UNIT-2

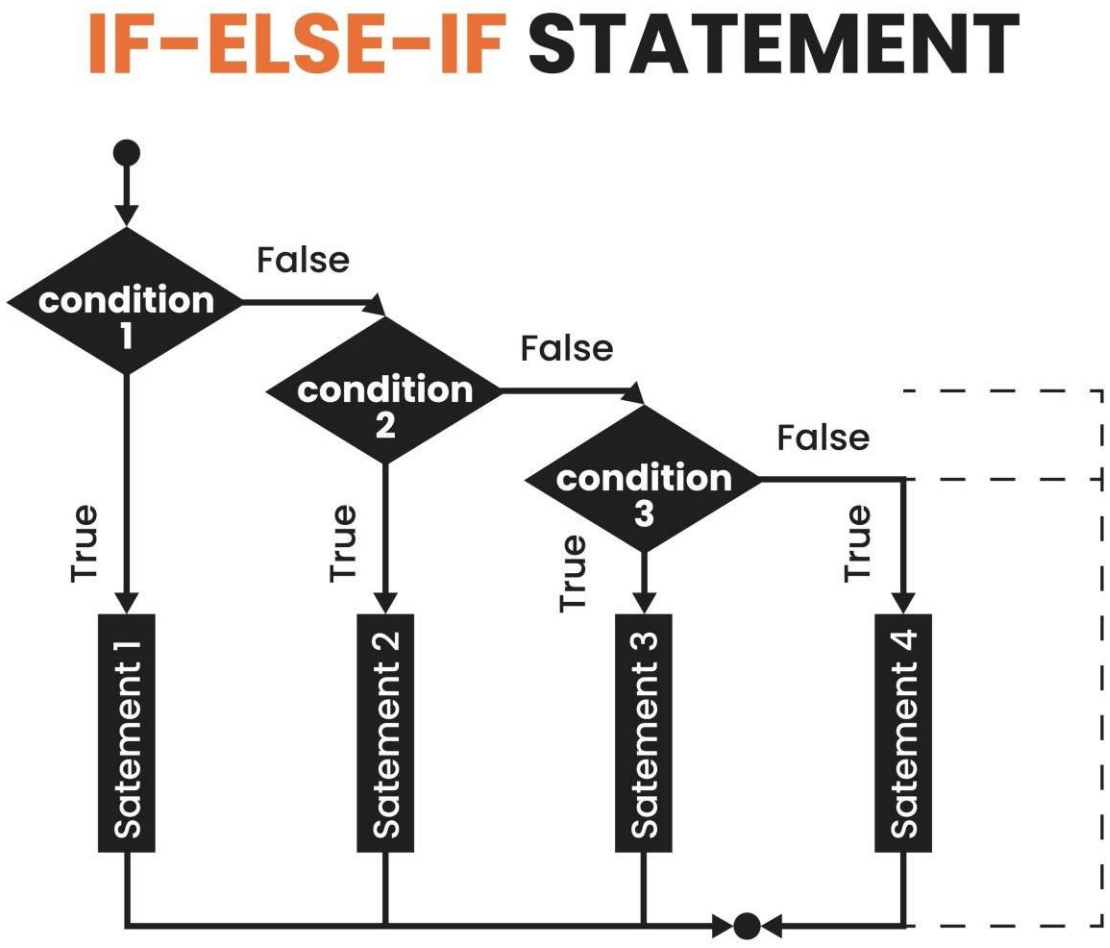
Lecture-12

Control Structures

The if Statement

Objective: To understand the basic syntax and functionality of the if statement for conditional execution.

Theory: The **if statement** is the most fundamental control structure. It executes a block of code only if a specified condition is True. If the condition is False, the code block is skipped entirely. This allows your program to make simple decisions.



Syntax:

if condition:

code to execute if condition is True

Program:

```
# Program 1: Basic if statement
age = 20
if age >= 18:
    print("You are eligible to vote.")



# Program 2: if statement with a false condition
temperature = 10
if temperature > 25:
    print("It's a hot day.")
```

Output:

You are eligible to vote.

The if-else Statement

Objective: To learn how to use the if-else statement to provide an alternative path of execution when the condition is False.

Theory: The **if-else statement** provides a two-way decision. The code inside the if block is executed if the condition is True, while the code inside the else block is executed if the condition is False.

Syntax:

if condition:

code to execute if condition is True else:

code to execute if condition is False

Program:

```
# Program 1: if-else statement for even/odd check
number = 7
```

```
if number % 2 == 0:
    print(f"{number} is an even number.")
else:
    print(f"{number} is an odd number.")



# Program 2: Another if-else example
is_raining = True
if is_raining:
    print("Remember to bring an umbrella.")
else:
```

Output:

7 is an odd number.

Remember to bring an umbrella.

The if-elif-else Statement

Objective: To understand how to handle multiple conditions using the if-elif-else structure.

Theory: The **if-elif-else statement** allows for multi-way decision-making. The conditions are checked sequentially. The code block for the first condition that evaluates to True is executed, and all other elif and else blocks are skipped. The else block is optional and acts as a catch-all for cases where none of the if or elif conditions are met.

Syntax:

if condition1:

code if condition1 is True elif condition2:

code if condition2 is True else:

code if all conditions are False

Program:

```
# Program 1: Grade classification
score = 85
if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
elif score >= 70:
    print("Grade: C")
else:
    print("Grade: D or F")



# Program 2: Simple temperature check
temperature = 22
if temperature < 0:
    print("It's freezing.")
elif temperature <= 15:
    print("It's a bit chilly.")
elif temperature <= 25:
    print("The weather is nice.")
```

Output:

Grade: B

The weather is nice.

Lecture-13

Functions and OOP: Control Structures

Functions:

A **function** is a block of code that is organized, reusable, and performs a single, related action. This helps to break down large programs into smaller, manageable parts.

- **Types of Functions:**
    - **Built-in Functions:** Functions that are part of Python's core, like print(), len(), sum(), and input().
    - **User-defined Functions:** Functions you create yourself to perform specific tasks in your code.
- **Types of Arguments:**
    - **Positional Arguments:** Arguments passed to a function based on their position or order.

```
def describe_pet(animal, name):
    print(f"I have a {animal} named {name}.")
describe_pet("dog", "Buddy")
```

    - **Keyword Arguments:** Arguments specified by name, allowing you to pass them in any order.

```
describe_pet(name="Buddy", animal="dog")
```

    - **Default Arguments:** A parameter is assigned a default value in the function definition, which is used if no argument is provided for it.

```
def greet(name, message="Hello"):
    print(f"{message}, {name}!")
greet("Alice")          # Uses default: "Hello, Alice!"
greet("Bob", "Hi")      # Overrides default: "Hi, Bob!"
```

- **Arbitrary Arguments (*args and **kwargs):**
  - *args (non-keyworded arguments): Allows a function to accept any number of positional arguments. They are packed into a **tuple**.
  - **kwargs (keyworded arguments): Allows a function to accept any number of keyword arguments. They are packed into a **dictionary**.

```
def show_info(*args, **kwargs):
    print("Positional arguments:", args)
    print("Keyword arguments:", kwargs)

show_info(1, "apple", age=25, city="New York")
# Output:

# Positional arguments: (1, 'apple')
# Keyword arguments: {'age': 25, 'city': 'New York'}
```

- **Variable Scope: The global Keyword:** The global keyword is used to modify a global variable from within a function.

```
x = 10  # A global variable
def modify_x():
    global x
    x = 20

modify_x()
print(x) # Output: 20
```

Overview of Object-Oriented Programming (OOP)

Objective: To provide a brief introduction to the core principles of Object-Oriented Programming (OOP) in Python by creating and using a simple class.

Theory: Object-Oriented Programming (OOP) is a paradigm that organizes code around **objects** rather than functions and logic. A **class** is a blueprint for creating objects, defining attributes (data) and methods (behavior). An **object** is an instance of a class. The four pillars of OOP are

Encapsulation, Inheritance, Polymorphism, and Abstraction.

Lecture-14

Functions: Conditional Branching

Control Statements and Logical Operators

Control statements determine the order in which your code is executed. Logical operators help you build complex conditions for these statements.

Logical Operators: Building Complex Conditions

Logical operators combine multiple conditional expressions into a single True or False result.

and: Returns True if **both** operands are True.

Truth Table:

Operand 1	Operand 2	Result
True	True	True
True	False	False
False	True	False
False	False	False

or: Returns True if **at least one** operand is True.

Truth Table:

Operand 1	Operand 2	Result
True	True	True
True	False	True
False	True	True
False	False	False

- **not:** Reverses the logical state. not True is False, and not False is True.

- **Sample Code with Logical Operators:**

```
# Example 1: `and`
is_admin = True
is_logged_in = True
if is_admin and is_logged_in:
    print("Access granted to admin panel.")

# Example 2: `or`
is_weekend = False
has_day_off = True
if is_weekend or has_day_off:
    print("Time to relax!")

# Example 3: `not`
is_valid = False
if not is_valid:
```

Control Statements: if, elif, else

This structure allows your program to make decisions and follow a specific "branch" of code.

- **if:** The most basic control statement. The code block is executed only if the condition is True.

```
score = 95
if score > 90:
    print("You got an A.")
```

- **if...else:** Provides a default path to take if the initial condition is False.

```
age = 17
if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")
```

- **if...elif...else:** Used to check for another condition if the previous if or elif conditions were False. You can have multiple elif blocks.

```
grade = 85
if grade >= 90:
    print("Excellent! You got an A.")
elif grade >= 80:
    print("Great job! You got a B.")
elif grade >= 70:
    print("Good effort. You got a C.")
else:
    print("You got a D or F. Keep studying!")
```

Lecture-15

Looping

Looping: The Power of Repetition

Looping statements are used to execute a block of code repeatedly.

for Loop: Iteration over a Sequence

A for loop is used to iterate over the items of any sequence (a list, tuple, dictionary, string, or range). It's great when you know the number of times you need to loop.

- **Iterating over Different Data Types:**

```
# Looping through a list of items
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
print(fruit)



# Looping through characters in a string
for char in "Python":
print(char)
# Looping through a dictionary's keys and values
student = {"name": "Charlie", "age": 20, "major": "Computer Science"}
for key, value in student.items():
print(f"Key: {key}, Value: {value}")
```

- **Useful Functions with for Loops:**

range(start, stop, step): Generates a sequence of numbers.

```
for i in range(1, 10, 2):
    print(i, end=" ") # Output: 1 3 5 7 9
```

enumerate(iterable): Returns both the index and the value of each item.

```
for index, fruit in enumerate(fruits):
    print(f"Fruit at index {index} is {fruit}.")
```

zip(iterable1, iterable2, ...): Combines items from multiple iterables.

```
names = ["Alice", "Bob"]
ages = [25, 30]
for name, age in zip(names, ages):
    print(f"{name} is {age} years old.")
```

while Loop: Repetition Based on a Condition

A **while loop** repeats as long as its condition is True. This is useful when the number of iterations is unknown beforehand.

- **Example with a Counter:**

```
count = 0
while count < 5:
    print(f"Count is {count}")
    count += 1
```

- **Example with a Sentinel Value:**

```
user_input = ""
while user_input.lower() != "quit":
    user_input = input("Enter a word (or 'quit' to exit): ")
    print(f"You entered: {user_input}")
```

- **Loop Control Statements:**
    - **break:** Exits the loop immediately.

```
for number in range(10):
    if number == 5:
        break
    print(number) # Output: 0 1 2 3 4
```

    - **continue:** Skips the rest of the code in the current iteration and moves to the next one.

```
for number in range(5):
    if number == 2:
        continue
    print(number) # Output: 0 1 3 4
```

pass: A null statement. It is used as a placeholder where a statement is syntactically required but you want no action to be performed.

```
for number in range(5):
    if number % 2 == 0:
        pass # This does nothing, just a placeholder
    else:
        print(number) # Output: 1 3
```

Lecture-16

Exception Handling

Exception Handling: Robust and Graceful Error Management

Exception handling is a critical part of writing stable, user-friendly programs. An **exception** is a special type of error that occurs during the execution of your program, but it can be "handled" or "caught" to prevent the program from crashing.

- **The Problem:** Without exception handling, a simple error like a user typing a letter instead of a number will cause your program to stop with a ValueError.
- # This will crash if the user enters a non-integer
- user_age = int(input("Enter your age: "))

- **The Solution: The try...except Block:** The try block contains the code that might cause an error. The except block contains the code that will run if an error occurs. This allows you to gracefully handle the error and give the user a helpful message.

Basic try...except:

```
try:
    result = 10 / 0  # This will cause a ZeroDivisionError
except:
    print("An error occurred.")
```

Output:

```
An error occurred.
```

Handling Specific Exceptions:

```
try:
    num = int(input("Enter a number: "))
except ValueError:
    print("Invalid input. Please enter a valid number.")
```

Output:

Case 1: Valid input
Input:

Enter a number: 25

Output:

(no output, because no error occurred)
num will now hold the value 25.

Case 2: Invalid input (e.g., abc)
Input:

Enter a number: abc

Output:

Invalid input. Please enter a valid number.

The try...except...else Block: The else block runs only if the try block completes successfully without any exceptions.

```
try:
    num = int(input("Enter a number: "))
    print("The number is:", num)
except ValueError:
    print("Invalid input.")
else:
    print("The try block executed successfully!")
```

Output:

If you enter a valid integer:

The number is: 10

The try block executed successfully!

If you enter an invalid value:

Enter a number: abc

The try...except...finally Block: The finally block **always** runs, regardless of whether an exception occurred. It's perfect for cleanup tasks, such as closing files or connections.

```
file = None
try:
    file = open("my_file.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("The file was not found.")
finally:
    if file:
        file.close()
        print("File has been closed.")
```

Output:

Case 1: File does not exist

If "my_file.txt" is not present in the same folder:

Output:

The file was not found.

(finally runs, but since file never opened successfully, file is None, so it won't try to close it.)

Case 2: File exists and has some text

Suppose my_file.txt contains:
Hello, Python!

Output:

File has been closed.

And the variable content will store:
Hello, Python!

Comprehensive Example:

```
while True: try:
num1_str = input("Enter a number: ") num2_str = input("Enter a second number: ")

num1 = int(num1_str) # Could raise a ValueError num2 = int(num2_str) # Could raise a ValueError

result = num1 / num2 # Could raise a ZeroDivisionError



except ValueError:
print("Invalid input. Please enter a valid number.")



except ZeroDivisionError:
print("You cannot divide by zero. Please try again.")



except Exception as e:
# This is a generic handler for any other unexpected error print(f"An unexpected error occurred: {e}")

else:
# The 'else' block runs ONLY if the 'try' block was successful. print(f"The result is {result}.")
break # Exit the loop on success



finally:
# The 'finally' block always runs, no matter what.
# It's perfect for cleanup, like closing files or connections. print("Operation attempt complete.")
```

Output:

Case 1: Invalid number input

Enter a number: abc

Enter a second number: 10

Output:

Invalid input. Please enter a valid number.
Operation attempt complete.

Case 2: Division by zero

Enter a number: 10

Enter a second number: 0

Output:

You cannot divide by zero. Please try again.
Operation attempt complete.

Case 3: Successful division

Enter a number: 20

Enter a second number: 5

Output:

The result is 4.0.

Operation attempt complete.

(The loop breaks here because division was successful.)

Lecture-17

Custom Functions in Python

🔹 **What is a Function?**

A **function** in Python is a block of organized, reusable code that is used to perform a single, related action. Functions help in dividing a large program into smaller, manageable, and reusable blocks of code.

👉 **Why use functions?**

1. **Reusability** – Write once, use many times.
2. **Readability** – Makes the program cleaner and easier to understand.
3. **Maintainability** – Easy to update or debug.
4. **Modularity** – Divides the program into logical parts.

🔹 **Defining a Function in Python**

The def keyword is used.

Syntax:

def function_name(parameters):
"""Optional docstring: explains what the function does""" # body of the function
return result

🔹 **Types of Functions**

1. **Built-in Functions**: Already available in Python (e.g., print(), len(), type()).
2. **User-defined Functions**: Created by the programmer using def.

Examples

Example 1: Function without arguments

- **Theory:**
  - A **function without arguments** does not take any input values.
  - It simply performs a task whenever it is called.
  - Such functions are useful when the output does not depend on user-provided data.

def welcome():
print("Hello, welcome to Python functions!") welcome()

Output:

Hello, welcome to Python functions!

Example 2: Function with arguments

- **Theory:**
  - Functions can take **arguments (parameters)**.
  - Arguments allow us to pass input values into the function.
  - The function can then use those inputs to perform tasks.

def greet(name): print("Hello", name, "!")

greet("Alice") greet("Bob")

Output:

Hello Alice ! Hello Bob !

Example 3: Function with default argument

- **Theory:**
  - A **default argument** is a value that is used if the user does not provide one.
  - This makes the function more flexible.
  - If the user gives a value, it overrides the default.

```
def power(base, exp=2):            # default exponent is 2 return base ** exp

print(power(5)) # 25
print(power(5, 3))          # 125
```

Output:

25

125

Example 4: Function returning multiple values

- **Theory:**
  - A function in Python can return **more than one value** at the same time.
  - This is done by separating the return values with a comma.
  - The returned values can be **unpacked** into multiple variables.

```
def calculate(a, b): return a+b, a-b, a*b
```

add, sub, mul = calculate(10, 5) print("Addition:", add) print("Subtraction:", sub) print("Multiplication:", mul)

Output:

Addition: 15

Subtraction: 5

Multiplication: 50

Example 5: Recursive function

- **Theory:**
  - A **recursive function** is a function that calls itself.
  - It is commonly used to solve problems that can be broken down into smaller, similar subproblems (e.g., factorial, Fibonacci series).
  - Every recursive function needs a **base case** to stop the recursion.

def factorial(n):

if n == 0 or n == 1: return 1

else:

return n * factorial(n-1) print("Factorial of 5:", factorial(5))

Output:

Factorial of 5: 120

Summary: Functions are essential in Python because they reduce repetition, make programs modular, and increase readability.

Lecture-18

Functions in random Module

Example 1: random()

📌 Returns a **floating-point number** between 0.0 and 1.0. import random

num = random.random()

print("Random number between 0 and 1:", num)

🔹 **Explanation**:

- Generates a pseudo-random decimal (float).
- Always in range [0.0, 1.0).

🔹 **Output (example):**

Random number between 0 and 1: 0.7365891444

Example 2: uniform(a, b)

📌 Returns a random **float between a and b**.

num = random.uniform(5, 10)

print("Random float between 5 and 10:", num)

🔹 **Explanation**:

- Useful when you want random decimal values in a range.
- Unlike randint, it can give decimals.

🔹 **Output (example):**

Random float between 5 and 10: 7.824329

Example 3: randint(a, b)

📌 Returns a **random integer between a and b** (inclusive).

```
num = random.randint(1, 6)
print("Random integer between 1 and 6:", num)
```

- **Explanation**:
  - Equivalent to rolling a dice 🎲.
  - Both end values are included.

- **Output (example):**

```
Random integer between 1 and 6: 4
```

Example 4: randrange(start, stop, step)

📌 Returns a random number from a range.

```
num = random.randrange(1, 10, 2)
print("Random odd number between 1 and 10:", num)
```

- **Explanation**:
  - Works like range(start, stop, step).
  - Picks a random element from that sequence.

- **Output (example):**

```
Random odd number between 1 and 10: 7
```

Example 5: choice(seq)

📌 Returns a random element from a list, tuple, or string.

fruits = ["apple", "banana", "cherry", "mango"

print("Random fruit:", random.choice(fruits))

- **Explanation**:
  - Selects one random element from a sequence.
  - Commonly used in games or quizzes.
- **Output (example):**

Random fruit: banana

Example 6: choices(seq, k=n)

📌 Returns a list of k random elements (with replacement).

colors = ["red", "blue", "green", "yellow"] print("Random 3 colors:", random.choices(colors, k=3))

- **Explanation**:
  - Unlike choice(), it can pick multiple items.
  - Items can repeat (sampling **with replacement**).
- **Output (example):**

Random 3 colors: ['blue', 'green', 'blue']

Example 7: sample(seq, k=n)

📌 Returns a list of k unique random elements (without replacement).

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

print("Random 4 numbers:", random.sample(numbers, 4))

Lecture-19

math Module

The **math module** in Python provides built-in mathematical functions and constants. These functions are useful for performing advanced math operations like square root, powers, trigonometry, and factorial.

3.1 sqrt()

import math print(math.sqrt(25))

Output:

5.0

🔎 **Theory**

- math.sqrt(x) returns the square root of a number x.
- The square root of a number is a value that, when multiplied by itself, gives the original number.
- Example: √25 = 5, because 5 × 5 = 25.
- Always returns a floating-point number (decimal).

3.2 ceil() and floor()

import math print(math.ceil(4.3)) print(math.floor(4.8))

Output:

5

4

🔎 **Theory**

- math.ceil(x) → Returns the **smallest integer greater than or equal to x**

(rounds UP).

- math.floor(x) → Returns the **largest integer less than or equal to x** (rounds DOWN).

- These are useful when handling **rounding operations** in programs like billing, statistics, or game scores.

3.3 pow() and factorial()

import math print(math.pow(2, 3)) print(math.factorial(5))

Output:

8.0

120

🔎 **Theory:**

- math.pow(a, b) → Returns a raised to the power b (a^b).
    - Example: $2^3 = 8$.
    - Always returns a float.
- math.factorial(n) → Returns the product of all positive integers up to n.
    - Example: $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$.
    - Only works for non-negative integers.

3.4 trigonometry

import math print(math.sin(math.pi/2)) print(math.cos(0))

Output:

1.0

1.0

🔎 **Theory:**

- math.sin(x) and math.cos(x) return the sine and cosine values of an angle.

- Angles must be in radians, not degrees.
    - To convert degrees → radians, use math.radians(degree_value).
- Examples:
    - sin(π/2) = 1
    - cos(0) = 1
- These are widely used in geometry, physics, engineering, and graphics programming.

3.5 gcd() – Greatest Common Divisor:

import math print(math.gcd(24, 36))

Output:

12

🎾 **Theory:**

- math.gcd(a, b) returns the largest integer that divides both numbers without remainder.
- Example: GCD(24, 36) = 12, because 12 is the largest number dividing both 24 and 36.
- Useful in fractions, simplifying ratios, and number theory.

3.6 log() – Logarithm:

import math

print(math.log(100)) # Natural log (base e) print(math.log(100, 10)) # Logarithm base 10

Output:

4.605170185988092

2.0

🔎 **Theory**

- math.log(x) → Natural logarithm of x (base e ≈ 2.718).
- math.log(x, base) → Logarithm with a custom base.
- Example: $\log_{10}(100) = 2$, since $10^2 = 100$.
- Very important in data science, probability, and growth models.

3.7 degrees() and radians():

```
import math

print(math.degrees(math.pi))        # Convert radians to degrees print(math.radians(180))    # Convert degrees to radians
```

Output:

180.0

3.141592653589793

🔎 **Theory**

- math.degrees(radian_value) → Converts radians into degrees.
- math.radians(degree_value) → Converts degrees into radians.
- Essential in trigonometry, since Python uses radians for functions like sin() and cos().

3.8 Constants: pi and e

```
import math

print(math.pi) # Value of pi print(math.e)          # Euler's number
```

Output:

3.141592653589793

2.718281828459045

🎾 Theory

- **math.pi → Value of π (ratio of circumference to diameter).**

- **math.e → Euler’s number, base of natural logarithms**

Lecture-20

time Module

The time module in Python is used to work with time-related tasks such as measuring execution time, handling delays, formatting dates, and getting system time.

4.1 time() – Current Time in Seconds

import time print(time.time())

Output (example):

1726551782.58463

🔎 **Theory**

- **Returns the current time in seconds since Epoch (Jan 1, 1970, 00:00:00 UTC).**
- **Useful for performance measurement and timestamps.**

4.2 ctime() – Readable Time

import time print(time.ctime())

Output (example):

Tue Sep 17 10:43:02 2025

🔎 **Theory**

- **Converts the current system time into a human-readable string.**
- **Format: "Day Month Date HH:MM:SS Year".**

4.3 sleep() – Delay Execution

import time print("Start") time.sleep(3)

print("End after 3 seconds")

Output:

Start

(3-second pause) End after 3 seconds

🎾 Theory

- time.sleep(n) pauses the program for n seconds.
- Useful in simulations, animations, or slowing down loops.

4.4 localtime() – Local Time as Struct

import time print(time.localtime())

Output (example):

time.struct_time(tm_year=2025, tm_mon=9, tm_mday=17, tm_hour=10, tm_min=43, tm_sec=2, tm_wday=1, tm_yday=260, tm_isdst=0)

🎾 Theory

- Returns current local time as a struct_time object (like a tuple).
- Contains year, month, day, hour, minute, second, weekday, etc.

4.5 strftime() – Format Date and Time

import time

now = time.localtime()

print(time.strftime("%Y-%m-%d %H:%M:%S", now))

Output (example):

2025-09-17 10:43:02

Theory

- **strftime(format, struct_time) formats time into a custom string.**

- **Common format codes:**

  - **%Y = Year (2025)**

  - **%m = Month (09)**

  - **%d = Day (17)**

  - **%H = Hour (24-hr)**

  - **%M = Minute**

  - **%S = Second**

4.6 gmtime() – UTC Time

import time print(time.gmtime())

Output (example):

time.struct_time(tm_year=2025, tm_mon=9, tm_mday=17, tm_hour=5, tm_min=13, tm_sec=2, tm_wday=1, tm_yday=260, tm_isdst=0)

Lecture-21

Python os Module

The os module in Python provides functions to interact with the operating system. It is mainly used for working with files, directories, and environment variables.

5.1 getcwd() – Get Current Working Directory

```
import os

print("Current directory:", os.getcwd())
```

Output (example):

Current directory: /home/user/project

🔎 Theory

- Returns the absolute path of the current working directory (CWD).
- The CWD is the folder where your Python script is running.

5.2 listdir() – List Files and Folders

```
import os

print(os.listdir())
```

Output (example):

['file1.py', 'data.txt', 'images', 'notes.docx']

🔎 Theory

- Lists all files and folders in the current directory.
- Helps to see directory contents.

5.3 mkdir() and rmdir() – Create and Remove Directory

```
import os

os.mkdir("new_folder")

print("After creation:", os.listdir())os.rmdir("new_folder")

print("After deletion:", os.listdir())
```

Output (example):

After creation: ['file1.py', 'data.txt', 'new_folder']

After deletion: ['file1.py', 'data.txt']

🔎 Theory

- os.mkdir(name) → Creates a new empty folder.
- os.rmdir(name) → Deletes an empty folder.

- ❌ If the folder is not empty, rmdir() will give an error.

5.4 rename() – Rename File or Folder

import os

os.rename("data.txt", "info.txt")

print(os.listdir())

Output (example):

['file1.py', 'info.txt', 'images']

🔎 Theory

- Renames a file or folder.
- If the target name already exists, it may overwrite (OS-dependent).

5.5 remove() – Delete File

import os

os.remove("old_file.txt")

🔎 Theory● Deletes the specified file permanently.

● ⚠ Use carefully: once deleted, it cannot be undone.

5.6 path Functions

import os

print(os.path.exists("file1.py")) # Check if file exists print(os.path.isfile("file1.py")) # Check if it is a file print(os.path.isdir("images")) # Check if it is a directory

Output (example):

True

True

True

🔎 Theory

● os.path.exists(path) → Checks whether a file/folder exists.

- os.path.isfile(path) → True if path is a file.
- os.path.isdir(path) → True if path is a directory.

5.7 getlogin() and environ

import os

print("Logged in as:", os.getlogin())

print("Path variable:", os.environ.get("PATH"))

Output (example):

Logged in as: student

Path variable: /usr/local/bin:/usr/bin:/bin

🔎 Theory● os.getlogin() → Returns the username of the logged-in user.

- os.environ → Access system environment

Lecture-22

Python shutil Module

The shutil module provides functions for file operations and directory management. It is more powerful than the os module for copying and removing folders.

6.1 copy() – Copy a File

import shutil

shutil.copy("file1.txt", "copy.txt")

Output (effect):

['file1.txt', 'copy.txt']

🔎 Theory

- shutil.copy(src, dest) copies the contents of src file to a new file dest.
- If dest exists, it may be overwritten.

6.2 copy2() – Copy with Metadata

import shutil

shutil.copy2("file1.txt", "backup.txt")

Output (effect):

['file1.txt', 'backup.txt']

🔎 Theory

- Similar to copy(), but also copies file metadata (creation time, modification time, permissions).
- Useful for making exact backups.6.3 copytree() – Copy Entire Folder

import shutil

shutil.copytree("data", "data_backup")

Output (effect):

['data', 'data_backup']

🔎 Theory

- Copies the entire folder (with all files/subfolders) to a new location.
- ❌ Raises error if data_backup already exists.

6.4 move() – Move/Rename File or Folder

import shutil

shutil.move("copy.txt", "renamed.txt")

Output (effect):

['file1.txt', 'renamed.txt']

🔎 Theory

- Moves a file/folder to another location.
- If used in the same directory, it works as a rename.

6.5 rmtree() – Delete Entire Folder

import shutil

⚠ Be careful! This will delete folder and all contents permanently shutil.rmtree("data_backup")

Output (effect):

['file1.txt', 'renamed.txt']

🔎 Theory● Removes the entire directory along with its files and subdirectories.

- Dangerous: Once deleted, data cannot be recovered.

6.6 disk_usage() – Disk Space Information

import shutil

usage = shutil.disk_usage("/") print("Total:", usage.total) print("Used:", usage.used) print("Free:", usage.free)

Output (example):

Total: 500107862016

Used: 320000000000

Free: 180107862016

🔎 Theory

- Returns disk usage statistics for the given path.
- Values are in bytes (convert to GB by dividing by 1024**3).

Lecture-23

sys, glob, and re

1. The sys Module

The sys module provides access to system-specific parameters and functions.

1.1 Python Version & Platform

```
import sys

print("Python Version:", sys.version)

print("Platform:", sys.platform)
```

Output (example):

```
Python Version: 3.12.0 (main, Oct 2024, ...)

[GCC 11.2.0]

Platform: win32
```

🔎 Theory

- sys.version → Returns current Python version details.
- sys.platform → Returns the operating system name (win32, linux, darwin for Mac).

1.2 Command Line Arguments

```
import sys

print("Arguments:", sys.argv)
```

Output (if run as python script.py hello):

```
Arguments: ['script.py', 'hello']
```

🔎 Theory

- sys.argv → List of arguments passed to the script from the command line.
- Useful for automation and batch processing.1.3 Exiting a Program

```
import sys

print("Before exit")

sys.exit()

print("This will not print")
```

Output:

```
Before exit
```

🔎 Theory

- sys.exit() → Immediately stops the program execution.

2. The glob Module

The glob module is used to search for files and directories using wildcards (*, ?).

2.1 List Python Files

```
import glob

print(glob.glob("*.py"))
```

Output (example):

['main.py', 'test.py', 'script.py']

🔎 Theory

- glob.glob("*.py") → Finds all files in the current folder that end with .py.

2.2 List CSV Files in a Folder

```
import glob

print(glob.glob("data/*.csv"))
```

Output (example):

['data/file1.csv', 'data/file2.csv'] 🔎 Theory

- Matches files inside data folder ending with .csv.

2.3 Recursive Search

```
import glob

print(glob.glob("**/*.txt", recursive=True))
```

Output (example):

['notes.txt', 'docs/readme.txt']

🔎 Theory

- recursive=True → Searches inside all subfolders as well.

3. The re Module (Regular Expressions)

The re module is used for pattern matching and text searching.

3.1 search() – Find a Pattern

```
import re

text = "I love Python"

result = re.search("Python", text)

print("Found at:", result.start())
```

Output:

Found at: 7

🔎 Theory

- re.search(pattern, text) → Finds first match of pattern in text.

3.2 findall() – Find All Matches

import re

text = "apple, banana, apple, mango" print(re.findall("apple", text))Output:

['apple', 'apple']

🔎 Theory

- Returns all matches of the pattern as a list.

3.3 sub() – Replace Pattern

import re

text = "I like Java"

print(re.sub("Java", "Python", text))

Output:

I like Python

🔎 Theory

- re.sub(old, new, text) → Replaces all occurrences of a pattern.

3.4 split() – Split Text by Pattern

import re

text = "one,two;three four"

print(re.split("[,;]", text))

Output:

['one', 'two', 'three', 'four']

🔎 Theory

- Splits text wherever it finds a comma, semicolon, or space.

Lecture-24

Python statistics Module

The statistics module in Python provides functions to perform mathematical statistics on numerical data.

It is widely used in data analysis, mathematics, and machine learning.

1. Mean (Average)

```
import statistics

data = [10, 20, 30, 40, 50]

print("Mean:", statistics.mean(data))
```

Output:

Mean: 30

🔎 Theory

- Mean = (Sum of all values) ÷ (Number of values)
- Example: (10 + 20 + 30 + 40 + 50) ÷ 5 = 30.

2. Median (Middle Value)

```
import statistics

data = [5, 7, 9, 11, 13]

print("Median:", statistics.median(data))
```

Output:

Median: 9

🔎 Theory

- The median is the middle value when data is sorted.
- If data has even numbers, median = average of two middle values.3. Mode (Most Frequent Value)

```
import statistics

data = [1, 2, 2, 3, 4, 4, 4, 5]

print("Mode:", statistics.mode(data))
```

Output:

Mode: 4

🔎 Theory

- Mode is the value that occurs most frequently.
- Example: In [1,2,2,3,4,4,4,5], mode = 4 because it appears 3 times.

4. Standard Deviation (Spread of Data)

```
import statistics

data = [10, 20, 30, 40, 50]

print("Standard Deviation:", statistics.stdev(data))
```

Output:

```
Standard Deviation: 15.811388300841896
```

🔎 Theory

- Standard Deviation (stdev) measures how much values deviate from the mean.
- A low stdev → values are close to mean.
- A high stdev → values are spread out.

5. Variance (Square of Standard Deviation)

```
import statistics

data = [10, 20, 30, 40, 50]

print("Variance:", statistics.variance(data))Output:
```

```
Variance: 250
```

🔎 Theory

- Variance = Average of squared differences from the mean.
- It shows the spread of data, like stdev but in squared units.

6. Harmonic Mean

```
import statistics

data = [2, 4, 4]

print("Harmonic Mean:", statistics.harmonic_mean(data))
```

Output:

```
Harmonic Mean: 3.0
```

🔎 Theory

- Harmonic Mean = $n \div (1/x_1 + 1/x_2 + \ldots + 1/x$