ApexaIQ Assignment Submission -02

Q1 What Is Python

1. Python is a high-level, interpreted programming language known for its simplicity and readability.

2. It is widely used in:

3. Web Development (Django, Flask)

4. Data Science and Machine Learning (Pandas, NumPy, TensorFlow)

5. Automation and Scripting

6. Game Development (Pygame)

Python has a large community and extensive libraries, making it beginner-friendly Python

Syntax : Python syntax is simple and readable – Indentation(spaces) defines code blocks instead of braces {}.

Example: Simple Python syntax

```
name = "Om"

if name == "Om":
   print("Hello, Om!")   # Indentation is important
```

Variables: Variables are used to store data. No need to declare type; Python infers it automatically.

```
x = 10        # Integer

name = "Om"     # String

pi = 3.14      # Float
```

Rules for variables names:

- Must start with a letter or underscore
- Cannot start with a number
- No spaces or special characters

Variable Naming Rules :

- Variable names can contain letters, numbers, and underscores.
- Variable names must start with a letter or underscore.

```
name = "om"

 age = 25

height = 5.6
```

Data Types in Python :

Python supports several built-in data types:

Integers (int): whole numbers(e.g,10 , -5).

Floats (float):Decimal numbers (e.g 3.14 ,-0.001).

String (str):Text data enclosed in quotes (e.g “hello” ,”python”)

Booleans (bool):Represent True or False

Lists : Ordered ,mutable collections(e.g [1,2,3,4])

Tuples : Ordered ,immutable collections(e.g (1,2,3))

Sets : Unordered collection of unique elements (e.g {1,2,3})

Dictionaries : Key – Value pairs(e.g {“name”:”Alice”,”age”:24})

Checking Data Types

- Use the `type()` function to check the data type of a variable.

```
print(type(10))        # Output: <class 'int'>
print(type("Hello"))   # Output: <class 'str'>
```

Control Flow and Loops :

If – Else Conditional Statements

What are conditional Statements ?

- Conditional statements allow you to execute code based on certain conditions
- Python uses if,elif , and else for decision-making

Syntax:

```
if condition1:
    # Code to execute if condition1 is
True elif condition2:
    # Code to execute if condition2 is
True else:
```

Example:

```
age = 18

if age < 18:
    print("You are a minor.")
elif age == 18:
    print("You just became an adult!")
else:
```

Match Case Statements In Python

What is Match Case ?

- Match – case is a new feature introduces in python 3.10 for pattern matching
- It simplifies complex condititional logic

Syntax:

```
match value:
    case pattern1:
        # Code to execute if value matches pattern1
    case pattern2:
        # Code to execute if value matches pattern2
    case _:
```

```
status = 404


match status:
    case 200:
        print("Success!")
    case 404:
        print("Not Found")
    case _:
        print("Unknown Status")
```

For Loops in Python

What are For Loops?

- For loops are used to iterate over a sequence (e.g., list, string, range).
- They execute a block of code repeatedly for each item in the sequence.

Syntax:

```
for item in sequence:
    # Code to execute for each item
```

Example:

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

for fruit in fruits:
    print(fruit)
```

Using range():

- The range() function generates a sequence of numbers
- Examples

```
for i in range(5):
    print(i)  # Output: 0, 1, 2, 3, 4
```

While Loops in Python

What are While Loops?

- While loops execute a block of code as long as a condition is `True`.
- They are useful when the number of iterations is not known in advance.
-

Syntax:

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

Example:

```
count = 0

while count < 5:
    print(count)
```

Infinite Loops:

- Be careful to avoid infinite loops by ensuring the condition eventually becomes `False`.
- Example of an infinite loop:

```
while True:
    print("This will run forever!")
```

Break , Continue Statements

Break :

- The break statement is used to exit a loop prematurely
- Example:

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

Continue :

- The continue statements skips the rest of the code in the current ireration and moves to the next iteration
- Example:

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

Functions (parameters, return values, default args,*args, **kwargs)

Defining Functions in Python

Functions help in reusability and modularity in Python.

Syntax:

```
def greet(name):
    return f"Hello, {name}!"

print(greet("Alice"))  # Output: Hello, Alice!
```

Function Arguments & Return Values

Functions can take parameters and return values Types of Arguments:

1. Positional Arguments

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

2 . Default Arguments

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

print(greet())  # Output: Hello, Guest!
```

3 .Keyword Arguments

```
def student(name, age):
    print(f"Name: {name}, Age: {age}")


student(age=20, name="Bob")
```

Decorators in Python

Decorators in Python are a powerful and expressive feature that allows you to modify or enhance functions and methods in a clean and readable way. They provide a way to wrap additional functionality around an existing function without permanently modifying it. This is often referred to as *metaprogramming*, where one part of the program tries to modify another part of the program at compile time.

Decorators use Python's higher-order function capability, meaning functions can accept other functions as arguments and return new functions.

Understanding Decorators

A decorator is simply a callable (usually a function) that takes another function as an argument and returns a replacement function. The replacement function typically *extends* or *alters* the behavior of the original function

Basic Example of a Decorator

```
def my_decorator(func): def
    wrapper():

        print("Something is happening before the function is call
        func()

        print("Something is happening after the function is calle
    return wrapper
```

Output:

```
Something is happening before the function is called.
Hello!

Something is happening after the function is called.
```

```
@my decorator

Def say_hello():

        Print("Hello")

Say_hello
```

Using Decorators with Arguments

Decorators themselves can also accept arguments. This requires another level of nesting: an outer function that takes the decorator's arguments and returns the

```
def repeat(n):
    def decorator(func):
        def wrapper(a):
            for _ in range(n):
                func(a)
        return wrapper
    return decorator


@repeat(3)
def greet(name):
    print(f"Hello, {name}!")
```

```
greet("world")
```

Output:

```
Hello, world!
Hello, world!
Hello, world!
```

Inheritance In Python : Inheritance is like a family tree. A child class (or *subclass*) inherits traits (attributes and methods) from its parent class (or *superclass*). This allows you to create new classes that are specialized versions of existing classes, without rewriting all the code

```
class Animal:  # Parent class (superclass)
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("Generic animal sound")

class Dog(Animal):  # Dog inherits from Animal (Dog is a subclass
    def speak(self):# We *override* the speak method (more on t
        print("Woof!")

class Cat(Animal):  # Cat also inherits from Animal
    def speak(self):
        print("Meow!")

# Create objects:
my_dog = Dog("Rover")
my_cat = Cat("Fluffy")

# They both have a 'name' attribute (inherited from Animal):
print(my_dog.name)  # Output: Rover
print(my_cat.name)  # Output: Fluffy

# They both have a 'speak' method, but it behaves differently:
my_dog.speak()  # Output: Woof!
my_cat.speak()  # Output: Meow!
```

Exception Handling In Python

Exceptions are events that occur during the execution of a program that disrupt the normal flow of instructions. Python provides a robust mechanism for handling

exceptions using try - except blocks. This allows your program to gracefully

recover from errors or unexpected situations, preventing crashes and providing informative error messages. You can also define your own custom exceptions to represent specific error conditions in your application.

Basic Exception Handling

The except block is the fundamental construct for handling exceptions:

•The try block contains the code that might raise an exception

- The `except` block contains the code that will be executed if a specific

  exception occurs within the `try` block.

```
try:
    x = 10 / 0  # This will raise a ZeroDivisionError
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

Output:

```
Cannot divide by zero!
```

Handling Multiple Exceptions

You can handle multiple types of exceptions using multiple except blocks or by specifying a tp

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("You can't divide by zero!")
except ValueError:
    print("Invalid input! Please enter a number.")



# Alternative using a tuple:
try:
  num = int(input("Enter a number: "))
  result = 10 / num
except (ZeroDivisionError, ValueError) as e:
  print(f"An error occurred: {e}")
```

Using else and finally

- Else : The else block is optional and is executed only if no exception occurs within the try block .its useful for code that should run only when the try block succeeds.
- Finally :The Finally block is also optional and is always executed , regardless of whether an exception occurred or not .its typically used for ceeanup operations such as closing files or releasing resources

```
try:
    file = open("test.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File not found!")
else:
    print("File read successfully.")
    print(f"File contents:\n{content}")
```

❖ Object Orinented Programming

What is OOP?

Imagine you're building with LEGOs. Instead of just having a pile of individual bricks (like in *procedural programming*), OOP lets you create pre-assembled units – like a car, a house, or a robot. These units have specific parts (data) and things they can do (actions).

That's what OOP is all about. It's a way of programming that focuses on creating "objects." An object is like a self-contained unit that bundles together:

- Data (Attributes): Information about the object. For a car, this might be its color, model, and speed.
- Actions (Methods): Things the object can do. A car can accelerate, brake, and turn.

Why Bother with OOP?

OOP offers several advantages:

- Organization: Your code becomes more structured and easier to navigate. Large projects become much more manageable.
- Reusability: You can use the same object "blueprints" (classes) multiple times, saving you from writing the same code over and over.
- Easier Debugging: When something goes wrong, it's often easier to pinpoint the problem within a specific, self-contained object.
- Real-World Modeling: OOP allows you to represent real-world things and their relationships in a natural way.

The Four Pillars of OOP

OOP is built on four fundamental principles:

1. Abstraction: Think of driving a car. You use the steering wheel, pedals, and gearshift, but you don't need to know the complex engineering under the hood. Abstraction means hiding complex details and showing only the essential information to the user.
2. Encapsulation: This is like putting all the car's engine parts inside a protective casing. Encapsulation bundles data (attributes) and the methods that

operate on that data *within* a class. This protects the data from being accidentally
changed or misused from outside the object. It controls access.

3. Inheritance: Imagine creating a "SportsCar" class. Instead of starting from scratch, you can build it upon an existing "Car" class. The "SportsCar" *inherits* all the features of a "Car" (like wheels and an engine) and adds its own special features (like a spoiler). This promotes code reuse and reduces redundancy.
4. Polymorphism: "Poly" means many, and "morph" means forms. This means objects of different classes can respond to the same "message" (method call) in their own specific way. For example, both a "Dog" and a "Cat" might have a `make_sound()` method. The dog will bark, and the cat will meow – same method name, different behavior.

2. Classes and Objects: The Blueprint and the Building

- Class: Think of a class as a blueprint or a template. It defines what an object *will be like* – what data it will hold and what actions it can perform. It doesn't create the object itself, just the instructions for creating it. It's like an architectural plan for a house.
- Object (Instance): An object is a *specific instance* created from the class blueprint. If "Car" is the class, then *your* red Honda Civic is an object (an instance) of the "Car" class. Each object has its own unique set of data. It's like the actual house built from the architectural plan.

Let's see this in Python:

```
class Dog:  # We define a class called "Dog"
    species = "Canis familiaris" # A class attribute (shared by a
```

```
    f __init__(self, name, breed): # The constructor (explaine
        self.name = name   # An instance attribute to store the
        self.breed = breed    # An instance attribute to store
        the

    def bark(self):          # A method (an action the dog can
        do print(f"{self.name} says Woof!")

# Now, let's create some Dog objects:
my_dog = Dog("Buddy", "Golden Retriever")  # Creating an object
c another_dog = Dog("Lucy", "Labrador")   # Creating another
obje

# We can access their attributes:
print(my_dog.name)       # Output: Buddy
print(another_dog.breed)  # Output: Labrador

# And make them perform actions:
my_dog.bark()            # Output: Buddy says
Woof! print(Dog.species) # Output: Canis
familiaris
```

self : Inside a class ,self is like saying “This particular object.” It’s a way for the object to refer to itself . it’s a always the first parameter in a method definition .but python handles it automatically when you call the method you don’t type self when calling the method Python inserts it for you.

Constructor : (__init__)

The `__init__` method is special. It’s called the constructor. It’s automatically run whenever you create a *new* object from a class.

What's it for? The constructor's job is to *initialize* the object's attributes – to give them their starting values. It sets up the initial state of the object.

```
class Dog:
    def __init__(self, name, breed):  # The constructor
        self.name = name      # Setting the name attribute
        self.breed = breed    # Setting the breed attribute

# When we do this:
my_dog = Dog("Fido", "Poodle")  # The __init__ method is automati

# It's like we're saying:
# 1. Create a new Dog object.
# 2. Run the __init__ method on this new object: #
    - Set my_dog.name to "Fido"
#    - Set my_dog.breed to "Poodle"
```

You can also set default values for parameters in the constructor, making them optional when creating an object:

```
class Dog:
    def __init__(self, name="Unknown", breed="Mixed"):
        self.name = name
        self.breed = breed

dog1 = Dog()          # name will be "Unknown", breed will be "Mi
dog2 = Dog("Rex")     # name will be "Rex", breed will be "Mixed"
dog3 = Dog("Bella", "Labrador") # name will be "Bella", breed wil
```

1. Modules and Pip - Using External Libraries

Importing Modules

Python provides built-in and third-party modules.

Example: Using the `math` module

```
import math

print(math.sqrt(16))  # Output: 4.0
```

Creating Your Own Module

Save this as `mymodule.py` :

```
def greet(name):
    return f"Hello, {name}!"
```

Import in another file:

```
import mymodule
print(mymodule.greet("Alice"))  # Output: Hello, Alice!
```

Installing External Libraries with `pip`

```
pip install requests
```

Example usage:

```
import requests
response = requests.get("https://api.github.com")
print(response.status_code)
```

List Comprehension and Dictionary Comprehension

List Comprehension is a short and efficient way to create lists using a single line of code

[expression for item in iterable if condition]

Example 1: Create a list of squares

```
squares = [x**2 for x in range(1, 6)]

print(squares)
```

Output : [1, 4, 9, 16, 25]

Example 2: Even Numbers only

Output : [0,2,4,6,8]

Dictionary Comprehension

Dictionary comprehension lets you create dictionaries quickly from iterables

Example 1: Square of numbers

```
Square_dict = {x: x**2 for x in range(1,6)}

Print(square_dict)
```

Example 2: Converts items to key-value pairs

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

lengths = {fruit: len(fruit) for fruit in fruits}

print(lengths)
```

Output : {'apple': 5, 'banana': 6, 'cherry': 6}

Python Coding Standards

Naming Convention : Naming convention make your code clean,readable and consistent.

- **Use snake_case**

```
user_name = "Om"

def calculate_total():

  pass
```

- **Use PascalCase / CamelCase**

```
class StudentProfile:

  pass
```

- **Constants**
- **Use UPPERCASE**

```
MAX_LIMIT = 100

PI = 3.14
```

Docstring (Documentation String)

Docstring describe what a function ,class or module does.They are written inside triple quotes(“”” “””) right after the definition.

Example :

```
def add_numbers(a, b):

  """

  Adds two numbers and returns the sum.
```

Parameters:

a (int): First number

b (int): Second number

Returns:

int: The sum of a and b

"""

return a + b

Comments: comments help others understand your code

Single – line comment : # This is a single-line comment

Multi – line comment : """

This is a multi-line comment.

Used to describe logic in detail.

"""

Types Of Testing

Unit Testing : Tests individual functions or components

Integration Testing : Tests how multiple modules work together

System Testing : Tests the complete application as a whole

Acceptance Testing : Ensures products meets user requirements

Regression Testing : Ensure new changes don't break old functionality

PEP 8 (Python Enhancements proposal 8)

PEP 8 defines Pythons official style guide for writing clean code

Rules	Example
Indentation	use 4 spaces per indent

Line Length	Max 79 characters per line
Blank Lines	Use blank lines between functions/classes
Imports	Keep imports at the top of the file
Spaces	Add spaces around operators

Solid Principles (for oop design)

S – Single Responsibility

O – Open /Closed

L – Liskov Substitution

I – Interface Segregation

D – Dependency Inversion

DRY principle : Avoid writing the same code multiple times – use functions ,loops ,or classes instead

Example : print(“Hello,om”)

print(“Hello,Patil”)

print(“Hello,Yash”)

Good(DRY):

```
def greet(name):
   print(f"Hello, {name}")


for person in ["Om", "Ramesh", "Patil"]:
   greet(person)
```

What Is API :

API (Application Programming Interface) is a set of rules that allows two software systems to communicate with each other.

It defines how requests and responses should be made between client and server.

Example : https://api.weather.com/data?city=Mumbai

Types Of APIs

Open /Public API : Available for public use

Partner API : Shared with specific partners

Private API : Used with in a company

Composite API: Combines multiples APIs into one response

HTTP Status Codes

HTTP status codes tell whether a request was successful or failed

Code	Meaning	Description
200	Ok	Request Successful
201	Created	New Resource Created
400	Bad Request	Invalid Input
401	Unauthorized	Missing/Invalid

403	Forbidden	No access rights
404	Not Found	Resourece not found
500	Internal Server Error	Server creashed

Response Formats

API commonly return data in json or xml format

JSON

```
{
  "name": "Om",
  "age": 21,
  "city": "Malkapur"
}
```

XML :

```
<student>
  <name>Om</name>
  <age>21</age>
  <city>Malkapur</city>
</student>
```

Types Of API Authentication

API Keys : Simple key sent with each request

Basic Auth : Username + Password

Bearer Token (OAuth2) : Secure token based Auth

JWT (JSON Web Token):Encoded user unfo for secure access

API Versioning and security

Versioning : Help maintain backward compatiability when API changes

/api/v1/users

/api/v2/users

Security Measures

- Use HTTPS instead of HTTP
- Validate inputs
- Limit request rate (Rate Limiting)
- Use tokens / OAuth2
- Never expose credentials in code

CRUD Operation in API

CRUD - > Create, Read , Update ,Delete

Operation	HTTP Method	Ex –URL
Create	POST	/api/users
Read	GET	/api/users/1
Update	PUT/PATCH	/api/users/1
Delete	DELETE	/api/users/1

Explore POSTMAN : POSTMAN is a popular tool to Test API easily – no coding required

Steps:

Open Postman

Select request type: GET, POST, etc.

Enter URL (e.g., https://jsonplaceholder.typicode.com/users)

Add headers or body

Click Send → View response in JSON format

Optimization and Efficiency

Good APIs are:

Fast: use caching (e.g., Redis)

Lightweight: avoid large responses

Paginated: break long lists into smaller chunks

Reusable: follow RESTful principles

Documented: clear docs (Swagger, OpenAPI)

Request Library in python

Python's `requests` library is used to **send API calls easily**.

Installation : pip install requests

GET Request Example :

```
import requests

response = requests.get("https://jsonplaceholder.typicode.com/users/1")

print(response.status_code)

print(response.json())
```

POST Request Example :

```
import requests


data = {"name": "Om", "city": "Malkapur"}
response = requests.post("https://jsonplaceholder.typicode.com/users", json=data)
print(response.status_code)
print(response.json())
```

Header + Token Example :

```
headers = {"Authorization": "Bearer my_api_token"}
response = requests.get("https://api.example.com/data", headers=headers)
print(response.json())
```

Software Development Life Cycle(SDLC) :

T he Software Development Life Cycle (SDLC) is the step-by-step process used to design, develop, test, and deploy software efficiently and with quality.

1. Requirement Analysis : Understand client needs and document software requirements.

2. System Design : Create architecture, design database, and UI mockups.

3. Implementation (Coding) : Developers write code according to design.

4. Testing : QA team verifies functionality and fixes bugs.

5. Deployment : Deliver the application to the user or production environment.

6. Maintenance: Provide updates, improvements, and bug fixes after release.

SDLC Models :

Waterfall : Sequential — each phase depends on the previous one.

Iterative : Development occurs in small parts (iterations).

Spiral : Combines design + risk analysis (best for large projects).

Agile : Continuous development and feedback (most popular).

Agile Basics : Agile is a flexible and iterative approach to software development that focuses on customer feedback, team collaboration, and continuous delivery.dback (most popular).

Agile Principles (from the Agile Manifesto):

1. Customer satisfaction through early and continuous delivery
2. Welcome changing requirements
3. Deliver working software frequently
4. Business and developers work together
5. Build projects around motivated individuals
6. Face-to-face communication is best
7. Working software is the measure of progress
8. Sustainable development pace
9. Continuous attention to quality and design

Version Controls: Version Control Systems (VCS) are tools that track changes in code over time, enabling collaboration and rollback if needed.

Benefits:

- Keeps history of every change
- Enables teamwork (multiple developers)
- Allows branching and merging
- Helps recover old versions

```
# Initialize a repository

git init

# Add files to staging

git add .

# Commit changes

git commit -m "Initial commit"

# Link to remote repository

git remote add origin https://github.com/username/repo.git

# Push changes to GitHub

git push -u origin main
```