checking.
    if a == A, a, B, b…Z, z: # Conceptual condition
        print("a:", type(a))
    elif a == 0, 1,….: # Conceptual condition
        print("a:", type(a))
    else:
        print("invalid data")
```


Flow Chart:



---

4. Iterative Constructs (Loops)

Iterative constructs allow a block of code to be executed repeatedly.

CS25C02

CS25C02

2.3.1 for loop

The for loop is used to iterate over a **sequence** or other **iterable object**. It executes the block of code once for each item in the sequence.

Syntax:

Python

```
For variable in sequence:
    Body of for loop.
```

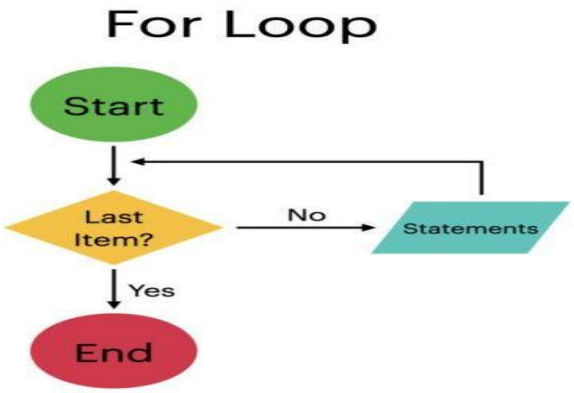
Example:

Python

```
for i in range (10):
    print(i)
```

O/P: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$.

Flow chart:



2.3.2 while loop

The while loop repeatedly executes a block of code as long as a specified **condition is true**. If the condition becomes false, the loop terminates.

Syntax:

Python

```
While condition:
```

```
# code block
```

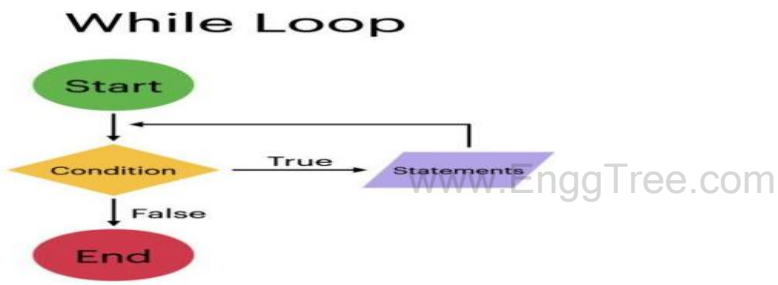
Example:

Python

```
count = 0
while count < 5:
    print(count)
    count = count + 1
```

O/P: $0, 1, 2, 3, 4$

Flowchart:



2.3.3 Nested Loop

A **nested loop** is a loop inside another loop. The inner loop completes all its iterations for every single iteration of the outer loop.

Flow Chart:

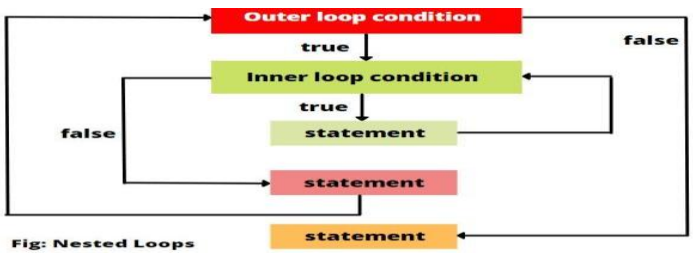


Fig: Nested Loops

Syntax:

Python

```
while condition1: # Outer loop
```


CS25C02

```
while condition2: # Inner loop
    # code block of inner loop
# code block of outer loop
```

Example:

Python

```
i = 0
while i < 3:        # Outer loop
    j = 0
    while j < 2:    # Inner loop
        print(f"i = {i}, j = {j}")
        j = j + 1
    i = i + 1
```


Output:

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

---

5. Jump Statements (Control Transfer) 🚀

Jump statements are used to transfer control within loops, altering the normal flow of iteration.

2.4.1 BREAK Statement

CS25C02

The break statement is used to **exit a loop permanently**. Regardless of the loop condition, when encountered, it immediately terminates the **inner most loop**.

Syntax:

Python

```
break
```

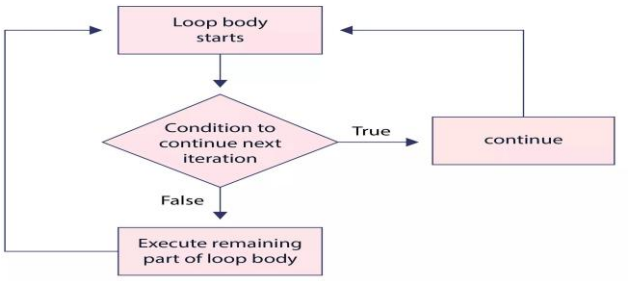
Example:

Python

```
for x in range(10):
    if x == 3:
        break
    print(x)
```

O/P: $0, 1, 2$

Flowchart:



2.4.2 CONTINUE Statement

- The continue statement **skips the current iteration** of the loop and moves to the next iteration.
- It does **not exit** the loop.

Syntax:

Python

```
Continue
```


CS25C02

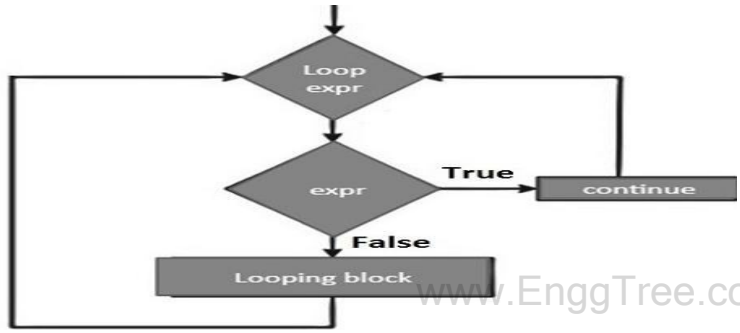
Example:

Python

```
for x in range(10):
    if x == 3:
        continue
    print(x)
```

O/P: $0, 1, 2, 4, 5, 6, 7, 8, 9$ (Note: 3 is skipped)

Flow Chart:



CS25C02

2.4.3 PASS: (DO NOTHING)

The pass statement is a **placeholder**. It is used when a statement is **syntactically required** but you don't want to execute any code.

Syntax:

Python

```
pass
```

Example:

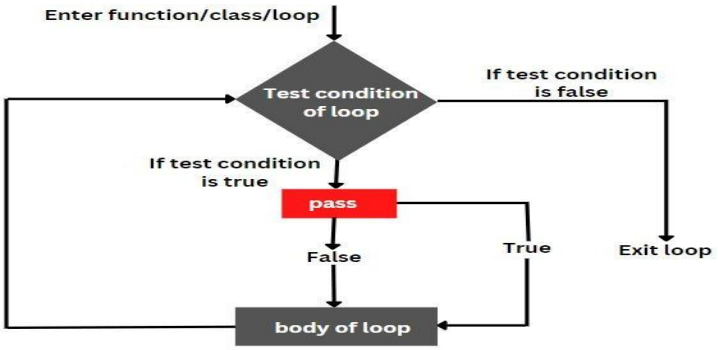
Python

```
for x in range(5):
    if x == 3:
        pass # To be implemented later
```

print(x)

O/P: $0, 1, 2, 3, 4$

Flow Chart:



---

6. Fruitful Functions and Recursion 🍎

2.5 Fruitful Functions

Functions that **return values** are called as **fruitful functions**.

- Return values
- Parameters
- Local and global scope (mentioned but not detailed)
- Function composition
- Recursion

2.5.1 Return values

A fruitful function is one that computes a result and **returns a value** to the caller using the **return** statement.

Syntax:

Python

```
def function_name (parameter):
    # function logic
```


CS25C02

```
    return value
```

Example:

Python

```
def add(a, b):
    return a + b
result = add(3, 4)
print(result)
```

O/P: 7

2.5.2 Parameters

Parameters are the values that are passed into a function. When the function is called, they are place holders that allow the function to accept input and perform its task with different values.

Syntax:

Python

```
def function_name(p1, p2): # p1, p2 are parameters
    # function body
```

Example:

Python

```
def greet(name):
    return f"Hello, {name}!"
greeting = greet("Joe")
print(greeting)
```

O/P: Hello, Joe!

2.5.3 Function Composition

CS25C02

It is the process of **combining two or more functions** to produce a new function. In Python, you can compose functions by **calling one function inside another** function.

Example:

Python

```
def square(x):
    return x * x
def double (x):
    return x * 2
def square_then_double (x):
    return double(square(x)) # Composition
# square_then_double(3) -> double(9) -> 18
```


2.5.4 Recursion

Recursion is a property in which one function calls **itself** repeatedly. The parameters are changed on each call.

CS25C02

Syntax:

Python

```
def recursive_function(parameter):
    if base_condition:
        return base_value
    else:
        return recursive_function(modified_parameter)
```

Example:

Python

```
def factorial(n):
```

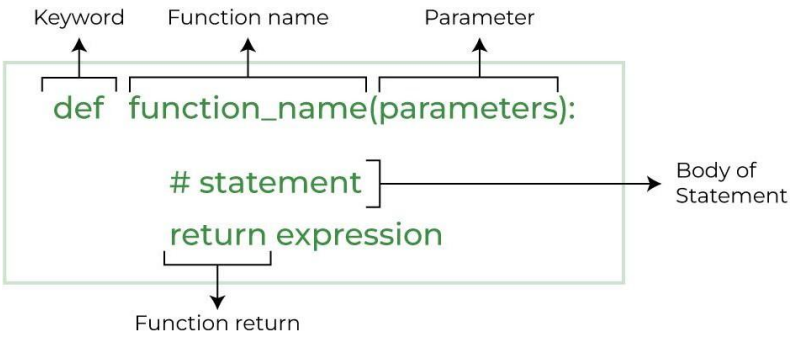
```
    if n == 0 or n == 1:
        return 1 # Base case
    else:
        return n * factorial(n - 1) # Recursive step
print(factorial(5))
```

O/P: 120

UNIT III FUNCTIONS

Hiding redundancy, complexity; Parameters, arguments and return values; formal vs actual arguments, named arguments, Recursive & Lambda Functions.

3. Introduction to Functions



Definition

A **function** is a **reusable block of code** that performs a specific task. Instead of writing the same code again and again, we put it inside a function and reuse it.

Why Functions are Needed?

- **Hiding Redundancy:** Repeated code can be **avoided**.
- **Hiding Complexity:** Large programs can be divided into smaller, simple parts.
- **Reusability:** Write once, use many times.
- **Readability:** Programs become easier to understand.

Example Program

Without function (Redundant)	With function (Reusable)
print("Area of rectangle:", 10 * 5)	def area(length, width):
print("Area of rectangle:", 7 * 4)	return length * width
	print("Area of rectangle:", area(10, 5))
Output: Area of rectangle: 50	print("Area of rectangle:", area(7, 4))

CS25C02

| Area of rectangle: 28 | Output: Area of rectangle: 50 |

| | Area of rectangle: 28 |

3.1 Need for Functions: Hiding Redundancy & Complexity

Definition

In programming, functions are used to **hide redundancy** and **reduce complexity**.

Hiding Redundancy

- Redundancy means repeating the same code multiple times.
- Functions help us avoid writing the same block of code again and again by grouping it into a reusable function.

Hiding Complexity

- Complex problems can be divided into smaller, easy-to-understand tasks using functions.
- Each function handles only one task, making the overall program easier to read, understand, and debug.

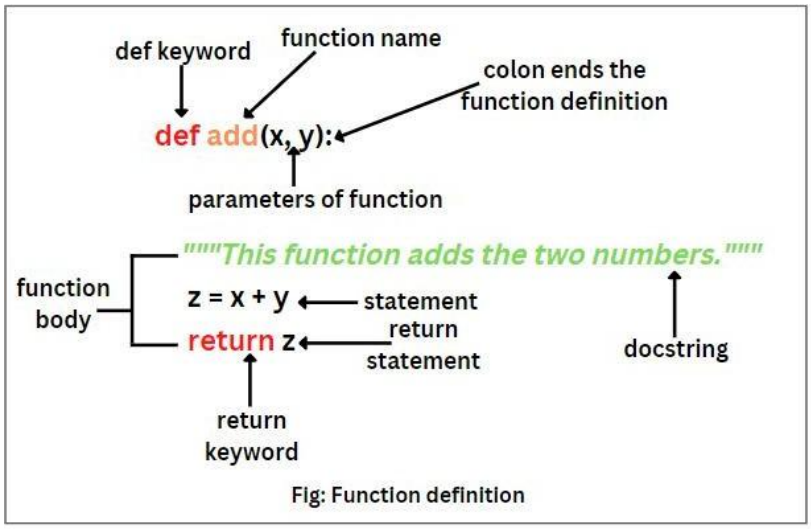


Fig: Function definition

CS25C02

Example 1: Without Function (Redundant Code)

Python

```
# Program without using functions
print("Square of 2 is:", 2 * 2)
print("Square of 3 is:", 3 * 3)
print("Square of 4 is:", 4 * 4)
```

Output:

Square of 2 is: 4

Square of 3 is: 9

Square of 4 is: 16

Here, the logic x*x is repeated again and again $\rightarrow$ **Redundancy**.

Example 2: With Function (Redundancy Removed)

Python


```
def square(x):
  return x * x
print("Square of 2 is:", square(2))
print("Square of 3 is:", square(3))
print("Square of 4 is:", square(4))
```

Output:

Square of 2 is: 4

Square of 3 is: 9

Square of 4 is: 16

Now, the code x*x is written only once inside the function $\rightarrow$ **Redundancy hidden**.

Example 3: Hiding Complexity with Functions

CS25C02

Python

```
def get_marks():
  return int(input("Enter marks:"))


def calculate_grade(marks):
  if marks >= 90:
    return "A"
  elif marks >= 75:
    return "B"
  elif marks >= 50:
    return "C"
  else:
    return "Fail"

def display_result(grade):
  print("Your Grade is:", grade)


# Main Program
marks = get_marks()
grade = calculate_grade(marks)
display_result(grade)
```

Output (Sample Run):

Enter marks:82

Your Grade is: B

Here, the complex task (grading system) is split into 3 smaller functions:

1. get_marks() – Input handling
2. calculate_grade() – Logic part
3. display_result() – Output display

This makes the program easier to read and maintain $\rightarrow$ Complexity hidden.

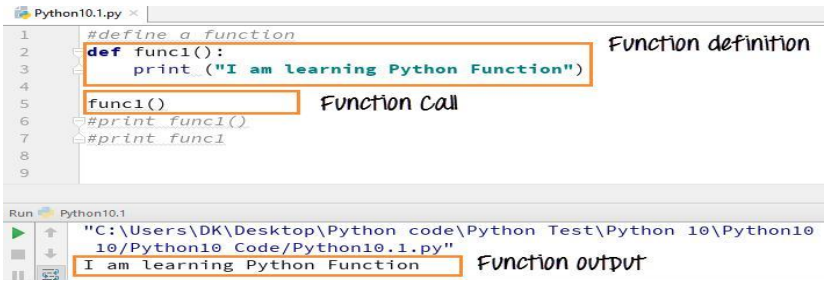
3.2 Function Definition and Calling

A function in Python is defined using the **def keyword**, followed by the function name, parameters, and a block of statements. Functions must be **called** to execute their code.

Syntax:

Python

```
def function_name(parameter1, parameter2, ...):
    # Function block (code to be executed)
    return result # Optional
```



Example:

Python

```
# Definition (Creates the function)
def say_hello():
  print("Hello!")
```

```
# Call (Executes the function)
say_hello()
```

- **Definition:** Creates the function and assigns it a name.
- **Call:** Executes the function by its name, optionally passing arguments.

3.3 Parameters and Arguments

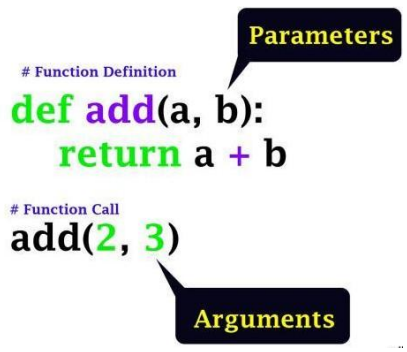
Functions can receive inputs through **parameters** and **arguments**.

- **Parameters:** Variables defined inside parentheses of the function **definition**. (Placeholders).
- **Arguments:** Actual values or variables passed during the function **call**. (Actual Data).

Example:

Python

```
def calculate_sum(a, b): # a and b are PARAMETERS
  return a + b


result = calculate_sum(10, 5) # 10 and 5 are ARGUMENTS
```



CS25C02

Types of Arguments:

- **Positional Arguments** – Order matters.

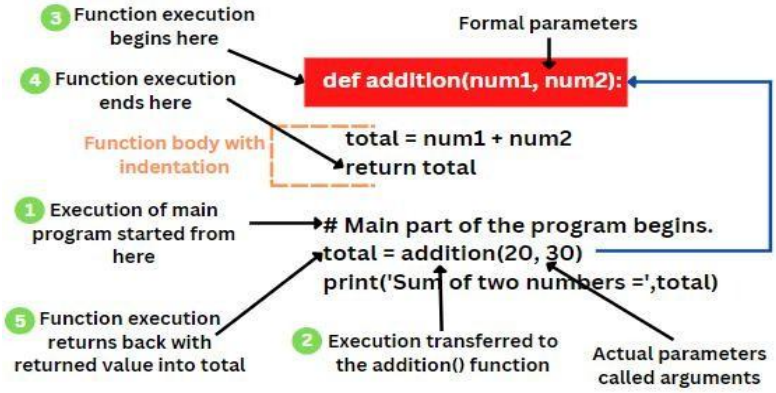
- **Default Arguments** – Provide fallback values.
- **Variable-Length Arguments** – *args (non-keyword), **kwargs (keyword arguments).

3.4 Return Values

Functions can send results back using the **return** statement.

- **Single Return:** Function returns one value.
- **Multiple Returns:** Functions can return **tuples** containing multiple values.

```
def divide(a, b):
    quotient = a // b
    remainder = a % b

    return quotient, remainder
```

Example:

Python

```
def safe_divide(a, b):
  if b == 0:
    return "Cannot divide by zero"
```


CS25C02

```
    return a / b


output1 = safe_divide(10, 2) # output1 will be 5.0
output2 = safe_divide(10, 0) # output2 will be "Cannot divide by zero"
```

Here, the function hides the complexity of division and exposes clear output.

3.5 Formal vs Actual Arguments

Definition

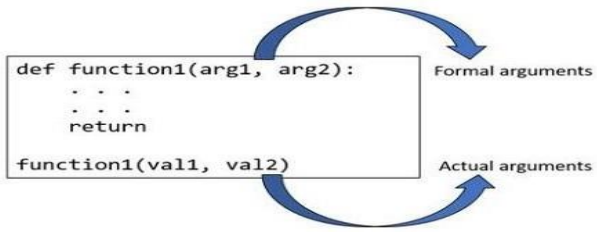
Formal Arguments

- The **variables declared in the function definition**.
- They act as **placeholders** to receive values when the function is called.
- Example: In def add(a, b):, **a and b are formal arguments**.

Actual Arguments

- The **real values** (or variables) that are **passed when calling the function**.
- They replace the formal arguments during execution.
- Example: In add(5, 10), the values **5 and 10 are actual arguments**.

Example Program 1: Simple Addition



Python

```
def add(a, b): # a, b are Formal Arguments
    result = a + b
```

```
  return result


sum_val = add(5, 10) # 5, 10 are Actual Arguments
print("Sum:", sum_val)
```

Output:

Sum: 15

Example Program 2: Greeting Function

Python

```
def greet(name): # name is the Formal Argument
  print("Hello,", name)


user = "Daniel"
greet(user) # user (value "Daniel") is the Actual Argument
```

Output:

Hello, Daniel

This distinction is important because formal arguments act as placeholders, while actual arguments supply data.

Basis	**Formal Argument**	**Actual Argument**
Definition	Variables in the function definition	Values given in the function call
Existence	Exists only inside the function (Local scope)	Exists in the calling program

Basis	Formal Argument	Actual Argument
Example (in add(a,b))	a, b	5, 10 in add(5,10)

3.6 Named Arguments

In Python, arguments can be passed by **explicitly mentioning the parameter name**, making code clearer and avoiding positional confusion.

Example:

Python

```
def power_calc(base, exponent):
  return base ** exponent


# Positional call (order matters)
print(power_calc(2, 3)) # Output: 8 (2^3)


# Named argument call (order doesn't matter, names are used)
print(power_calc(exponent=3, base=2)) # Output: 8 (2^3)
```


Advantages:

- Improves **readability**.
- Reduces errors in large functions.
- Allows skipping some parameters if defaults are defined.

3.7 Recursive Functions

CS25C02

A **recursive function** is one that **calls itself**. It is often used to solve problems that can be broken down into smaller, similar sub-problems.

Rules for Recursion:

1. A **base case** must stop recursion.
2. Each recursive call should **simplify the problem**.

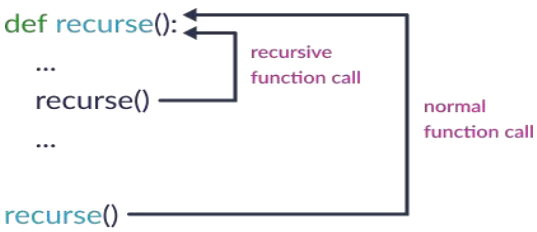
Example – Factorial:

The factorial of a number n is $n \times (n-1)!$

Python

```
def factorial(n):
  if n == 0 or n == 1:
    return 1  # Base Case: Stops the recursion
  else:
    # Recursive Call: Simplifies the problem (n-1)
    return n * factorial(n - 1)


print(factorial(5))
```


Output: 120 ($5 \times 4 \times 3 \times 2 \times 1$)



- **Advantages:** Reduces code length, natural representation of mathematical definitions.
- **Disadvantages:** Higher memory usage (stack calls), risk of **infinite recursion** if the base case is missing.

CS25C02

3.8 Lambda (Anonymous) Functions

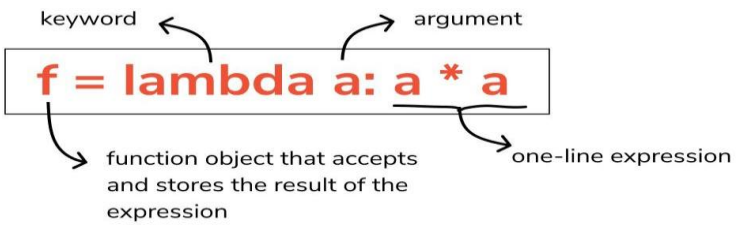
Lambda functions are small, **unnamed functions** created with the **lambda keyword**. They are useful for short operations where defining a full function is unnecessary.

Syntax:

Python

lambda arguments: expression

Example:



Python


```
# Define a lambda function to add two numbers

add_lambda = lambda a, b: a + b


# Call the lambda function

print(add_lambda(10, 7))
```

Output: 17

- Can be used with higher-order functions like map(), filter(), and reduce().

Example with filter():

Python

```
numbers = [1, 2, 3, 4, 5, 6]

# Filter to keep only even numbers

even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
```


CS25C02

```
print(even_numbers)
```

Output: [2, 4, 6]

CS25C02

CS25C02

UNIT -4 STRINGS & COLLECTIONS

String Comparison, Formatting, Slicing, Splitting, Stripping, List, tuples, and dictionaries, basic list operators, searching and sorting lists; dictionary literals, adding and removing keys, accessing and replacing values. Practical: String manipulations and operations on lists, tuples, sets, and dictionaries. (Minimum three)

4. Collections and String Manipulation

4.1 Strings

A **String** is basically a **sequence of characters.**

Example:

Python

```
str1 = "HELLO PYTHON"

str2 = 'Hello python 12'

print(str1)

print(str2)
```

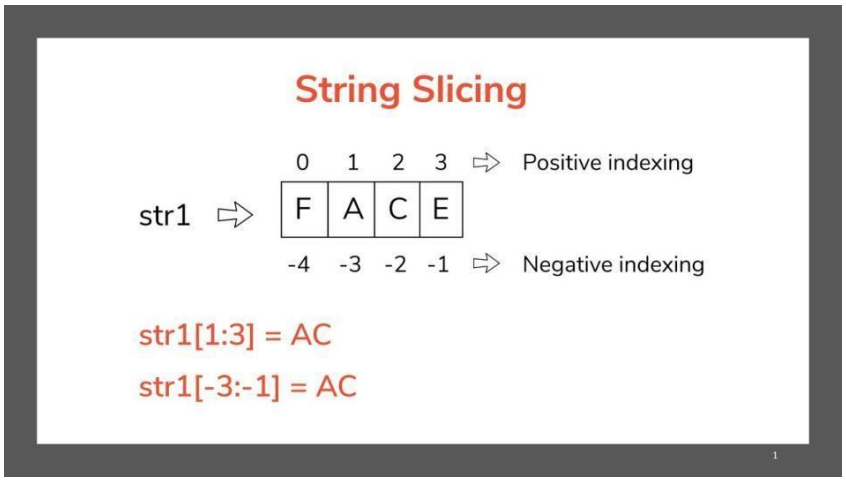
O/P:

```
HELLO PYTHON

Hello python 12
```

4.1.1 String Slicing:

A **substring** of a string is obtained by taking a **slice**. Similarly, we can slice a list to refer to a sublist of items.

Syntax: string[start:end:step]



Example:

Python

```
str_val = "Hello python"
print(str_val[0:5])
print(str_val[0:])
print(str_val[8:])
print(str_val[1:12])
```

O/P:

```
Hello
Hello python
hon
ello python
```

4.1.2 Immutability:

Strings in Python are **immutable**. We **cannot change** the existing string.

Python

```
str_val = "Hello python"
```


CS25C02

str_val[0] = 'h' # Attempting to change the first character

Output: TypeError (String does not support item assignment)

To make the desired changes, we need to take a **new string** and manipulate it as per our requirements.

Python

str_val = "Hello python"

new_str = str_val[0].upper() + str

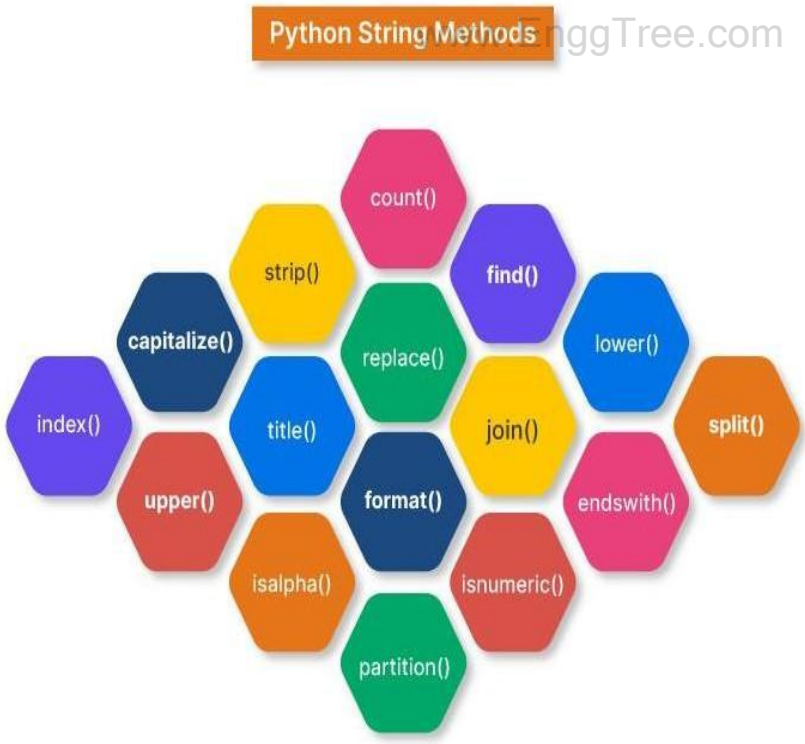
_val[1:] print(new_str)

O/P:

Hello python

4.1.3 String Functions and Methods:

CS25C02



1. String Concatenation:

Joining of two or more strings is called String Concatenation (using the + operator).

Python

```
str1 = "Hello"

str2 = "Python"

print(str1 + str2) # Output: HelloPython
```

2. String Comparison:

Comparison operators (>, <=, ==) are used based on alphabetical order (ASCII/Unicode values).

Python

```
str_a = "Hello"

str_b = "Python"

print(str_a > str_b) # Output: False (H comes before P)
```

3. String Repetition:

We can repeat the string using the * operator.

Python

```
str_rep = "Hello"

print(str_rep * 4)

# Output: Hello Hello Hello Hello.
```

4. Membership Test:

The membership of a particular character is determined using the keywords in and not in.

Python

```
print('H' in "Hello") # Output: True

print('z' in "Hello") # Output: False
```

Python String Methods:

1. **count():** Counts the occurrences of a substring.

CS25C02

2. **capitalize():** Capitalizes the first letter of the string.
3. **find():** Finds the first index of the needed value (returns -1 if not found).
4. **index():** Similar to find(), but raises a ValueError if not found.
5. **isalnum():** Returns true if all characters are alphabets or numbers.
6. **isdigit():** Returns true if all characters are numbers.
7. **islower():** Checks if all cased characters are lowercase.
8. **isupper():** Checks if all cased characters are uppercase.

String Module (Legacy/Advanced):

The string module contains a number of constants and functions to process the strings. We need to import it at the beginning (import string).

1. capwords(): Displays the first letter of each word in the string as capital.

Syntax: string.capwords(str)

Python


```
# After import string

str_val = "hello Python Program"

# string.capwords(str_val)

# O/P: Hello Python Program
```

2. upper() and lower() (Available as methods on string objects: str.upper(), str.lower()).

Syntax: String.upper(str), String.lower(str)

3. Translation of character to other form:

The maketrans() method returns a translation table for passing to translate(), that will map each character in from-ch into the character at the same position in to-ch. The from-ch and to-ch must have the same lengths.

Syntax: string.maketrans(from-ch, to_ch) (Used on Python 2; in Python 3, it's a string method.)

CS25C02

Example (Conceptual):

Python

```
str_original = "I love programming in python"

translation_table = str_original.maketrans("aeo", "012")

str_translated = str_original.translate(translation_table)

print(str_original)

print(str_translated)
```

O/P:

I love programming in python

I l2v1 pr2gramming in pyth2n

In the above program, we have generated a translation table for the characters a, e, and o. These characters will be mapped to 0, 1, and 2 respectively using the function maketrans. The actual conversion of the given string will take place using the function translate.

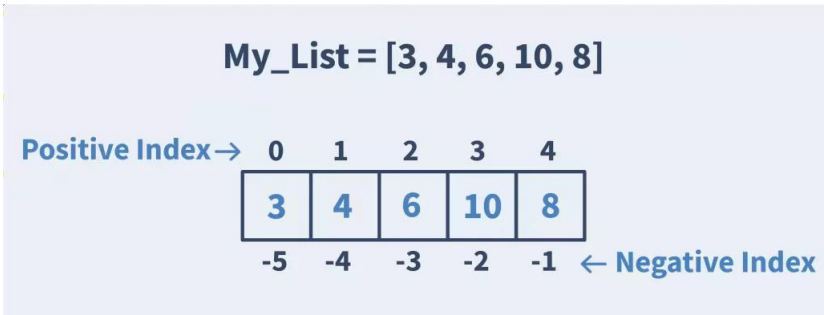
CS25C02

---

5. Python Collections Overview 📦

4.2.1 Lists

- Arrays and lists are both used in Python to store data.
- NumPy is used to create an array in Python (for numerical operations).
- A list is a versatile data structure that can store any type of data and can be indexed.
- The functions which can be performed in list and array are distinct.



Comparison of Call by Value vs. Call by Reference (Mutability)

Concept	Description	Example (Conceptual)
Call by Value (Immutable/Local Variable)	The original variable (e.g., an integer) cannot be modified in place inside the function. A **new copy is created**. Tuples are Immutable.	def change_val(x): x = 15; print("Inside function:", x) a = 5; change_val(a); print("Outside function:", a) **O/P:** Inside function: 15 Outside function: 5
Call by Reference (Mutable/Global Variable)	Mutable objects (e.g., lists) can be **modified in place**. Changes inside a function reflect outside. **Lists are mutable**.	def change_list(lst): lst[3] = 10; print("Inside function:", lst)

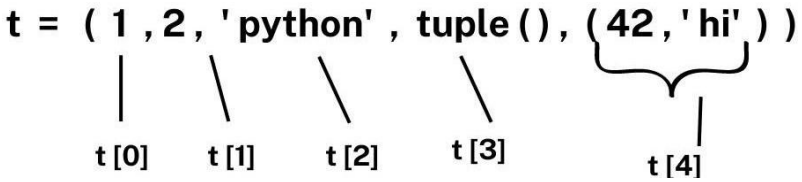
Concept	Description	Example (Conceptual)

		my_list = [1, 2, 3, 4];
		change_list(my_list);
		print("Outside function:", my_list)
		
		O/P: Inside function: [1, 2, 3, 10]
		
		Outside function: [1, 2, 3, 10]

CS25C02

4.2.2 Tuple

- A **Tuple** is a **sequence of values**. The values can be of any type and they are indexed as Integers (starting from 0).
- **Tuples are immutable** (cannot be changed after creation).
- Tuples are typically a comma-separated list of values.



Python

```
# Standard way with parentheses

Tuple1 = ('A', 'a', 'E', 'e', 'I', 'i', 'O', 'o', 'U', 'u')

# Creating without parentheses (Python infers it)

Tuple2 = 'A', 'a', 'E', 'e', 'I', 'i', 'O', 'o', 'U', 'u'
```

It is **not necessary** to enclose Tuples within parentheses.

- To create a tuple with a **single element**, we have to include a **comma** at the end.

Python

```
t = (1,)

print(type(t)) # Output: <class 'tuple'>
```

- Another way to create a tuple is by the built-in tuple() function. If we create a Tuple with no arguments, that is referred to as an **empty tuple**.

Python

```
t1 = tuple() # Empty tuple

t2 = tuple('fruit') # Tuple from an iterable

print(t2) # Output: ('f', 'r', 'u', 'i', 't')

print(t2[2]) # Output: u
```


Tuples are immutable, which means we cannot update or change the values of a tuple, but we can **replace one tuple with another**.

Example (Reassigning a new tuple):

Python

```
t = (1, 2, 3, 4, 5)

# Creates a NEW tuple by concatenating:

t = ('a',) + t[1:]

print(t) # O/P: ('a', 2, 3, 4, 5)
```

4.2.2.1 Tuple Assignment

1. An assignment to all of the elements in a tuple using a single assignment statement.
2. Python has a very powerful tuple assignment feature that allows a tuple of variables on the left of an assignment to be assigned values from a tuple on the right of the assignment.

CS25C02

3. The left side is a tuple of variables, the right side is a tuple of values; each value is assigned to its respective variable.
4. All the expressions on the right side are evaluated before any of the assignments.
5. Naturally, the no. of variables on the left and the no. of values on the right have to be the same.

Tuple Assignment Example:

Python

(a, b, c) = (1, 2, 3) # a=1, b=2, c=3

(x, y, z) = 1, 2 # Error: need more than 3 values to Unpack

4.2.2.2 Tuple Packing / Unpacking

- In **tuple packing**, the values on the right are 'packed' together in a tuple.

Python

t = 10, "hello", 3.14 # Packing

print(t) # Output: (10, 'hello', 3.14)

- In **tuple unpacking**, the values in a tuple on the right are 'unpacked' into the variables on the left. The right side can be any kind of sequence.

Python

a, b, c = t # Unpacking

print(a) # Output: 10

print(c) # Output: 3.14

4.2.3 Sets

- **Definition: Unordered collection of unique items**, written inside {}.
- **Key features:**
    - No **duplicate** elements.

CS25C02

- Supports mathematical operations like **union, intersection, difference**.

Example:

Python

nums = {1, 2, 2, 3, 4}

print(nums) # Output: {1, 2, 3, 4} (Duplicates removed)

4.2.4 Dictionaries:

A **Dictionary** organizes information in Python by **association** (key-value pairs), not by position (like lists). In Python, a dictionary associates a set of **keys** with data **values**.

A Python dictionary is written as a sequence of **key-value pairs** separated by commas. These pairs are called **entities** or **items**. The entire sequence is enclosed in **curly braces** {}.

NOTE: Elements in dictionaries are accessed via **keys** and not by their position.

Python

student = {"name": "Daniel", "age": 23} print(student ["name"]) # Output:

Daniel **4.2.4.1 Dictionary Operations**

Operation	Description	Example	Output
Create Dictionary	Define using {} with key–value pairs.	student = {"name":"Daniel", "age":23}	{'name': 'Daniel', 'age': 23}

CS25C02

Operation	Description	Example	Output
Access Value	Use key inside [].	student["name"]	Daniel
Get Value (safe)	Use .get() method (returns None if key is absent).	student.get("age")	23
Add Item	Assign new key–value.	student["course"] = "Python"	{'name':'Daniel', 'age':23, 'course':'Python'}
Update Value	Reassign value to existing key.	student["age"] = 24	{'name':'Daniel', 'age':24}
Remove Item (pop)	pop(key) removes by key and returns the value.	student.pop("age")	{'name':'Daniel'}
Remove Last Item	popitem() removes last inserted key-value pair.	student.popitem()	Remaining dict without last pair
Delete Key	Use del keyword.	del student["name"]	{} (if it was the only item)

CS25C02

Operation	Description	Example	Output
Clear Dictionary	Remove all items.	student.clear()	{}
Keys	Get all keys.	student.keys()	dict_keys(['name', 'age'])
Values	Get all values.	student.values()	dict_values(['Daniel', 23])
Items	Get key–value pairs.	student.items()	dict_items([('name', 'Daniel'), ('age', 23)])
Loop Keys	Iterate dictionary (by default iterates keys).	for k in student: print(k)	name age
Loop Items	Iterate key–value pairs.	for k, v in student.items(): print(k, v)	name Daniel age 23

CS25C02

4.2.4.2 Dictionary Methods

Method	Description	Example	Output
clear()	Removes all elements.	student.clear()	{}

Method	Description	Example	Output
copy()	Returns a shallow copy.	x = student.copy()	{'name':'Daniel','age':23}
fromkeys(seq, value)	Creates a new dictionary with keys from a sequence and a common value.	dict.fromkeys(['a','b'],'zero')	{'a':'zero','b':'zero'}
get(key, default)	Returns value for key; returns default if key not found.	student.get('age', 0)	23
items()	Returns view object with all key–	student.items()	dict_items([('name','Daniel'), ('age',23)])

CS25C02

Method	Description	Example	Output
	value pairs.		
keys()	Returns view object with all keys.	student.keys()	dict_keys(['name','age'])
values()	Returns view object with all values.	student.values()	dict_values(['Daniel', 23])
pop(key, default)	Removes item with given key and returns its value.	student.pop('age')	23
popitem()	Removes and returns the **last inserted** key–	student.popitem()	('age', 23)

CS25C02

Method	Description	Example	Output
	value pair.		
setdefault(key, default)	Returns value of key; if key not present, inserts with default value.	student.setdefault('course','Python')	Adds 'course':'Python' if not present
update(dict2)	Updates dictionary with key–value pairs from another dictionary.	student.update({'age':24})	{'name':'Daniel','age':24}

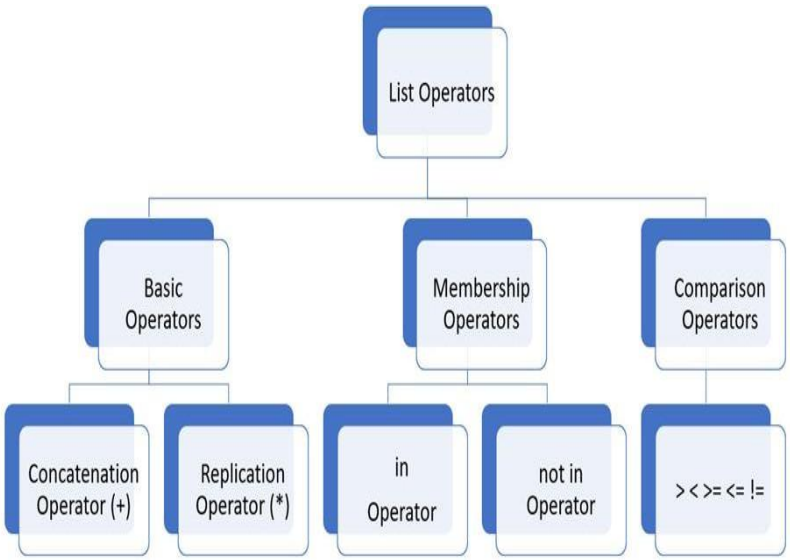
CS25C02

---

6. List Operations, Searching, and Sorting 📋

4.3 Basic List Operations

4.3.1 Basic List Operators in Python



Operator	Description	Example	Output
+	**Concatenates** two lists.	[1,2,3] + [4,5]	[1,2,3,4,5]
*	**Repeats** list elements.	[1,2] * 3	[1,2,1,2,1,2]
in	Checks if element **exists** in list (Membership).	3 in [1,2,3]	True
not in	Checks if element **does not exist**.	5 not in [1,2,3]	True
len()	Returns number of items.	len([10,20,30])	3
min()	Returns smallest element.	min([4,7,1])	1
max()	Returns largest element.	max([4,7,1])	7
sum()	Returns sum of elements.	sum([1,2,3,4])	10

CS25C02

4.3.2 List Methods in Python

(Assuming lst = [1, 2, 3] initially for examples)

Method	Description	Example	Output
append(x)	Adds element to the **end**.	lst.append(10)	[1,2,3,10]
insert(i, x)	Inserts element at **index i**.	lst.insert(1,99)	[1,99,2,3]
extend(list2)	Adds **all** elements of another list.	lst.extend([4,5])	[1,2,3,4,5]
remove(x)	Removes **first occurrence** of element x.	lst.remove(2)	[1,3,4]
pop(i)	Removes and **returns** element at index i (default last).	lst.pop()	Returns last element
clear()	Removes all elements.	lst.clear()	[]
index(x)	Returns **index** of first occurrence of x.	lst.index(3)	2
count(x)	Returns number of times x appears.	lst.count(2)	1
sort()	**Sorts list in place** (ascending by default).	lst.sort()	[1,2,3,4]

CS25C02

Method	Description	Example	Output
reverse()	**Reverses** the order of elements (in place).	lst.reverse()	[4,3,2,1]
copy()	Returns a shallow copy of the list.	lst.copy()	[1,2,3]

4.4 Searching and Sorting Lists

Lists are **mutable** (values can be changed, added, removed) and support **indexing and slicing** like strings.

4.4.1 Searching in Lists

Searching means finding whether an element exists in a list or locating its position.

1. Using Membership Operators (in, not in)

Simplest way to check existence.

Python

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

print("banana" in fruits) # True

print("kiwi" not in fruits) # True
```

2. Using index() Method

Returns the first index position of the given element. Raises a ValueError if the element is not found.

Python

```
my_list = [10, 20, 30]

print(my_list.index(20)) # 1

# my_list.index(50) # ValueError
```

3. Linear Search (Manual Search)

CS25C02

Checks each element one by one until the target is found. Works on unsorted lists.

Example:

Python

```
def linear_search(lst, target):
  for i in range(len(lst)):
    if lst[i] == target:
      return i
  return -1 # Not found


numbers = [15, 8, 42, 23, 4]
print(linear_search(numbers, 23)) # 3
print(linear_search(numbers, 100)) # -1
```


4. Binary Search (Efficient for Sorted Lists)

Faster but works only on sorted lists. Repeatedly divides the list into halves.

Example using bisect module:

Python

```
# After import bisect
# bisect.bisect_left(sorted_list, target)
# The bisect module is used for insertion/searching in sorted lists.
```

5. Searching with List Comprehension

Can be used to find all positions of an element.

Python

```
data = ["a", "b", "a"]
all_a_indices = [i for i, x in enumerate(data) if x == "a"]
```


CS25C02

print(all_a_indices) # [0, 2]

Summary (Searching)

- Simple Search $\rightarrow$ in, not in, index().
- Linear Search $\rightarrow$ Works for any list but slower for large data.
- Binary Search $\rightarrow$ Works only on **sorted lists**, faster ($\log n$ time).

4.4.2 Sorting in Lists

Sorting means arranging the elements of a list in a particular order (ascending or descending).

1. **Using sort() Method**
    - Sorts the list **in place** (changes the original list).
    - Default is **ascending** order.
    - Arguments: reverse=True (descending), key=function (custom logic).

CS25C02

Example:

Python

```
numbers = [5, 2, 9, 1, 7]

numbers.sort()

print(numbers) # [1, 2, 5, 7, 9] (Original list changed)

numbers.sort(reverse=True)

print(numbers) # [9, 7, 5, 2, 1]
```

2. **Using sorted() Function**
    - Returns a **new sorted list** (does **not** change the original).
    - Same parameters as sort().

Example:

Python

```
original = [5, 2, 9]
new_sorted = sorted(original)
print(original) # [5, 2, 9] (Original unchanged)
print(new_sorted) # [2, 5, 9]
```

3. Sorting with key Parameter

Allows sorting based on a specific rule (e.g., string length, absolute value).

- **Sort by string length:**

Python

```
words = ["a", "ccc", "bb"]
words.sort(key=len) # Sort based on the length returned by len()
print(words) # ['a', 'bb', 'ccc']
```


- **Sort case-insensitive:**

Python

```
names = ["Apple", "banana"]
names.sort(key=str.lower) # Sort based on lowercase version
print(names) # ['Apple', 'banana']
```

4. Reversing a List (Not Sorting)

The reverse() method only reverses the order without considering numerical or alphabetical sorting.

Python

```
list_rev = [1, 3, 2]
list_rev.reverse()
print(list_rev) # [2, 3, 1]
```

4.4.2.1 Difference between sort() and sorted()

Feature	sort()	sorted()
Changes Original List	**Yes** (in-place modification)	**No** (returns a new list)
Return Value	None	New sorted list
Speed	Slightly faster (no new list creation)	Slightly slower (creates copy)

Summary (Sorting)

- sort() $\rightarrow$ modifies the original list.
- sorted() $\rightarrow$ returns a new sorted list.
- Python uses **Timsort** (Hybrid of Merge + Insertion Sort) internally for efficiency.

CS25C02

---

7. Dictionary Details and Operations 📚

4.5 Dictionary Literals, Keys, and Values

A dictionary in Python is an **unordered, mutable collection of key-value pairs**. It allows fast access to data using a **unique key** instead of an index.

4.5.1 Dictionary Literals

- A dictionary literal is created using **curly braces {}** with key-value pairs.
- **Syntax:** dictionary = {key1: value1, key2: value2, key3: value3}
- Keys must be **unique and immutable** (string, number, tuple). Values can be any data type.

Example:

Python

```
student = {"name": "Daniel", "age": 23, "department": "CSE"}
```

print(student)

Output: {'name': 'Daniel', 'age': 23, 'department': 'CSE'}

4.5.2 Dictionary Keys

- Keys act as **identifiers** for values and **must be unique**.
- If a duplicate key is added, the latest value **overwrites** the old one.
- Keys must be **hashable** (immutable).

Example (Overwrite):

Python

```
data = {"a": 1, "a": 2}
print(data) # Output: {'a': 2} (1 is overwritten by 2)
```

Accessing Keys:

Python

```
print(student.keys()) # dict_keys(['name', 'age', 'course'])
```


4.5.3 Dictionary Values

- Values can be **any data type** (list, dictionary, etc.).
- Values can be **repeated** (not unique).

Accessing a value using a key:

Python

```
print(student["name"]) # Arun (Raises KeyError if key DNE)
print(student.get("marks")) # [80, 85, 90] (Returns None if key DNE)
```

4.5.4 Dictionary Items (Keys + Values)

The items() method returns both keys and values as pairs (tuples).

Python

```
print(student.items()) # dict_items([('name', 'Arun'), ('age', 20), ('marks', [80, 85, 90])])
```


CS25C02

4.5.6 Applications of Dictionary Keys & Values

- **Student Database:** Roll number $\rightarrow$ Name.
- **Inventory Management:** Product ID $\rightarrow$ Quantity.
- **Phonebook:** Name $\rightarrow$ Phone number.

4.6 Accessing Values

Values are accessed using their keys.

Syntax: dictionary_name[key] or dictionary_name.get(key)

Example:

Python

```
student = {"name": "Daniel", "age": 23}

print("Name:", student["name"])

print("Age:", student["age"])
```


4.6.1 Adding Keys

Add a new key-value pair by assigning a value to a new key.

Syntax: dictionary_name[new_key] = value

Example:

Python

```
student["year"] = 3

# {'name': 'Daniel', 'age': 23, 'department': 'CSE', 'year': 3}
```

4.7 Replacing Values

Replace a value by assigning a new value to an **existing key**.

Syntax: dictionary_name[existing_key] = new_value

Example:

Python

CS25C02

student["age"] = 24

{'name': 'Daniel', 'age': 24, 'department': 'CSE', 'year': 3}

4.7.1 Removing Keys

Keys can be removed using del or pop().

Method	Syntax	Description
del	del dictionary[key]	Removes the key-value pair.
pop()	dictionary.pop(key)	Removes the pair and **returns the value**.

Example:

Python

student = {'name': 'Daniel', 'age': 24, 'department': 'CSE', 'year': 3}

removed_year = student.pop('year') # Returns 3

del student['department']

Result: {'name': 'Daniel', 'age': 24}

CS25C02

4.8 Dictionary Operations in Python (Summary Table)

Operation	Syntax / Method	Example Code	Output
Dictionary Literal (Creation)	dict_name = {key: value, ...}	student = {"name": "Daniel", "age": 23}	{'name': 'Daniel', 'age': 23}
Access Value	dict_name[key], dict_name.get(key)	student["name"]	Daniel

Operation	Syntax / Method	Example Code	Output
Add New Key	dict_name[new_key] = value	student["year"] = 3	... 'year': 3}
Replace Value	dict_name[existing_key] = new_value	student["age"] = 24	... 'age': 24 ...}
Remove Key (del)	del dict_name[key]	del student["year"]	... removed
Remove Key (pop)	dict_name.pop(key)	student.pop("age")	Returns 23
Check Key Exists	key in dict_name	"name" in student	True
Get All Keys	dict_name.keys()	student.keys()	dict_keys(['name', 'age'])

CS25C02

4.9 Adding, Removing, and Replacing Elements

Data Structure	Adding	Removing	Replacing
Lists (Mutable)	list.append(x), list.insert(i, x)	list.pop(i), list.remove(x), del list[i]	list[i] = new_value

Data Structure	Adding	Removing	Replacing
Dictionaries (Mutable)	dict[new_key] = value	dict.pop(key), del dict[key]	dict[existing_key] = new_value
Sets (Mutable)	set.add(x), set.update(iterable)	set.remove(x), set.discard(x)	**Not applicable** (Unordered, no indexing)

CS25C02

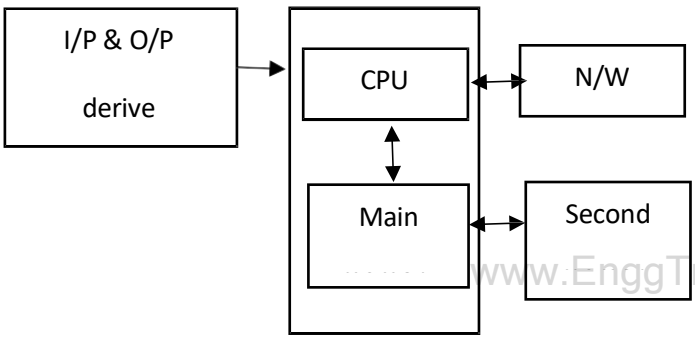
UNIT -5 FILES OPERATIONS

Create, Open, Read, Write, Append and Close files. Manipulating directories, OS and Sys modules, reading/writing text and numbers, from/to a file; creating and reading a formatted file (csv, tab-separated, etc.)

5. File Handling and System Interaction 📁

5.0 Introduction

A **File** is a named location on a disk to store related information. Since **RAM (Random Access Memory)** is volatile (loses its data when the computer is turned off), we use files for future use of the data.

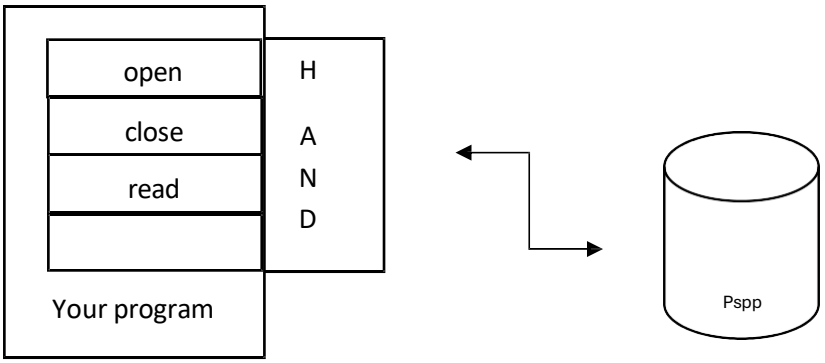


CS25C02

5.1 File Handling Concepts

The basic process of file handling involves three steps:

Step	Method	Description
1	**Open**	Establishes a link to the file on disk.
2	**Operation**	Reading, Writing, or Appending data.
3	**Close**	Saves changes and releases the file link/resources.



File Types:

- Text file
- Binary file

Text File:

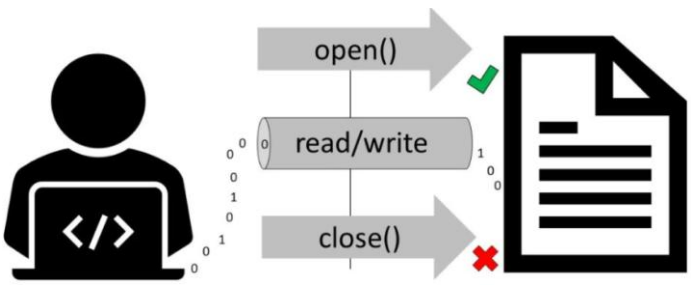
- It is a type of file which is used to store textual information.
- Structured as a sequence of lines.
- Data stored in a text file remains even if you shut down and restart a computer (non-volatile).

Binary File:

- Is any type of file that is **not a text file** (e.g., images, compiled programs, audio).

CS25C02

5.2 Creating and Opening Files



Opening a File:

In Python, there is a built-in function called **open()** to access a file.

The open() function Syntax:

Python

file_object = open(file_name, access_mode, buffering, encoding, ...)

- **file_name** $\rightarrow$ The name of the file that we want to access.
- **access_mode** $\rightarrow$ It determines the mode in which the file has to be opened (e.g., read, write, append).
- **buffering** $\rightarrow$ Controls buffering (0 for no buffering, 1 for line buffering, >1 for buffer size, <0 for system default).

Example:

Python

file = open("data.txt", "r") # Opens data.txt in read mode

Python File Modes:

Mode	Description
'r'	Open a file for **reading** (default). File pointer is at the beginning.
'w'	Open a file for **writing**. Creates a new file if it doesn't exist. **Overwrites** existing content.
'a'	Open a file for **appending** at the end. Creates file if it doesn't exist.
'rb'	Open a file for **reading only in a binary format**.
'wb'	Opening a file for **writing only in a binary format**.
'rt'	Opens a file for **reading** in text format (equivalent to 'r').

CS25C02

Mode	Description
'wt'	Opens a file for **writing** in text format (equivalent to 'w').
'r+'	Opens a file for both **reading and writing**.
'w+'	Opens a file for both **writing and reading** (Overwrites content).
'a+'	Open a file for both **appending and reading**.
'rb+'	Open a file for reading and writing in a **binary file**.
'wb+'	Open a file for writing and reading in a **binary file**.
'ab'	Open a file for appending in **binary format**.
'ab+'	Open a file for both appending and reading in **binary format**.

CS25C02

Closing a File:

In Python, the **close()** method is used to close a file, saving changes and freeing system resources.

Syntax:

Python

```
file_object.close()
```

Example:

Python

```
file = open("sample.txt", "w")

# operations...

file.close()
```

5.3 Reading and Writing Files

Reading a File

- To read a file in Python, we must open a file in a **reading mode** ('r' or 'r+').
- We can use the **read(size)** method to read a specific number of characters (size).
- If the size parameter is not specified, it reads the whole file and returns the content as a string.

Reading Methods:

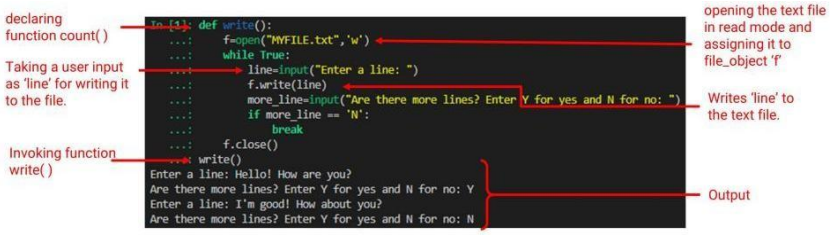
| Method | Description |

| :--- | :--- |

| file.read(size) | Reads size bytes/chars, or entire file if size is omitted. |

| file.readline() | Reads a single line from the file. |

| file.readlines() | Reads all lines and returns them as a list of strings. |



Example:

Python

```
file = open("data.txt", "r")

content = file.read(10) # Reads the first 10 characters

print(content)

file.close()
```

Writing a File

For writing contents to a file, we must open it in a **write mode** ('w', 'a', or 'r+').

Writing Methods:

| Method | Description |

| :--- | :--- |

| file.write(string) | Writes the specified string to the file. |

| file.writelines(list_of_strings) | Writes a list of strings to the file (does not add newlines automatically). |

Example:

Python

```
file = open("output.txt", "w")
file.write("First line.\n")
file.write("Second line.")
file.close()
```


CS25C02

5.4 Append and Close Files in Python

Append Mode ('a' or 'a+')

- Adds data at the **end** of an existing file **without deleting** previous content.
- 'a' $\rightarrow$ Append only.
- 'a+' $\rightarrow$ Append and Read.

Syntax:

Python

```
file = open("log.txt", "a")
file.write("New entry.\n")
```

file.close()

Using with (Recommended for File Handling)

The **with statement** automatically handles file closing, even if errors occur. This is often called a **context manager**.

Python

with open("filename.txt", "a") as file:

file.write("Text to append\n") # Automatically closes file when 'with' block exits

Key Points:

- Always **close files** after operations to ensure data integrity.
- Append mode prevents overwriting existing data.

5.5 Directory Manipulation (os and sys Modules)

CS25C02

Python provides the **os** and **sys** modules to interact with the underlying operating system and the Python runtime environment, respectively.

What is the os Module?

OS stands for the **Operating System module**. It allows developers to interact with the **operating system**, providing functions for **file and directory manipulation** (e.g., getting the current working directory, creating folders).

What is the sys Module?

Sys stands for the **System module**. It helps us to interact with the **Python runtime environment** (the interpreter). It provides access to variables used or maintained by the Python interpreter, such as command-line arguments and system-specific parameters.

Summary of Differences:

| Feature | os Module | sys Module |

| :--- | :--- | :--- |

| Purpose | Deals with operating system dependent functions (file handling, directories). | Provides access to Python interpreter variables and functions (runtime environment). |

| Usage | Used for file paths, creating/removing directories, environment variables. | Used for command-line arguments (sys.argv), exiting programs (sys.exit()), Python version info. |

| Example Functions | os.getcwd(), os.mkdir(), os.remove(), os.environ | sys.argv, sys.exit(), sys.version, sys.path |

5.5.1 Accessing Details

```
# PythonCode.py executed with 'python PythonCode.py arg1 arg2 '
import sys
import os

# Accessing command-line arguments using sys.argv
script_name = sys.argv[0]
arguments = sys.argv[1:]

# Getting the absolute path of the script using os.path
script_path = os.path.abspath(script_name)

print(&amp;quot;Script Name:&amp;quot;, script_name)
print(&amp;quot;Arguments:&amp;quot;, arguments)
print(&amp;quot;Script Path:&amp;quot;, script_path)
```

Output:

- **sys Module** can access the command line arguments (sys.argv).
- **os Module** can access the current working directory (os.getcwd()).

Example (Conceptual Code):

Python

import sys

print(sys.argv) # Accesses command line arguments

import os

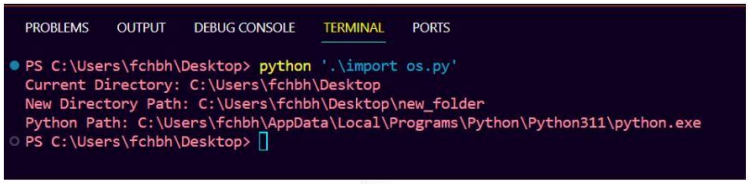
print(os.getcwd()) # Gets the Current Working Directory

5.5.2 Returning Path

CS25C02

```python
import os
import sys


# Getting current working directory using os.getcwd()
current_directory = os.getcwd()

# Creating a new directory using os.mkdir()
new_directory1 = os.path.join(current_directory, 'new_folder')
os.mkdir(new_directory1)

# Checking Python path using sys.executable
python_path = sys.executable

print(&amp;quot;Current Directory:&amp;quot;, current_directory)
print(&amp;quot;New Directory Path:&amp;quot;, new_directory1)
print(&amp;quot;Python Path:&amp;quot;, python_path)
```

Output:



Output

- **os Module** gives the Python Path within the system (related to files/directories).
- **sys Module** gives the Python Path (related to where the interpreter looks for modules, sys.path).

Example (Conceptual Code):

Python

import os

print(os.environ['PATH']) # Accessing OS environment path

import sys

print(sys.path) # List of directories where Python searches for modules

5.5.4 Reading/Writing Text and Numbers (File I/O Example)

(This section re-demonstrates file I/O operations for numbers, requiring type conversion.)

Writing numbers:

Python

num = 123

CS25C02

```
with open("numbers.txt", "w") as f:
    f.write(str(num) + "\n") # Numbers must be converted to string before writing
```

Reading numbers:

```
Python
with open("numbers.txt", "r") as f:
    read_num_str = f.readline().strip() # Read as string and remove whitespace
    read_num_int = int(read_num_str) # Convert string back to integer
    print(read_num_int)
```

5.7 Reading/Writing Formatted Files (CSV, TSV)

5.7.1 Working with CSV files in Python

A **CSV (Comma Separated Values)** file is a plain text file where fields are separated by **commas**. Python's built-in **csv** module simplifies working with these files.

5.7.1.1 Reading a CSV file (Reader Object)

Reading is done using the csv.reader object, which treats each row as a **list of strings**.

Example (Conceptual Code):

```
Python
import csv
rows = []
with open('data.csv', 'r') as csvfile:
    csvreader = csv.reader(csvfile)
    headers = next(csvreader) # Get headers (first row)
    for row in csvreader:
```


CS25C02

```
    rows.append(row)

# The 'rows' list now contains data, where each row is a list.
```

```
import csv
filename = "aapl.csv"  # File name
fields = []  # Column names
rows = []    # Data rows

with open(filename, 'r') as csvfile:
    csvreader = csv.reader(csvfile)  # Reader object

    fields = next(csvreader)  # Read header
    for row in csvreader:     # Read rows
        rows.append(row)

    print("Total no. of rows: %d" % csvreader.line_num)  # Row count

print('Field names are: ' + ', '.join(fields))

print('\nFirst 5 rows are:\n')
for row in rows[:5]:
    for col in row:
        print("%10s" % col, end=" ")
    print('\n')
```

Output

```
Total no. of rows: 185
Field names are:Date, Open, High, Low, Close, Adj Close, Volume

First 5 rows are:

2014-09-29 100.589996 100.690002  98.040001  99.620003  93.514290  142718700

2014-10-06  99.949997 102.379997  98.309998 100.730003  94.556244  280258200

2014-10-13 101.330002 101.779999  95.180000  97.669998  91.683792  358539800

2014-10-20  98.320000 105.489998  98.220001 105.220001  98.771042  358532900

2014-10-27 104.849998 108.040001 104.699997 108.000000 101.380676  220230600
```

5.7.1.2 Reading CSV Files Into a Dictionary (csv.DictReader)

This class reads each row as a **dictionary**, using the headers as **keys**.

Example (Conceptual Code):

Python

```
import csv

data_list = []

with open('employees.csv', 'r') as file:
    dict_reader = csv.DictReader(file)
    for row in dict_reader:
        data_list.append(row)

# The 'data_list' now contains dictionaries: [{'ID': '1', 'Name': '...'}, ...]
```

5.7.1.3 Writing to a CSV file (Writer Object)

Writing is done using csv.writer and the writerow() or writerows() methods.

Example (Conceptual Code):

Python

CS25C02

```
import csv
fields = ['Name', 'Age']
rows = [['Alice', 30], ['Bob', 25]]

with open('names.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(fields) # Writes the header row
    writer.writerows(rows) # Writes multiple data rows
```

5.7.1.4 Writing a dictionary to a CSV file (csv.DictWriter)

This class is used to map dictionary keys to CSV column headers.

Example (Conceptual Code):

Python


```
import csv
data = [{'Name': 'Alice', 'Age': 30}, {'Name': 'Bob', 'Age': 25}]
fieldnames = ['Name', 'Age']

with open('output_dict.csv', 'w', newline='') as csvfile:
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader() # Writes column headers based on fieldnames
    writer.writerows(data) # Writes all dictionaries as CSV rows
```

5.7.2 Simple Ways to Read TSV Files in Python

A **TSV (Tab-Separated Values)** file is similar to a CSV file, but fields are separated by a **tab character (\t)**.

5.7.2.1 Using read_csv(sep='\t') (Pandas Library)

CS25C02

The pandas library can handle TSV files effortlessly by setting the sep parameter to '\t'.

Example (Conceptual Code):

Python

```
import pandas as pd

# Sample TSV file: Dani.tsv

df = pd.read_csv("Dani.tsv", sep='\t')

# df is a DataFrame that simplifies viewing and analyzing tabular data.
```

5.7.2.2 Using csv.reader() (Built-in)

The built-in csv module can read TSV files by explicitly setting the delimiter parameter to '\t'.

Example (Conceptual Code):

Python


```
import csv

with open('Dani.tsv', 'r') as csvfile:
    tsv_reader = csv.reader(csvfile, delimiter='\t')
    for row in tsv_reader:
        print(row) # Each line is printed as a list of values.
```

5.7.2.3 Using Generator Expression (Minimal)

A lightweight, memory-friendly approach that processes each line as it's read using simple string methods.

Example (Conceptual Code):

Python

```
with open('Dani.tsv', 'r') as f:
    data_gen = (line.strip().split('\t') for line in f)
```


CS25C02

```
for row in data_gen:
    print(row)
```

5.7.2.4 Using csv.DictReader() for TSV

csv.DictReader() can also be used for TSV files by specifying the delimiter. It reads each row as an **ordered dictionary**.

Example (Conceptual Code):

Python

```
import csv
with open('Dani.tsv', 'r') as csvfile:
    dict_reader = csv.DictReader(csvfile, delimiter='\t')
    for row in dict_reader:
        print(row) # Each row is a dictionary using headers as keys.
```


CS25C02

UNIT -6 PACKAGES

Built-in modules, User-Defined modules, Numpy, SciPy, Pandas, Scikitlearn

6. Introduction to Modules and Packages 📦

What is a Module in Python?

A **module** is a simple Python file that contains a **collection of functions and global variables**, having a **.py** extension. It is an executable file. To organize all the modules, we have the concept called **Package** in Python.

Examples of modules:

1. Datetime
2. Regex
3. Random, etc.

Example (Conceptual):

Save the code in a file called demo_module.py:

Python

```
# demo_module.py

def myModule(val):
    print("This is My Module :", val)
```

Import the module named demo_module and call the myModule function:

Python

```
import demo_module

demo_module.myModule("Math")

# Output: This is My Module : Math
```

What is a Package in Python?

CS25C02

A **package** is a simple directory having a **collection of modules**. This directory contains Python modules and also an **__init__.py** file, by which the interpreter interprets it as a Package. The package is simply a namespace. A package can also contain sub-packages inside it.

Examples of Packages:

1. Numpy
2. Pandas

Example (Conceptual Structure):

MyPackage/

|-- __init__.py

|-- moduleA.py

|-- submodule/

| |-- __init__.py

| |-- moduleB.py

CS25C02

What is a Library in Python?

A **library** is a collection of related functionality of codes that allows you to perform many tasks without writing your own code. It is a reusable chunk of code that we can use by importing it into our program.

General Assumption: While a package is a collection of modules, a library is often considered a **collection of packages** that serves a high-level purpose (e.g., data science, machine learning).

Example:

Importing the pandas library and calling the read_csv method using an alias of pandas, pd.

Python

import pandas as pd

df = pd.read_csv("data.csv")

6.1 Built-in Modules

A Python Module is a file that contains built-in functions, classes, and variables. Python comes with many modules, each with its specific work.

6.1.1 What is a Python Module

A Python module is a file containing Python definitions and statements. A module can define functions, classes, and variables, and can also include runnable code. Grouping related code into a module makes the code easier to understand, use, and logically organized.

6.1.2 Create a Python Module

To create a Python module, write the desired code and save it in a file with a **.py** extension.

Example: calc.py

Python

```
# A simple module, calc.py

def add(x, y):

    return (x+y)


def subtract(x, y):

    return (x-y)
```


6.1.3 Import Module in Python

We can import the functions and classes defined in one module to another using the **import** statement.

When the interpreter encounters an import statement, it imports the module if the module is present in the **search path**. (A search path is a list of directories that the interpreter searches for a module.)

6.1.4 Syntax to Import Module in Python

Python

```
import module_name
```

Note: This imports the module only. To access the functions inside the module, the **dot (.) operator** is used.

Importing Modules in Python Example

Python

```
import calc

result = calc.add(10, 5)

# Output: 15
```

6.1.5 Python Import From Module

Python's **from** statement lets you import specific attributes (functions, classes, variables) from a module **without importing the module as a whole**.

Import Specific Attributes Example:

Python

```
# importing sqrt() and factorial from the module math

from math import sqrt, factorial

# We no longer need the 'math.' prefix

print(sqrt(16))     # Output: 4.0

print(factorial(6)) # Output: 720
```

6.1.6 Import All Names

The * symbol used with the from import statement is used to import **all** the names from a module to the current namespace.

Syntax:

Python

CS25C02

from module_name import *

What does import * do in Python?

The use of * has its advantages and disadvantages. If you know exactly what you will be needing from the module, it is not recommended to use * (to avoid namespace conflicts), else do so.

6.1.7 Module Search Path

Whenever a module is imported in Python, the interpreter looks for several locations:

1. It searches for the module in the **current directory**.
2. If not found, it searches each directory in the shell variable **PYTHONPATH** (an environment variable, consisting of a list of directories).
3. If that also fails, it checks the **installation-dependent list of directories** configured at the time Python is installed.

6.1.8 Directories List for Modules

The **sys.path** is a built-in variable within the sys module. It contains a list of directories that the interpreter will search for the required module.

Python

```
import sys

# print(sys.path)

# Output (Example structure): ['/home/user/workspace', '/usr/lib/python3.x', ...]
```

6.1.9 Renaming the Python Module

We can rename the module while importing it using the **as** keyword.

Syntax:

Python

```
Import Module_name as Alias_name
```

Example:

CS25C02

Python

```
import math as m

print(m.pi)

# Output: 3.14159265...
```

6.2 Packages

- **Packages** are namespaces which contain many modules and other packages.
- Along with modules, the package contains an **__init__.py** file. In fact, to be a package, there must be a file called __init__.py in the folder.

6.2.1 How to create a Package (Conceptual Example)

This involves setting up a directory structure and calling functions from modules within it.

Structure:


```
MyMaths/

|-- __init__.py

|-- Add.py

|-- Sub.py

|-- Mul.py

|-- Div.py

testing_arithmetic.py (Driver Program)
```

Example: Add.py content

Python

```
def add_fun(x, y):

    print(x + y)
```

(Similar structure for Sub.py, Mul.py, Div.py)

Driver Program (testing_arithmetic.py):

The modules are imported using the package structure:
Package.Module.Attribute.

Python

```
# Step 9: Import the functions (Assuming the directory structure works)

import MyMaths.Add.Add  # Assuming the module is named Add.py

import MyMaths.Sub.Sub

import MyMaths.Mul.Mul

import MyMaths.Div.Div


# Step 10: Calling the functions

print("The Addition of 10 and 20 is:")

MyMaths.Add.Add.add_fun(10, 20)

# ... other calls ...


# Output:

# The addition of 10 and 20 is: 30

# The Subtraction of 10 and 20 is: -10

# The Multiplication of 10 and 20 is: 200

# The Division of 10 and 20 is: 0.5
```

6.3 Built-in Modules (Standard Library)

Python built-in modules are a set of libraries that come **pre-installed** with the Python installation. They provide a wide range of functionalities, from file operations to mathematical computations.

Command to list all available modules:

Python

help('modules')

6.3.1 Advantages of Built-in Modules in Python

- **Reduced Development Time:** Used without installing external modules or writing lengthy code.
- **Optimized Performance:** Some are optimized using low-level code for efficiency.
- **Consistency & Standardization:** Part of the Python Standard Library, providing a consistent way to solve problems.
- **Documentation:** Comprehensive official documentation makes them easier to learn and utilize.
- **Maintainability:** Maintained by the core Python team, ensuring updates, bug fixes, and long-term viability.
- **Reduced Risk:** Avoids the risks associated with third-party libraries (security, discontinued support).

Key Built-in Modules Summary

Module	Purpose	Example Functionality
json	Encode/Decode **JSON** data.	json.dumps() (Python dict to JSON string).
tkinter	Standard **GUI** (Graphical User Interface) library.	Create windows, buttons, and text boxes.

Module	Purpose	Example Functionality
random	Generates **random numbers** and selections.	random.randint(1, 10), random.choice(list).
math	Advanced **mathematical functions** and constants.	math.sqrt(), math.pi.
datetime	Manipulation and reading of **date and time** values.	datetime.date.today(), datetime.datetime.now().time().
os	Interact with the **Operating System** (OS-level functionality).	os.getcwd() (Current Working Directory).
sys	Interact with the **Python runtime environment**.	sys.version, sys.argv (Command-line arguments).
re	Support for **Regular Expressions** (pattern matching).	re.search(), re.findall().
hashlib	Provides algorithms for **message digests or hashing**.	hashlib.sha256(), hashlib.md5().

CS25C02

Module	Purpose	Example Functionality
calendar	Operations and manipulations related to **calendars**.	calendar.month(year, month).
heapq	Functions for implementing **heaps** (priority queues).	heapq.heapify(), heapq.heappop().

6.5 User Defined Modules

User-defined modules are the modules which are created by the user to simplify their project. They can contain functions, classes, variables, and other code that you can reuse.

CS25C02

How to create a user-defined module?

1. Write the code for the functions/classes.
2. Save the file with a .py extension (e.g., calculator.py).

Example: calculator.py

Python

```
# calculator.py

def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

Using the User-Defined Module:

Python

```
import calculator
```

```
result = calculator.add(10, 5)
print(result)
# Output: 15
```

Importing Specific Attributes:

```
from calculator import subtract
result = subtract(10, 5)
print(result)
# Output: 5
```

Importing All Attributes:

```
from calculator import *
result = add(10, 5)
print(result)
# Output: 15
```


CS25C02

6.6 Popular Python Packages for Engineers

Python's versatility stems from its huge set of external libraries (often distributed as packages) which are essential for specialized tasks in fields like engineering, data science, and machine learning.

Library	Purpose	Key Features
NumPy	**Numerical computing**	Multidimensional arrays, element-wise operations, linear algebra.

Library	Purpose	Key Features
Pandas	**Data analysis** and manipulation	DataFrame & Series, time series handling, missing data cleaning.
Scikit-learn	**Machine learning**	Classification, regression, clustering, easy API.
Matplotlib	**Data visualization**	Line, bar, scatter plots; pyplot for simple plotting.
SciPy	**Scientific computing**	Integrals, optimization, statistical functions.
TensorFlow/PyTorch	**Deep learning**	Tensors, dynamic computation graph, runs on CPU/GPU.
Requests	**HTTP requests**	Supports GET, POST, cookies, API interaction.
Beautiful Soup	**Web scraping**	Parse HTML/XML, extract data easily.

CS25C02

These four packages—**NumPy, SciPy, Pandas, and Scikit-learn**—form the foundational stack for scientific computing, data analysis, and machine learning in Python. They are often used together in engineering and data-intensive applications.

1. NumPy (Numerical Python) 🔢

NumPy is the **fundamental package** for numerical computation in Python. It provides support for large, multi-dimensional arrays and matrices, along with a

large collection of high-level mathematical functions to operate on these arrays efficiently.

Key Features

- **ndarray:** The core object is the N-dimensional array (ndarray), which stores data in a contiguous block of memory, making operations much faster than standard Python lists.
- **Vectorization:** Enables operations to be performed element-wise without explicit looping, leading to significant performance gains (vectorization).
- **Mathematical Operations:** Includes functions for linear algebra, Fourier transforms, and random number generation.

Example: Array Operations

We perform a simple element-wise addition and multiplication, showcasing the power of vectorization.

Python

import numpy as np

CS25C02

```
# 1. Create NumPy arrays (Vectorization)
array_a = np.array([1, 2, 3, 4])
array_b = np.array([10, 20, 30, 40])

# 2. Element-wise operations (much faster than lists)
sum_array = array_a + array_b
product_array = array_a * 2

# 3. Linear Algebra (Dot Product)
dot_product = np.dot(array_a, array_b)
```

print("Original Array A:", array_a)

print("Element-wise Sum (A + B):", sum_array)

print("Element-wise Product (A * 2):", product_array)

print("Dot Product (A . B):", dot_product)

Output	**Explanation**
Original Array A: [1 2 3 4]	The base data structure.
Element-wise Sum (A + B): [11 22 33 44]	Computes $1+10, 2+20,$ etc.
Element-wise Product (A * 2): [2 4 6 8]	Multiplies every element by 2.
Dot Product (A . B): 300	Calculates $(1\times 10) + (2\times 20) + (3\times 30) + (4\times 40)$.

CS25C02

---

2. SciPy (Scientific Python) 🔬

SciPy is built upon NumPy and provides a vast library of modules for **scientific and technical computing**. It focuses on numerical routines like optimization, integration, interpolation, signal processing, and specialized statistical functions.

Key Features

- **Specialized Modules:** Offers modules for specific domains: scipy.integrate, scipy.optimize, scipy.linalg, scipy.stats, etc.
- **Algorithms:** Implements highly efficient numerical algorithms for complex mathematical problems that are rarely found in core Python or NumPy.
- **Interoperability:** Seamlessly integrates with NumPy arrays.

Example: Optimization (Minimizing a Function)

We use scipy.optimize to find the minimum value of a simple function, $f(x) = x^2 + 5x + 6$.

Python

```
import numpy as np

from scipy.optimize import minimize


# 1. Define the function to minimize

def quadratic_function(x):

    # f(x) = x^2 + 5x + 6. Minimum should be near x = -2.5

    return x**2 + 5 * x + 6


# 2. Provide an initial guess for x

x0 = np.array([0])


# 3. Run the optimization algorithm

result = minimize(quadratic_function, x0)


print(f"Function minimized at x = {result.x[0]:.2f}")

print(f"Minimum function value = {result.fun:.2f}")
```

Output	Explanation
Function minimized at x = -2.50	SciPy found the x value where the slope is zero (the vertex of the parabola).

Output	Explanation
Minimum function value = -0.25	This is the lowest possible value of the function.

---

3. Pandas (Panel Data) 📊

Pandas is the primary library for **data manipulation and analysis**. It introduces two powerful data structures: the **Series** (1D labeled array) and the **DataFrame** (2D labeled table, similar to a spreadsheet or SQL table).

Key Features

- **DataFrame:** The most popular structure, providing labeled rows and columns, making data easy to index, query, and manipulate.
- **Data Cleaning:** Excellent tools for handling missing data (NaN), merging, reshaping, and grouping data.
- **I/O Tools:** Easily read and write data from various formats (CSV, Excel, SQL databases).

Example: Data Analysis and Aggregation

We create a DataFrame, calculate a new column, and perform basic grouping.

Python

```
import pandas as pd


# 1. Create a DataFrame (data dictionary)
data = {
    'City': ['Delhi', 'Mumbai', 'Delhi', 'Chennai'],
    'Sales': [100, 150, 120, 180],
    'Cost': [50, 75, 60, 90]
```


CS25C02

```
}
df = pd.DataFrame(data)

# 2. Add a new calculated column
df['Profit'] = df['Sales'] - df['Cost']

# 3. Group and aggregate data
city_summary = df.groupby('City')['Profit'].sum()

print("--- DataFrame ---")
print(df)
print("\n--- Summary by City ---")
print(city_summary)
```


Output	**Explanation**
Profit column added (e.g., $100-50=50$).	Calculation across an entire column (vectorized).
City \n Delhi 110 \n Mumbai 75 \n Chennai 90	Grouped the DataFrame by City and summed the Profit for each city (e.g., Delhi: $50 + 60 = 110$).

---

4. Scikit-learn (sklearn) 🧠

Scikit-learn is the essential library for **Machine Learning (ML)** tasks in Python. It provides efficient tools for common ML algorithms, including classification, regression, clustering, model selection, and dimensionality reduction.

Key Features

- **Comprehensive Algorithms:** Implements dozens of supervised and unsupervised learning algorithms (Linear Regression, KNN, Decision Trees, SVM, etc.).
- **Consistent API:** All models follow a standardized API: model.fit(X, y) for training and model.predict(X) for prediction, making it easy to switch between models.
- **Integration:** Designed to work perfectly with NumPy arrays (input/output) and Pandas DataFrames.

Example: Simple Linear Regression

We use sklearn to train a simple Linear Regression model to predict a target variable based on input data.

Python

```
import numpy as np

from sklearn.linear_model import LinearRegression


# 1. Sample Data (X = feature, y = target)

# X needs to be 2D (matrix of features)

X = np.array([[1], [2], [3], [4], [5]])

y = np.array([2, 4, 5, 4, 5]) # Target (y is roughly 1*X + 2)


# 2. Create the model instance

model = LinearRegression()


# 3. Train the model (FITTING)

model.fit(X, y)
```


CS25C02

```
# 4. Predict a new value (PREDICTING)
new_X = np.array([[6]])
prediction = model.predict(new_X)

print(f"Trained Coefficient (Slope): {model.coef_[0]:.2f}")
print(f"Prediction for X=6: {prediction[0]:.2f}")
```

Output	**Explanation**
Trained Coefficient (Slope): 0.60	The model found the best-fit line with a slope of 0.60.
Prediction for X=6: 6.20	Using the trained model, it predicts the y value when $X=6$ is 6.20.

CS25C02