Day 2:

1. Python Fundamentals :

Python uses:

1.Web development :Frameworks like Django, Flask
2.Data Science and Analysis: Libraries like Pandas , NumPy , Matplotlib
3. Machine Learning : TensorFlow , PyTorch, Scikit-learn
4. Game Devlopment: Libries like Pygame
5. Web Scraping : Tools like BeautifulSoup , Scrapy

Why Learn Python? ...

- Easy Syntax
- Build-in libraries for Beginners
- Project oriented Learing

Python Comments:

- Single line : # this is single line comment
- Multi line: " " " this is
  Multiline comment " " "

Rules for Naming Variables:

To use variables effectively, we must follow Python's naming rules:

- Variable names can only contain letters, digits and underscores (_).
- A variable name cannot start with a digit.
- Variable names are case-sensitive (myVar and myvar are different).
- Avoid using Python keywords (e.g., if, else, for) as variable names.

List of Keywords in Python

True	False	None	and	else	elif	For	While
or	not	is	if	break	continue	Pass	try
class	import	from	in	except	finally	raise	assert
as	del	global	with	def	return	lambda	Yield
nonlocal	Async	Await					

Python Data Types:

- **Numeric –** int, float, complex
- **Sequence Type –** string, list, tuple
- **Mapping Type –** dict
- **Boolean –** bool
- **Set Type –** set, frozenset
- **Binary Types –** bytes, bytearray, memoryview

1.Numeric – int, float, complex numbers

- **Integers** – This value is represented by int class. It contains positive or negative whole numbers (without fractions or decimals). In Python, there is no limit to how long an integer value can be.
  Example: 5342, -533333
- **Float** – This value is represented by the float class. It is a real number with a floating-point representation. It is specified by a decimal point. Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation.
  Example : 3433.34 , -5415.96
- **Complex Numbers** – A complex number is represented by a complex class. It is specified as *(real part) + (imaginary part)j* .
  Example: – 2+3j

code….

```
a = 5
print(type(a))
b = 5.0
print(type(b))
c = 2 + 4j
print(type(c))
```

#output….

```
<class 'int'>
<class 'float'>
<class 'complex'>
```

2.Sequence Types

Sequences are ordered collections of elements, where each element has an index.

a) str (String)

- A string is an immutable sequence of characters enclosed within single, double, or triple quotes.
- Strings support various operations like slicing, concatenation, and formatting.

  #**code: …**

---

s = "Hello, World!"

print(s[0])

 # Output: H

---

b) list (List)

- A list is a mutable, ordered collection of elements.
- It can store elements of different types.
- Lists allow indexing, slicing, and modification.

#code: …
lst = [10, "Python", 3.14]
lst.append(20) # Adds an element at the end
print(lst)

 # Output: [10, "Python", 3.14, 20]

---

c) tuple (Tuple)

- A tuple is an immutable, ordered collection of elements.
- Once created, its elements cannot be changed.

  #**Code: …**

---

tup = (1, 2, "Hello")
print(tup[1])

 # Output: 2

---

3. Set Types

Sets are unordered collections of unique elements.

a) set (Set)

- A set is a mutable collection of unique, unordered elements.
- It does not allow duplicate values.

Example:

```
s = {1, 2, 3, 4, 4}
print(s)

# Output: {1, 2, 3, 4}
```

b) frozenset (Frozen Set)

- A frozenset is an immutable version of a set.
- Once created, its elements cannot be changed.

Example:

```
fs = frozenset({1, 2, 3})
print(fs)
```

4. Mapping Type

Mappings store key-value pairs.

dict (Dictionary)

- A dictionary is a mutable, unordered collection of key-value pairs.
- Keys must be unique and immutable (e.g., strings, numbers, or tuples).

Example:

```
d = {"name": "Pavan", "age": 21}
print(d["name"])

 # Output: Pavan
```

5. Boolean Type

Boolean values represent truth values.

bool (Boolean)

- A boolean type has only two values: True or False.
- It is often used in conditional statements.

Example:

```
is_active = True
print(5 > 3)

 # Output: True
```

6. Binary Types

Binary types deal with raw byte data.

a) bytes

- A bytes object is an immutable sequence of bytes.
- It is useful for handling binary data like images or files.

Example:

```
b = b"Hello"
print(b[0])

 # Output: 72 (ASCII of 'H')
```

b) bytearray

- A bytearray is a mutable version of bytes.

Example:

```
ba = bytearray([65, 66, 67])
ba[0] = 68
print(ba)

 # Output: bytearray(b'DBC')
```

c) memoryview

- A memoryview provides a view of a bytes or bytearray object without copying data.

  Example:

```
mv = memoryview(b"Python")
print(mv[0])

# Output: 80 (ASCII of 'P')
```

Conditional Statements:

1. if Statement

The if statement executes a block of code only if the condition is True.

Syntax:

```
if condition:
        # Code to execute if condition is True
```

```
Example:
age = 18
if age >= 18:
   print("You are eligible to vote.")

#Output: ...
You are eligible to vote.
```

2. if-else Statement

The if-else statement executes one block of code if the condition is True, and another block if it is False.

Syntax:

```
if condition:
```

```
    # Code if condition is True
else:
    # Code if condition is False
```

Example:

```
age = 16
if age >= 18:
    print("You are eligible to vote.")
else:
    print("You are not eligible to vote.")

#Output: ….
You are not eligible to vote.
```

---

3. if-elif-else Statement

The if-elif-else statement is used when there are multiple conditions to check.

Syntax:

```
if condition1:
    # Code if condition1 is True
elif condition2:
    # Code if condition2 is True
else:
    # Code if none of the conditions are True
```

Example:

```
marks = 85
if marks >= 90:
    print("Grade: A")
elif marks >= 75:
    print("Grade: B")
else:
    print("Grade: C")

#Output: ….
Grade: B
```

4. Nested if Statement

A nested if statement means an if statement inside another if statement.

Syntax:

```
if condition1:
    if condition2:
        # Code if both condition1 and condition2 are True
```

Example:

```
num = 10
if num > 0:
    print("Positive number")
    if num % 2 == 0:
        print("Even number")
#Output: …
Positive number
Even number
```

Loops in Python:

1. for Loop

The for loop is used for iterating over a sequence such as a list, tuple, dictionary, or string.

Syntax:

```
for variable in sequence:
    # Code to execute
```

Example 1: Iterating Over a List

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)

Output:
apple
```

```
banana
cherry
```

Example 2: Using range() in a for Loop

```
for i in range(1, 6):
    print(i)
Output:
1
2
3
4
5
```

2. while Loop

The while loop executes a block of code as long as the condition is True.

Syntax:

```
while condition:
    # Code to execute
```

Example: Printing Numbers from 1 to 5

```
i = 1
while i <= 5:
    print(i)
    i += 1  # Increment to avoid infinite loop
Output:
1
2
3
4
5
```

Exception Handling :

Exception handling in Python allows programs to handle runtime errors gracefully instead of crashing. Python provides built-in mechanisms to **identify, handle, and manage errors** using try, except, else, raise and finally statements.

1. Handling Exceptions using try-except

Python provides a try-except block to catch exceptions and prevent program crashes.

Syntax:

```
try:
    # Code that may raise an exception
except ExceptionType:
    # Code to execute if an exception occurs
```

Example: Handling ZeroDivisionError

```
try:
    x = 10
    y = 0
    result = x / y
    print(result)
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
```

Output:

```
Error: Cannot divide by zero.
```

The program does not crash and prints a user-friendly error message instead.

2. Handling Multiple Exceptions

We can handle multiple exceptions separately using different except blocks.

```
Example: Handling Different Errors
try:
    x = int(input("Enter a number: "))  # User might enter a non-numeric value
    y = 0
    result = x / y  # Division by zero
except ValueError:
```

```
    print("Error: Invalid input! Please enter a number.")
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
```

Output (if user enters "abc"):

Error: Invalid input! Please enter a number.

Output (if user enters 10):

Error: Cannot divide by zero.

---

3. Using a Single except Block for Multiple Exceptions

You can handle multiple exceptions in a single except block by specifying them inside parentheses.

Example:

```
try:
    x = int(input("Enter a number: "))
    y = 0
    result = x / y
except (ValueError, ZeroDivisionError) as e:
    print(f"Error: {e}")
```

Output:

If the user enters "abc", it will print:

Error: invalid literal for int() with base 10: 'abc'

If the user enters 10, it will print:

Error: division by zero

---

4. Using else with try-except

The else block runs **only if no exception occurs** in the try block.

Example:

```
try:
    x = int(input("Enter a number: "))
    y = int(input("Enter another number: "))
    result = x / y
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Invalid input! Please enter a number.")
else:
    print("Division result:", result)
```

Output (if input is valid and no errors occur):

```
Division result: 5.0
```

5.Using finally Block

The finally block always executes, **whether an exception occurs or not**. It is useful for resource cleanup (e.g., closing files or database connections).

Example:

```
try:
    f = open("example.txt", "r")
    content = f.read()
    print(content)
except FileNotFoundError:
    print("Error: File not found.")
finally:
    print("Closing the file.")
```

Output (if file does not exist):

```
Error: File not found.
Closing the file.
```

6. Raising Custom Exceptions using raise

You can manually trigger exceptions using the raise keyword.

Example: Raising a Custom Exception

```
age = int(input("Enter your age: "))
if age < 18:
    raise ValueError("You must be 18 or older to proceed.")
else:
    print("You are eligible.")
```

Output (if user enters 16):

```
ValueError: You must be 18 or older to proceed.
```

7. Defining Custom Exceptions

Python allows you to define your own exception classes by inheriting from Exception.

Example: Creating a Custom Exception

```
class UnderAgeError(Exception):
    def __init__(self, message="You must be 18 or older."):
        self.message = message
        super().__init__(self.message)

try:
    age = int(input("Enter your age: "))
    if age < 18:
        raise UnderAgeError()
    print("You are eligible.")
except UnderAgeError as e:
    print(f"Custom Exception: {e}")
```

Output (if user enters 16):

Custom Exception: You must be 18 or older.

Error Handling in Python :

1. Syntax Error (Compile Time error):
   These occur when the Python interpreter encounters incorrect syntax in the code.

2. Exceptions(Runtime Error) :
   Exceptions occur when a program is running but encounters an operation that is invalid. These errors **do not stop code execution immediately if handled properly**.

2.Login Module in Python:

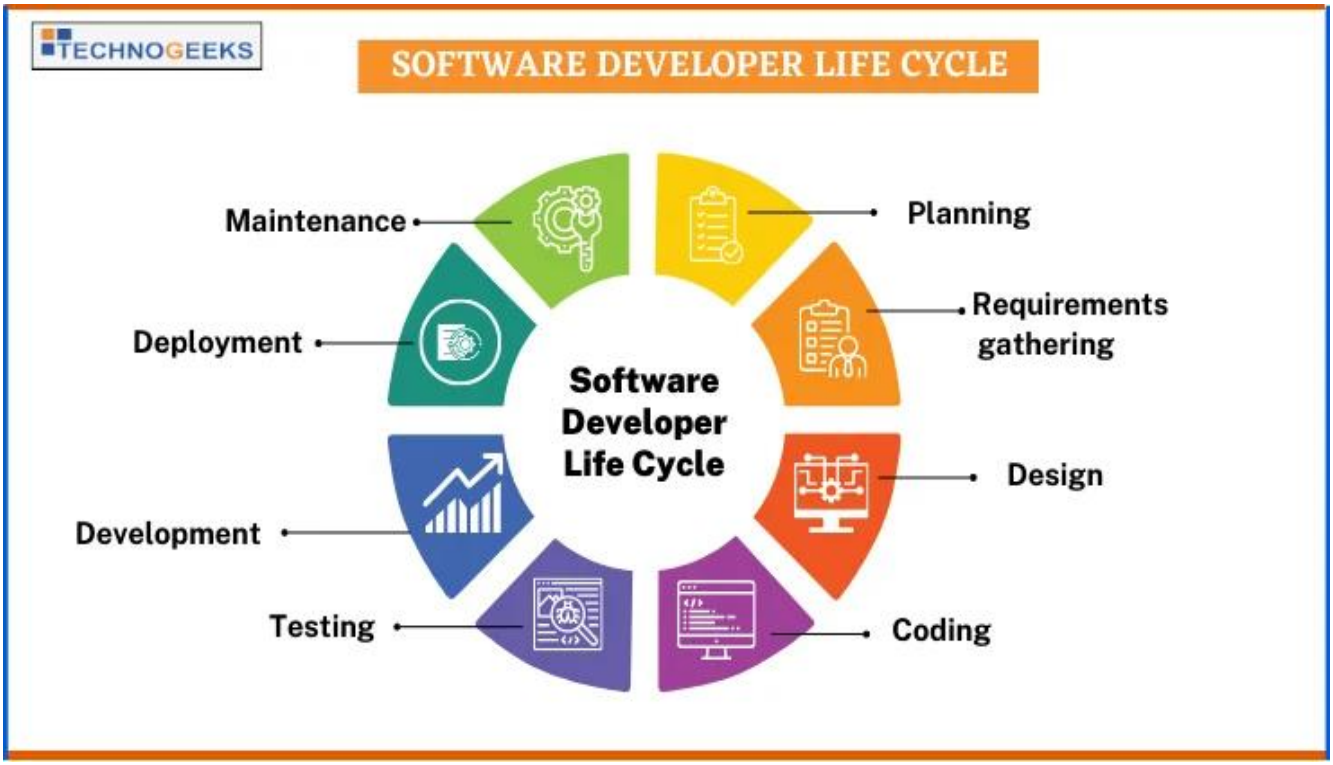
In Python, the login module is a set of tools and functions that allow you to authenticate users and manage their sessions. The login module can be used to secure web applications, restrict access to certain areas of a website, and authenticate users with various third-party services.

Some of the key features of the login module include:

1. **Authentication:** The login module can authenticate users based on their username and password or using a third-party authentication service such as OAuth.
2. **Session Management:** The login module can manage user sessions, allowing users to remain logged in even after leaving the website or closing their browser.

3. **Security:** The login module provides various security features such as encryption of sensitive data and protection against common security attacks such as cross-site scripting (XSS) and SQL injection.
4. **Customization:** The login module can be customized to fit the specific needs of your application. You can customize the look and feel of the login page, the authentication process, and the user session management.

3. SDLC life Cycle:



The Software Development Life Cycle (SDLC) is a structured framework that outlines the process of developing software. It includes planning, designing, coding, testing, and maintaining the software. The goal of SDLC is to ensure that the software meets user requirements and quality standards.

Phases of SDLC

- **Requirements analysis**: Analyzing the requirements for the software
- **Planning**: The initial phase of the SDLC
- **Design**: Designing the software
- **Coding**: Writing the code for the software
- **Testing**: Testing the software for quality and functionality
- **Deployment**: Making the software available for use
- **Maintenance**: Maintaining the software to ensure it continues to work properly

Project Management:

Project management (PM) is the process of planning and managing a project from start to finish. It involves coordinating resources and activities to meet deadlines and achieve the desired outcome.

What does project management include?

- **Planning**: Creating a plan for the project, including a timeline and milestones
- **Execution**: Carrying out the plan, including assigning tasks and managing resources
- **Monitoring and control**: Tracking progress and making adjustments as needed
- **Closing**: Completing the project and documenting the results

What are some project management approaches?

- **Agile**: An approach that focuses on delivering requirements incrementally throughout the project
- **Hybrid**: A combination of different methodologies, such as Agile and Waterfall, to create a process that fits the project and team
- **Waterfall**: A traditional project management approach

Agile Methodology :

It's an iterative and incremental process that's designed to help teams develop products in environments where change are made on frequent basis .

Agile methodology is a project management framework that breaks down projects into phases and encourages teams to reflect and improve after each phase

Steps of Agile methodology:

1. Requirement Gatering