# Python Research Topics -

Indentation, comments

Data types

Loops

Conditional statement

Functions

Oops

File handling

Exception handling

Numpy and pandas

Selenium for scrapping

Regex

Multithreading/multiprocessing

Concurrency vs Parallelism

Other Topics -

SDLC

Agile and Scrum

Code version control

Doc

Risk management

Python coding standards

PEP-8

Comments/Docstring

Error handling and logging

Efficient code

Various Principles

Unit testing - validation

Ruff, black

## 1. Indentation, comments

**What is Indentation?**

Indentation in Python refers to the spaces or tabs used at the beginning of a line of code.

It is crucial for defining the structure of the code, such as blocks within loops, conditionals, functions, and classes.

Python uses indentation instead of brackets to denote code blocks, which enhances readability and simplifies the syntax.

Comments in Python

**What are Comments?**

Comments in Python are annotations added to the code to explain what the code does.

They are ignored by the interpreter and are useful for making code more understandable and for debugging purposes

**Types of Comments:**

1. **Single-Line Comments:**
    - Start with the # symbol and continue until the end of the line.
    - Example: # This is a single-line comment
2. **Multiline Comments:**
    - Python does not have a specific syntax for multiline comments. However, you can use multiple single-line comments or enclose text in triple quotes (''' or """) to create a multiline comment.
    - Example: ''' This is another way to create a multiline comment. '''
3. **Docstring Comments:**
    - These are used to document functions, classes, and modules. They are enclosed in triple quotes (''' or """) and are placed immediately after the function or class definition.
    - Example:
    ```
    def greet(name):
        """This function prints a greeting message."""
        print(f"Hello, {name}!")
    ```

## 2. Data types

**Python Data Types**

Python has several built-in data types that are categorized into different groups based on their characteristics and uses. Here's an overview of the main data types in Python:

**1. Numeric Types**

- **Integers (int):** Whole numbers, e.g., 5.
- **Floats (float):** Decimal numbers, e.g., 3.1415.
- **Complex Numbers (complex):** Numbers with real and imaginary parts, e.g., 1+2j.

**2. Sequence Types**

- **Strings (str):** Text enclosed in quotes, e.g., "Hello". Strings are immutable.
- **Lists (list):** Ordered collections of items, e.g., [1][2][3]. Lists are mutable.
- **Tuples (tuple):** Ordered, immutable collections of items, e.g., (1, 2, 3).
- **Range (range):** A sequence of numbers, e.g., range(1, 5).

**3. Mapping Type**

- **Dictionaries (dict):** Unordered collections of key-value pairs, e.g., {"name": "John", "age": 30}. Dictionaries are mutable.

**4. Set Types**

- **Sets (set):** Unordered collections of unique items, e.g., {"apple", "banana"}. Sets are mutable.
- **Frozensets (frozenset):** Immutable sets, e.g., frozenset({"apple", "banana"}).

**5. Boolean Type**

- **Boolean (bool):** Represents truth values, either True or False.

**6. Binary Types**

- **Bytes (bytes):** Immutable sequences of integers in the range 0 <= x < 256, e.g., b'Hello'.
- **Bytearray (bytearray):** Mutable sequences of integers in the range 0 <= x < 256.
- **Memoryview (memoryview):** A buffer interface for binary data.

**7. None Type**

- **NoneType (None):** Represents the absence of a value.

**Checking Data Types**

You can determine the data type of a variable using the type() function:

```
x = 5

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

**Converting Data Types**

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

```
x = "5"

y = int(x)

print(type(y))  # Output: <class 'int'>
```

**Citations:**

1. https://www.w3schools.com/python/python_datatypes.asp
2. https://www.w3schools.com/python/gloss_python_built-in_data_types.asp

## 3. Loops

**Loops in Python**

Loops are essential control structures in Python that allow you to execute a block of code repeatedly. Python supports three main types of loops: **for loops**, **while loops**, and **nested loops**.

**1. For Loops**

**Purpose:** Used to iterate over sequences like lists, tuples, dictionaries, sets, and strings.

**Syntax:**

```
for iterating_var in sequence:
   statements
```

**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 expression:
    statements
```

**Example:**

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

**3. Nested Loops**

**Purpose:** Use one loop inside another to handle multi-dimensional data.

**Example:**

```
colors = ["red", "green", "blue"]
shapes = ["circle", "square", "triangle"]
for color in colors:
    for shape in shapes:
        print(f"{color} {shape}")
```

**Loop Control Statements**

- **Break:** Terminates the loop.
- **Continue:** Skips the current iteration and moves to the next.
- **Pass:** Used when a statement is syntactically required but no execution is needed.
- **Else:** Executes a block of code when the loop finishes normally (not by a break).

**Additional Features**

- **Range Function:** Used with for loops to iterate over a sequence of numbers.

```
for i in range(5):
    print(i)
```

- **Else Clause in Loops:** Executes when the loop completes without a break.

```
for i in range(5):
    print(i)
else:
    print("Loop finished")
```

**Citations:**

1. https://www.scholarhat.com/tutorial/python/while-for-nested-loop
2. https://www.simplilearn.com/tutorials/python-tutorial/python-loops
3. https://www.w3schools.com/python/python_for_loops.asp

## 4. Conditional statements

Conditional statements in Python are used to control the flow of a program based on conditions or decisions. The main types of conditional statements are **if**, **if-else**, **if-elif-else**, and **nested if** statements.

**1. If Statement**

**Purpose:** Execute a block of code if a condition is true.

**Syntax:**

```
if condition:
  # Execute this code if the condition is True
```

**Example:**

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

**2. If-Else Statement**

- **Purpose:** Execute one block of code if a condition is true and another if it is false.
- **Syntax:**

```
if condition:
  # Execute this code if the condition is True
else:
  # Execute this code if the condition is False
```

- **Example:**

```
num = 5
if num > 10:
  print("Number is greater than 10")
else:
  print("Number is less than or equal to 10")
```

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

- **Purpose:** Check multiple conditions and execute different blocks of code based on which condition is true.
- **Syntax:**

```
if condition1:
  # Execute this code if condition1 is True
elif condition2:
  # Execute this code if condition1 is False and condition2 is True
else:
  # Execute this code if all conditions are False
```

- **Example:**

```
score = 85
if score >= 90:
  print("Grade A")
elif score >= 80:
  print("Grade B")
```

```
else:
    print("Grade C or below")
```

**4. Nested If Statement**

- **Purpose:** Use conditional statements inside other conditional statements.
- **Syntax:**

```
if condition1:
    if condition2:
        # Execute this code if both conditions are True
    else:
        # Execute this code if condition1 is True but condition2 is False
else:
    # Execute this code if condition1 is False
```

- **Example:**

```
num = 15
if num > 10:
    if num % 2 == 0:
        print("Number is greater than 10 and even")
    else:
        print("Number is greater than 10 and odd")
else:
    print("Number is less than or equal to 10")
```

**Logical Operators**

- **And (and):** Both conditions must be true.
- **Or (or):** At least one condition must be true.
- **Not (not):** Inverts the truth value of a condition.

**Example with Logical Operators**

```
age = 25
income = 5000

if age >= 18 and income > 4000:
    print("Eligible for loan")
else:
    print("Not eligible for loan")
```

**Ternary Operator**

**Purpose:** A concise way to write simple if-else statements.

**Syntax:**

```
result = value_if_true if condition else value_if_false
```

- **Example:**

```
num = 5
result = "Even" if num % 2 == 0 else "Odd"
print(result)
```

Conditional statements are fundamental in programming and are used extensively in decision-making processes within Python scripts.

**Citations:**

1. https://www.programiz.com/python-programming/if-elif-else
2. https://www.youtube.com/watch?v=wIXfXYf17ok
3. https://www.w3schools.com/python/python_conditions.asp

## 5. Functions

Functions in Python are blocks of code that perform a specific task. They can take arguments, return values, and be reused throughout a program. Here's an overview of functions in Python:

**Types of Functions**

1. **Built-in Functions:**
   - These are pre-defined functions in Python, such as print(), len(), sum(), etc. They are always available and can be used directly in your code.
2. **User-defined Functions:**
   - These are functions created by users to perform specific tasks. They are defined using the def keyword.
3. **Recursive Functions:**
   - These functions call themselves to solve problems that can be broken down into smaller instances of the same problem.
4. **Lambda Functions:**
   - These are small, anonymous functions defined using the lambda keyword. They are useful for quick, simple operations.
5. **Higher-Order Functions:**
   - These functions take other functions as arguments or return functions as output. Examples include map(), filter(), and reduce().

**Defining a Function**

To define a function in Python, you use the def keyword followed by the function name and parameters in parentheses.

```
def greet(name):
  """Prints a greeting message."""
  print(f"Hello, {name}!")
```

**Calling a Function**

You call a function by its name followed by parentheses containing any required arguments.

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

**Function Arguments**

- **Positional Arguments:** Passed in the order they are defined.
- **Keyword Arguments:** Passed by parameter name, allowing any order.
- **Default Arguments:** Parameters with default values if not provided.
- **Variable-Length Arguments:** *args for positional and kwargs for keyword arguments.

**Returning Values**

Functions can return values using the return statement. If no return statement is provided, the function returns None by default.

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

result = add(3, 5)
print(result)  # Output: 8
```

**Example of a Recursive Function**

```python
def factorial(n):
  if n == 1:
    return 1
  else:
    return n * factorial(n - 1)

print(factorial(5))  # Output: 120
```

**Example of a Lambda Function**

```python
add = lambda x, y: x + y
print(add(3, 5))  # Output: 8
```

Functions are essential for organizing code, improving readability, and enhancing reusability in Python programs.

**Citations:**

1. https://www.w3schools.com/python/python_ref_functions.asp
2. https://www.shiksha.com/online-courses/articles/types-of-functions-in-python/
3. https://www.simplilearn.com/tutorials/python-tutorial/python-functions

# 6. OOP

## Object-Oriented Programming (OOP) in Python

Object-Oriented Programming (OOP) is a programming paradigm that revolves around the concept of objects and classes. Python supports OOP and provides several features to implement it effectively.

### Key Concepts in OOP

1. **Classes:**
   - A blueprint or template that defines the properties and behaviors of an object.
   - Classes are essentially templates for creating objects.
2. **Objects:**
   - Instances of classes.
   - Each object has its own set of attributes (data) and methods (functions).
3. **Inheritance:**
   - A mechanism where one class can inherit the properties and methods of another class.
   - The child class inherits all the attributes and methods of the parent class and can also add new attributes or override the ones inherited from the parent class.
4. **Polymorphism:**
   - The ability of an object to take on multiple forms.
   - This can be achieved through method overriding or method overloading.
5. **Encapsulation:**
   - The concept of bundling data and methods that manipulate that data into a single unit.
   - It helps in hiding the implementation details from the outside world.
6. **Abstraction:**
   - The practice of showing only the necessary information to the outside world while hiding the background details or implementation.

**Implementing OOP in Python**

**Classes and Objects**

```python
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

    def print_details(self):
        print(f"Brand: {self.brand}, Model: {self.model}, Year: {self.year}")

# Creating an object
my_car = Car("Toyota", "Corolla", 2015)
my_car.print_details()
```

**Inheritance**

```python
class Vehicle:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def print_brand(self):
        print(f"Brand: {self.brand}")

class Car(Vehicle):
    def __init__(self, brand, model, year):
        super().__init__(brand, model)
        self.year = year
    def print_details(self):
        self.print_brand()
        print(f"Model: {self.model}, Year: {self.year}")

my_car = Car("Toyota", "Corolla", 2015)
my_car.print_details()
```

**Polymorphism**

```
class Shape:
    def area(self):
        pass

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side ** 2

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14159 * (self.radius ** 2)

square = Square(4)
circle = Circle(5)
```

```
print(square.area())
print(circle.area())
```

**Encapsulation**

```
class BankAccount:
    def __init__(self):
        self.__balance = 0

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited ${amount}. Current balance: ${self.__balance}")
        else:
            print("Invalid deposit amount.")

    def get_balance(self):
        return self.__balance

account = BankAccount()
account.deposit(1000)
print(f"Current balance: ${account.get_balance()}")
```

**Abstraction**

```
python
from abc import ABC, abstractmethod

class AbstractPayment(ABC):
    @abstractmethod
    def make_payment(self, amount):
        pass
```

```
class CreditCardPayment(AbstractPayment):
    def make_payment(self, amount):
        print(f"Paid ${amount} using credit card.")


class PayPalPayment(AbstractPayment):
    def make_payment(self, amount):
        print(f"Paid ${amount} using PayPal.")


payment = CreditCardPayment()
payment.make_payment(100)
```

**Benefits of OOP**

- **Modularity:** Code is organized into modules (classes) that are easier to manage and maintain.
- **Reusability:** Classes can be reused in multiple parts of a program.
- **Easier Debugging:** Issues can be isolated to specific classes or objects.
- **Improved Readability:** Code is more understandable due to the use of meaningful class and method names.

OOP is a powerful approach to programming that helps create robust, maintainable, and scalable software systems.

Source: https://www.geeksforgeeks.org/python-oops-concepts/

## 7. File Handling

File handling in Python allows reading, writing, and manipulating files stored on a system. Python provides built-in functions to work with files using the open() function.

The open() function is used to open a file. It returns a file object that allows interaction with the file.

Syntax:

```
file = open("filename", "mode")
```

The filename parameter specifies the name of the file to be opened. The mode parameter specifies how the file should be opened.

File modes:
"r" - Read mode (default). Opens the file for reading and raises an error if the file does not exist.
"w" - Write mode. Creates a new file if it does not exist and overwrites if it does.
"a" - Append mode. Adds data at the end of the file. Creates a new file if it does not exist.
"x" - Exclusive creation mode. Fails if the file already exists.
"r+" - Read and write mode. File must exist.
"w+" - Write and read mode. Overwrites the existing file.
"a+" - Append and read mode.

Example:
```
file = open("example.txt", "w")
```

Reading a file:
The read() method reads the entire content of the file.
```
file = open("example.txt", "r")
content = file.read()
print(content)
file.close()
```

The readline() method reads one line at a time.
```
file = open("example.txt", "r")
line = file.readline()
print(line)
file.close()
```

The readlines() method reads all lines into a list.
```
file = open("example.txt", "r")
lines = file.readlines()
print(lines)
file.close()
```

Writing to a file:
The write() method writes data to a file and overwrites existing content.
```
file = open("example.txt", "w")
file.write("Hello, World!")
file.close()
```

The writelines() method writes multiple lines from a list.
```
file = open("example.txt", "w")
```

```
file.writelines(["Hello\n", "Python\n"])
file.close()
```

Appending to a file:

The append mode adds new content at the end of an existing file.

```
file = open("example.txt", "a")
file.write("\nAppending new line")7
file.close()
```

Closing a file:

It is important to close a file after use to free system resources.

```
file = open("example.txt", "r")
content = file.read()
file.close()
```

Using the with statement automatically closes the file after execution.

```
with open("example.txt", "r") as file:
content = file.read()
```

File handling exceptions:

To handle errors, such as file not found, use try-except.

```
try:
file = open("non_existent.txt", "r")
content = file.read()
except FileNotFoundError:
print("File not found!")
finally:
file.close()
```

Checking if a file exists:

Before opening a file, check its existence using the os module.

```
import os
if os.path.exists("example.txt"):
print("File exists")
else:
print("File does not exist")
```

Deleting a file:

To delete a file, use the os.remove() method.

```
import os
os.remove("example.txt")
```

Working with directories:

Create a directory using os.mkdir().

```
import os
os.mkdir("new_folder")
```

List files in a directory using os.listdir().

```
import os
print(os.listdir("."))
```

Remove a directory using os.rmdir().

```
import os
os.rmdir("new_folder")
```

Summary:
Use open() to open a file.
Always close files using close() or with open().
Use different modes such as "r", "w", and "a" based on needs.
Handle exceptions with try-except.
Use the os module for file and directory operations.

Citations:

1. https://www.w3schools.com/python/python_file_handling.asp
2. https://www.youtube.com/watch?v=BRrem1k3904

## 8. Exception handling

Exception handling in Python is a crucial feature that allows developers to manage errors and unexpected situations gracefully. The primary components of exception handling in Python are:

**Try and Except Blocks**
The basic structure of exception handling in Python uses try and except blocks:

```
try:
    # Code that might raise an exception
except ExceptionType:
    # Code to handle the exception
```

When an exception occurs in the try block, the program flow immediately moves to the corresponding except block.

**Multiple Except Blocks**
You can handle different types of exceptions separately:

```
try:
    # Code that might raise exceptions
except ZeroDivisionError:
    print("Cannot divide by zero")
except ValueError:
    print("Invalid input")
except:
    print("An unexpected error occurred")
```

Python executes the first matching except clause it encounters.

**Else Clause**

The else clause executes if no exceptions occur in the try block:

```
try:
    result = 5 / 2
except ZeroDivisionError:
    print("Cannot divide by zero")
else:
    print(f"Result: {result}")
```

**Finally Clause**

The finally clause always executes, regardless of whether an exception occurred or not:

```
try:
    file = open("example.txt", "r")
    # File operations
except FileNotFoundError:
    print("File not found")
finally:
    file.close()
```

The finally clause is useful for cleanup operations.

**Raising Exceptions**

You can raise exceptions explicitly using the raise keyword:

```
x = -1
if x < 0:
    raise ValueError("x cannot be negative")
```

**Best Practices**

1. Use specific exception types rather than bare except clauses.
2. Handle exceptions as close to the source as possible.
3. Use the finally clause for cleanup operations.
4. Avoid suppressing exceptions unless you have a good reason.

Exception handling in Python allows for more robust and error-resistant code, improving the overall reliability and user experience of your programs.

**Citations:**

1. https://www.datacamp.com/tutorial/exception-handling-python
2. https://www.freecodecamp.org/news/error-handling-in-python-introduction/
3. https://www.w3schools.com/python/python_try_except.asp

## 9. Numpy and Pandas

NumPy and Pandas are two essential Python libraries used for numerical computing and data analysis.
NumPy (Numerical Python) provides support for multi-dimensional arrays, mathematical functions, and linear algebra operations. Pandas (Python Data Analysis Library) is built on top of NumPy and is used for data manipulation and analysis.
NumPy:
NumPy provides the array object, called ndarray, which allows efficient storage and computation.
Creating a NumPy array:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

Basic array operations:

```
import numpy as np
arr = np.array([1, 2, 3])
print(arr + 2)
print(arr * 3)
print(arr.sum())
print(arr.mean())
```

Array indexing and slicing:

```
import numpy as np
arr = np.array([10, 20, 30, 40, 50])
print(arr[1])
print(arr[1:4])
```

Creating multi-dimensional arrays:

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr)
Reshaping arrays:
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6])
reshaped = arr.reshape(2, 3)
print(reshaped)
```

Mathematical operations:

```
import numpy as np
arr1 = np.array([1, 2, 3])
```

```
arr2 = np.array([4, 5, 6])
print(arr1 + arr2)
print(np.dot(arr1, arr2))
```
Generating random numbers:
```
import numpy as np
rand_arr = np.random.rand(3, 3)
print(rand_arr)
```
Pandas:

Pandas provides two primary data structures: Series and DataFrame.

Creating a Pandas Series:
```
import pandas as pd
s = pd.Series([10, 20, 30, 40])
print(s)
```
Creating a DataFrame:
```
import pandas as pd
data = {"Name": ["Alice", "Bob"], "Age": [25, 30]}
df = pd.DataFrame(data)
print(df)
```
Reading and writing data:
```
import pandas as pd
df = pd.read_csv("data.csv")
df.to_csv("output.csv", index=False)
```
Accessing data in a DataFrame:
```
import pandas as pd
df = pd.DataFrame({"Name": ["Alice", "Bob"], "Age": [25, 30]})
print(df["Name"])
print(df.iloc[0])
```
Filtering data:
```
import pandas as pd
df = pd.DataFrame({"Name": ["Alice", "Bob"], "Age": [25, 30]})
filtered_df = df[df["Age"] > 25]
print(filtered_df)
```
Sorting data:
```
import pandas as pd
df = pd.DataFrame({"Name": ["Alice", "Bob"], "Age": [30, 25]})
sorted_df = df.sort_values("Age")
print(sorted_df)
```
Handling missing values:
```
import pandas as pd
df = pd.DataFrame({"Name": ["Alice", None], "Age": [25, None]})
```

```
df.fillna("Unknown", inplace=True)
print(df)
```

Merging and joining DataFrames:

```
import pandas as pd
df1 = pd.DataFrame({"ID": [1, 2], "Name": ["Alice", "Bob"]})
df2 = pd.DataFrame({"ID": [1, 2], "Score": [90, 85]})
merged_df = pd.merge(df1, df2, on="ID")
print(merged_df)
```

Source: https://www.w3schools.com/python/numpy/numpy_intro.asp
https://www.w3schools.com/python/pandas/pandas_intro.asp

## 10. Selenium for scrapping

Selenium is a Python library used for web scraping and browser automation. Unlike BeautifulSoup, which parses static HTML, Selenium can interact with dynamic web pages that require JavaScript execution.

Installing Selenium:

```
pip install selenium
```

Setting up a WebDriver:
Selenium requires a browser driver such as ChromeDriver for Google Chrome. Download the appropriate driver and provide its path in your code.

```
Example of launching a browser:
from selenium import webdriver
driver = webdriver.Chrome()
driver.get("https://example.com")
```

Locating elements:
Selenium provides multiple methods to find elements on a webpage:

Finding an element by ID:

```
from selenium import webdriver
driver = webdriver.Chrome()
driver.get("https://example.com")
element = driver.find_element("id", "element_id")
print(element.text)
```

Finding elements by class name:

```
from selenium import webdriver
driver = webdriver.Chrome()
driver.get("https://example.com")
elements = driver.find_elements("class name", "class_name")
for element in elements:
print(element.text)
```

Finding elements using XPath:

```
from selenium import webdriver
driver = webdriver.Chrome()
driver.get("https://example.com")
element = driver.find_element("xpath", "//h1")
print(element.text)
```

Interacting with elements:

Clicking a button:

```
from selenium import webdriver
driver = webdriver.Chrome()
driver.get("https://example.com")
button = driver.find_element("id", "submit_button")
button.click()
```

Entering text in an input field:

```
from selenium import webdriver
driver = webdriver.Chrome()
driver.get("https://example.com")
input_field = driver.find_element("name", "username")
input_field.send_keys("test_user")
```

Extracting data:

```
from selenium import webdriver
driver = webdriver.Chrome()
driver.get("https://example.com")
content = driver.find_element("tag name", "p").text
print(content)
```

Handling dynamic content:

Websites that use JavaScript to load content may require waiting mechanisms.

Implicit wait:

```
from selenium import webdriver
driver = webdriver.Chrome()
```

```
driver.implicitly_wait(10)
driver.get("https://example.com")
```

Explicit wait:

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

driver = webdriver.Chrome()
driver.get("https://example.com")
wait = WebDriverWait(driver, 10)
element = wait.until(EC.presence_of_element_located((By.ID, "dynamic_element")))
print(element.text)
```

Handling pop-ups and alerts:

Dismissing an alert:

```
from selenium import webdriver
from selenium.common.exceptions import NoAlertPresentException

driver = webdriver.Chrome()
driver.get("https://example.com")
try:
alert = driver.switch_to.alert
alert.accept()
except NoAlertPresentException:
print("No alert found")
```

Scrolling a webpage:

```
from selenium import webdriver

driver = webdriver.Chrome()
driver.get("https://example.com")
driver.execute_script("window.scrollTo(0, document.body.scrollHeight);")
```

Closing the browser:

driver.close() - Closes the current tab.
driver.quit() - Closes all tabs and ends the session.

Note:
Selenium is useful for scraping JavaScript-heavy websites.
It automates browser interactions like clicking buttons and filling forms.
Waiting mechanisms help handle dynamic content.
Web scraping with Selenium can be combined with BeautifulSoup for efficient parsing.

## 11. Regex

Regular Expressions (Regex) in Python are used for pattern matching and text processing. The re module provides functions to work with regex.

Installing the re module:

The re module is built into Python, so no installation is required.

Importing the re module:

```
import re
```

Basic regex functions:

Searching for a pattern:

```
import re
text = "The price is 100 dollars."
pattern = r"\d+"
match = re.search(pattern, text)
print(match.group())
```

Finding all matches in a string:

```
import re
text = "There are 3 apples and 4 oranges."
pattern = r"\d+"
matches = re.findall(pattern, text)
print(matches)
```

Replacing text using regex:

```
import re
text = "My phone number is 123-456-7890."
pattern = r"\d{3}-\d{3}-\d{4}"
new_text = re.sub(pattern, "XXX-XXX-XXXX", text)
print(new_text)
```

Splitting a string based on a pattern:

```
import re
text = "apple,banana;orange grape"
pattern = r"[;, ]+"
result = re.split(pattern, text)
print(result)
```

Common regex patterns:

\d - Matches any digit (0-9).

\D - Matches any non-digit character.

\w - Matches any word character (letters, digits, underscore).

\W - Matches any non-word character.

\s - Matches any whitespace character (space, tab, newline).

\S - Matches any non-whitespace character.

. - Matches any character except a newline.

^ - Matches the start of a string.
$ - Matches the end of a string.

-
    - Matches 0 or more occurrences of the pattern.
-
    - Matches 1 or more occurrences of the pattern.

      ? - Matches 0 or 1 occurrence of the pattern.
      {n} - Matches exactly n occurrences of the pattern.
      {n,} - Matches n or more occurrences.
      {n,m} - Matches between n and m occurrences.

Example of validating an email address:

```
import re
pattern = r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+.[a-zA-Z]{2,}$"
email = "example@email.com"
if re.match(pattern, email):
print("Valid email")
else:
print("Invalid email")
```

Example of extracting phone numbers:

```
import re
text = "Contact us at 987-654-3210 or 123-456-7890."
pattern = r"\d{3}-\d{3}-\d{4}"
phone_numbers = re.findall(pattern, text)
print(phone_numbers)
```

Note:
Regex is used for pattern matching and text processing.
The re module provides functions like search(), findall(), sub(), and split().
Common patterns include \d for digits, \w for word characters, and . for any character.
Regex is useful for validation, data extraction, and text manipulation.

## 12. Multithreading/multiprocessing

Multithreading and multiprocessing in Python allow executing multiple tasks simultaneously to improve performance.

**Multithreading**

Multithreading allows multiple threads to run concurrently within the same process. It is useful for tasks that involve waiting, such as I/O operations (file handling, network requests). Python threads run in the same memory space but are limited by the Global Interpreter Lock (GIL), meaning they cannot achieve true parallel execution for CPU-intensive tasks.

Example of multithreading:

```
import threading

def print_numbers():
    for i in range(5):
        print(i)

thread = threading.Thread(target=print_numbers)
thread.start()
thread.join()
```

**Multiprocessing**

Multiprocessing creates separate processes that run independently, each with its own memory space. It is ideal for CPU-bound tasks like data processing and calculations, as it bypasses the GIL and allows true parallel execution.

Example of multiprocessing:

```
import multiprocessing

def print_numbers():
    for i in range(5):
        print(i)

process = multiprocessing.Process(target=print_numbers)
process.start()
process.join()
```

**Concurrency vs. Parallelism**

- **Concurrency** means multiple tasks are managed at the same time but not necessarily executed simultaneously. It is achieved using multithreading where tasks take turns using the CPU.
- **Parallelism** means multiple tasks run simultaneously on multiple CPU cores. It is achieved using multiprocessing.

Example:

- A web server handling multiple requests at the same time is concurrency.
- A data processing task running on multiple cores simultaneously is parallelism.

**Note:**

- **Multithreading** is useful for I/O-bound tasks but limited by the GIL.
- **Multiprocessing** is better for CPU-bound tasks as it allows true parallel execution.
- **Concurrency** handles multiple tasks at once, while **parallelism** executes multiple tasks simultaneously.

## 13. SDLC

Software Development Life Cycle (SDLC) is a structured process used for planning, creating, testing, and deploying software applications. It ensures high-quality software is developed efficiently.

**Phases of SDLC**

1. **Requirement Analysis**
    - Understand and document software requirements from stakeholders.
    - Identify business needs, user expectations, and constraints.
2. **Planning**
    - Define project scope, budget, timeline, and risks.
    - Create a development roadmap and assign resources.
3. **Design**
    - Create system architecture, database design, and user interface mockups.
    - Define technologies, frameworks, and tools for development.
4. **Development (Implementation)**
    - Write code based on the design specifications.
    - Developers follow coding standards and best practices.
5. **Testing**
    - Identify and fix bugs using unit, integration, and system testing.
    - Ensure the software meets functional and performance requirements.
6. **Deployment**
    - Release the software to users or clients.
    - Deploy in a staging or production environment.
7. **Maintenance**
    - Provide updates, bug fixes, and performance improvements.
    - Handle user feedback and ensure smooth operation.
    -

**Agile Model in SDLC**

The **Agile Model** is a software development approach that focuses on iterative development, collaboration, and flexibility. Unlike traditional models like Waterfall, Agile allows for continuous feedback and improvements throughout the development process.

**Key Principles of Agile (as per Agile Manifesto)**

1. **Individuals and interactions over processes and tools.**
2. **Working software over comprehensive documentation.**
3. **Customer collaboration over contract negotiation.**
4. **Responding to change over following a plan.**

**Phases of Agile Model**

1. **Concept & Requirement Gathering**
   - Identify high-level project goals and user needs.
   - Define the initial product backlog (list of features).
2. **Planning**
   - Break work into small deliverable units called **sprints** (typically 2-4 weeks).
   - Assign tasks and define sprint goals.
3. **Design & Development**
   - Implement features incrementally.
   - Frequent collaboration with stakeholders.
4. **Testing**
   - Continuous testing during and after development.
   - Automated and manual testing to ensure quality.
5. **Deployment & Release**
   - Deliver functional software at the end of each sprint.
   - Gather feedback and make necessary improvements.
6. **Review & Feedback**
   - Conduct sprint reviews and retrospectives.
   - Adapt and improve based on user feedback.

**Agile Frameworks**

1. **Scrum**
    - Uses short development cycles (sprints).
    - Daily stand-up meetings and sprint planning.
    - Roles: Product Owner, Scrum Master, Development Team.
2. **Kanban**
    - Focuses on visualizing work using a Kanban board.
    - Continuous delivery without fixed sprints.
3. **Extreme Programming (XP)**
    - Emphasizes frequent releases, test-driven development (TDD), and pair programming.
4. **Lean Development**
    - Minimizes waste and maximizes efficiency.
5. **Scaled Agile Framework (SAFe)**
    - Adapts Agile for large organizations with multiple teams.

**Advantages of Agile**

- Faster delivery of functional software.
- High adaptability to changing requirements.
- Continuous feedback improves product quality.
- Encourages collaboration and customer satisfaction.

**Disadvantages of Agile**

- Requires strong communication and team coordination.
- Less effective for projects with fixed scope and deadlines.
- Frequent changes can lead to scope creep.

**Note:**

- Agile is an iterative and flexible software development model.
- It emphasizes customer collaboration, continuous improvement, and adaptability.
- Frameworks like Scrum, Kanban, and XP support Agile development.
- Agile is best suited for dynamic projects with evolving requirements.

## 14. Agile and Scrum

**Agile** is a software development methodology that promotes **iterative development, flexibility, and continuous feedback**. It focuses on delivering small, working increments of software rather than a complete product at once.

**Scrum** is a framework within Agile that organizes work into **sprints** (typically 2-4 weeks). It involves roles like **Product Owner, Scrum Master, and Development Team**.

**Scrum Process**:

1. **Sprint Planning** – Define goals for the sprint.
2. **Daily Stand-ups** – Short meetings to track progress.
3. **Sprint Review** – Demonstrate completed work.
4. **Sprint Retrospective** – Identify improvements for the next sprint.

## 15. Code version control

Code version control is a **system that tracks changes in code** over time, allowing multiple developers to work on a project efficiently.

**Git** is the most widely used version control system.

**Key Git Commands**:

- git init – Initialize a repository.
- git clone – Copy an existing repository.
- git add – Stage changes for commit.
- git commit -m "message" – Save changes with a message.
- git push – Upload changes to a remote repository.
- git pull – Fetch and merge updates from a remote repository.

Git commit types: https://www.linkedin.com/pulse/commit-types-git-mushroomsoft-it-etyae

## 16. Doc

Documentation refers to **written records** that describe a project's functionality, design, and usage.

**Types of Documentation**:

1. **Code Documentation** – Docstrings, comments, API references.
2. **User Documentation** – Manuals, guides, FAQs.
3. **Technical Documentation** – System architecture, requirements, and workflows.

**Tools**: Markdown, Sphinx, Doxygen, Javadoc.

## 17. Risk management

Risk management in software development involves **identifying, assessing, and mitigating risks** that can impact the project.
**Types of Risks**:
1. **Technical Risks** – Bugs, outdated technology.
2. **Operational Risks** – Lack of resources, delays.
3. **Security Risks** – Data breaches, cyber threats.
4. **Business Risks** – Budget overruns, market changes.

**Risk Management Process**:
1. Identify risks.
2. Analyze their impact.
3. Prioritize and plan mitigation strategies.
4. Monitor and update risk plans regularly.

## 18. Python coding standards and pep8

**Python Coding Standards and PEP 8**
Python coding standards ensure that code is readable, maintainable, and follows best practices. **PEP 8** (Python Enhancement Proposal 8) is the official style guide for writing Python code.
**1. Indentation**
Use **4 spaces per indentation level**. Do not use tabs.
Example:

```
def example():
    print("Follow indentation rules")
```

**2. Line Length**

Keep lines of code within **79 characters**. For docstrings and comments, use **72 characters**. Use line continuation (\) or parentheses for long lines.

Example:

```
long_variable = ("This is a very long line that should be wrapped "
                 "using parentheses.")
```

**3. Blank Lines**

- Use **two blank lines** before defining a function or class.
- Use **one blank line** between functions inside a class.

  Example:

  ```
  class Example:

      def method_one(self):
          pass

      def method_two(self):
          pass
  ```

**4. Imports**

- Import **one module per line**.
- Use **absolute imports** whenever possible.
- Place imports in the order:
  1. Standard library imports
  2. Third-party library imports
  3. Local application imports

     Example:

     ```
     import os
     import sys

     import numpy as np
     import pandas as pd
     ```

from mymodule import myfunction

**5. Naming Conventions**

- Variables and functions: **lowercase_with_underscores**
- Constants: **UPPERCASE_WITH_UNDERSCORES**
- Class names: **CamelCase**
- Private methods: **_single_leading_underscore**

  Example:

  ```
  class MyClass:
      def __init__(self):
  ```

```
        self.variable_name = 10

    def _private_method(self):
        pass
```

**6. Whitespace Usage**

- Avoid extra spaces inside parentheses, brackets, or braces.
- Use **a single space** around operators and after commas.

Example:

```
correct = (1, 2, 3)
incorrect = ( 1, 2, 3 )

x = 10 + 5  # Correct
y=10+5      # Incorrect
```

**7. Comments and Docstrings**

- Use **# for inline comments** and place them **above the code** they describe.
- Use **""" triple quotes """** for multi-line docstrings.

Example:

```
def add_numbers(a, b):
    """Returns the sum of two numbers."""
    return a + b
```

**8. Function and Class Definitions**

- Keep function definitions concise.
- Use **type hints** when possible.

Example:

```
def greet(name: str) -> str:
    return f"Hello, {name}"
```

**9. Avoid Unnecessary Complexity**

- Use **list comprehensions** instead of loops when appropriate.
- Use **f-strings** for string formatting.

Example:

```
squares = [x**2 for x in range(10)]
print(f"Squares: {squares}")
```

**10. Exception Handling**

- Use try-except blocks for error handling.
- Handle specific exceptions instead of using a generic except.

Example:

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

**Summary**

- Follow **PEP 8** for consistent coding style.
- Use **4 spaces** for indentation.
- Keep **lines under 79 characters**.
- Follow **naming conventions** and maintain **proper spacing**.
- Write **clear comments and docstrings**.
- Use **efficient coding practices** for readability and performance.

These guidelines help in writing **clean, readable, and maintainable** Python code.

## 19. Error handling and logging

Error handling ensures a program continues running smoothly despite runtime errors. Python uses try-except blocks for handling exceptions. Logging helps track issues and debugging.

**Example of Error Handling:**

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

Logging Example:

```
import logging
logging.basicConfig(level=logging.INFO)
logging.info("This is an info message")
```

## 20. Efficient code

Efficient code optimizes execution time, memory usage, and readability. Best practices include:

- Using **list comprehensions** instead of loops.
- **Avoiding unnecessary variables** and redundant computations.

- Using **efficient data structures** (e.g., set instead of list for lookups).
- Implementing **lazy loading** and **generators** where needed.

**Example:**

```
squares = [x**2 for x in range(10)]  # Faster than using a for-loop
```

## 21. Various Principles

**DRY (Don't Repeat Yourself)** – Avoid duplicate code.
**KISS (Keep It Simple, Stupid)** – Write simple, readable code.
**YAGNI (You Ain't Gonna Need It)** – Don't add unnecessary features.
**Separation of Concerns** – Keep functions and modules focused.
**Test-Driven Development (TDD)** – Write tests before code.

## 22. Unit Testing Validation

Unit testing verifies individual code units work as expected. Python's unittest module provides built-in support.

**Pytest** is a popular Python testing framework that simplifies unit testing with minimal boilerplate code. It automatically discovers test functions and provides useful assertion error messages.

**Installing Pytest**

To install pytest, run:

```
pip install pytest
```

**Writing a Test with Pytest**

A test function should start with test_ and use assert statements.

**Example:**

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


def test_add():
    assert add(2, 3) == 5
    assert add(-1, 1) == 0
    assert add(0, 0) == 0
```

**Running Tests**

Save the test file (e.g., test_math.py) and run:

```
pytest
```

**Using Fixtures for Setup and Teardown**

Fixtures help initialize test data or resources before running tests.

**Example:**

```
import pytest


@pytest.fixture
def sample_data():
    return {"a": 2, "b": 3}


def test_add(sample_data):
    assert add(sample_data["a"], sample_data["b"]) == 5
```

**Parameterized Testing**

To test multiple inputs efficiently, use @pytest.mark.parametrize.

**Example:**

```
import pytest


@pytest.mark.parametrize("a, b, expected", [(2, 3, 5), (-1, 1, 0), (0, 0, 0)])
def test_add(a, b, expected):
    assert add(a, b) == expected
```

**Handling Exceptions in Tests**

Use pytest.raises to test for expected exceptions.

**Example:**

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


def test_divide():
    with pytest.raises(ValueError):
        divide(10, 0)
```

## 23. Ruff and Black

**Ruff** – A fast Python linter that checks for code style, errors, and performance issues.
**Use**:

```
ruff check my_script.py
```

**Black** – A Python code formatter that enforces PEP 8 style.
**Use**: `black my_script.py`

## Extra-

## List and Dictionary Operations:

**List and Dictionary Operations in Python**

**1. List Operations**

A **list** is an ordered, mutable collection that allows duplicate values.

**Creating a List**

```
my_list = [1, 2, 3, 4, 5]
```

**Accessing Elements**

```
print(my_list[0])   # First element
print(my_list[-1])  # Last element
```

**Modifying Elements**

```
my_list[1] = 10  # Change second element
```

**Adding Elements**

```
my_list.append(6)        # Adds to the end
my_list.insert(2, 15)     # Inserts 15 at index 2
```

**Removing Elements**

```
my_list.remove(10)  # Removes first occurrence of 10
popped = my_list.pop(2)  # Removes and returns element at index 2
del my_list[0]  # Deletes first element
```

**Slicing a List**

```
print(my_list[1:4])  # Elements from index 1 to 3
print(my_list[:3])   # First 3 elements
print(my_list[-3:])  # Last 3 elements
```

**List Comprehension**

```
squared = [x**2 for x in my_list]
```

**Sorting a List**

```
my_list.sort()       # Sorts in ascending order
my_list.sort(reverse=True)  # Sorts in descending order
```

**Finding Elements**

```
print(3 in my_list)  # Check if 3 exists in the list
print(my_list.index(4))  # Get index of 4
```

**Iterating Through a List**

```
for item in my_list:
    print(item)
```

**List Length**

```
print(len(my_list))
```

**2. Dictionary Operations**

A **dictionary** is an unordered collection of key-value pairs, where keys are unique.

**Creating a Dictionary**

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}
```

**Accessing Values**

```
print(my_dict["name"])  # Output: Alice

print(my_dict.get("age"))  # Output: 25
```

**Adding and Updating Keys**

```
my_dict["job"] = "Engineer"  # Adding new key

my_dict["age"] = 26  # Updating existing key
```

**Removing Elements**

```
del my_dict["city"]  # Remove a key-value pair

popped_value = my_dict.pop("age")  # Remove and return value
```

**Iterating Through a Dictionary**

```
for key, value in my_dict.items():

    print(key, value)
```

**Checking if a Key Exists**

```
print("name" in my_dict)  # Output: True
```

**Getting All Keys and Values**

```
keys = my_dict.keys()  # Get all keys

values = my_dict.values()  # Get all values
```

**Merging Dictionaries**

```
new_dict = {"country": "USA", "salary": 50000}

my_dict.update(new_dict)
```

**Dictionary Length**

```
print(len(my_dict))
```

**Dictionary Comprehension**

```
squared_numbers = {x: x**2 for x in range(5)}
```

**Note:**

- **Lists** store ordered, mutable elements with duplicates.

- **Dictionaries** store key-value pairs with unique keys.
- Use append(), insert(), remove(), and slicing for list operations.
- Use keys(), values(), update(), and pop() for dictionary operations.
- List and dictionary comprehensions provide efficient ways to process data.

## BeautifulSoup Vs Selenium

**BeautifulSoup**

- A Python library used for **parsing and extracting data** from HTML and XML.
- Works well with **static** web pages where content is already loaded.
- Relies on **requests** or urllib to fetch the webpage.
- Faster than Selenium since it does not interact with a browser.
- Does not handle JavaScript-rendered content.

**Example:**

```
from bs4 import BeautifulSoup

import requests


url = "https://example.com"

response = requests.get(url)

soup = BeautifulSoup(response.text, "html.parser")


title = soup.find("title").text

print(title)
```

**Selenium**

- A web automation tool that can **interact with web pages dynamically**.
- Suitable for scraping **JavaScript-heavy** websites where content loads dynamically.
- Uses a real browser (Chrome, Firefox, etc.), making it **slower** than BeautifulSoup.
- Allows actions like clicking, scrolling, and form submission.

**Example:**

```
from selenium import webdriver


driver = webdriver.Chrome()

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


title = driver.title

print(title)


driver.quit()
```

**Conclusion**

- Use **BeautifulSoup** when working with simple, static pages for fast and efficient scraping.
- Use **Selenium** when dealing with dynamic pages that require interaction or JavaScript execution.
- Sometimes, both can be **combined**—using Selenium to load the page and BeautifulSoup to parse the content.

## Encapsulation:

Encapsulation is an Object-Oriented Programming (OOP) principle that restricts direct access to data and methods, allowing controlled access through public methods. It helps in **data hiding** and **security** by preventing unintended modifications.

**How Encapsulation Works in Python?**

Encapsulation in Python is implemented using **access modifiers**:

1. **Public Members** – Accessible from anywhere.
2. **Protected Members (_var)** – Indicated by a single underscore, meant for internal use but still accessible.
3. **Private Members (__var)** – Indicated by double underscores, not directly accessible outside the class.

**Example of Encapsulation in Python**

```
class BankAccount:
    def __init__(self, account_number, balance):
        self.account_number = account_number  # Public variable
        self._bank_name = "ABC Bank"  # Protected variable
        self.__balance = balance  # Private variable

    def deposit(self, amount):
        """Public method to deposit money"""
        self.__balance += amount

    def get_balance(self):
        """Public method to access private balance"""
        return self.__balance

# Creating an object
account = BankAccount("123456", 1000)

# Accessing public member
print(account.account_number)  # Allowed

# Accessing protected member (not recommended)
print(account._bank_name)  # Allowed but should be avoided

# Accessing private member (will raise an error)
# print(account.__balance)  # AttributeError

# Using public method to access private data
print(account.get_balance())  # Allowed: Outputs 1000
```

**Key Points**

- Private variables can be accessed using **getter/setter methods** to maintain control.
- Attempting to access a private variable directly will result in an AttributeError.
- Encapsulation ensures **data security, integrity, and controlled access** to class properties.

## XPath in Web Scraping and Automation

**XPath (XML Path Language)** is a query language used to navigate and locate elements in an **XML or HTML document**. It is widely used in **web scraping** and **automated testing** with Selenium.

**Types of XPath**

1. **Absolute XPath**: Starts from the root (/html/...) and follows a direct path.
    - Example: /html/body/div/h1
    - **Cons:** Breaks if the structure changes.
2. **Relative XPath**: Starts from anywhere in the document using //, making it flexible.
    - Example: //h1 (Finds all <h1> elements)

**Common XPath Expressions**

- //tagname[@attribute='value'] → Selects an element by attribute.
    - Example: //input[@id='username']
- //tagname[text()='text_value'] → Finds an element by text.
    - Example: //button[text()='Login']
- //tagname[contains(@attribute, 'value')] → Finds elements containing partial text.
    - Example: //div[contains(@class, 'header')]

**Using XPath in Selenium (Example)**

```
from selenium import webdriver


driver = webdriver.Chrome()

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

```
# Using XPath to locate an element
element = driver.find_element("xpath", "//input[@id='search']")
element.send_keys("XPath Example")

driver.quit()
```

**Advantages of XPath**

- Works for both **HTML and XML**.
- Can **traverse elements** in any direction (parent, child, sibling).
- More **precise** than CSS selectors in some cases.

**Disadvantages**

- **Slower** compared to CSS selectors.
- Can break if the **DOM structure changes frequently**.

**Conclusion:** XPath is a powerful tool for locating elements but should be used carefully for **dynamic** websites where CSS selectors may be a better choice.

## REST API in Python

A **REST API (Representational State Transfer API)** allows communication between different systems using HTTP methods such as **GET, POST, PUT, DELETE**. In Python, REST APIs can be built using frameworks like **Flask** and **FastAPI**.

**Key HTTP Methods in REST API**

1. **GET** – Retrieve data.
2. **POST** – Send data to create a resource.
3. **PUT** – Update an existing resource.
4. **DELETE** – Remove a resource.

**Creating a Simple REST API using Flask**

```
from flask import Flask, jsonify, request

app = Flask(__name__)
```

```
# Sample data
data = {"message": "Hello, World!"}

# GET request
@app.route("/api", methods=["GET"])
def get_data():
    return jsonify(data)

# POST request
@app.route("/api", methods=["POST"])
def update_data():
    new_data = request.json
    data.update(new_data)
    return jsonify({"message": "Data updated", "data": data})

if __name__ == "__main__":
    app.run(debug=True)
```

- **jsonify()** converts Python dictionaries into JSON format.
- **request.json** retrieves incoming JSON data.
- The API runs on http://127.0.0.1:5000/api.

**Consuming REST API with requests Library**

```
import requests

response = requests.get("http://127.0.0.1:5000/api")
print(response.json())  # Output: {'message': 'Hello, World!'}
```

**Building REST API with FastAPI (Asynchronous & Fast)**

```
from fastapi import FastAPI
```

```
app = FastAPI()


@app.get("/api")
async def read_data():
    return {"message": "Hello, World!"}
```

Run with uvicorn:

```
uvicorn filename:app --reload
```

**Advantages of REST API in Python**

- **Lightweight and scalable**
- **Supports multiple data formats (JSON, XML, etc.)**
- **Can be easily tested and consumed**

**Conclusion:** REST APIs in Python are commonly built using **Flask** for simplicity and **FastAPI** for speed and async support.

## REST API vs SOAP vs GraphQL

**1. REST API (Representational State Transfer)**

- Uses **HTTP methods** like GET, POST, PUT, DELETE.
- Data is exchanged in **JSON** or **XML** formats.
- **Stateless**: Each request is independent, and the server does not store client data.
- **Simple, scalable, and widely used** in web services.

Example:

```
GET /users/1
```

Response:

```
{"id": 1, "name": "John"}
```

**2. SOAP (Simple Object Access Protocol)**

- Uses **XML format** for messaging.
- **Strict protocol with built-in security (WS-Security)**.
- **Stateful or Stateless**, depending on implementation.
- **Slower and more complex** due to XML parsing and strict rules.

Example SOAP request (XML):

```
<Envelope>
  <Body>
    <GetUser>
      <id>1</id>
    </GetUser>
  </Body>
</Envelope>
```

**3. GraphQL**

- **Query language** for APIs, developed by Facebook.
- Clients can request **specific data fields** instead of receiving a full response.
- **More efficient** than REST for complex queries.

Example GraphQL Query:

```
{
 user(id: 1) {
  name
  email
 }
}
```

Response:

```
{"user": {"name": "John", "email": "john@example.com"}}
```

**Comparison**

| **Feature** | **REST API** | **SOAP** | **GraphQL** |
|---|---|---|---|
| Format | JSON/XML | XML | JSON |
| Flexibility | Medium | Low | High |
| Performance | Fast | Slow | Faster for complex queries |
| Security | Moderate | High (WS-Security) | Moderate |
| Use Case | Web & mobile apps | Enterprise & banking | Complex queries & modern APIs |

**Conclusion**

- **REST** is simple, widely used, and flexible.
- **SOAP** is more secure but complex and slower.
- **GraphQL** provides precise queries and efficiency for large-scale applications.

## What is Stateless?

**Stateless** means that a system **does not store client-specific data** between requests. Each request is treated as **independent** and contains all the necessary information for the server to process it.

**Example of Stateless System (REST API)**

A client sends a request:

GET /user/1

The server responds with:

{"id": 1, "name": "John"}

- The server **does not remember** previous interactions.
- If the client makes another request, it must send all required details again.

**Characteristics of Stateless Systems**

1. **No session data is stored on the server** (each request is independent).

2. **Scalable and reliable** (easy to distribute across multiple servers).
3. **Simpler implementation** (no need for session management).

**Example of a Stateful System (Opposite of Stateless)**

- A **banking application** where login credentials are stored for the session.
- A **shopping cart** where the server remembers selected items.

**Conclusion**

- **Stateless systems** are used in **REST APIs, cloud computing, and microservices** for scalability.
- **Stateful systems** are used when **user sessions and history** need to be maintained, like in banking or e-commerce.

## What is CORS Policy?

**CORS (Cross-Origin Resource Sharing)** is a security feature in web browsers that restricts web pages from making requests to a **different domain** than the one that served the web page. This prevents unauthorized access to resources from different origins.

**Why is CORS Needed?**

By default, browsers follow the **Same-Origin Policy (SOP)**, which blocks cross-origin requests to protect users from malicious websites. CORS allows controlled access to resources from different domains while maintaining security.

**How CORS Works?**

When a browser makes a cross-origin request, the server must send the correct **CORS headers** in the response to allow or block access.

**Example of CORS Headers**

A response from the server must include:

```
Access-Control-Allow-Origin: https://example.com
```

This allows requests **only** from https://example.com.

To allow requests from any domain:

```
Access-Control-Allow-Origin: *
```

**Common CORS Errors and Fixes**

1. **Error:**

   ```
   Access to fetch at 'https://api.example.com' from origin 'http://localhost:3000' has been blocked by CORS policy.
   ```

**Fix:** The server must include the correct Access-Control-Allow-Origin header.

2. **Allow Multiple Methods:**

   ```
   Access-Control-Allow-Methods: GET, POST, PUT, DELETE
   ```

3. **Allow Custom Headers:**

   ```
   Access-Control-Allow-Headers: Content-Type, Authorization
   ```

**Conclusion**

CORS is an essential security feature in browsers that prevents unauthorized cross-origin requests while allowing **legitimate** interactions between different domains through proper server configurations.

## Proxy and Endpoints

**1. What is a Proxy?**

A **proxy** is an intermediary server that sits between a client and a destination server. It forwards requests from clients to the server and sends the responses back to the client.

**Types of Proxies:**

- **Forward Proxy**: Acts on behalf of the client, masking the client's IP.
- **Reverse Proxy**: Sits in front of the server, handling client requests for security, caching, or load balancing.

**Example of a Proxy in Python (Using requests module)**

```
proxies = {
   "http": "http://your-proxy.com:8080",
```

```
    "https": "https://your-proxy.com:8080",
}
response = requests.get("http://example.com", proxies=proxies)
print(response.text)
```

**2. What is an Endpoint?**

An **endpoint** is a specific URL where an API or web service provides access to resources. Each endpoint corresponds to a specific function, like fetching data, updating a record, or deleting a resource.

**Example of API Endpoints:**

- https://api.example.com/users → Fetch all users
- https://api.example.com/users/1 → Fetch user with ID 1
- https://api.example.com/users/1/orders → Fetch orders for user 1

**Example of Using an Endpoint in Python (requests module)**

```
import requests

url = "https://api.example.com/users"
response = requests.get(url)

print(response.json())  # Prints user data
```

**Key Differences: Proxy vs Endpoint**

| **Feature** | **Proxy** | **Endpoint** |
|---|---|---|
| Purpose | Intermediary between client and server | Specific URL for accessing an API |
| Usage | Improves security, load balancing, caching | Provides data or services via an API |
| Example | http://proxyserver.com:8080 | https://api.example.com/users |

**Conclusion:**

- **Proxies** enhance security and manage traffic between clients and servers.

- **Endpoints** define the access points for interacting with an API.

## Rate Limiting and Pagination for Rate Limiting:

**1. What is Rate Limiting?**

Rate limiting controls the number of requests a client can make to a server within a specific time frame. It helps prevent abuse, protect APIs, and ensure fair resource usage.

**Example of Rate Limiting Rules:**

- **100 requests per minute per user**
- **10 requests per second per IP address**

**How Rate Limiting Works?**

1. **Client makes a request.**
2. **Server checks the request count for the time window.**
3. **If the limit is exceeded, the request is rejected with a 429 (Too Many Requests) response.**
4. **If under the limit, the request is processed.**

**Example of Rate Limit Headers in an API Response:**

```
X-RateLimit-Limit: 100  # Maximum requests allowed

X-RateLimit-Remaining: 10  # Remaining requests in the time window

X-RateLimit-Reset: 60  # Time (in seconds) until the limit resets
```

**Rate Limiting in Flask Using Flask-Limiter:**

```
from flask import Flask

from flask_limiter import Limiter


app = Flask(__name__)

limiter = Limiter(app, key_func=lambda: "user")


@app.route("/data")
```

```
@limiter.limit("5 per minute")  # 5 requests per minute
def get_data():
    return "This is a rate-limited API."

if __name__ == "__main__":
    app.run()
```

**2. Pagination for Rate Limiting**

Pagination is used to limit the number of records returned per request to reduce server load and avoid excessive data retrieval.

**Types of Pagination:**

1. **Offset-based Pagination**
   - Uses limit and offset parameters.
   - Example:
   - GET /users?limit=10&offset=20
   - Retrieves 10 users, skipping the first 20.
2. **Cursor-based Pagination**
   - Uses a cursor (e.g., timestamp or ID) instead of offset.
   - Example:
   - GET /users?cursor=abcd1234&limit=10
   - More efficient for large datasets.

**Example of Pagination in Python (Using requests):**

```
import requests

url = "https://api.example.com/users?limit=10&offset=0"
response = requests.get(url)

print(response.json())  # Prints paginated data
```

**Benefits of Pagination for Rate Limiting:**

- Reduces server load by limiting response size.
- Prevents excessive requests and API abuse.
- Improves performance and response time.

**Conclusion:**

- **Rate Limiting** controls request frequency to prevent abuse.
- **Pagination** optimizes data retrieval by breaking large results into smaller chunks.
- Both techniques help maintain **API efficiency and performance**.