### **Week 2:**

**1. What are Comments in Python?**

🡺Comments in Python are **text annotations** added to the code to explain what the code does. They are ignored by the Python interpreter and are used to make the code more readable and understandable.

**Types of Comments in Python**

1. **Single-Line Comments**:
   * Start with the # symbol and continue until the end of the line.
   * Example:

python

*# This is a single-line comment*

**print**("Hello, World!")

1. **Multi-Line Comments**:
   * Python does not have a specific syntax for multi-line comments. However, you can use multiple single-line comments or a **docstring** (triple quotes """...""" or '''...''') for this purpose.
   * Example using single-line comments:

python

*# This is a multi-line comment*

*# spanning multiple lines*

**print**("Hello, World!")

* + Example using triple quotes:

python

"""

This is a multi-line comment

using triple quotes.

"""

**print**("Hello, World!")

1. **Docstring Comments**:
   * These are used to document functions, classes, and modules. They are typically placed immediately after the definition and are enclosed in triple quotes.
   * Example:

python

**def** greet(name):

"""This function greets a person."""

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

**Why Use Comments?**

1. **Readability**: Comments make the code easier to understand for both the developer and others.
2. **Debugging**: They can be used to temporarily disable parts of the code during testing.
3. **Collaboration**: Comments help other developers understand the code's intent and logic.

**2. What is Indentation in Python?**

🡺Indentation in Python refers to the **whitespaces** (spaces or tabs) used at the beginning of a line to define code blocks. It is essential for structuring code within functions, loops, conditionals, and classes.

**Key Points**

1. **Consistency**: Use either spaces or tabs consistently.
2. **Standard**: 4 spaces per indentation level is recommended.
3. **No Indentation at the Start**: The first line should not be indented.

**Example**

python

**if** 5 > 2:

**print**("Five is greater than two!")

Incorrect indentation leads to **IndentationError**. Consistent indentation is crucial for readability and syntax correctness.

**3.** **Data Types in Python**

🡺Python has several built-in data types that can be categorized into the following groups:

**1. Numeric Types**

* **int**: Whole numbers (e.g., 5).
* **float**: Decimal numbers (e.g., 5.5).
* **complex**: Complex numbers (e.g., 5j).

**2. Sequence Types**

* **str**: Strings (e.g., "Hello").
* **list**: Ordered lists (e.g., [1, 2,[3]).
* **tuple**: Ordered, immutable lists (e.g., (1, 2, 3)).
* **range**: A sequence of numbers (e.g., range(1, 6)).

**3. Mapping Type**

* **dict**: Key-value pairs (e.g., {"name": "John"}).

**4. Set Types**

* **set**: Unordered, unique items (e.g., {1, 2, 3}).
* **frozenset**: Immutable sets (e.g., frozenset({1, 2, 3})).

**5. Boolean Type**

* **bool**: True or False values.

**6. Binary Types**

* **bytes**: Binary data (e.g., b"Hello").
* **bytearray**: Mutable binary data.
* **memoryview**: Buffer interface for binary data.

**7. None Type**

* **NoneType**: Represents the absence of a value.

**Checking Data Type**

You can use the type() function to check the type of a variable:

python

x = 5

**print**(type(x)) *# Output: <class 'int'>*

**Type Conversion**

Python allows converting between data types using functions like int(), float(), str(), etc.:

python

x = 5

y = str(x) *# Converts int to str*

**print**(type(y)) *# Output: <class 'str'>*

**Mutable vs Immutable Types**

* **Mutable Types**: Can be changed after creation (e.g., list, dict).
* **Immutable Types**: Cannot be changed after creation (e.g., tuple, str).

**4.**  **Loops in Python**

🡺Python has two primary types of loops: **For Loops** and **While Loops**. Additionally, **Nested Loops** are used to execute loops within other loops.

**1. For Loops**

* **Purpose**: Iterate over a sequence (like lists, tuples, strings, dictionaries, or sets).
* **Syntax**: for iterating\_var in sequence:
* **Example**:

python

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

**for** fruit **in** fruits:

**print**(fruit)

**2. While Loops**

* **Purpose**: Execute a block of code as long as a condition is true.
* **Syntax**: while condition:
* **Example**:

python

i = 0

**while** i < 5:

**print**(i)

i += 1

**Loop Control Statements**

1. **break**: Stops the loop entirely.
2. **continue**: Skips the current iteration and moves to the next.
3. **pass**: Used when a stateent is required syntactically but no action is needed.

**Additional Features**

* **range() Function**: Used with for loops to iterate over a sequence of numbers.
* **else Clause**: Can be used with for loops to execute code when the loop finishes normally (not by a break statement).

**5.**  **Conditional Statements in Python**

🡺Conditional statements in Python are used to execute different blocks of code based on conditions or decisions. The primary types are **if**, **if-else**, and **if-elif-else** statements.

**1. If Statement**

* **Purpose**: Execute a block of code if a condition is true.
* **Syntax**: if condition: code\_to\_execute
* **Example**:

python

grade = 85

**if** grade >= 60:

**print**("You passed the exam.")

**2. If-Else Statement**

* **Purpose**: Choose between two blocks of code based on a condition.
* **Syntax**: if condition: code\_if\_true else: code\_if\_false
* **Example**:

python

grade = 85

**if** grade >= 60:

**print**("You passed the exam.")

**else**:

**print**("You failed the exam.")

**3. If-Elif-Else Statement**

* **Purpose**: Select from multiple conditions and execute different blocks of code.
* **Syntax**: if condition1: code1 elif condition2: code2 else: code3
* **Example**:

python

grade = 85

**if** grade >= 90:

**print**("A")

**elif** grade >= 80:

**print**("B")

**elif** grade >= 70:

**print**("C")

**else**:

**print**("F")

**Short-Hand If-Else (Ternary Operator)**

* **Purpose**: Simplify if-else statements into one line.
* **Syntax**: result\_if\_true if condition else result\_if\_false
* **Example**:

python

grade = 85

result = "Pass" **if** grade >= 60 **else** "Fail"

**print**(result)

**Logical Operators**

* **and**: Both conditions must be true.
* **or**: At least one condition must be true.
* **Example**:

python

a = 200

b = 33

c = 500

**if** a > b **and** c > a:

**print**("Both conditions are true")

**Match Statement (Python 3.10+)**

* **Purpose**: Similar to if-elif-else but with a more concise syntax for multiple conditions.
* **Example**:

python

grade = 85

**match** grade:

**case** grade **if** grade >= 90:

**print**("A")

**case** grade **if** grade >= 80:

**print**("B")

**case** **\_**:

**print**("Other grade")

**6. Functions in Python**

🡺A function is a reusable block of code that performs a specific task.  
It helps in:  
Code Reusability – Write once, use multiple times.  
Modularity – Break a large program into smaller parts.  
Readability – Makes the code structured and easier to understand.  
Maintainability – Easier to modify and debug.

**Types of Functions in Python**

Python has two types of functions:

1️ **Built-in Functions** – Predefined functions like print(), len(), sum(), max(), min(), etc.  
2️ **User-Defined Functions** – Functions created by the user for specific tasks.

**Defining and Calling a Function**

**Syntax**

def function\_name(parameters):

"""Docstring (optional): Describes the function"""

# Function body (code)

return result

**Example: Simple Function Without Parameters**

def greet():

print("Hello, welcome to Python!")

greet() # Calling the function

**Output:**

Hello, welcome to Python!

**Function with Parameters**

**Example: Function with One Parameter**

def greet(name):

print("Hello", name + "!")

greet("Alice")

greet("Bob")

**Output:**

Hello Alice!

Hello Bob!

**Example: Function with Multiple Parameters**

def add(a, b):

result = a + b

print("Sum:", result)

add(5, 3)

add(10, 20)

**Output:**

Sum: 8

Sum: 30

**Return Statement**

The return statement sends back a result to the caller.

**Example: Function with Return Statement**

def square(n):

return n \* n

result = square(4)

print("Square:", result)

**Output:**

Square: 16

**Default Parameter Values**

If no argument is passed, Python uses the default value.

**Example:**

def greet(name="Guest"):

print("Hello", name + "!")

greet()

greet("Alice")

**Output:**

Hello Guest!

Hello Alice!

**7. Object-Oriented Programming (OOP)**

🡺OOP is a programming paradigm based on objects that contain both data (attributes) and functions (methods). It allows for better code organization, reusability, and scalability.

**1. Class and Object**

A **class** is a blueprint for creating objects, and an **object** is an instance of a class.

**Syntax:**

class ClassName:

def \_\_init\_\_(self, attribute1, attribute2):

self.attribute1 = attribute1

self.attribute2 = attribute2

def method(self):

print("This is a method inside a class.")

# Creating an object

obj = ClassName("value1", "value2")

print(obj.attribute1) # Accessing attributes

obj.method() # Calling method

**Example:**

class Car:

def \_\_init\_\_(self, brand, color):

self.brand = brand

self.color = color

def display(self):

print(f"This car is a {self.color} {self.brand}.")

# Creating an object of Car class

car1 = Car("Toyota", "Red")

car1.display()

**2. Inheritance**

Inheritance allows a class (child class) to inherit attributes and methods from another class (parent class).

**Syntax:**

class ParentClass:

def method(self):

print("This is a method from the parent class.")

class ChildClass(ParentClass):

pass # Inherits everything from ParentClass

obj = ChildClass()

obj.method() # Calling inherited method

**Example:**

class Animal:

def speak(self):

print("Animals make sounds.")

class Dog(Animal):

def bark(self):

print("Dog barks!")

# Creating object

dog = Dog()

dog.speak() # Inherited method

dog.bark() # Own method

**3. Encapsulation**

Encapsulation restricts access to certain attributes/methods to protect data.

**Syntax:**

class ClassName:

def \_\_init\_\_(self):

self.\_\_private\_attribute = "Private" # Private variable

def get\_private(self):

return self.\_\_private\_attribute # Accessor method

**Example:**

class BankAccount:

def \_\_init\_\_(self, balance):

self.\_\_balance = balance # Private variable

def deposit(self, amount):

self.\_\_balance += amount

def get\_balance(self):

return self.\_\_balance # Getter method

# Creating object

account = BankAccount(1000)

account.deposit(500)

print(account.get\_balance()) # Output: 1500

**4. Polymorphism**

Polymorphism allows the same method to have different behaviors in different classes.

**Syntax:**

class ParentClass:

def method(self):

print("Parent class method.")

class ChildClass:

def method(self):

print("Child class method.")

# Polymorphism in action

obj1 = ParentClass()

obj2 = ChildClass()

for obj in (obj1, obj2):

obj.method() # Different behavior

**Example:**

class Bird:

def speak(self):

print("Birds chirp.")

class Dog:

def speak(self):

print("Dogs bark.")

# Using polymorphism

for animal in [Bird(), Dog()]:

animal.speak()

**5. Abstraction**

Abstraction hides implementation details and only shows relevant information using abstract classes.

**Syntax:**

from abc import ABC, abstractmethod

class AbstractClass(ABC):

@abstractmethod

def abstract\_method(self):

pass

**Example:**

from abc import ABC, abstractmethod

class Vehicle(ABC):

@abstractmethod

def fuel\_type(self):

pass

class Car(Vehicle):

def fuel\_type(self):

print("Car runs on petrol.")

# Creating object

car = Car()

car.fuel\_type()

**8. File Handling**

🡺File handling in Python allows us to create, read, write, append, and delete files. Python provides built-in functions to interact with files using the open() function.

**Basic File Handling Operations**

**1. Opening a File (open())**

* open("filename", "mode") is used to open a file.
* **Modes:**
  + "r" → Read (default, file must exist)
  + "w" → Write (creates a new file or overwrites existing content)
  + "a" → Append (adds data to an existing file)
  + "x" → Create (creates a file, but gives an error if it already exists)
  + "b" → Binary mode (e.g., "rb", "wb")
  + "t" → Text mode (default)

**Syntax:**

file = open("filename.txt", "mode")

# Perform file operations

file.close() # Always close the file

**2. Reading a File (r mode)**

Reads the content of an existing file.

**Example:**

file = open("example.txt", "r") # Open in read mode

content = file.read() # Read entire content

print(content)

file.close() # Close the file

**3. Writing to a File (w mode)**

Creates a new file or overwrites existing content.

**Example:**

file = open("example.txt", "w") # Open in write mode

file.write("Hello, this is a new file!") # Write to file

file.close() # Close the file

**4. Appending to a File (a mode)**

Adds data to an existing file without erasing the previous content.

**Example:**

file = open("example.txt", "a") # Open in append mode

file.write("\nThis is an appended line.")

file.close()

**5. Reading File Line by Line**

Reads the file one line at a time using readline() or readlines().

**Example:**

file = open("example.txt", "r")

for line in file:

print(line.strip()) # Print each line without extra newlines

file.close()

**6. Using with Statement (Recommended)**

The with statement automatically closes the file.

**Example:**

with open("example.txt", "r") as file:

content = file.read()

print(content) # File automatically closes after the block

**7. File Deletion**

We can delete a file using the os module.

**Example:**

import os

os.remove("example.txt") # Deletes the file

**9. Exception Handling**

🡺Exception handling is a mechanism to handle runtime errors (exceptions) gracefully without crashing the program. Python provides the try-except block to catch and handle errors.

**Syntax:**

try:

# Code that may cause an exception

except ExceptionType:

# Code to handle the exception

else:

# Code to execute if no exception occurs (optional)

finally:

# Code that executes no matter what (optional)

**Example: Handling Division by Zero Error**

try:

num1 = int(input("Enter a number: "))

num2 = int(input("Enter another number: "))

result = num1 / num2 # May cause ZeroDivisionError

print("Result:", result)

except ZeroDivisionError:

print("Error: Cannot divide by zero!")

except ValueError:

print("Error: Invalid input! Please enter numbers.")

else:

print("Division successful!")

finally:

print("Execution completed.") # Runs no matter what

**Common Exception Types**

* ZeroDivisionError → Division by zero
* ValueError → Invalid value (e.g., entering text instead of a number)
* IndexError → Accessing an index out of range
* KeyError → Accessing a non-existent dictionary key
* FileNotFoundError → Trying to open a missing file

**10. NumPy and Pandas**

**1. NumPy (Numerical Python)**

🡺NumPy is a Python library for numerical computing. It provides support for **multi-dimensional arrays** and **mathematical operations** like linear algebra, statistics, and element-wise operations.

**Installation:**

pip install numpy

**Basic Operations in NumPy**

|  |  |
| --- | --- |
| Operation | Syntax & Example |
| Creating an array | np.array([1, 2, 3]) |
| Creating zeros array | np.zeros((2,2)) |
| Creating ones array | np.ones((3,3)) |
| Creating random array | np.random.rand(2,2) |
| Shape of array | arr.shape |
| Reshaping array | arr.reshape(3,2) |
| Slicing an array | arr[0:2] |
| Element-wise addition | arr1 + arr2 |
| Matrix multiplication | np.dot(A, B) |

**Example: NumPy Operations**

import numpy as np

arr1 = np.array([[1, 2], [3, 4]])

arr2 = np.array([[5, 6], [7, 8]])

print("Addition:\n", arr1 + arr2)

print("Multiplication:\n", arr1 \* arr2)

result = np.dot(arr1, arr2)

print("Matrix Multiplication:\n", result)

reshaped = arr1.reshape(1, 4)

print("Reshaped Array:\n", reshaped)

**2. Pandas (Data Analysis Library)**

Pandas is a Python library for **data analysis and manipulation**. It provides two main data structures:

* **Series** (1D labeled array)
* **DataFrame** (2D table with labeled rows & columns)

**Installation:**

pip install pandas

**Basic Operations in Pandas**

|  |  |
| --- | --- |
| Operation | Syntax & Example |
| Creating a Series | pd.Series([10, 20, 30]) |
| Creating a DataFrame | pd.DataFrame({'A': [1,2], 'B': [3,4]}) |
| Read CSV file | pd.read\_csv("file.csv") |
| Write CSV file | df.to\_csv("file.csv", index=False) |
| View first 5 rows | df.head() |
| View last 5 rows | df.tail() |
| Get column names | df.columns |
| Selecting a column | df['column\_name'] |
| Filtering data | df[df['column'] > 50] |
| Sorting data | df.sort\_values(by='column') |
| Grouping data | df.groupby('column').sum() |
| Checking missing values | df.isnull().sum() |

**Example: Pandas Operations**

import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie'],

'Age': [25, 30, 35],

'Salary': [50000, 60000, 70000]}

df = pd.DataFrame(data)

print("DataFrame:\n", df)

print("\nFirst 2 rows:\n", df.head(2))

print("\nSalary column:\n", df['Salary'])

filtered\_df = df[df['Age'] > 28]

print("\nFiltered Data:\n", filtered\_df)

**NumPy vs. Pandas**

|  |  |  |
| --- | --- | --- |
| Feature | NumPy | Pandas |
| Data Type | Arrays | DataFrames & Series |
| Usage | Numerical operations | Data analysis & manipulation |
| Speed | Faster for computations | Slower due to overhead |
| Flexibility | Less flexible | More flexible for handling structured data |

**11. Selenium for Web Scraping**

🡺Selenium is an automation tool that allows us to control web browsers programmatically. It is widely used for web scraping, testing web applications, and automating browser tasks.

**Selenium Useful for Web Scraping**

Unlike other libraries like BeautifulSoup and requests, Selenium is useful when:  
Websites use JavaScript to load data dynamically.  
You need to interact with elements (click buttons, fill forms).  
You want to scrape data from complex websites requiring user actions.

**Installation**

First, install Selenium and the WebDriver for your browser:

pip install selenium

Download the appropriate WebDriver (e.g., ChromeDriver for Chrome)

**Syntax: Basic Selenium Web Scraping**

from selenium import webdriver

from selenium.webdriver.common.by import By

driver = webdriver.Chrome() # or webdriver.Firefox()

driver.get("https://example.com")

element = driver.find\_element(By.TAG\_NAME, "h1")

print("Heading:", element.text)

driver.quit()

**Example: Scraping a website**

Let's scrape **quotes** from [Quotes to Scrape](http://quotes.toscrape.com/).

from selenium import webdriver

from selenium.webdriver.common.by import By

from selenium.webdriver.common.keys import Keys

import time

driver = webdriver.Chrome()

driver.get("http://quotes.toscrape.com/")

quotes = driver.find\_elements(By.CLASS\_NAME, "text")

for quote in quotes:

print(quote.text)

driver.quit()

**Benefits of Using Selenium for Scraping**

Handles JavaScript: Unlike BeautifulSoup, Selenium loads dynamic content.  
Interacts with elements: Can click buttons, fill forms, scroll pages.  
Automates browsing tasks: Useful for web automation and testing.  
Supports multiple browsers: Chrome, Firefox, Edge, etc.

**12. Regular Expressions (Regex)**

🡺Regular Expressions (regex) are patterns used to match, search, and manipulate strings. Python provides the re module for working with regex.

**How is Regex Useful?**

Validating emails, phone numbers, or passwords  
Searching for patterns in text (e.g., finding all dates in a document)  
Replacing or modifying text (e.g., removing special characters)  
Extracting data from structured text

**3. Basic Syntax and Operations**

|  |  |
| --- | --- |
| Operation | Syntax & Example |
| Importing regex | import re |
| Search a pattern | re.search(pattern, string) |
| Find all matches | re.findall(pattern, string) |
| Replace text | re.sub(pattern, replace, string) |
| Split text | re.split(pattern, string) |

**4. Example: Basic Regex Operations**

import re

# Sample text

text = "My email is example@gmail.com and my phone number is 9876543210."

# 1. Search for an email

email\_pattern = r"\b[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b"

email = re.search(email\_pattern, text)

if email:

print("Email found:", email.group())

# 2. Find all numbers in text

numbers = re.findall(r"\d+", text)

print("Numbers found:", numbers)

# 3. Replace numbers with [REDACTED]

modified\_text = re.sub(r"\d+", "[REDACTED]", text)

print("Modified text:", modified\_text)

**5. Common Regex Patterns**

|  |  |  |
| --- | --- | --- |
| Pattern | Description | Example Match |
| \d | Any digit (0-9) | "123" in "abc123" |
| \D | Any non-digit | "abc" in "abc123" |
| \w | Any word character (a-z, A-Z, 0-9, \_) | "hello" in "hello!" |
| \W | Any non-word character | "!" in "hello!" |
| \s | Any whitespace (space, tab) | " " in "hello world" |
| \b | Word boundary | Matches at the start or end of a word |
| ^ | Start of string | "^Hello" matches "Hello world" |
| $ | End of string | "world$" matches "Hello world" |
| . | Any character except newline | "c.t" matches "cat", "cut" |
| \* | 0 or more occurrences | "go\*" matches "g", "go", "goo" |
| + | 1 or more occurrences | "go+" matches "go", "goo" (not "g") |
| {n,m} | Between n and m occurrences | "a{2,4}" matches "aa", "aaa", "aaaa" |
| [abc] | Any one of the characters | "[aeiou]" matches any vowel |
| `(abc | def)` | Matches either "abc" or "def" |

**6. Example: Extracting Phone Numbers**

import re

text = "Call me at 9876543210 or 123-456-7890."

pattern = r"\b\d{10}\b|\b\d{3}-\d{3}-\d{4}\b"

phones = re.findall(pattern, text)

print("Phone Numbers:", phones)

**7. Benefits of Using Regex**

Fast and efficient text processing  
Powerful pattern-matching capabilities  
Works with various data formats (text, logs, HTML)

**13. Multithreading**

🡺Multithreading is a technique where multiple threads run concurrently within a single process. It allows tasks to execute in parallel, improving efficiency in I/O-bound operations (e.g., file reading, network requests).

**How is Multithreading Useful?**

Running multiple tasks simultaneously (e.g., downloading multiple files)  
Improves performance for I/O-bound tasks (networking, file operations)  
Keeps the program responsive (e.g., GUI applications)

**Syntax: Creating and Running Threads**

Python's threading module is used for multithreading.

import threading

def print\_numbers():

for i in range(5):

print(f"Number: {i}")

thread1 = threading.Thread(target=print\_numbers)

thread2 = threading.Thread(target=print\_numbers)

thread1.start()

thread2.start()

thread1.join()

thread2.join(

print("Both threads completed!")

**Example: Multithreading with Sleep**

import threading

import time

def task(name):

for i in range(3):

time.sleep(1) # Simulates a delay

print(f"Task {name}: {i}")

t1 = threading.Thread(target=task, args=("A",))

t2 = threading.Thread(target=task, args=("B",))

t1.start()

t2.start()

t1.join()

t2.join()

print("All tasks completed!")

**Output (Threads Run in Parallel)**

Task A: 0

Task B: 0

Task A: 1

Task B: 1

Task A: 2

Task B: 2

All tasks completed!

**Using Locks to Prevent Race Conditions**

If multiple threads modify shared data, we may encounter **race conditions** (unexpected behavior). Use threading.Lock() to prevent this.

import threading

lock = threading.Lock()

counter = 0

def increment():

global counter

for \_ in range(100000):

with lock: # Locking ensures only one thread modifies `counter` at a time

counter += 1

t1 = threading.Thread(target=increment)

t2 = threading.Thread(target=increment)

t1.start()

t2.start()

t1.join()

t2.join()

print("Final Counter:", counter) # Without lock, the value might be incorrect!

**7. Benefits of Multithreading**

Runs multiple tasks concurrently  
Improves efficiency for I/O-heavy applications  
Keeps UI applications responsive

🔹 **Limitations:** Due to Python’s GIL, multithreading does not provide true parallel execution for CPU-bound tasks. Use multiprocessing for that.

**14. Concurrency vs. Parallelism in Python**

|  |  |  |
| --- | --- | --- |
| Feature | Concurrency (Multitasking) | Parallelism (True Simultaneous Execution) |
| Definition | Multiple tasks start, run, and complete in overlapping time periods (but not necessarily at the same moment). | Multiple tasks run at the exact same time using multiple CPU cores. |
| Execution | Tasks take turns running but share the same CPU. | Tasks run simultaneously on different CPU cores. |
| Used For | I/O-bound tasks (e.g., web scraping, database calls, network requests). | CPU-bound tasks (e.g., heavy computations, AI/ML training, image processing). |
| Example | Web scraper downloading multiple pages asynchronously. | Running multiple AI models on different CPU cores. |
| Implementation in Python | threading (Multithreading) and asyncio (Asynchronous programming). | multiprocessing (Multiple processes). |
| Global Interpreter Lock (GIL) Effect | Affected by GIL (threads don’t run truly in parallel). | Not affected by GIL (each process has its own memory space). |
| Efficiency | More efficient for tasks that involve waiting (e.g., reading from a file). | More efficient for tasks that require heavy computation. |
| Example Code | threading.Thread(target=my\_function) or asyncio.run(my\_function()) | multiprocessing.Process(target=my\_function) |

## Other topics

**1. Software Development Life Cycle (SDLC)**

🡺The Software Development Life Cycle (SDLC) is a structured process for developing software applications. It consists of multiple phases, ensuring software is built systematically, efficiently, and with minimal risks.

**Phases of SDLC**

1. **Requirement Analysis**
   * Understanding and gathering business requirements.
   * Stakeholders (clients, users, developers) discuss the scope and functionalities.
   * Example: A company needs a website with an online store and a payment gateway.
2. **Planning**
   * Defining project scope, budget, and timelines.
   * Identifying risks and resource allocation.
   * Example: Estimating 6 months for development, assigning 4 developers and 2 testers.
3. **Design**
   * **High-level design (HLD):** System architecture, tech stack selection.
   * **Low-level design (LLD):** Detailed class structures, database schema, UI/UX mockups.
4. **Development (Implementation)**
   * Writing code based on the design documents.
   * Follows coding standards (e.g., PEP-8 in Python).
   * Example: Backend development using Django, frontend with React.
5. **Testing**
   * Detecting and fixing bugs.
   * Types of testing: Unit, Integration, System, User Acceptance Testing (UAT).
6. **Deployment**
   * Deploying software in a live environment.
   * May use CI/CD pipelines for automation.
7. **Maintenance & Support**
   * Fixing post-release bugs.
   * Updating software for improvements or security patches.

**2. Agile and Scrum**

**Agile**

Agile is a software development methodology focused on iterative development, collaboration, and flexibility.

It values:

* Individuals over processes
* Working software over documentation
* Customer collaboration
* Responding to change

**Scrum**

Scrum is a framework within Agile that organizes development into short, iterative cycles called sprints (1-4 weeks).

**Scrum Roles**

* Product Owner: Defines features and prioritizes backlog items.
* Scrum Master: Facilitates the team, removes obstacles.
* Development Team: Developers, testers, designers.

**Scrum Artifacts**

* Product Backlog: List of all required features.
* Sprint Backlog: Features selected for the current sprint.
* Increment: The working software delivered at the end of each sprint.

**Scrum Events**

1. Sprint Planning – Team decides what to build in a sprint.
2. Daily Standup (Daily Scrum) – 15-minute updates on progress.
3. Sprint Review – Demonstrate completed work to stakeholders.
4. Sprint Retrospective – Analyze what went well and what needs improvement.

**3. Code Version Control**

Version control systems (VCS) track changes in code, enabling multiple developers to collaborate efficiently.

**Types of Version Control**

* Local Version Control – Saves versions on a local machine.
* Centralized Version Control (CVCS) – Uses a single central server (e.g., SVN).
* Distributed Version Control (DVCS) – Developers have local copies (e.g., Git).

**Git – The Most Popular VCS**

* Git Commands:
  + git init – Initialize a repository.
  + git clone <repo\_url> – Copy a repository.
  + git add <file> – Stage changes.
  + git commit -m "message" – Save changes.
  + git push – Upload changes to a remote repository.
  + git pull – Get the latest changes from the remote.

**4. Documentation**

Documentation is essential for maintainability, collaboration, and onboarding new developers.

**Types of Documentation**

1. Code Documentation – Comments and docstrings in code.
2. API Documentation – Details on how to use an API (Swagger, Postman).
3. User Documentation – Guides for end users.
4. Technical Documentation – Architecture diagrams, database schemas.

Example: A Python library like NumPy provides detailed docstrings so developers understand its functions easily.

**5. Risk Management**

Risk management helps identify, assess, and mitigate potential risks in software development.

**Risk Types**

* Technical Risks: Bugs, security vulnerabilities.
* Business Risks: Changing client requirements.
* Operational Risks: Server failures, deployment issues.

**Risk Management Process**

1. Identify risks – e.g., outdated libraries causing security vulnerabilities.
2. Assess impact – High, medium, or low severity.
3. Mitigate risks – Backup plans, extra testing.

**6. Python Coding Standards**

Python has conventions to write clean and readable code. The most widely accepted standard is PEP-8.

**7. PEP-8 (Python Enhancement Proposal 8)**

PEP-8 is a Python style guide with best practices:

* Use 4 spaces per indentation.
* Limit lines to 79 characters.
* Use meaningful variable names (total\_price instead of tp).
* Write comments/docstrings properly (explained below).

Example:

def add\_numbers(a, b):

"""Returns the sum of two numbers."""

return a + b

**8. Comments and Docstrings**

**Comments**

* Inline comments:
* x = 5 # This is an inline comment
* Block comments: Used for explaining complex logic.

**Docstrings**

Used to document functions, classes, and modules.

def greet(name):

"""Returns a greeting message."""

return f"Hello, {name}!"

**9. Error Handling and Logging**

* Error Handling – Using try-except blocks to catch exceptions.
* Logging – Recording errors using the logging module instead of print().

Example:

import logging

logging.basicConfig(level=logging.ERROR)

try:

x = 1 / 0

except ZeroDivisionError as e:

logging.error(f"Error occurred: {e}")

**10. Writing Efficient Code**

* **Use List Comprehensions:**
* squares = [x\*\*2 for x in range(10)]
* **Use Generators for Memory Efficiency:**
* def count\_up():
* for i in range(10):
* yield i
* **Avoid Unnecessary Loops:** Use vectorized operations in NumPy and pandas.

**11. Various Software Development Principles**

* DRY (Don't Repeat Yourself) – Avoid redundant code.
* KISS (Keep It Simple, Stupid) – Keep code simple.
* SOLID (Object-Oriented Design Principles) – Improve maintainability.
* YAGNI (You Aren’t Gonna Need It) – Don’t add unnecessary features.

**12. Unit Testing & Validation**

* **Unit Testing** – Testing individual components using unittest or pytest.
* **Validation** – Ensuring input data is correct before processing.

Example using unittest:

import unittest

def add(a, b):

return a + b

class TestAdd(unittest.TestCase):

def test\_sum(self):

self.assertEqual(add(2, 3), 5)

**13. Ruff & Black (Python Linters & Formatters)**

* **Ruff:** A fast Python linter that enforces code quality.
* **Black:** A formatter that automatically formats Python code.

Example usage:

black script.py # Formats the file

ruff check script.py # Checks for issues