Interactive CARES

পাইথন সহজে শেখার হ্যান্ডবুক

- ওয়েব ডেভেলপমেন্ট
- মেশিন লার্নিং
- ডাটা সায়েন্স

পাইথনের শুরু ভেরিয়েবল কমেন্টস এনভায়রনমেন্ট

Introduction to Python

Overview of Python:

- Python is a high-level, interpreted programming language known for its simplicity and readability.
- It was developed by Guido van Rossum and first released in 1991.
- Python emphasizes code readability with its notable use of significant whitespace.

Why Learn Python?

- **Simplicity:** Easy to read and write.
- **Versatility:** Used in web development, data analysis, AI, scientific computing, and more.
- **Large Community:** Extensive libraries and frameworks available.
- **Career Opportunities:** High demand in various industries.

Real-world Applications:

- Web Development (e.g., Django, Flask)
- Data Science and Machine Learning (e.g., Pandas, scikit-learn)
- Automation and Scripting
- Game Development (e.g., Pygame)
- Embedded Systems

Installing Python

- **Step-by-Step Installation:**
  - **Windows:**
    1. Download the installer from the official Python website.
    2. Run the installer and check the box to add Python to your PATH.
    3. Click “Install Now” and follow the prompts.
  - **macOS:**
    1. Download the installer from the Python website.
    2. Open the .pkg file and follow the instructions.
    3. Verify installation by opening the terminal and typing python3 --version .
  - **Linux:**
    1. Open your terminal.
    2. Update your package list: sudo apt update .
    3. Install Python 3: sudo apt install python3 .
- **Verifying Installation:**
  - Open a terminal or command prompt.
  - Type python --version or python3 --version to check the installed version.

Install Pycharm

PyCharm is a popular Integrated Development Environment (IDE) for Python development. Here's a step-by-step guide to installing PyCharm on your computer:

Step 1: Download PyCharm

1. Go to the official PyCharm website: JetBrains PyCharm
2. You will see two versions: Professional and Community. The Community edition is free and open-source, while the Professional edition offers more features but requires a license. Choose the version that suits your needs and click the "Download" button.

Step 2: Install PyCharm

For Windows:

1. Once the download is complete, open the installer (pycharm-community-*.exe for the Community edition).
2. Follow the installation wizard:
   - Click "Next" to continue.
   - Choose the installation location and click "Next."
   - Select the installation options you prefer, such as creating a desktop shortcut or associating .py files with PyCharm.
   - Click "Install" to begin the installation process.
3. After the installation is complete, click "Finish" to exit the installer. You can choose to run PyCharm immediately if you wish.

Writing and Running Your First Python Program

Hello, World! Program

```
print(”Hello, World!”)
```

Running the Program:

- Save the code in a file named hello.py .
- Open a terminal or command prompt and navigate to the directory containing hello.py .
- Run the script by typing python hello.py or python3 hello.py .

Using the Python Interactive Shell:

- Open a terminal or command prompt.
- Type python or python3 to enter the interactive shell.
- Type the code directly:

```
print(”Hello, World!”)
```

Understanding How Python Code Works

To understand how Python code works, we'll look at a simple example and explain each step involved in its execution.

Example: Greeting Program

```
# greeting.py

# Step 1: Get the user's name
name = input("Enter your name: ")
# Step 2: Print a personalized greeting
print("Hello, " + name + "!")
```

Steps Involved:

For Windows:

1. Reading the Source Code:
   - The Python interpreter reads the source code from the file greeting.py .
2. Bytecode Compilation:
   - The source code is translated into bytecode by the interpreter.
   - Bytecode is a set of instructions that can be executed by the Python Virtual Machine (PVM).
3. Execution by PVM:
   - The PVM executes the bytecode instructions line-by-line.

Understanding Code Execution & Introduce with debugging

- Debugging goes beyond finding bugs; it's crucial from development to production and understanding code.
- It allows you to see what's happening at each line, making it easier to understand complex logic step-by-step.
- Small mistakes causing many errors can be quickly identified and fixed through debugging.
- Debugging helps break down and test large functions incrementally, avoiding the need to write and test all at once.
- It's useful for understanding other people's code, especially in varied coding styles and unfamiliar projects.
- Debugging improves testing, performance, and code quality across multiple languages, not just Python, including
- JavaScript, Java, and C#

```
# Calculate the area of a rectangle
length = 5 # Length of the rectangle
width = 3 # Width of the rectangle
area = length * width # Area formula: length * width
print("Area:", area)
```

Python Comments

Single-line Comments: Use the # symbol.

```
# This is a single-line comment
```

Multi-line Comments: Enclose comments in triple quotes.

```
"""
This is a multi-line comment
that spans multiple lines.
"""
```

Best Practices:

- Write clear and concise comments.
- Use comments to explain the purpose of the code, not obvious details.

```
# Calculate the area of a rectangle
length = 5 # Length of the rectangle
width = 3 # Width of the rectangle
area = length * width # Area formula: length * width
print("Area:", area)
```

Python Variables

Definition:

- Variables store data values.
- Python is dynamically typed, so you don't need to declare a variable type explicitly.

Assigning Values:

```
x = 5
name = "Alice"
is_student = True
```

Naming Conventions:

- Descriptive Names: Use meaningful and descriptive names to make your code self-explanatory. For example, use total_cost instead of tc .
- Lowercase with Underscores: Variable names should be written in lowercase letters and words should be separated by underscores for readability. For example, student_name instead of studentName .
- Avoid Reserved Words: Do not use Python reserved keywords as variable names, such as class , for , if , etc.
- Start with a Letter or Underscore: Variable names must start with a letter (a-z, A-Z) or an underscore (_). They cannot start with a number.

- No Special Characters: Variable names should only contain letters, numbers, and underscores. Avoid using special characters like ! , @ , # , etc.
- Case Sensitivity: Remember that variable names are case-sensitive. For example, myVariable and myvariable are two different variables.
- Short but Meaningful: While being descriptive, try to keep variable names reasonably short. For example, num_students is better than number_of_students_in_the_class .
- Use Singular Nouns: Use singular nouns for variables that hold a single value, and plural nouns for variables that hold collections. For example, student for a single student, and students for a list of students.
- Consistency: Be consistent with your naming conventions throughout your code to maintain readability and ease of understanding.
- Avoid Double Underscores: Do not use double underscores at the beginning and end of variable names, as these are reserved for special use in Python (e.g., __init__ , __main__).

Basic Operations:

```
a = 10
b = 20
sum = a + b
print(sum) # Output: 30
```

ডেটা টাইপ মিউটেবল ইমিউটেবল ও কনভার্সন

- **Data Types in Python**

Numeric Types

- int: Integer numbers, e.g., 5 , -3 , 42 .
- float: Floating-point numbers, e.g., 3.14 , -0.001 , 2.0 .
- complex: Complex numbers with real and imaginary parts, e.g., 1 + 2j , 3 - 4j .

```
x = 5 # int
y = 3.14 # float
z = 1 + 2j # complex
```

Numeric Types Practical Use Case

- int : Whole numbers without decimal points. Used for counting and indexing.
- float : Numbers with decimal points. Used for precise calculations and measurements.
- complex : Numbers with real and imaginary parts. Used for advanced mathematical computations.

String Type

- str: A sequence of characters, e.g., ”hello” , ’world’ .

```
greeting = ”Hello, world!”
```

String Types Practical Use Case

- Collect and Store Feedback: Gather customer feedback and store it in a list of strings.
- Extract Useful Information: Identify key phrases or sentiments to understand customer opinions.
- Format Responses: Prepare feedback data for reporting or display, enhancing readability.

Sequence Types

- list: Ordered, mutable collection of items, e.g., [1, 2, 3] , ['a', 'b', 'c'] .

```
fruits = ['apple', 'banana', 'cherry']
# It may have diff types of data
fruits = [1, 3.4, True ,'cherry']

# May have duplicate data
fruits = ['apple', 'apple', 'apple']

# List has index
print(fruits[0])
```

- tuple: Ordered, immutable collection of items, e.g., (1, 2, 3) , ('a', 'b', 'c') .

```
coordinates = (10, 20,40)
# It may have diff types of data
coordinates = (10, "20",4.0)

# May have duplicate data
coordinates = (10, 10,10)

# has index
print(coordinates[0])
```

- range: Represents an immutable sequence of numbers, commonly used in loops, e.g., range(5) , range(1, 10, 2) .

```
numbers = range(1, 10)

# Using Loop
numbers = range(1, 10)
for number in numbers:
        print(number)

# Converting List
print(list(numbers))

# Use Star
print(*numbers)

# Means Default Start from 0
numbers = range(10)

# Means Range After 2 Step
numbers = range(1, 10, 2)
```

String Types Practical Use Case

- List: Used for storing a collection of items that can be modified. Ideal for tasks where you need to add, remove, or change items frequently.
- Tuple: Used for storing a collection of items that should not be changed. Perfect for read-only data or fixed collections of items, like coordinates or configuration settings.
- Range: Used for generating a sequence of numbers. Commonly used in loops for iterating a specific number of times or creating sequences of numbers efficiently.

Mapping Type

- dict: Unordered, mutable collection of key-value pairs, e.g., {'name': 'Alice', 'age': 25}

```
person = {'name': 'Alice', 'age': 25}
print(person['name'])
print(person)
```

Mapping Type Practical Use Case

- Storing Employee Data: Use dictionaries to store employee information with unique IDs as keys and details (name, position, salary) as values.
- Accessing Employee Data: Retrieve specific employee details quickly using their unique ID as the key.
- Updating Employee Records: Easily update or modify employee information in the dictionary by accessing the relevant key.

Set Types

- set: Unordered, mutable collection of unique items, e.g., {1, 2, 3} , {'a', 'b', 'c'} .

```
# Must have unique data
unique_numbers = {1, 2, 3}

# Duplicate data avoided
unique_numbers = {1, 2,2, 3,3,3}
```

- frozenset: Unordered, immutable collection of unique items, e.g., frozenset([1, 2, 3]) .

```
# Must have unique data
immutable_set = frozenset([1, 2, 3])

# Duplicate data avoided
immutable_set = frozenset([1,2, 2, 3])
```

Set Types Practical Use Case

- Set: Used for storing a collection of unique items. Ideal for tasks that require eliminating duplicates or performing mathematical set operations like unions, intersections, and differences.
- Frozenset: An immutable version of a set. Suitable for scenarios where a set of unique items needs to be hashable, such as using sets as dictionary keys or elements of another set.

Boolean Type

- bool: Represents True or False .

```
is_active = True
```

Boolean Type Practical Use Case

- Authentication Status: Use a boolean variable to track if a user is logged in (True) or not (False).
- Conditional Statements: Use booleans in if statements to execute different code blocks based on conditions, such as granting access to certain features only if the user is authenticated.
- Validation Checks: Use booleans to validate user inputs or data integrity, such as checking if an input meets specific criteria (True) or not (False).

None Type

- NoneType: Represents the absence of a value or a null value.

```
result = None
```

None Type Practical Use Case

- Function with No Return Value: Use None to indicate that a function does not return a value. This is useful for functions that perform actions rather than calculations.
- Default Parameter Values: Use None as a default parameter value to signify that no argument was passed, allowing for flexible function definitions and behavior.
- Placeholder for Optional Data: Use None as a placeholder for optional or missing data, making it clear when a variable is intentionally left unset or waiting for a value.

▪ Checking Data Types

```
x = 10
print(type(x)) # Output: <class 'int'>

y = 3.14
print(type(y)) # Output: <class 'float'>

message = "Hello"
print(type(message)) # Output: <class 'str'>

is_valid = True
print(type(is_valid)) # Output: <class 'bool'>
```

Checking Data Types Use Case

- Input Validation: Ensure that user inputs are of the expected type before processing them.
- Function Arguments: Validate function arguments to prevent type errors and ensure correct operation.
- Data Processing: Confirm data types during processing to apply appropriate operations and avoid errors.
- Configuration Loading: Verify the types of configuration settings loaded from files or environment variables.
- Dynamic Data Handling: Handle data that can come in v arious types (e.g., JSON parsing) by checking types before processing.

Mutable vs. Immutable Data Types:

- **Mutable: Can be changed after creation (e.g., lists, dictionaries).**
- **Immutable: Cannot be changed after creation (e.g., strings, tuples).**

Immutable Data Types

Immutable objects cannot be modified after their creation. Any operation that seems to modify an immutable object will actually create a new object. Immutable types include.

Integers (int): Whole numbers, positive or negative.

```
a = 5
initial_id = id(a)
a = 10 # Creates a new integer object with value 10
new_id=id(a)
```

Floating-point numbers (float): Numbers with a decimal point.

```
b = 3.14
initial_id = id(b)
b = 2.71 # Creates a new float object with value 2.71
new_id=id(b)
```

Strings (str): Sequences of characters.

```
s = ”hello”
initial_id = id(s)
s = ”world” # Creates a new string object with value ”world”
new_id=id(s)
```

Tuples (tuple): Ordered collections of items.

```
t = (1, 2, 3)
initial_id = id(t)
t = (4, 5, 6) # Creates a new tuple object with different values
new_id=id(t)
```

Frozen Sets (frozenset): Immutable sets.

```
fs = frozenset([1, 2, 3])
initial_id = id(fs)
fs = frozenset([4, 5, 6]) # Creates a new frozenset object
      with different values
new_id=id(fs)
```

Immutable Practical Use Cases

- Configuration Settings: Store application settings in tuples to ensure they are not accidentally modified.
- User Roles: Define fixed user roles (e.g., admin, editor, viewer) using tuples for security and integrity.
- API Endpoints: Use tuples to store API endpoints, ensuring the URLs remain constant.
- Coordinates: Store geographical coordinates as tuples to maintain their integrity throughout the application.
- Cache Keys: Use frozensets for cache keys to ensure that key combinations remain consistent and hashable.

▪ Mutable Data Types

Mutable objects can be modified after their creation. Operations that modify mutable objects do not create new objects but rather change the existing object. Mutable types include:

Lists (list): Ordered collections of items.

```
l = [1, 2, 3]
initial_id = id(l)
l[0] = 4 # Modifies the existing list object
new_id = id(l)
```

Dictionaries (dict): Collections of key-value pairs.

```
d = {'a': 1, 'b': 2}
initial_id = id(d)
d['a'] = 3 # Modifies the existing dictionary object
new_id = id(d)
```

</WEB
DEVELOPMENT with
python, Django & React

Batch 11

ক্যারিয়ার পাথে এনরোল করতে স্ক্যান করুন

Sets (set): Unordered collections of unique items.

```
s = {1, 2, 3}
initial_id = id(s)
s.add(4) # Modifies the existing set object
new_id = id(s)
```

Mutable Practical Use Cases

- User Sessions: Use dictionaries to store session data, allowing dynamic updates of user-specific information.
- Shopping Cart: Implement shopping carts using lists to add, remove, or modify items based on user actions.
- Form Data: Collect and modify form inputs using dictionaries, making it easy to validate and process user submissions.
- Real-time Notifications: Maintain a list of notifications for users, allowing additions and deletions as new events occur.
- Dynamic UI Elements: Use lists or dictionaries to manage dynamic elements like user-generated content or interactive components that change based on user interaction.

▪ Type Conversion

Explicit Type Conversion: The programmer manually converts a data type using functions like int(), float(), or str().

```
x = "123"
y = int(x) # Convert string to integer
z = float(x) # Convert string to float
a = str(456) # Convert integer to string

print(y) # Output: 123
print(z) # Output: 123.0
print(a) # Output: "456"
```

Implicit Type Conversion: Python automatically converts one data type to another during operations without explicit instruction from the programmer.

```
x = 10
y = 3.14
z = x + y # x is converted to float
print(z) # Output: 13.14
```

Handling Conversion Errors

```
try:
    x = ”abc”
    y = int(x)
except Exception as e:
    print(f”An error occurred: {e}”)
```

Type Conversion Use Case

- User Input Handling: Convert string inputs from forms into integers or floats for calculations.
- Data Processing: Convert data types when reading from or writing to files to ensure correct data formats.
- Mathematical Operations: Convert data to appropriate numeric types for accurate mathematical operations.
- JSON Parsing: Convert data types when parsing JSON to ensure correct types for further processing.
- Database Interaction: Convert data types to match database schema requirements when inserting or retrieving data.

- **Example: Simple Calculator**

```
# Simple Addition
num1 = input(”Enter first number: ”)
num2 = input(”Enter second number: ”)

# Convert input strings to integers
num1 = int(num1)
num2 = int(num2)

# Calculate the sum
sum = num1 + num2

# Print the result
print(”The sum is:”, sum)
```

- **Example: Greeting Program**

```
# Greeting Program
name = input(”Enter your name: ”)

# Print a personalized greeting
print(”Hello, ” + name + ”!”)
```

- **Example: Temperature Converter (Celsius to Fahrenheit)**

```
# Temperature Converter (Celsius to Fahrenheit)
celsius = input("Enter temperature in Celsius: ")

# Convert input string to float
celsius = float(celsius)

# Calculate Fahrenheit
fahrenheit = (celsius * 9/5) + 32

# Print the result
print("Temperature in Fahrenheit:", fahrenheit)
```

- **Example: Even or Odd Checker**

```
# Even or Odd Checker
num = input("Enter a number: ")

# Convert input string to integer
num = int(num)

# Check if the number is even or odd
if num % 2 == 0:
    print(num, "is even")
else:
    print(num, "is odd")
```

- **Example: Simple Interest Calculator**

```
# Simple Interest Calculator
principal = input("Enter the principal amount: ")
rate = input("Enter the rate of interest: ")
time = input("Enter the time (in years): ")

# Convert input strings to float
principal = float(principal)
rate = float(rate)
time = float(time)

# Calculate simple interest
interest = (principal * rate * time) / 100

# Print the result
print("The simple interest is:", interest)
```

স্ট্রিং এবং স্ট্রিং ম্যানিপুলেশন

Strings

- Strings in Python are sequences of characters enclosed within single (' '), double (" "), or triple quotes (''' ''' or """ """).
- They are immutable, meaning they cannot be changed once created.

```
# Single quotes
string1 = 'Hello, World!'

# Double quotes
string2 = "Hello, World!"

# Triple quotes
string3 = '''Hello,
World!'''

# Triple quotes can span multiple lines
string4 = """Hello,
World!"""
```

Single Quotes (' ')

- Used to create string literals.
- Typically used for short strings or when the string itself contains double quotes.
- Can be escaped using a backslash (\ ).
- Best for short strings, especially when the string contains double quotes.

```
single_quote_str = 'Hello, World!'
print(single_quote_str) # Output: Hello, World!

# Using single quotes inside the string
quote_in_str = 'He said, "Hello, World!"'
print(quote_in_str) # Output: He said, "Hello, World!" - no need to escape since the inner quotes are double quotes.

# Using single quotes inside the string with escaping
escaped_quote_in_str = 'He said, \'Hello, World!\'"
print(escaped_quote_in_str) # Output: He said, 'Hello, World!' - escaped using a backslash (\).
```

▪ Double Quotes (" ")

- **Also used to create string literals.**
- **Preferred when the string contains single quotes to avoid escaping.**
- **Can be escaped using a backslash (\ ).**
- **Best for short strings, especially when the string contains single quotes**

```
double_quote_str = "Hello, World!"
print(double_quote_str) # Output: Hello, World!

# Using single quotes inside the string
quote_in_str = "It's a wonderful day!"
print(quote_in_str) # Output: It's a wonderful day! - no need to escape since the inner quotes are single quotes.

# Using double quotes inside the string with escaping
escaped_quote_in_str = "He said, \"Hello, World!\""
print(escaped_quote_in_str) # Output: He said, "Hello, World!" - escaped using a backslash (\).
```

▪ Triple Single Quotes (''' ''')

- Used for multi-line strings or docstrings.
- Can contain both single and double quotes without escaping.
- Preserves the formatting, including line breaks and indentation.
- Ideal for multi-line strings and when the string contains both single and double quotes

```
triple_single_quote_str = '''This is a string
that spans multiple lines.
It can contain both "double quotes" and 'single quotes'
without escaping.'''
print(triple_single_quote_str)

# Output:
# This is a string
# that spans multiple lines.
# It can contain both "double quotes" and 'single quotes'
without escaping.
```

▪ Triple Double Quotes (""" """)

- Functionally identical to triple single quotes.
- Often used for docstrings (multi-line comments) in functions, classes, and modules.
- Preserves the formatting, including line breaks and indentation.
- Also ideal for multi-line strings and commonly used for docstrings

```
triple_double_quote_str = """This is another string
that spans multiple lines.
It also can contain both "double quotes" and 'single
quotes' without escaping."""
print(triple_double_quote_str)

# Output:
# This is another string
# that spans multiple lines.
# It also can contain both "double quotes" and 'single
 quotes' without escaping.
```

- **String Indexing**

Positive Indexing

- Starts from 0 and goes up to len(string) - 1 .
- Index 0 refers to the first character, index 1 to the second character, and so on.

```
text = "Hello, World!"
print(text[0]) # Output: 'H' (first character)
print(text[7]) # Output: 'W' (eighth character)
```

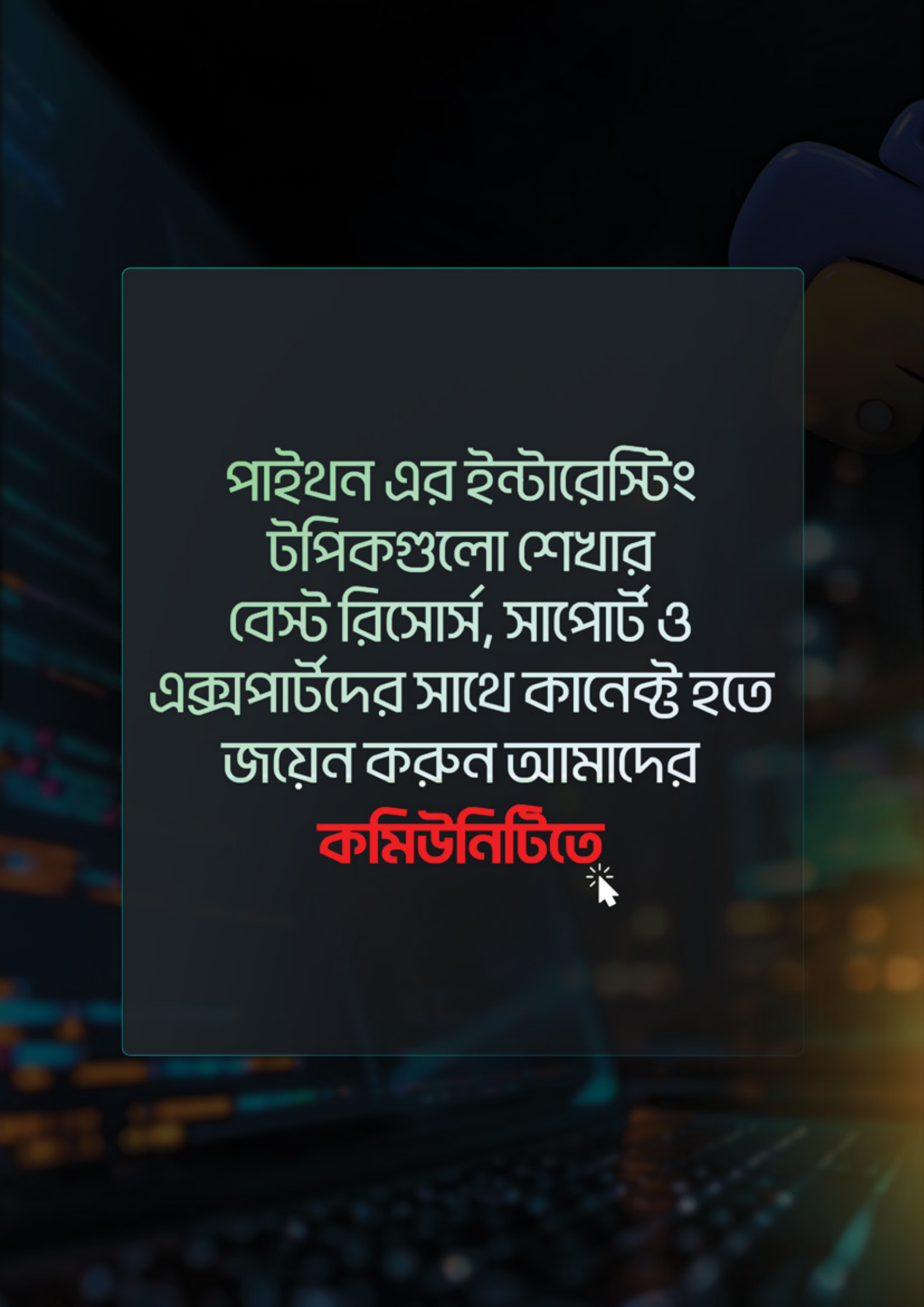
Negative Indexing

- Starts from -1 and goes backwards from the end of the string.
- Index -1 refers to the last character, index -2 to the second last character, and so on.

```
text = ”Hello, World!”
print(text[-1]) # Output: ’!’ (last character)
print(text[-2]) # Output: ’d’ (second last character)
```

String Indexing Use Case

- Extracting Substrings: Retrieve specific parts of a string, such as a substring or a single character.
- Reversing Strings: Access characters in reverse order.
- Manipulating User Input: Modify or analyze parts of user-provided strings, like form inputs.
- Parsing Data: Extract specific fields from structured data formats.
- Validation and Formatting: Check and adjust the format of strings, such as dates or IDs.


পাইথন এর ইন্টারেস্টিং
টপিকগুলো শেখার
বেস্ট রিসোর্স, সাপোর্ট ও
এক্সপার্টদের সাথে কানেক্ট হতে
জয়েন করুন আমাদের
কমিউনিটিতে

String Slicing

String slicing in Python allows you to extract a portion of a string using a colon (:) syntax. The basic form of slicing is string[start:stop:step] , where:

- start is the index where the slice starts (inclusive).
- stop is the index where the slice ends (exclusive).
- step determines the step size or the increment between each index.

Here are the detailed examples based on the given string text = ”Hello, World!” :

- Extracts a Substring from Index 0 to 4

```
text = ”Hello, World!”
print(text[0:5])
```

- Extracts from Index 7 to the End

```
print(text[7:]) # Output: ’World!’
```

- Extracts from the Start to Index 4

```
print(text[:5])
```

- Extracts Every Second Character

```
print(text[::2])
```

- Reverses the String

```
print(text[::-1])
```

- Extracting a Substring with a Specific Step

```
text = ”Hello, World!”
# Extract every third character starting from index 0
print(text[0::3]) # Output: ’Hl r!’
```

- Extracting a Substring from the Middle

```
text = ”Hello, World!”
# Extract substring from index 3 to 8
print(text[3:8]) # Output: ’lo, W’
```

Slicing Use Case

- Extracting Substrings: Retrieve specific parts of a string, such as words or sentences.
- Reversing Strings: Easily reverse the entire string or specific parts of it.
- Formatting Strings: Modify parts of a string to fit a certain format or extract meaningful data.
- Analyzing Data: Extract specific fields from structured data formats like dates or file paths.
- Cleaning Data: Remove unwanted parts of a string or reformat it.

String Concatenation

String concatenation is the process of combining two or more strings into one. In Python, this can be done using the + operator.

Using the + operator:

```
string1 = ”Hello”
string2 = ”World”
combined = string1 + ”, ” + string2 + ”!”
print(combined) # Output: Hello, World!
```

Using join() method:

```
string1 = ”Hello”
string2 = ”World”
combined = ”, ”.join([string1, string2]) + ”!”
print(combined) # Output: Hello, World!
```

Using formatted string literals (f-strings) (Python 3.6+):

```
string1 = ”Hello”
string2 = ”World”
combined = f”{string1}, {string2}!”
print(combined) # Output: Hello, World!
```

- Using the format() method:

```
string1 = "Hello"
string2 = "World"
combined = "{}, {}!".format(string1, string2)
print(combined) # Output: Hello, World!
```

- Using % formatting:

```
string1 = "Hello"
string2 = "World"
combined = "%s, %s!" % (string1, string2)
print(combined) # Output: Hello, World!
```

String Concatenation Use Case

- Building Dynamic Messages: Combine strings to create dynamic text for user messages or logs.
- URL Construction: Assemble URLs from different parts, such as base URLs and query parameters.
- File Paths: Construct file paths by combining directory names and file names.
- Template Strings: Create templates by merging fixed text with dynamic data.
- Data Formatting: Combine multiple pieces of data into a formatted string for display or storage.

String Repetition

String repetition is the process of repeating a string a specified number of times. This can be done using the * operator.

```
# Defining a string
repeat_str = ”Hello! ”

# Repeating the string 3 times
repeat = repeat_str * 3

# Printing the repeated string
print(repeat) # Output: ’Hello! Hello! Hello! ’
```

String Repetition Use Case

- Generating Patterns: Create repeated patterns or borders for text-based interfaces or displays.
- Formatting Output: Repeat characters or strings to format output consistently, like underlining headings.
- Initialization: Quickly initialize a string with repeated characters for placeholders or data preparation.
- Creating Repeated Messages: Generate repeated warning or notification messages for emphasis.
- Visual Separators: Use repeated strings as visual separators in logs or reports.

- **String Methods**

```
# Define a string for demonstration
text = ”hello world”

# Convert to uppercase
print(”Uppercase:”, text.upper()) # Output: ’HELLO WORLD’

# Convert to lowercase
text = ”HELLO WORLD”
print(”Lowercase:”, text.lower()) # Output: ’hello world’

# Capitalize the first letter
text = ”hello world”
print(”Capitalize:”, text.capitalize()) # Output: ’Hello world’

# Title case (capitalize first letter of each word)
print(”Title case:”, text.title()) # Output: ’Hello World’

# Swap case (invert case of each letter)
text = ”Hello World”
print(”Swap case:”, text.swapcase()) # Output:
’hELLO wORLD’

# Replace a substring
text = ”hello world”
print(”Replace:”, text.replace(”world”, ”Python”)) #
Output: ’hello Python’

# Split the string into a list
text = ”hello-world”
words = text.split(”-”) # Splits on hyphen print(words) #
Output: [’hello’, ’world’]
```

```
# Join a list into a string
words = ['hello', 'world']
print("Join:", ' '.join(words)) # Output: 'hello world'

# Strip whitespace from both ends
text = " hello world "
print("Strip:", text.strip()) # Output: 'hello world'

# Remove leading whitespace
print("Left strip:", text.lstrip()) # Output: 'hello world '

# Remove trailing whitespace
print("Right strip:", text.rstrip()) # Output: ' hello world'

# Check if string starts with a substring
text = "hello world"
print("Starts with 'hello':", text.startswith("hello")) #
Output: True

# Check if string ends with a substring
print("Ends with 'world':", text.endswith("world")) #
Output: True

# Find the position of a substring
print("Find 'world':", text.find("world")) # Output: 6

# Count occurrences of a substring
print("Count 'o':", text.count("o")) # Output: 2

# Check if all characters are alphanumeric
print("Is alphanumeric:", text.isalnum()) # Output: False

# Check if all characters are alphabetic
text = "hello"
print("Is alphabetic:", text.isalpha()) # Output: True
```

```
# Check if all characters are digits
text = "12345"
print("Is digit:", text.isdigit()) # Output: True

# Check if the string contains only whitespace
text = " "
print("Is whitespace:", text.isspace()) # Output: True

# Check if the string is titlecased
text = "Hello World"
print("Is titlecased:", text.istitle()) # Output: True

# Example of combining methods
# Capitalizing each word in a sentence
sentence = "this is a sample sentence."
capitalized_sentence = sentence.title()
print("Capitalized sentence:", capitalized_sentence) #
Output: 'This Is A Sample Sentence.'

# Removing extra spaces and converting to uppercase
text = " hello world "
cleaned_text = text.strip().upper()
print("Cleaned and uppercase:", cleaned_text) #
Output: 'HELLO WORLD'
```

String Methods Practical Use Case

- Data Cleaning: Remove unwanted characters, trim whitespace, and standardize text formats.
- Text Analysis: Count occurrences, find substrings, and analyze text content.
- User Input Processing: Validate and sanitize user inputs from forms or other sources.
- Formatting Output: Prepare and format strings for display or reporting.
- Generating Dynamic Text: Construct dynamic messages, URLs, or file paths based on variable data.

নাম্বার এবং ম্যাথ বিস্তারিত

- **Numbers**

Python supports several types of numbers: integers, floating-point numbers (floats), and complex numbers.

Basic Arithmetic Operations

```
# Define some numbers
a = 10
b = 3
c = 3.14

# Addition
print("Addition:", a + b) # Output: 13

# Subtraction
print("Subtraction:", a - b) # Output: 7

# Multiplication
print("Multiplication:", a * b) # Output: 30

# Division
print("Division:", a / b) # Output: 3.3333333333333335

# Floor Division (integer division)
print("Floor Division:", a // b) # Output: 3

# Modulus (remainder)
print("Modulus:", a % b) # Output: 1

# Exponentiation (power)
print("Exponentiation:", a ** b) # Output: 1000
```

Arithmetic Operations Use Case

- Financial Calculations: Calculate interest, total payments,
- and loan amortization schedules.
- Data Analysis: Perform statistical calculations like mean,
- median, and standard deviation.
- Graphics and Gaming: Calculate positions, velocities, and
- accelerations for animations.
- Unit Conversion: Convert units, such as from miles to
- kilometers or Celsius to Fahrenheit.
- Recipe Scaling: Adjust ingredient quantities based on the
- number of servings.

Type Conversion

```
x = 10 # Integer
y = 3.14 # Float

# Convert int to float
print("Convert int to float:", float(x)) # Output: 10.0

# Convert float to int
print("Convert float to int:", int(y)) # Output: 3

# Convert int to complex
print("Convert int to complex:", complex(x)) #
Output: (10+0j)
```

Math

```
import math

# Square root
print("Square root:", math.sqrt(16)) # Output: 4.0

# Power
print("Power:", math.pow(2, 3)) # Output: 8.0

# Trigonometric functions
print("Sine of 90 degrees:", math.sin(math.radians(90))) # Output: 1.0
print("Cosine of 0 degrees:", math.cos(math.radians(0))) # Output: 1.0

# Logarithmic functions
print("Natural log of 10:", math.log(10)) # Output: 2.302585092994046
print("Log base 10 of 10:", math.log10(10)) # Output: 1.0

# Factorial
print("Factorial of 5:", math.factorial(5)) # Output: 120

# Greatest common divisor
print("GCD of 48 and 180:", math.gcd(48, 180)) # Output: 12

# Absolute value
print("Absolute value of -7.5:", math.fabs(-7.5)) # Output: 7.5

# Floor and Ceiling
print("Floor of 3.7:", math.floor(3.7)) # Output: 3
print("Ceiling of 3.7:", math.ceil(3.7)) # Output: 4

# Constants
print("Pi:", math.pi) # Output: 3.141592653589793
print("Euler's number:", math.e) # Output: 2.718281828459045
```

Math Functions Use Case

- Financial Calculations: Compute compound interest, loan amortization schedules, and investment growth using exponential and logarithmic functions.
- Data Analysis: Perform statistical analyses such as calculating mean, median, standard deviation, and correlation coefficients.
- Scientific Computing: Solve equations, perform trigonometric calculations, and analyze physical phenomena.
- Game Development: Calculate angles, distances, and collision detection using trigonometric and geometric functions.
- Engineering: Design and analyze systems, perform signal processing, and compute stress and strain using advanced mathematical functions.

▪ Operator Precedence

Operator precedence determines the order in which operators are evaluated in an expression. Operators with higher precedence are evaluated before operators with lower precedence

Operator Precedence Table (from highest to lowest)

1. Exponentiation (``)**
2. Unary plus, Unary minus, Bitwise NOT (+x , -x , ~x)
3. Multiplication, Division, Floor division, Modulus (* , / , // , %)
4. Addition, Subtraction (+ , -)
5. Bitwise shift (<< , >>)
6. Bitwise AND (&)
7. Bitwise XOR (^)
8. Bitwise OR (|)

9. Comparisons, Identity, Membership (== , != , > , < , >= , <= , is , is not , in , not in)
10. Logical NOT (not)
11. Logical AND (and)
12. Logical OR (or)

Example 1: Exponentiation vs. Multiplication

```
result = 2 ** 3 * 2
print("2 ** 3 * 2:", result) # Output: 16
# Explanation: 2 ** 3 is evaluated first (8), then 8 * 2 = 16
```

Example 2: Multiplication vs. Addition

```
result = 10 + 3 * 2
print("10 + 3 * 2:", result) # Output: 16
# Explanation: 3 * 2 is evaluated first (6), then 10 + 6 = 16
```

Operator Precedence use case

- Mathematical Expressions: Ensure correct order of operations in complex calculations involving multiple arithmetic operators.
- Data Analysis: Accurately compute expressions in data processing pipelines where multiple operations are performed sequentially.
- Programming Logic: Implement conditional statements and loops with mixed logical and comparison operators.
- Financial Calculations: Calculate investment returns, loan payments, and other financial metrics accurately by respecting operator precedence.
- Game Development: Evaluate expressions involving multiple operations, such as calculating positions, velocities, and collision responses.

কন্ট্রোল ফ্লো এবং লজিক্যাল অপারেশনস

If Statements

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

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

Else Statements

The else statement executes a block of code if the if condition is False .

```
age = 16
if age >= 18:
    print("You are an adult.")
else:
    print("You are not an adult.")
```

Else If (Elif) Statements

The elif statement allows you to check multiple conditions. It stands for "else if" and can be used when you need to check more than one condition.

```
age = 16
if age >= 18:
    print("You are an adult.")
elif age >= 13:
    print("You are a teenager.")
else:
    print("You are a child.")
```

Combining If , Elif , and Else Statements

```
score = 75

if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
elif score >= 70:
    print("Grade: C")
elif score >= 60:
    print("Grade: D")
else:
    print("Grade: F")
```

Nested If Statements

You can also nest if statements within other if statements to check more complex conditions.

```
age = 20
has_permission = True

if age >= 18:
    if has_permission:
        print("You can enter the club.")
    else:
        print("You need permission to enter the club.")
else:
    print("You are not allowed to enter the club.")
```

if Else use case

- User Authentication: Check if the entered username and password match the stored credentials and grant or deny access.
- Form Validation: Validate user input in forms and provide feedback or error messages.
- Payment Processing: Determine if a payment transaction is successful or if an error occurred, and handle each case accordingly.
- Data Filtering: Filter data based on specific criteria, such as filtering out invalid entries from a dataset.
- Weather Forecasting: Display different messages or actions based on weather conditions, such as suggesting an umbrella if it's going to rain.
- Inventory Management: Check if stock levels are sufficient to fulfill an order and alert if more inventory is needed.
- Game Logic: Determine game outcomes based on player actions or states, such as winning, losing, or drawing a game.
- Personalized Greetings: Provide personalized greetings or messages based on the time of day or user preferences.
- Discount Application: Apply discounts to purchases based on customer status, such as member, non-member, or special promotions.
- File Handling: Check if a file exists before attempting to read or write to prevent errors and handle cases where the file is missing.

for Loop in Python

The for loop in Python is used to iterate over a sequence (such as a list, tuple, dictionary, set, or string) or other iterable objects.

Iterating Over a List

```
# Example of iterating over a list
fruits = ['apple', 'banana', 'cherry']
for fruit in fruits:
    print(fruit)
```

Iterating Over a String

```
# Example of iterating over a string
word = "hello"
for letter in word:
    print(letter)
```

Using range() Function

```
# Example of using range() function
for i in range(5):
    print(i)
```

Iterating Over a Dictionary

```
# Example of iterating over a dictionary
student_scores = {'Alice': 90, 'Bob': 85, 'Charlie': 92}
for student, score in student_scores.items():
    print(f"{student}: {score}")
```

Iterating Over a Set

```
# Example of iterating over a set
unique_numbers = {1, 2, 3, 4, 5}
for number in unique_numbers:
    print(number)
```

Using break Statement

```
# Example of using break statement
for number in range(10):
    if number == 5:
        break
    print(number)
```

Using continue Statement

```
# Example of using break statement
for number in range(10):
    if number == 5:
        break
    print(number)
```

for loop use case

- Data Processing: Iterate over a list of data points to perform calculations or transformations.
- File Handling: Read and process lines in a file sequentially.
- Generating Reports: Create summaries or reports by iterating over data records.
- Batch Processing: Apply operations to a batch of items, such as resizing images or processing transactions.
- Automating Tasks: Automate repetitive tasks like sending emails or making API calls.

</WEB
DEVELOPMENT with
python, Django & React

Batch 11

ক্যারিয়ার পাথে এনরোল করতে স্ক্যান করুন

- Iterating Over Dictionaries: Access keys and values in a dictionary for tasks like configuration or data analysis.
- Matrix Operations: Perform operations on matrices or 2D arrays, such as addition, multiplication, or transposition.
- Building User Interfaces: Generate dynamic UI components by iterating over data models.
- Simulation and Modeling: Run simulations by iterating over time steps or model parameters.
- Web Scraping: Extract information from web pages by iterating over HTML elements.

While Loops in Python

Iterating Over a List

```
# Iterating over a list with a while loop
fruits = ['apple', 'banana', 'cherry']
index = 0
while index < len(fruits):
        print(fruits[index])
        index += 1
```

Iterating Over a String

```
# Iterating over a string with a while loop
word = "hello"
index = 0
while index < len(word):
        print(word[index])
        index += 1
```

Using range() Function

```
# Simulating range() with a while loop
start = 0
end = 5
while start < end:
    print(start)
    start += 1
```

Iterating Over a Dictionary

```
# Iterating over a dictionary with a while loop
student_scores = {'Alice': 90, 'Bob': 85, 'Charlie': 92}
keys = list(student_scores.keys())
index = 0
while index < len(keys):
    key = keys[index]
    print(f"{key}: {student_scores[key]}")
    index += 1
```

Iterating Over a Set

```
# Iterating over a dictionary with a while loop
student_scores = {'Alice': 90, 'Bob': 85, 'Charlie': 92}
keys = list(student_scores.keys())
index = 0
while index < len(keys):
    key = keys[index]
    print(f"{key}: {student_scores[key]}")
    index += 1
```

Using break Statement

```
# Using break statement in a while loop
counter = 0
while counter < 10:
    if counter == 5:
        break
    print(counter)
    counter += 1
```

Using continue Statement

```
# Using continue statement in a while loop
counter = 0
while counter < 10:
    counter += 1
    if counter % 2 == 0:
        continue
    print(counter)
```

While loop use case

- User Input Validation: Continuously prompt the user for input until valid data is provided.
- Reading Files: Read data from a file until the end of the file is reached.
- Polling for Changes: Continuously check for changes in data or status until a condition is met.
- Implementing Timers: Create countdown timers or delay loops.
- Game Loops: Run the main loop of a game, which continues until the game is over.
- Retry Logic: Retry an operation until it succeeds or a maximum number of attempts is reached.

- Simulations: Run simulations that proceed until a certain condition is met.
- Processing Queues: Process items from a queue until it is empty.
- Progress Tracking: Track and update progress until a task is complete.
- Generating Sequences: Generate a sequence of numbers or data until a certain condition is reached.

What about other's type of loop

- **In Python, there are for and while loops, but there is no direct equivalent to the do-while loop found in some other programming languages.**
- **Additionally, there is no for in , for of , or forEach loop syntax specifically like in JavaScript**

Logical Operators in Python

Logical operators are used to combine conditional statements. The most common logical operators in Python are and , or , and not .

1. and Operator

The and operator returns True if both conditions are True. If either condition is False , the result is False .

```
age = 20
has_permission = True

if age >= 18 and has_permission:
    print("You can enter the club.")
else:
    print("You cannot enter the club.")
```

2. or Operator

The or operator returns True if at least one of the conditions is True . If both conditions are False , the result is False .

```
age = 16
has_permission = True

if age >= 18 or has_permission:
    print("You can enter the club.")
else:
    print("You cannot enter the club.")
```

4. Combining Logical Operators

You can combine multiple logical operators to form more complex conditions.

```
age = 20
has_permission = False
is_vip = True

if (age >= 18 and has_permission) or is_vip:
    print("You can enter the club.")
else:
    print("You cannot enter the club.")
```

Logical Operators Use case

- Access Control: Check multiple conditions to grant or deny access to resources.
- Input Validation: Validate multiple input criteria simultaneously.
- Search Functionality: Filter search results based on multiple criteria.
- Feature Toggles: Enable or disable features based on various conditions.
- Data Filtering: Filter data records based on multiple conditions.
- E-commerce: Apply discounts and promotions based on combined conditions.
- Game Development: Determine game state changes based on multiple player actions or game conditions.
- Scheduling: Check for multiple availability conditions before scheduling an event.
- Configuration Management: Apply configuration settings based on multiple environment variables or settings.
- Monitoring and Alerts: Trigger alerts based on combined system monitoring conditions.

Comparison Operators in Python

Comparison operators are used to compare two values and return a Boolean result (True or False). These operators are essential for making decisions in your code using conditional statements.

1. Equal to (==)

The == operator checks if two values are equal.

```
x = 5
y = 5
print(x == y) # True
```

2. Not equal to (!=)

The != operator checks if two values are not equal.

```
x = 5
y = 3
print(x != y) # True
```

3. Greater than (>)

The > operator checks if the value on the left is greater than the value on the right.

```
x = 7
y = 5
print(x > y) # True
```

4. Less than (<)

The < operator checks if the value on the left is less than the value on the right.

```
x = 3
y = 5
print(x < y) # True
```

5. Greater than or equal to (>=)

```
x = 5
y = 5
print(x >= y) # True
```

6. Less than or equal to (<=)

```
x = 5
y = 7
print(x <= y) # True
```

Comparison Operators Use case

- User Authentication: Verify if entered credentials match stored credentials.
- Input Validation: Ensure user input meets specific criteria, such as age or date range.
- Sorting Data: Compare elements to sort lists, tuples, or other data structures.
- Conditional Formatting: Apply different formatting based on data values, such as highlighting high scores.
- Inventory Management: Check stock levels and trigger reorder processes if inventory falls below a certain threshold.
- Financial Transactions: Validate if transactions exceed credit limits or fall within acceptable ranges.
- Performance Monitoring: Compare current system metrics against baseline values to trigger alerts.
- Game Development: Determine outcomes based on player scores or in-game conditions.
- Access Control: Grant or deny access based on user roles or permissions.
- Data Analysis: Filter and segment data based on comparison criteria.

ডেটা স্ট্রাকচারের এক্সপ্লোরেশন

Lists

- Ordered: Lists maintain the order of elements. This means that when you add elements to a list, they retain their position, and you can access elements using their index. The order of elements is preserved during iteration.
- Mutable: Lists are mutable, meaning you can modify them after creation. You can add, remove, or change elements within a list without creating a new list.
- Allow duplicates: Lists can contain duplicate elements. This allows you to have multiple occurrences of the same value within a list.
- Heterogeneous: Lists can hold elements of different data types. For example, a list can contain integers, strings, floats, and even other lists.
- Dynamic size: Lists in Python are dynamic, meaning their size can change as you add or remove elements.

```
fruits = [”apple”, ”banana”, ”cherry”]
print(fruits)
```

List Methods

1. append()

Adds an element to the end of the list.

```
fruits = [”apple”, ”banana”, ”cherry”]
fruits.append(”orange”)
print(fruits)
```

2. insert()

Inserts an element at a specified position.

```
fruits = [”apple”, ”banana”, ”cherry”]
fruits.insert(1, ”kiwi”)
print(fruits)
```

3. extend()

Extends the list by adding elements from another list.

```
fruits = [”apple”, ”banana”, ”cherry”]
more_fruits = [”grape”, ”melon”]
fruits.extend(more_fruits)
print(fruits)
```

4. remove()

Removes the first occurrence of the specified element.

```
fruits = [”apple”, ”banana”, ”cherry”]
fruits.remove(”banana”)
print(fruits)
```

5. pop()

Removes the element at the specified position (default is the last element) and returns it.

```
fruits = [”apple”, ”banana”, ”cherry”]
last_fruit = fruits.pop()
print(last_fruit)
print(fruits)
```

6. clear()

Removes all elements from the list.

```
fruits = [”apple”, ”banana”, ”cherry”]
fruits.clear()
print(fruits)
```

7. index()

Returns the index of the first occurrence of the specified element.

```
fruits = [”apple”, ”banana”, ”cherry”]
index = fruits.index(”banana”)
print(index)
```

8. count()

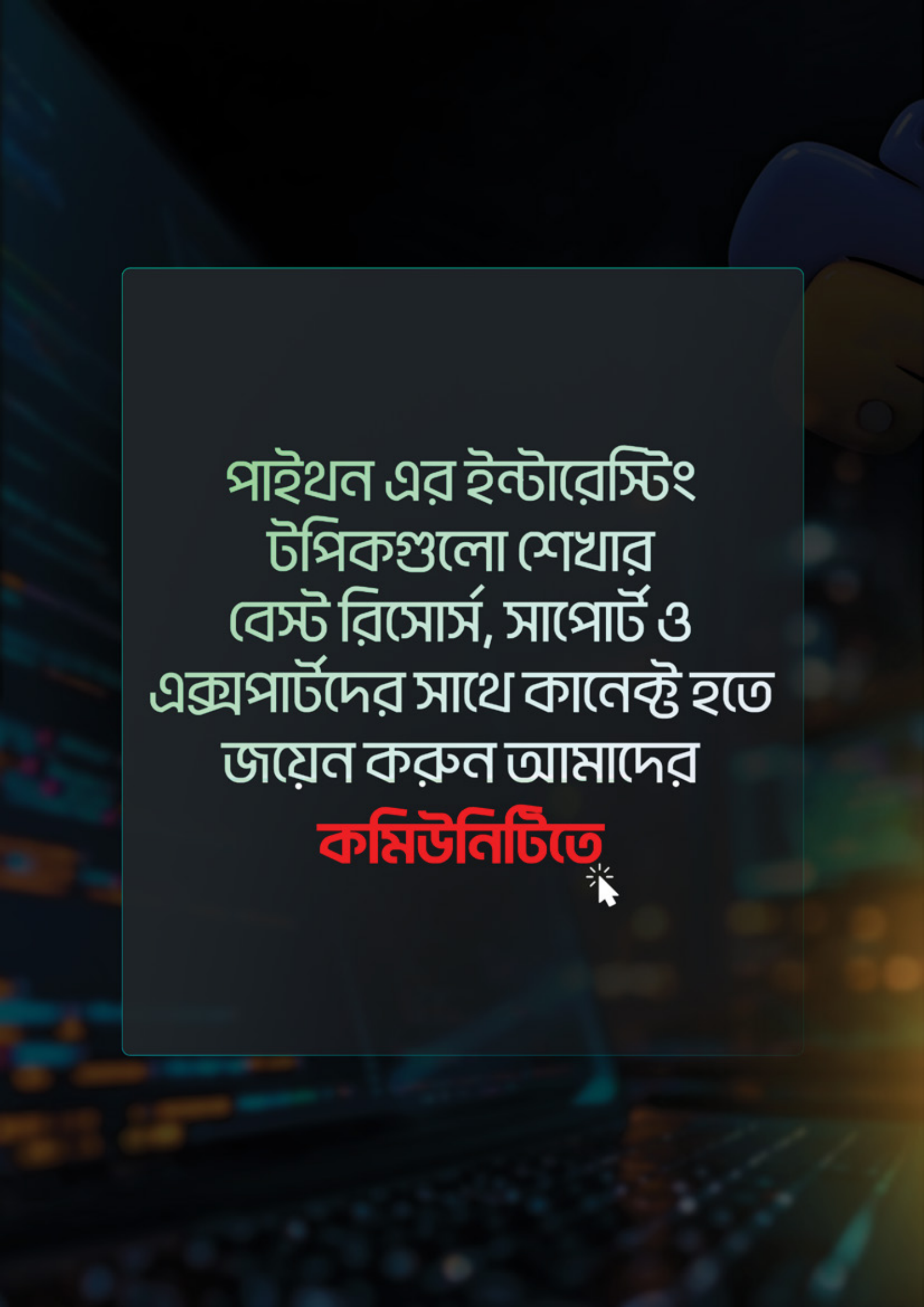
Returns the number of occurrences of the specified element.

```
fruits = [”apple”, ”banana”, ”cherry”]
count = fruits.count(”apple”)
print(count)
```

9. sort()

Sorts the list in ascending order by default.

```
numbers = [3, 1, 4, 1, 5, 9, 2]
numbers.sort()
print(numbers)
```


পাইথন এর ইন্টারেস্টিং
টপিকগুলো শেখার
বেস্ট রিসোর্স, সাপোর্ট ও
এক্সপার্টদের সাথে কানেক্ট হতে
জয়েন করুন আমাদের
কমিউনিটিতে

10. reverse()

Reverses the order of the list.

```
numbers = [3, 1, 4, 1, 5, 9, 2]
numbers.reverse()
print(numbers)
```

12. len()

Returns the number of elements in the list.

```
fruits = [”apple”, ”banana”, ”cherry”]
length = len(fruits)
print(length)
```

13. List Slicing

You can access a range of elements using slicing.

```
fruits = [”apple”, ”banana”, ”cherry”, ”date”, ”fig”, ”grape”]
print(fruits[1:4]) # [’banana’, ’cherry’, ’date’]
print(fruits[:3]) # [’apple’, ’banana’, ’cherry’]
print(fruits[3:]) # [’date’, ’fig’, ’grape’]
print(fruits[-3:]) # [’date’, ’fig’, ’grape’]
```

14. Looping Through a List

Sorts the list in ascending order by default.

```
fruits = [”apple”, ”banana”, ”cherry”, ”date”, ”fig”, ”grape”]
for fruit in fruits:
    print(fruit)
```

List Use Case

- Storing User Data: Keep a list of user names, email addresses, or IDs for easy access and manipulation.
- Managing To-Do Lists: Track tasks and their statuses in a to-do list application.
- Inventory Management: Maintain a list of product items, quantities, and details in a store inventory system.
- Processing Orders: Store and process customer orders in an e-commerce application.
- Collecting Survey Responses: Gather and analyze survey responses from multiple participants.
- Scheduling Events: Organize and manage a list of events or appointments in a calendar application.
- Data Analysis: Store and manipulate datasets for statistical analysis or machine learning.
- Playlist Management: Keep track of songs, videos, or other media items in a playlist.
- Shopping Cart: Store items added to a shopping cart in an online shopping system.
- Tracking Scores: Maintain a list of scores or results for games or competitions.

Tuples

- Ordered: Like lists, tuples maintain the order of elements. The order in which elements are added is preserved, and they can be accessed using an index.
- Immutable: Once a tuple is created, it cannot be modified. You cannot add, remove, or change elements in a tuple after its creation. This immutability makes tuples suitable for use as keys in dictionaries and ensures that the data remains consistent.
- Allow duplicates: Tuples can contain duplicate elements. This allows multiple occurrences of the same value within a tuple.
- Heterogeneous: Tuples can hold elements of different data types. For example, a tuple can contain integers, strings, floats, and even other tuples.
- Fixed size: The size of a tuple is fixed upon creation. Unlike lists, you cannot change the size of a tuple after it is created.

```
fruits = (”apple”, ”banana”, ”cherry”)
print(fruits)
print(fruits[0]) # Output: apple
print(fruits[1]) # Output: banana
print(fruits[2]) # Output: cherry
```

Tuple Methods

1. count()

Returns the number of times a specified value appears in the tuple.

```
numbers = (1, 2, 3, 2, 4, 2)
count_of_twos = numbers.count(2)
print(count_of_twos) # Output: 3
```

2. index()

Returns the index of the first occurrence of the specified value.

```
numbers = (1, 2, 3, 2, 4, 2)
index_of_three = numbers.index(3)
print(index_of_three) # Output: 2
```

3. Looping Through a Tuple

You can loop through the elements of a tuple using a for loop.

```
fruits = (”apple”, ”banana”, ”cherry”)
for fruit in fruits:
    print(fruit)
```

4. Tuple Slicing

You can access a range of elements in a tuple using slicing.

```
fruits = (”apple”, ”banana”, ”cherry”)
print(fruits[1:3]) # Output: (’banana’, ’cherry’)
print(fruits[:2]) # Output: (’apple’, ’banana’)
print(fruits[1:]) # Output: (’banana’, ’cherry’)
print(fruits[-2:]) # Output: (’banana’, ’cherry’)
```

5. Converting Between Lists and Tuples

You can convert lists to tuples and vice versa using the tuple() and list() functions.

```
# List to tuple
my_list = [1, 2, 3]
my_tuple = tuple(my_list)
print(my_tuple) # Output: (1, 2, 3)

# Tuple to list
my_tuple = (4, 5, 6)
my_list = list(my_tuple)
print(my_list) # Output: [4, 5, 6]
```

Tuples Use case

- Immutable Data Storage: Store read-only configuration settings that should not be altered during program execution.
- Return Multiple Values: Return multiple values from a function efficiently.
- Fixed Collections: Store a fixed collection of related data, such as coordinates (x, y, z) or RGB color values.
- Dictionary Keys: Use tuples as keys in dictionaries for composite key lookups.
- Database Records: Represent rows of database records as tuples for easy and consistent access.
- Data Integrity: Ensure data integrity by using tuples for sequences of data that should remain constant.
- Grouping Data: Group heterogeneous data types together logically, such as an employee record with a name, ID, and position.
- Efficient Iteration: Iterate over a fixed set of elements without the overhead of mutable data structures.
- Named Tuples: Use named tuples to create self-documenting code and improve code readability.
- Function Arguments: Pass a fixed set of parameters to functions, ensuring the parameters remain unchanged.

Sets in Python

- Unordered: The elements in a set do not have a defined order. Unlike lists or tuples, when you iterate over a set, the items may appear in a different order each time, and they cannot be accessed using an index.
- Mutable: Sets are mutable, meaning you can add or remove elements after the set is created. However, the elements within the set must be immutable types, such as strings, numbers, or tuples.
- No duplicates: Sets do not allow duplicate elements. If you try to add a duplicate element to a set, it will be ignored, ensuring that all elements in the set are unique.
- Heterogeneous: Like lists and tuples, sets can hold elements of different data types. A set can contain integers, strings, floats, and even other immutable types.
- Dynamic size: The size of a set can change as you add or remove elements. There is no fixed limit on the number of elements a set can hold.
- Efficient membership testing: Sets are optimized for checking whether a specific element is part of the set, making them ideal for operations that require fast membership testing.

```
fruits = {"apple", "banana", "cherry"}
print(fruits)
```

- **Set Methods**

1. add()

Adds an element to the set.

```
fruits = {”apple”, ”banana”, ”cherry”}
fruits.add(”orange”)
print(fruits)
```

2. update()

Adds multiple elements to the set.

```
fruits = {”apple”, ”banana”, ”cherry”}
fruits.update([”grape”, ”melon”])
print(fruits)
```

3. remove()

Removes the specified element from the set. Raises a KeyError if the element is not found.

```
fruits = {”apple”, ”banana”, ”cherry”}
fruits.remove(”banana”)
print(fruits)
```

4. discard()

Removes the specified element from the set if it is present.

```
fruits = {’apple’, ’cherry’, ’grape’, ’melon’, ’orange’}
fruits.discard(”melon”)
print(fruits)
```

5. pop()

Removes and returns an arbitrary element from the set. Raises a KeyError if the set is empty.

```
fruits = {'apple', 'cherry', 'grape', 'melon', 'orange'}
element = fruits.pop()
print(element)
print(fruits)
```

6. clear()

Removes all elements from the set.

```
fruits = {'apple', 'cherry', 'grape', 'melon', 'orange'}
fruits.clear()
print(fruits)
```

7. union()

Returns a new set with all elements from both sets.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
result = set1.union(set2)
print(result) # Output: {1, 2, 3, 4, 5}
```

8. intersection()

Returns a new set with only the elements that are common to both sets.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
result = set1.intersection(set2)
print(result)
```

9. difference()

Returns a new set with elements in the first set that are not in the second set.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
result = set1.difference(set2)
print(result)
```

10. symmetric_difference()

Returns a new set with elements in either set but not in both.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
result = set1.symmetric_difference(set2)
print(result)
```

11. issubset()

Checks if all elements of the set are present in another set.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
print(set1.issubset(set2))
```

12. issuperset()

Checks if the set contains all elements of another set.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
print(set1.issuperset({1, 2}))
```

13. isdisjoint()

Checks if two sets have no elements in common.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
print(set1.isdisjoint(set2))
```

Sets Use case

- Removing Duplicates: Automatically eliminate duplicate entries from a collection of data.
- Membership Testing: Quickly check if an item is part of a collection.
- Mathematical Set Operations: Perform union, intersection, difference, and symmetric difference operations on collections.
- Unique Elements: Maintain a collection of unique items, such as unique user IDs or product codes.
- Filtering Data: Filter out non-unique items from a list or other iterable.
- Tagging System: Implement a system to tag items where each tag is unique.
- Graph Algorithms: Use sets to manage adjacency lists for nodes in graph algorithms.
- Tracking Seen Items: Keep track of items that have already been processed or visited in loops or algorithms.
- Sparse Data Structures: Represent sparse datasets efficiently where only a few elements out of a large range are present.
- Event Handling: Manage a set of unique event handlers or callbacks in event-driven programming.

Dictionary

- Unordered: Dictionaries in Python store key-value pairs without a specific order. While the insertion order is preserved in Python 3.7 and later, dictionaries do not support indexing like lists or tuples.
- Mutable: Dictionaries are mutable, meaning you can change them after they are created. You can add, remove, or modify key-value pairs in a dictionary at any time.
- Key uniqueness: Each key in a dictionary must be unique. If you try to insert a duplicate key, the existing value associated with that key will be overwritten with the new value.
- Heterogeneous: Both keys and values in a dictionary can be of any data type. Keys are typically immutable types (like strings, numbers, or tuples), while values can be of any type, including other dictionaries.
- Dynamic size: Dictionaries in Python can grow or shrink as you add or remove key-value pairs. There is no fixed limit on the number of elements a dictionary can hold.
- Efficient key-based access: Dictionaries provide fast access to values based on their keys, making them ideal for use cases where you need to quickly retrieve, add, or modify data associated with specific keys.

```
student = {
    "name": "Alice",
    "age": 23,
    "major": "Computer Science"
}
print(student)
print(student["name"]) # Output: Alice
print(student.get("age")) # Output: 23
```

Dictionary Methods

1. keys()

Returns a view object containing the dictionary's keys.

```
student = {
    "name": "Alice",
    "age": 23,
    "major": "Computer Science"
}
print(student.keys())
```

2. values()

Returns a view object containing the dictionary's values.

```
student = {
    "name": "Alice",
    "age": 23,
    "major": "Computer Science"
}
print(student.values())
```

3. items()

Returns a view object containing the dictionary's key-value pairs.

```
student = {
    "name": "Alice",
    "age": 23,
    "major": "Computer Science"
}
print(student.items())
```

4. update()

Updates the dictionary with the specified key-value pairs.

```
student = {
    "name": "Alice",
    "age": 23,
    "major": "Computer Science"
}
student.update({"age": 24, "graduated": True})
print(student)
```

5. pop()

Removes the specified key and returns its value. Raises a KeyError if the key is not found.

```
student = {
    "name": "Alice",
    "age": 23,
    "major": "Computer Science"
}
age = student.pop("age")
print(age) # Output: 24
print(student)
```

6. popitem()

Removes and returns the last inserted key-value pair as a tuple.

```
student = {'name': 'Alice', 'age': 24, 'major':
'Computer Science', 'graduated': True}
item = student.popitem()
print(item) # Output: ('graduated', True)
print(student)
```

7. clear()

Removes all elements from the dictionary.

```
student = {
    "name": "Alice",
    "age": 23,
    "major": "Computer Science"
}
student.clear()
print(student) # Output: {}
```

8. get()

Returns the value for the specified key if it exists, otherwise returns the default value.

```
student = {
    "name": "Alice",
    "age": 23,
    "major": "Computer Science"
}
age = student.get("age", "Not Available")
print(age)
```

10. copy()

Returns a shallow copy of the dictionary.

```
student = {
    "name": "Alice",
    "age": 23,
    "major": "Computer Science"
}
student_copy = student.copy()
print(student_copy)
```

Dictionary Use case

- Storing Configuration Settings: Keep configuration settings for applications where each setting has a unique key.
- Fast Lookups: Quickly retrieve values based on unique keys, such as looking up a user's profile by user ID.
- Counting Occurrences: Count the frequency of items in a dataset, such as word counts in a document.
- Caching Results: Store and retrieve the results of expensive computations to avoid redundant calculations.
- Mapping Relationships: Represent relationships between items, such as mapping employee IDs to employee names.
- Storing JSON Data: Work with JSON data structures, which are naturally represented as dictionaries in Python.
- Organizing Data: Organize and group related pieces of information together, such as storing contact information.
- Database Records: Represent records from a database where each record is a dictionary with column names as keys.
- User Preferences: Store user preferences and settings for applications.
- Nested Data Structures: Create complex data structures by nesting dictionaries within dictionaries to represent hierarchical data.

</WEB
DEVELOPMENT with
python, Django & React

Batch 11

ক্যারিয়ার পাথে এনরোল করতে স্ক্যান করুন

ফাংশন এবং স্কোপ ম্যানেজমেন্ট

Defining & Calling a Function

- Functions in Python are defined using the def keyword. They allow you to encapsulate code into reusable blocks. Functions can take arguments, return values, and have optional parameters with default values.

```
# Define the function
def say_hello():
    print("Hello")

# Call the function
say_hello()
```

Functions Use case

- Code Reusability: Encapsulate code logic in reusable functions to avoid duplication and simplify maintenance.
- Modular Programming: Break down complex problems into smaller, manageable functions that each handle a specific task.
- Data Processing Pipelines: Create functions to process data step-by-step, such as in ETL (Extract, Transform, Load) processes.
- Event Handling: Define functions to handle specific events or actions in event-driven programming.
- Mathematical Computations: Implement mathematical formulas and algorithms as functions for clarity and reuse.
- API Interactions: Encapsulate API calls in functions to simplify making requests and handling responses.

- Automation Scripts: Create functions to automate repetitive tasks, such as file manipulation or data entry.
- Custom Sorting and Filtering: Define custom key functions for sorting and filtering data collections.
- Testing and Validation: Write functions to perform tests and validations on data or other functions.
- User Input Handling: Encapsulate logic for validating and processing user input in functions.

Function with Parameters

Function parameters allow you to pass data to a function. Parameters are specified within the parentheses in the function definition.

Required Parameters

Required parameters are the most common type. These parameters must be provided when calling the function.

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

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

Default Parameters

Default parameters allow you to specify a default value for a parameter. If the argument is not provided, the default value is used.

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

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

Variable-Length Arguments (*args)

The *args parameter allows you to pass a variable number of positional arguments to a function. The arguments are passed as a tuple.

```
def greet(*names):
    for name in names:
        print(f"Hello, {name}!")

greet("Alice", "Bob", "Charlie")
```

Keyword Arguments (`kwargs)**

The **kwargs` parameter allows you to pass a variable number of keyword arguments to a function. The arguments are passed as a dictionary.

```
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_info(name="Alice", age=25, city="New York")
```

- **Return Statements**

  The return statement is used to exit a function and return a value (or values) to the caller. This allows functions to produce outputs that can be used elsewhere in the code. A function can return multiple values, return nothing (implicitly or explicitly), or return complex objects like lists, dictionaries, or custom objects.

Returning a Single Value

A function can return a single value.

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

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

Returning Multiple Values

A function can return multiple values as a tuple.

```
def get_min_max(numbers):
    return min(numbers), max(numbers)

min_val, max_val = get_min_max([1, 2, 3, 4, 5])
print(f"Min: {min_val}, Max: {max_val}") # Output:
Min: 1, Max: 5
```

Returning Nothing

A function can return nothing, either implicitly or explicitly.

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

result = greet("Alice")
print(result) # Output: None
```

```
def greet(name):
    print(f"Hello, {name}!")
    return None

result = greet("Alice")
print(result) # Output: None
```

Returning a List

```
def get_even_numbers(n):
    return [x for x in range(n) if x % 2 == 0]

evens = get_even_numbers(10)
print(evens) # Output: [0, 2, 4, 6, 8]
```

Returning a Dictionary

```
def get_person_info(name, age):
    return {"name": name, "age": age}

info = get_person_info("Alice", 25)
print(info) # Output: {'name': 'Alice', 'age': 25}
```

Using Return for Early Exit

```
def find_first_even(numbers):
    for number in numbers:
        if number % 2 == 0:
            return number
    return None

result = find_first_even([1, 3, 5, 6, 7])
print(result) # Output: 6
```

Lambda Functions

Lambda functions are small, anonymous functions defined using the lambda keyword. They are often used for short, simple operations and are particularly useful when you need a simple function for a short period of time.

Basic Lambda Function

```
# A lambda function that multiplies two numbers
multiply = lambda: 2+3
print(multiply())

multiply = lambda x, y: x * y
print(multiply(2, 3)) # Output: 6
```

Lambda for Simple Calculations

```
# A lambda function that adds two numbers
add = lambda x, y: x + y
print(add(5, 3)) # Output: 8

# A lambda function that returns the greater of
  two numbers
greater = lambda a, b: a if a > b else b
print(greater(10, 20)) # Output: 20
```

Using Lambda with map()

The map() function applies a given function to all items in an iterable (like a list) and returns a map object (an iterator).

```
# map(function, iterable)
numbers = [1, 2, 3, 4]
squared = map(lambda x: x ** 2, numbers)
print(list(squared)) # Output: [1, 4, 9, 16]
```

Using Lambda with filter()

The filter() function constructs an iterator from elements of an iterable for which a function returns true.

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers)) # Output: [2, 4, 6]
```

Using Lambda with sorted()

The sorted() function returns a new sorted list from the elements of any iterable. You can pass a lambda function to the key parameter to determine the sort order.

```
points = [(2, 3), (1, 2), (4, 1)]
sorted_points = sorted(points, key=lambda point: point[1])
print(sorted_points) # Output: [(4, 1), (1, 2), (2, 3)]
```

When to Use Lambda Functions

- Simple Functions: Use lambdas for simple, small functions that are not reused elsewhere.
- Inline Use: When you need a function for a short period of time
- Readability: For very simple operations, lambdas can sometimes make the code more concise and readable.

When Not to Use Lambda Functions

- Complex Operations: Avoid lambdas for complex functions that require multiple lines of code. Use a regular function defined with def instead.
- Reusability: If the function needs to be reused, it's better to define it with def .

Local and Global Variables

Understanding the scope of variables is crucial for writing effective and bug-free code. In Python, the scope of a variable determines where in the program you can access that variable. The two main scopes for variables are local and global

- Local Variables: Defined within a function and can only be accessed inside that function.
- Global Variables: Defined outside any function and can be accessed throughout the program.
- Modifying Global Variables: Use the global keyword to modify global variables inside a function.

Local Variables

Local variables are defined inside a function and can only be accessed within that function. They are created when the function is called and destroyed when the function exits.

```
def my_function():
    local_var = 10
    print("Inside the function, local_var:", local_var)

my_function()
# Trying to access local_var outside the function will
  cause an error
# print(local_var) # This will raise a NameError
```

Global Variables

Global variables are defined outside any function and can be accessed throughout the program, both inside and outside functions.

```
global_var = 20

def my_function():
    print("Inside the function, global_var:", global_var)

my_function()
print("Outside the function, global_var:", global_var)
```

Modifying Global Variables Inside a Function

To modify a global variable inside a function, you need to use the global keyword. Without this keyword, Python treats the variable as a local variable within the function scope.

```
global_var = 20

def my_function():
    global global_var
    global_var = 30
    print("Inside the function, global_var:", global_var)

my_function()
print("Outside the function, global_var:", global_var)
```

Local and Global Variables with the Same Name

If a local variable has the same name as a global variable, the local variable will shadow the global variable within the function scope.

```
global_var = 20

def my_function():
        global_var = 10 # This is a local variable
         print("Inside the function, global_var:", global_var)

my_function()
print("Outside the function, global_var:", global_var)
```

ফাইল এবং ডিরেক্টরি ব্যবস্থাপনা

Use Case

- Reading and Writing Text Files: Open, read, and write data to text files for tasks such as logging, data storage, and configuration.
- CSV File Processing: Read from and write to CSV files to handle tabular data for data analysis and reporting.
- Log File Management: Maintain and manage log files to record application events and errors.
- Configuration Files: Read application configuration settings from files to customize behavior without changing code.
- Data Serialization: Serialize and deserialize data using formats like JSON or XML to save program state or exchange data between systems.
- Binary File Handling: Work with binary files to read and write non-text data, such as images, videos, or executable files.
- File Searching and Filtering: Search for specific content within files and filter results based on certain criteria.
- File Uploads and Downloads: Handle file uploads and downloads in web applications to manage user files.
- Temporary File Storage: Create and manage temporary files to hold intermediate data during processing tasks.
- File Compression and Decompression: Compress and decompress files to save disk space or prepare for transfer over the network.

Creating a File

```
# Create a new file
with open("example.txt", "w") as file:
    file.write("This is a new file.")
```

Reading from a File

```
# Read the content of a file
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

Writing to a File

```
# Write to a file
with open("example.txt", "w") as file:
    file.write("Writing new content to the file.")
```

Renaming a File

```
import os

# Rename a file
os.rename("example.txt", "new_example.txt")
```

Deleting a File

```
import os

# Delete a file
os.remove("new_example.txt")
```

Creating a Directory

```
import os
# Create a new directory
os.mkdir("new_directory")
```

Reading a Directory

```
import os
# List all files and directories in the current directory
print(os.listdir("."))
```

Writing to a File in a Directory

```
# Create a file in the new directory and write to it
with open("new_directory/example.txt", "w") as file:
file.write("Writing content to a file in a new directory.")
```

Renaming a Directory

```
import os
# Rename a directory
os.rename("new_directory", "renamed_directory")
```

Renaming a Directory

```
import os
# Remove a file inside the directory first
os.remove("renamed_directory/example.txt")
# Delete the directory
os.rmdir("renamed_directory")
```

Creating a Directory

```
import os

# Create a new directory
os.mkdir("new_directory")
```

Reading a Directory

```
import os

# List all files and directories in the current directory
print(os.listdir("."))
```

Writing to a File in a Directory

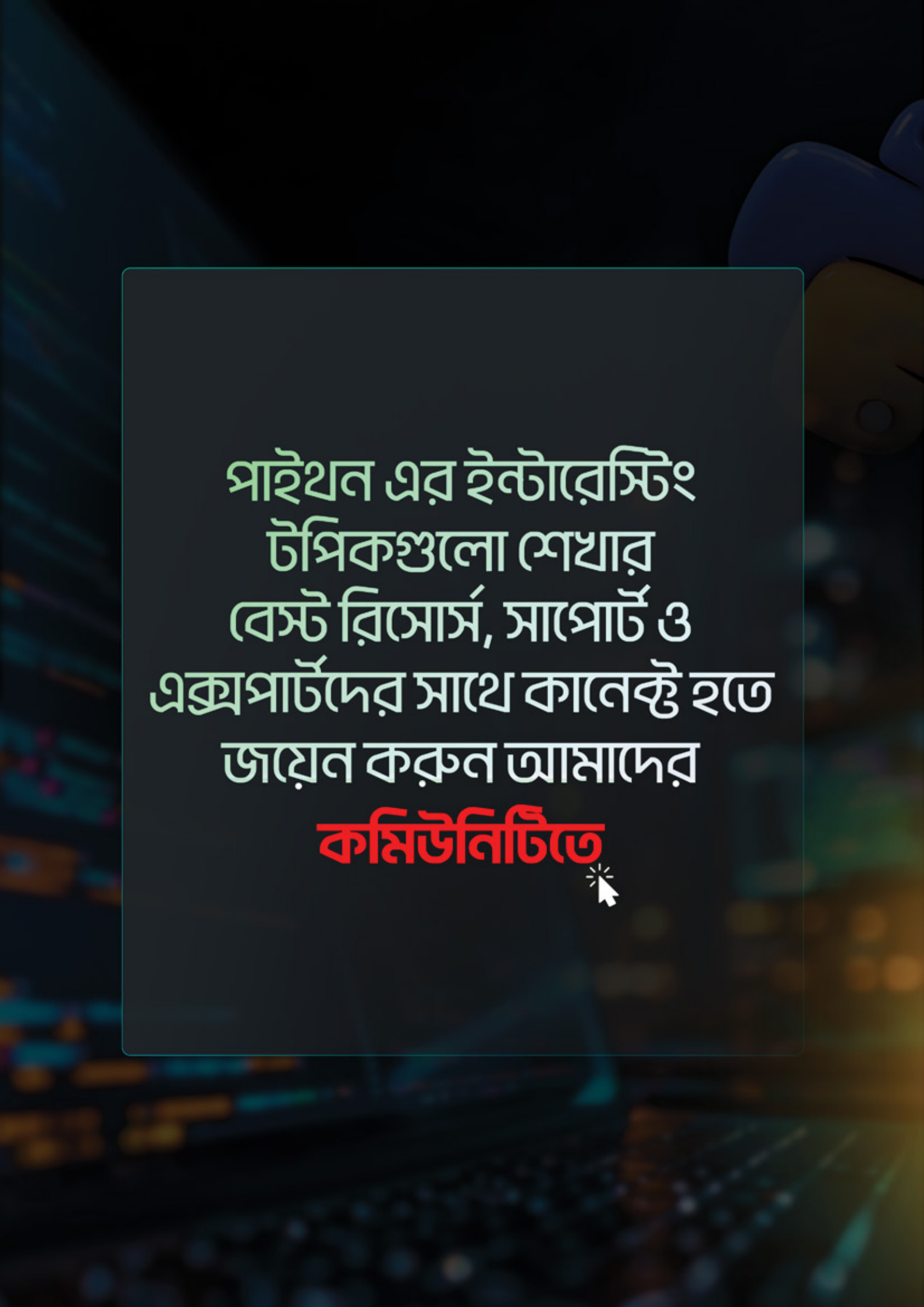
```
# Create a file in the new directory and write to it
with open("new_directory/example.txt", "w") as file:
file.write("Writing content to a file in a new directory.")
```

Renaming a Directory

```
import os

# Rename a directory
os.rename("new_directory", "renamed_directory")
```

Renaming a Directory

```
import os

# Remove a file inside the directory first
os.remove("renamed_directory/example.txt")

# Delete the directory
os.rmdir("renamed_directory")
```


পাইথন এর ইন্টারেস্টিং
টপিকগুলো শেখার
বেস্ট রিসোর্স, সাপোর্ট ও
এক্সপার্টদের সাথে কানেক্ট হতে
জয়েন করুন আমাদের
কমিউনিটিতে

Creating a Directory

```
import os

# Create a new directory
os.mkdir("new_directory")
```

Reading a Directory

```
import os

# List all files and directories in the current directory
print(os.listdir("."))
```

Writing to a File in a Directory

```
# Create a file in the new directory and write to it
with open("new_directory/example.txt", "w") as file:
file.write("Writing content to a file in a new directory.")
```

Renaming a Directory

```
import os

# Rename a directory
os.rename("new_directory", "renamed_directory")
```

Renaming a Directory

```
import os

# Remove a file inside the directory first
os.remove("renamed_directory/example.txt")

# Delete the directory
os.rmdir("renamed_directory")
```

- **Read Files Name From Directory and Print List**

```
import os
directory_path = ”demo”
file_list = os.listdir(directory_path)

print(file_list)

print(”Files in the directory:”)
for file_name in file_list:
    print(file_name)
```

- **Creating Zip File**

```
import zipfile
with zipfile.ZipFile(”my_archive.zip”, ”w”) as zipf:
    zipf.write(”example.txt”)
    zipf.write(”large_file.txt”)
```

- **Extract from Zip File**

```
import zipfile
zip_file_path = ”my_archive.zip”
with zipfile.ZipFile(zip_file_path, ”r”) as zipf:
    zipf.extractall()
    extracted_files = zipf.namelist()
    print(”Extracted files:”, extracted_files)
```

- **Make Zip Form Directory**

```
import shutil

# Specify the directory to be zipped
directory_to_zip = "demo"

# The name of the output zip file (without extension)
output_zip_name = "demo_archive"

# Create a zip archive
shutil.make_archive(output_zip_name, 'zip', directory_to_zip)

print(f"Directory '{directory_to_zip}' has been zipped into '{output_zip_name}.zip'")
```

- **Make CSV File Form List**

```
import csv

# Data to write
data = [
  ["Name", "Salary", "Designation", "Department", "Location"],
  ["Alice", 70000, "Software Engineer", "IT", "New York"],
  ["Bob", 85000, "Data Scientist", "Data", "San Francisco"],
  ["Charlie", 60000, "System Administrator", "IT", "Chicago"],
  ["David", 95000, "Product Manager", "Product", "Boston"],
  ["Eve", 72000, "UX Designer", "Design", "Los Angeles"]
]

# Path to the CSV file
csv_file_path = "example.csv"

# Writing to the CSV file
with open(csv_file_path, mode='w', newline='') as file:
        csv_writer = csv.writer(file)
        csv_writer.writerows(data)

print(f"CSV file '{csv_file_path}' created successfully!")
```

- **Make List From CSV File**

```
import csv

# Path to the CSV file
csv_file_path = ”example.csv”

# Initialize an empty list to hold the rows
data_list = []

# Reading the CSV file and storing it in a list
with open(csv_file_path, mode=’r’) as file:
        csv_reader = csv.reader(file)

        # Iterate over each row in the CSV file
        for row in csv_reader:
              data_list.append(row)

# Print the list
print(”List from CSV file:”)
for row in data_list:
      print(row)
```

এক্সেপশন হ্যান্ডেলিং এবং এর বিশেষ প্রয়োগ

Use Case

- User Input Validation: Validate user inputs and handle errors gracefully, prompting users to provide correct information.
- File Operations: Manage errors that occur during file operations, such as file not found, permission denied, or read/write errors.
- Network Communication: Handle network-related errors like timeouts, connection issues, and invalid responses from servers.
- API Interactions: Gracefully handle errors when making API requests, such as dealing with unavailable services, rate limits, or invalid API keys.
- Database Access: Manage exceptions that occur during database operations, like connection failures, query errors, or transaction issues.
- Data Parsing: Handle errors that arise when parsing data from various formats, such as JSON, XML, or CSV, ensuring robust data processing.
- Arithmetic Operations: Catch exceptions in mathematical computations, such as division by zero or overflow errors, to maintain program stability.
- Resource Management: Ensure that resources (files, network connections, etc.) are properly closed or released even if an error occurs during their use.
- Logging Errors: Record detailed error information to log files for debugging and maintenance purposes.
- User Authentication: Handle exceptions during user authentication processes, such as invalid credentials or expired tokens.

Error List

- ArithmeticError : The base class for all errors related to arithmetic operations.
    - ZeroDivisionError : Raised when dividing by zero.
    - OverflowError : Raised when the result of an arithmetic operation is too large to be represented.
    - FloatingPointError : Raised when a floating-point operation fails.
- AttributeError : Raised when an attribute reference or assignment fails.
- EOFError : Raised when the input() function hits an end-of-file condition (EOF) without reading any data.
- ImportError : Raised when an import statement fails to find the module definition or when a from...import fails to find a name that is to be imported.
    - ModuleNotFoundError : A subclass of ImportError raised when a module cannot be found.
- IndexError : Raised when a sequence subscript is out of range.
- KeyError : Raised when a dictionary key is not found.
- KeyboardInterrupt : Raised when the user hits the interrupt key (Ctrl+C or Delete).
- MemoryError : Raised when an operation runs out of memory.
- NameError : Raised when a local or global name is not found.
    - UnboundLocalError : A subclass of NameError that is raised when a local variable is referenced before it has been assigned.

- OSError : The base class for operating system-related errors.
    - FileNotFoundError : Raised when a file or directory is requested but doesn't exist.
    - PermissionError : Raised when trying to run an operation without the necessary access rights.
    - IsADirectoryError : Raised when a file operation is requested on a directory.
- RuntimeError : Raised when an error is detected that doesn't fall in any other category.
    - NotImplementedError : Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.
- SyntaxError : Raised when the parser encounters a syntax error.
    - IndentationError : Raised when there's an indentation issue.
    - TabError : Raised when tabs and spaces are mixed inconsistently in indentation.
- TypeError : Raised when an operation or function is applied to an object of inappropriate type.
- ValueError : Raised when a function receives an argument of the correct type but an inappropriate value.
- StopIteration : Raised by the next() function to indicate that there are no further items produced by the iterator.
- ZeroDivisionError : Raised when dividing or performing modulo operations by zero.

Simple Try-Except

```
try:
    # Attempt to divide by zero
    result = 10 / 0
except ZeroDivisionError:
    # Handle the division by zero error
    print("Error: Cannot divide by zero.")
```

Handling Multiple Exceptions

```
try:
    # Attempt to read a non-existent file and divide by a
    non-numeric value
    with open("non_existent_file.txt") as file:
        content = file.read()
    result = 10 / int(content) # Assuming the content
        should be an integer
except FileNotFoundError:
    # Handle the file not found error
    print("Error: File not found.")
except ValueError:
    # Handle the error if content is not a valid integer
    print("Error: File content is not a valid number.")
except ZeroDivisionError:
    # Handle the division by zero error
    print("Error: Cannot divide by zero.")
```

Catching All Exceptions

```
try:
    # Code that may raise an exception
    with open("example.txt") as file:
        content = file.read()
    result = 10 / int(content)
except Exception as e:
    # Handle any exception
    print(f"An error occurred: {e}")
```

Finally Block

```
try:
    # Code that may raise an exception
    with open("example.txt") as file:
        content = file.read()
    result = 10 / int(content)
except ZeroDivisionError:
    # Handle the division by zero error
    print("Error: Cannot divide by zero.")
finally:
    # This block will be executed no matter what
    print("Execution completed.")
```

- **Finally Block**

```
try:
    # Code that may raise an exception
    with open("example.txt") as file:
        content = file.read()
    result = 10 / int(content)
except ZeroDivisionError:
    # Handle the division by zero error
    print("Error: Cannot divide by zero.")
except FileNotFoundError:
    # Handle the file not found error
    print("Error: File not found.")
except ValueError:
    # Handle the error if content is not a valid integer
    print("Error: File content is not a valid number.")
else:
    # This block will be executed if no exceptions occur
    print("Division successful, result is:", result)
finally:
    # This block will be executed no matter what
    print("Execution completed.")
```

- **Combined Example**

```
try:
    with open("example.txt", "r") as file:
        content = file.read()
    # Attempting to convert content to integer and
      perform division
    number = int(content.strip())
    result = 10 / number
except FileNotFoundError:
    print("Error: File not found.")
except ValueError:
    print("Error: File content is not a valid number.")
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
else:
    print("File read and division successful, result is:",
    result)
finally:
    print("Execution completed.")
```

জেসন এবং ডেটটাইম ব্যবস্থাপনা

JSON (JavaScript Object Notation) is a lightweight data i nterchange format that is easy for humans to read and write and easy for machines to parse and generate. Python provides a built-in module called json to work with JSON data. This module can be used to convert Python objects to JSON strings and vice versa.

Key Functions in the json Module

1. json.dumps() : Converts a Python object into a JSON string.
2. json.loads() : Converts a JSON string into a Python object.
3. json.dump() : Writes a Python object as a JSON string to a file.
4. json.load() : Reads a JSON string from a file and converts it into a Python object.

Convert Python Object to JSON String

```
import json

# Python dictionary
person = {
        "name": "John",
        "age": 30,
        "city": "New York",
        "hasChildren": False,
        "titles": ["engineer", "programmer"]
}

# Convert Python dictionary to JSON string
person_json = json.dumps(person)
print(person_json)
```

Convert JSON String to Python Object

```
import json

# JSON string
person_json = '{"name": "John", "age": 30, "city":
"New York", "hasChildren": false, "titles":
["engineer", "programmer"]}'

# Convert JSON string to Python dictionary
person = json.loads(person_json)
print(person)
```

Write JSON String to a File

```
import json

# Python dictionary
person = {
        "name": "John",
        "age": 30,
        "city": "New York",
        "hasChildren": False,
        "titles": ["engineer", "programmer"]
}

# Write JSON string to a file
with open('person.json', 'w') as json_file:
        json.dump(person, json_file)
```

Read JSON String from a File

```
import json

# Read JSON string from a file
with open('person.json', 'r') as json_file:
    person = json.load(json_file)

print(person)
```

Pretty Printing JSON

```
import json

# Python dictionary
person = {
    "name": "John",
    "age": 30,
    "city": "New York",
    "hasChildren": False,
    "titles": ["engineer", "programmer"]
}

# Convert Python dictionary to JSON string with pretty
  printing
person_json = json.dumps(person, indent=4)
print(person_json)
```

- **Working With Date, Year, Month, Day, Date, Time, WeekDay, Hour, Minute, Second**

```
import datetime

current_datetime = datetime.datetime.now() # Current
                                Date
print(current_datetime.year) # Current Year
print(current_datetime.month) # Current Month
print(current_datetime.day) # Current Day
print(current_datetime.date()) # Current Date
print(current_datetime.time()) # Current Time
print(current_datetime.weekday()) # Current weekday
print(current_datetime.hour) # Current Hour
print(current_datetime.minute) # Current Minute
print(current_datetime.second) # Current second
print(current_datetime.microsecond) # Current
microsecond
```

- **Current Date Time With Expected Format**

```
import datetime
current_datetime = datetime.datetime.now()
formatted_datetime = current_datetime.strftime
("%d/%m/%Y %H:%M:%S")
print("Current Date and Time:", formatted_datetime)
```

Handling Time Zones

Python's datetime module can handle time zones with the help of the pytz library. Here's how to work with time zones

```
import pytz
import datetime

# Set a timezone
timezone = pytz.timezone("America/New_York")
ny_time = datetime.datetime.now(timezone)
print("New York Time:", ny_time)

# Convert to another timezone
london_timezone = pytz.timezone("Europe/London")
london_time = ny_time.astimezone(london_timezone)
print("London Time:", london_time)
```

Calculating Time Differences

```
import datetime

# Subtracting two dates
date1 = datetime.datetime(2024, 9, 1)
date2 = datetime.datetime(2023, 9, 1)
difference = date1 - date2
print("Difference:", difference)

# Adding 10 days to a date
new_date = date1 + datetime.timedelta(days=10)
print("New Date:", new_date)
```

পাইথন অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং

Introduction to Object-Oriented Programming (OOP) in Python

Object-Oriented Programming (OOP) is a programming paradigm that uses ”objects” to design applications and computer programs. These objects represent real-world entities and can contain data (attributes) and methods (functions).

Key Concepts of OOP

1. Class: A blueprint for creating objects. It defines a set of attributes and methods that the created objects will have.
2. Object: An instance of a class. It is created using the class blueprint and can have its own unique data.
3. Attributes: Variables that belong to an object. They describe the object’s state.
4. Methods: Functions that belong to an object. They define the behavior of the object.
5. Inheritance: A way to form new classes using classes that have already been defined. It helps to reuse and enhance existing code.
6. Encapsulation: The bundling of data and methods that operate on the data within one unit, e.g., a class. It restricts direct access to some of the object’s components.
7. Polymorphism: The ability to present the same interface for different data types. It allows methods to be used interchangeably.

OOP Use Case

- Creating Reusable Components: Design reusable classes for common functionalities like logging, data access, or UI elements.
- Modeling Real-World Entities: Represent real-world entities like customers, products, and orders in an e-commerce application with classes.
- Encapsulation: Encapsulate data and related methods within classes to hide implementation details and expose a clear interface.
- Inheritance for Reusability and Extensibility: Create a base class with common functionality and extend it in derived classes for specialized behavior.
- Design Patterns Implementation: Implement common design patterns (Singleton, Factory, Observer) to solve recurring design problems.
- APIs and Web Services: Develop APIs and web services with classes to manage requests, responses, and data handling.
- Database Interaction: Use classes to represent database tables and manage CRUD operations through Object-Relational Mapping (ORM) frameworks.
- Data Structures: Implement custom data structures (linked lists, trees, graphs) using classes and objects.
- Testing and Mocking: Use classes to create mock objects for unit testing, isolating the code being tested from its dependencies.
- Security and Access Control: Implement security features like authentication and authorization using classes to manage user roles and permissions.
- File and Resource Management: Manage file operations, network connections, and other resources with classes that ensure proper resource handling and cleanup.
- Business Logic Implementation: Encapsulate complex business rules and logic within classes to maintain clarity and separation from other parts of the application.

Creating Classes and Objects

```
class MyClass:
    x = 5

p1 = MyClass()
print(p1.x)
```

```
class MyClass:
      x = 5
      y = 10

def addTwo(self):
        z=10
        print(self.x+self.y+z)

p1 = MyClass()
p1.addTwo()
```

- self is essential for methods to access or modify the instance's attributes.
- It distinguishes between instance attributes and local variables within methods.
- Without self , the method would not know which instance's attributes to operate on.

- **Class Variables and Methods**

Class Variables

Class variables are variables that are shared among all instances of a class. They are defined within the class but outside any instance methods.

Class Methods

Class methods are methods that operate on the class itself rather than on instances of the class. They are defined using the @classmethod decorator and take cls as their first parameter, which refers to the class itself.

```
class MyClass:
#Class Variables
    x = 5
    y=10

#Class Methods

def addTwo(self,extra):
    print(self.x+self.y+10)

def addNew(self):
    self.addTwo(10)

p1 = MyClass()
p1.addTwo(100)
p1.addNew()
```

Constructors

- Constructors are special methods in Python that are automatically called when an object of a class is created.
- The primary purpose of a constructor is to initialize the instance variables of a class.
- In Python, the constructor method is always named __init__ .

Without Parameter

```
class MyClass:
    def __init__(self):
        print("I am constructor")

p1 = MyClass()
```

With Parameter

```
class MyClass:
    def __init__(self,msg):
        print(msg)

p1 = MyClass("I am constructor")
```

Change Class Variable Using Constructor Params

```
class MyClass:
    y=0
    x=0
    def __init__(self):
        self.x = 10
        self.y = 20

p1 = MyClass()

print(p1.x)
print(p1.y)
```

- **Instance Variables and Methods**

Instance Variables:

Attributes that are unique to each instance of a class. They are typically defined in the __init__ method.

Instance Methods:

Functions that operate on instance variables. They take self as the first parameter to refer to the instance on which the method is called.

```
class Employee:

def __init__(self,company_name,employee_count):
        self.company_name = company_name #
                                Instance variable
        self.employee_count = employee_count #
                                 Instance variable

# Instance method
def display_company_info(self):
    print(f"Company Name: {self.company_name},
    Total Employees: {self.employee_count}")

emp1 = Employee("Google",20000)

emp1.display_company_info()
```

Inheritance

- Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class to inherit attributes and methods from another class.
- The class that is inherited from is called the parent class or base class, and the class that inherits is called the child class or derived class
- This mechanism promotes code reusability and can help in creating a more organized and manageable code structure.

</WEB
DEVELOPMENT with
python, Django & React

Batch 11

ক্যারিয়ার পাথে এনরোল করতে স্ক্যান করুন

PASS

- When you want to define a class or function that you plan to implement later, you can use pass to avoid syntax errors.
- When you have a loop or conditional statement and you do not want it to execute any code for a specific condition.
- When you are writing code and want to temporarily skip the implementation of a certain block while you focus on other parts.

Single Inheritance

Definition: A child class inherits from one parent class.

```
class Father:
    def addTwo(self, num1, num2):
        return num1 + num2

class Son1(Father):
    pass

son1 = Son1()
print(son1.addTwo(5, 3)) # Output: 8
```

Multiple Inheritance

Definition: A child class inherits from more than one parent class.

```
class Father:
    def addTwo(self, num1, num2):
        return num1 + num2

class Mother:
    def subtractTwo(self, num1, num2):
        return num1 - num2

class Son1(Father, Mother):
    pass
son1 = Son1()
print(son1.addTwo(5, 3)) # Output: 8
print(son1.subtractTwo(5, 3)) # Output: 2
```

Multilevel Inheritance

Definition: A child class inherits from a parent class, which in turn inherits from another parent class.

```
class Father:
    def addTwo(self, num1, num2):
        return num1 + num2

class Son1(Father):
    pass

class Grandson(Son1):
    pass

grandson = Grandson()
print(grandson.addTwo(5, 3)) # Output: 8
```

- **Constructor at inheritance situations**

When Parent class has, but the child class has not

```
class Father:
    x=10
    y=10
    def __init__(self):
        print(self.x+self.y)

class Son(Father):
    pass

obj = Son()
```

When child class has, but the parent class has not

```
class Father:
    x=10
    y=10

class Son(Father):
    def __init__(self):
        print(self.x + self.y)

obj = Father()
```

When Parent and child both has contractor

```
class Father:
    a = 10
    b = 20
def __init__(self):
    print(self.a + self.b)

class Son(Father):
    x = 100
    y = 200

    def __init__(self):
        super().__init__()
        print(self.x + self.y)
obj = Son()
```

Why is super().__init__() needed?

- Inheritance of Initialization Logic: The Father class might contain essential setup logic in its constructor that the Son class needs. Without calling super().__init__() , that logic will not be executed, and any initialization in Father would be skipped.
- Avoid Code Duplication: If the Son class should also perform the initialization defined in Father , super().__init__() ensures that the constructor of the base class runs without duplicating the logic in Son .
- Object Creation Chain: In Python's inheritance, calling super().__init__() is a best practice to ensure proper object creation, especially in complex cases where multiple classes are involved.

Accessing the Parent's Constructor

```
class Father:
    def __init__(self):
        print("Father's constructor")

class Son(Father):
    def __init__(self):
        # Access Father's constructor using super()
        super().__init__()
        print("Son's constructor")

obj = Son()
```

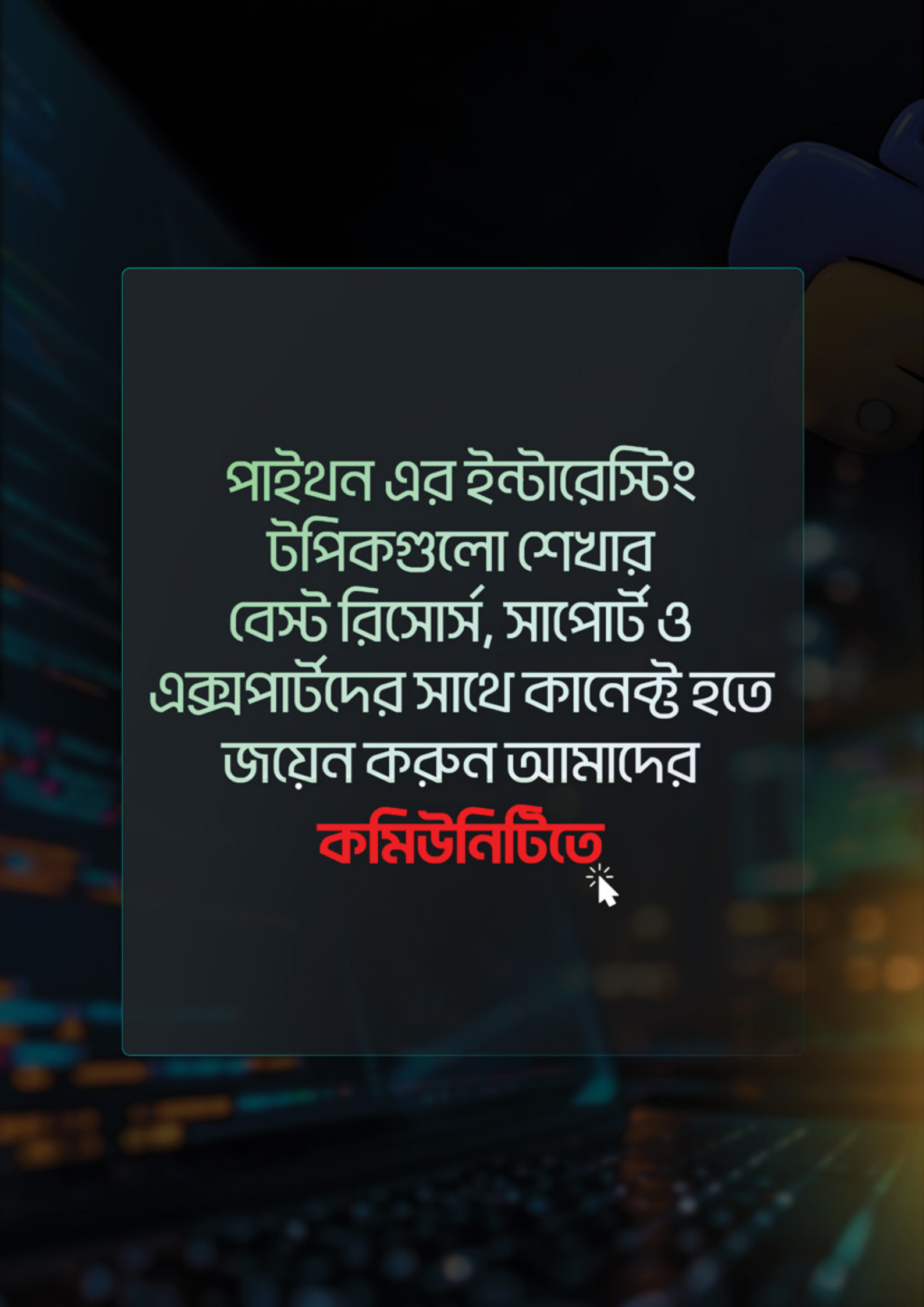
- **Static Properties in inheritance situations**

If Parent has static properties, child can access as it is like parent

```
class Father:
    a = 10
    b = 20

    @staticmethod
    def addtwo():
        print(Father.a + Father.b)

class Son(Father):
    pass
Son.addtwo()
Father.addtwo()

print(Father.a)
print(Father.b)
```

If Child has static properties, Parent can't access as it is like child

```
class Father:
    pass

class Son(Father):
    a = 10
    b = 20

    @staticmethod
    def addtwo():
        print(Son.a + Son.b)

Son.addtwo()
Father.addtwo()

print(Son.a)
print(Son.b)
```


পাইথন এর ইন্টারেস্টিং
টপিকগুলো শেখার
বেস্ট রিসোর্স, সাপোর্ট ও
এক্সপার্টদের সাথে কানেক্ট হতে
জয়েন করুন আমাদের
কমিউনিটিতে

How child can access parents static and non-static properties

```
class Father:
    a = 10
    b = 20

    @staticmethod
    def addtwo():
        print(Father.a + Father.b)

    def addthree(self):
    print(self.a + self.b+10)

class Son(Father):

    def addnew(self):
        self.addthree()
        Father.addtwo()

obj = Son()
obj.addnew()
```

Method Overriding

- Method overriding occurs when a subclass provides a new implementation for a method that is already defined in its superclass.
- This allows the subclass to modify or extend the behavior of that method.

Child can override parent method

```
class Father:
    def addTwo(self, num1, num2):
        return num1 + num2

class Son(Father):
    def addTwo(self, num1, num2):
        # Overriding the method
        print(f"Adding {num1} and {num2} in Son")
        return num1 + num2 + 1 # Modified behavior

# Creating instances of Father and Son
father = Father()
son = Son()

# Calling the addTwo method from Father and Son
print(father.addTwo(5, 3)) # Output: 8
print(son.addTwo(5, 3)) # Output: Adding 5 and 3 in Son
                        # 9
```

Method overloading

- Method overloading is a concept where multiple methods have the same name but different parameters (number or type).
- Python doesn't support method overloading in the same way as some other languages like Java or C++.
- However, you can achieve similar behavior using default arguments or variable-length arguments.

Method Overloading Using Default Arguments

In this example, we will create a Father class with a method addNumbers that can take either two or three arguments.

```
class Father:
    def addNumbers(self, num1, num2, num3=0):
        return num1 + num2 + num3

# Creating an instance of Father
father = Father()

# Calling the addNumbers method with two arguments
print(father.addNumbers(5, 3)) # Output: 8

# Calling the addNumbers method with three arguments
print(father.addNumbers(5, 3, 2)) # Output: 10
```

Method Overloading Using Variable-Length Arguments

In this example, we will use variable-length arguments to allow the method to accept any number of arguments.

```
class Father:
    def addNumbers(self, *args):
        return sum(args)

# Creating an instance of Father
father = Father()

# Calling the addNumbers method with two arguments
print(father.addNumbers(5, 3)) # Output: 8

# Calling the addNumbers method with three arguments
print(father.addNumbers(5, 3, 2)) # Output: 10

# Calling the addNumbers method with four arguments
print(father.addNumbers(1, 2, 3, 4)) # Output: 10
```

Abstract class in python

In Python, an abstract class is a class that cannot be instantiated directly and often serves as a blueprint for other classes. It is used when you want to define common behavior (methods) for a group of related classes, but you don't want to allow the creation of an object of the abstract class itself. Abstract classes are essential for designing interfaces in object-oriented programming.

Key Points of Abstract Classes:

1. Purpose: Define methods that must be implemented by subclasses.
2. Cannot be instantiated: You cannot create an object of an abstract class directly.
3. Abstract methods: Methods declared in the abstract class but without any implementation.
4. Implementation in Subclasses: Subclasses are required to provide implementations for the abstract methods.

To make the Father class abstract in Python, you need to:

1. Import the ABC (Abstract Base Class) module from the abc library.
2. Use the ABC class as a base class for Father .
3. Mark the method addtwo as abstract using the @abstractmethod decorator.

```
from abc import ABC, abstractmethod

# Define the abstract class
class Father(ABC):
    x = 10
    y = 20

    # Define an abstract method
    @abstractmethod
    def addtwo(self):
        pass

# Subclass that implements the abstract method
class Son(Father):

    # Provide implementation for the abstract method
    def addtwo(self):
        print(self.x + self.y)

# Instantiate Son class and call the method
obj = Son()
obj.addtwo()

# You cannot instantiate Father directly now:
# obj2 = Father() # This will raise an error because Father is abstract
```

Access modifiers

In Python, access modifiers control the visibility and accessibility of class attributes and methods. While Python does not have strict access control like some other programming languages (e.g., Java or C++), it uses naming conventions to indicate access levels. The three main types of access modifiers in Python are:

Public (No leading underscore)

- Attributes and methods are public by default.
- They can be accessed and modified both inside and outside the class.
- There are no restrictions on public attributes.

```
class Car:
    def __init__(self, brand):
        self.brand = brand # Public attribute

    def display_brand(self): # Public method
        return f"The brand is {self.brand}"

car = Car("Toyota")
print(car.brand) # Accessing public attribute
print(car.display_brand()) # Accessing public method
```

Protected (Single leading underscore: _)

- Attributes and methods prefixed with a single underscore _ are protected.
- This is a convention indicating that these members should not be accessed or modified outside the class (or its subclasses), but they are still accessible.
- This is a ”soft” access control, meaning it is more of a warning to developers.

```
class Car:
    def __init__(self, brand):
        self._brand = brand # Protected attribute

    def _display_brand(self): # Protected method
        return f”The brand is {self._brand}”

car = Car(”Toyota”)
print(car._brand) # Technically accessible, but not recommended
print(car._display_brand()) # Accessible, but should be used with caution
```

Private (Double leading underscore: __)

- Attributes and methods prefixed with double underscores (__) are considered private.
- These members are not meant to be accessed or modified outside the class. Python applies name mangling to private members to make them harder to access from outside the class.
- Private attributes and methods are intended for internal use only within the class.

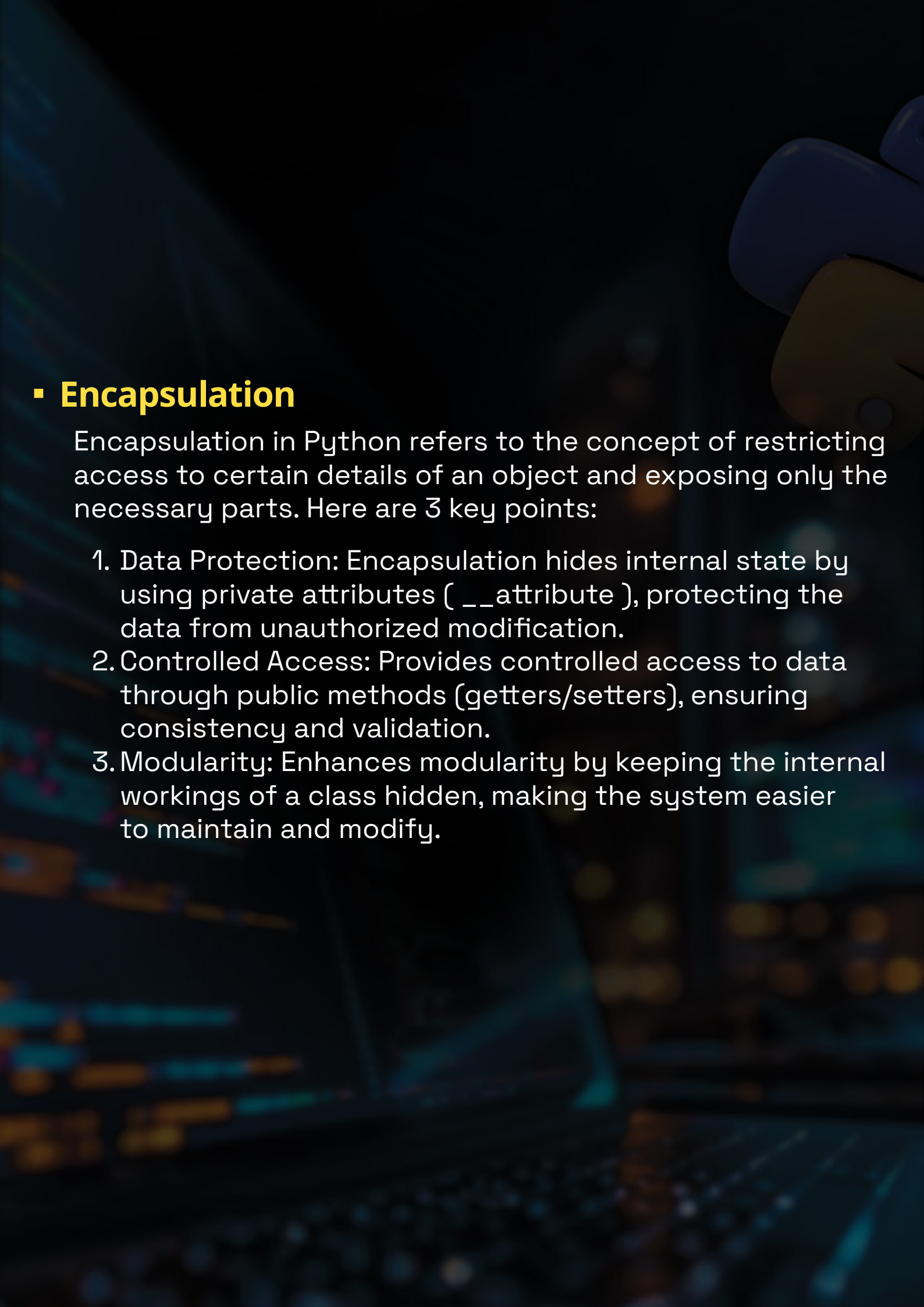
```
class Car:
    def __init__(self, brand):
    self.__brand = brand # Private attribute

    def __display_brand(self): # Private method
        return f"The brand is {self.__brand}"

    def get_brand(self): # Public method to access
    private attribute
        return self.__brand

car = Car("Toyota")
# print(car.__brand) # Raises AttributeError, can't
access directly
print(car.get_brand()) # Access through a public method
```

Getter Setter

Python has a way to define getters and setters, typically using the property() function or the @property decorator. Here's how they work:

- Getters: Used to access (get) the value of a private or protected attribute.
- Setters: Used to set (update) the value of a private or protected attribute.

```
class CAR:
    __BRAND = "Toyota"

    @property
    def BRAND(self):
        return self.__BRAND

    @BRAND.setter
    def BRAND(self,value):
        self.__BRAND=value

OBJ = CAR()
OBJ.BRAND="Suzuki"
print(OBJ.BRAND
```



- **Encapsulation**

  Encapsulation in Python refers to the concept of restricting access to certain details of an object and exposing only the necessary parts. Here are 3 key points:

  1. Data Protection: Encapsulation hides internal state by using private attributes (__attribute), protecting the data from unauthorized modification.
  2. Controlled Access: Provides controlled access to data through public methods (getters/setters), ensuring consistency and validation.
  3. Modularity: Enhances modularity by keeping the internal workings of a class hidden, making the system easier to maintain and modify.

```
class BankAccount:
    __balance = 0

    # Deposit money
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited {amount}")
        else:
            print("Invalid deposit amount!")

    # Withdraw money
    def withdraw(self, amount):
        if 0 < amount and amount<= self.__balance:
            self.__balance -= amount
            print(f"Withdrew {amount}. New balance:
            {self.__balance}")
        else:
            print("Invalid or insufficient funds for
            withdrawal!")

    # Check balance
    def check_balance(self):
        eturn self.__balance

# Create an account
account = BankAccount()

# Deposit money
account.deposit(500)

# Withdraw money
account.withdraw(200)

# Check Balance
print(account.check_balance())
```

Polymorphism

Polymorphism is one of the core concepts of object-oriented programming (OOP), along with inheritance, encapsulation, and abstraction. Polymorphism allows objects of different classes to be treated as objects of a common superclass. It provides a way to use a single interface to represent different underlying forms (data types).

Key Points of Polymorphism

1. Definition: Polymorphism means "many shapes" and allows methods to do different things based on the object it is acting upon.
2. Types of Polymorphism:
    - Compile-time Polymorphism (Method Overloading): Achieved by function overloading. Not directly supported in Python, but can be simulated using default arguments or variable-length arguments.
    - Runtime Polymorphism (Method Overriding): Achieved when a method in a subclass has the same name, return type, and parameters as in its superclass.

```
class Father:
    def addTwo(self, num1, num2):
        return num1 + num2

class Son(Father):
    def addTwo(self, num1, num2):
        # Overriding the method
        print(f"Adding {num1} and {num2} in Son")
        return num1 + num2 + 1 # Modified behavior

# Function to demonstrate polymorphism
def demonstrate_polymorphism(obj, num1, num2):
    return obj.addTwo(num1, num2)

# Creating instances of Father and Son
father = Father()
son = Son()

# Demonstrating polymorphism
print(demonstrate_polymorphism(father, 5, 3)) #
Output: 8
print(demonstrate_polymorphism(son, 5, 3)) #
Output: Adding 5 and 3 in Son
# 9
```

পাইথন মডিউল এন্ড প্যাকেজ ডেভেলপমেন্ট

Creating and using modules in Python is a fundamental part of writing clean, maintainable code. A module is simply a Python file with a .py extension that contains definitions and statements, such as functions, classes, or variables. You can import these modules into other Python files to reuse the code.

Step-by-Step Guide to Creating and Using Python Modules

Step 1: Create a Python Module

First, create a Python file that will serve as your module. Let's call this file mymodule.py .

```
# mymodule.py

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

def subtract(a, b):
    return a - b

class MathOperations:
    def multiply(self, a, b):
        return a * b

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

Step 2: Create a Python Script to Import the Module

Next, create another Python file where you will import and use the module you just created. Let's call this file main.py .

```
# main.py

# Importing the entire module
import mymodule

result_add = mymodule.add(5, 3)
result_subtract = mymodule.subtract(5, 3)
print(f"Addition: {result_add}")
print(f"Subtraction: {result_subtract}")

# Importing specific functions and classes from the module
from mymodule import MathOperations

math_ops = MathOperations()
result_multiply = math_ops.multiply(5, 3)
result_divide = math_ops.divide(10, 2)
print(f"Multiplication: {result_multiply}")
print(f"Division: {result_divide}")
```

Explanation

1. Creating a Module (mymodule.py):
    - This file defines two functions, add and subtract , and a class MathOperations with methods for multiplication and division.
2. Importing the Module (main.py):
    - You can import the entire module using import mymodule and then access its functions and classes using the module name (mymodule.add , mymodule.subtract).
    - Alternatively, you can import specific items from the module using from mymodule import MathOperations and use them directly.

Module Search Path

When you import a module, Python searches for the module in the following locations:

1. The directory containing the input script (or the current directory when no file is specified).
2. The list of directories contained in the PYTHONPATH environment variable.
3. The installation-dependent default path (e.g., /usr/lib/python3.9).

Creating a Package

A package is a way of organizing related modules into a directory hierarchy. To create a package, you need to create a directory and include an __init__.py file (which can be empty) in it.

Step 1: Create a Package Directory

Let's create a package named mypackage .

```
mypackage/
    __init__.py
    mymodule.py
```

Step 2: Use the Package in Your Script

Now, you can use the package in your main.py script.

```
# main.py

# Importing from the package
from mypackage import mymodule

result_add = mymodule.add(5, 3)
result_subtract = mymodule.subtract(5, 3)
print(f"Addition: {result_add}")
print(f"Subtraction: {result_subtract}")

from mypackage.mymodule import MathOperations

math_ops = MathOperations()
result_multiply = math_ops.multiply(5, 3)
result_divide = math_ops.divide(10, 2)
print(f"Multiplication: {result_multiply}")
print(f"Division: {result_divide}")
```

Explanation

1. Package Structure:
    - The directory mypackage contains an __init__.py file and the mymodule.py file. This structure tells Python that mypackage is a package.

2. Importing from the Package:
    - You can import the mymodule module from the mypackage package using from mypackage import mymodule .
    - You can also import specific classes or functions directly from the module within the package using from mypackage.mymodule import MathOperations .

</WEB
DEVELOPMENT with
python, Django & React

Batch 11

ক্যারিয়ার পাথে এনরোল করতে স্ক্যান করুন

Beginner Question and Answer

Python Basics & Syntax

1. What is Python, and what are some advantages of using it?
    - Python is a high-level, interpreted programming language known for its simplicity and readability. Advantages include ease of learning, vast libraries, and community support.
2. What is the difference between Python 2 and Python 3?
    - Python 2 is outdated and no longer supported, while Python 3 includes improved syntax, better Unicode support, and more efficient memory management.
3. Can you explain the difference between list and tuple in Python?
    - List is mutable (can be changed), while tuple is immutable (cannot be changed after creation).
4. What are dictionaries in Python? How do they work?
    - Dictionaries store key-value pairs. You access values by keys, e.g., dict[key] .
5. How would you define a function in Python?
    - A function in Python is defined using def followed by the function name and parameters:
    - def my_function():
6. What is the purpose of the self keyword in Python?
    - self refers to the instance of the class, allowing access to the class's attributes and methods within its own definition.

7. Can you explain how to handle exceptions in Python?
    - Exceptions in Python are handled using try , except , and optionally finally for cleanup.
8. What is a lambda function, and when would you use one?
    - A lambda function is an anonymous, one-line function. It's used for short operations where defining a full function isn't necessary.
9. How do you create a virtual environment in Python?
    - To create a virtual environment, run:
    - python -m venv myenv.
10. What are Python's built-in data types?
    - Built-in data types in Python include int , float , str , list , tuple , dict , set , and bool .

Control Flow

11. How does a for loop work in Python? Can you give an example?
    - A for loop iterates over a sequence (like a list). Example:

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

12. Explain the difference between break and continue in loops.
    - break exits the loop completely,
    - while continue skips the current iteration and moves to the next one.

13. What is the purpose of the else block in loops in Python?
    - The else block in loops runs when the loop completes without encountering a break .

14. Can you explain how an if statement works in Python with an example?
    - An if statement checks a condition and executes code if it's true. Example:

```
if x > 5:
  print("x is greater than 5")
```

15. What is the difference between == and is in Python?
    - == checks value equality,
    - while is checks object identity (if they refer to the same memory location).

16. How does the pass statement work in Python?
    - The pass statement does nothing and is used as a placeholder.

17. What is the purpose of the try...except block, and how does it work?
    - The try...except block catches and handles exceptions.
    - If an error occurs in the try block, control moves to the except block.

Functions and Arguments

18. What is the difference between positional arguments and keyword arguments in Python functions?
    - Positional arguments are passed in order,
    - while keyword arguments are passed with explicit names.
    - Example: def func(a, b): (positional) vs. def func(a, b=5): (keyword).

19. How would you define a default argument in Python?
    - A default argument in Python has a predefined value if no argument is passed.
      Example:

```
def greet(name=”John”):
    print(name)
```

20. What are variable-length arguments in Python? How do you use them?
    - Variable-length arguments allow passing any number of arguments.
    - Use *args for non-keyword
    - and **kwargs for keyword arguments.

21. Can you explain the concept of function return values with an example?
    - A function return value sends back a result from the function. Example:

```
def add(x, y):
    return x + y
```

22. What does the *args and **kwargs syntax mean in Python functions?
    - *args collects non-keyword variable-length arguments as a tuple,
    - and `kwargs`** collects keyword arguments as a dictionary. Example:

```
def add(x, y):
    return x + y
```

Object-Oriented Programming

23. What is a class in Python? How do you create one?
    - A class in Python is a blueprint for creating objects.
    - It's created using the class keyword. Example:

```
class MyClass:
    pass
```

24. Can you explain the concept of inheritance in Python?
    - Inheritance allows a class to inherit attributes and methods from another class. Example:

```
class Animal:
    pass
class Dog(Animal):
    pass
```

25. What is method overriding, and how does it work in Python?

- Method overriding occurs when a subclass provides a specific implementation of a method defined in its superclass.

26. What is the purpose of the __init__ method in Python classes?

- The __init__ method is the constructor in Python classes, used for initializing the object's state. Example:

```
class MyClass:
    def __init__(self, name):
        self.name = name
```

27. What is the difference between staticmethod and classmethod in Python?

- staticmethod does not require self or cls
- and is used for utility functions,
- while classmethod requires cls
- and is used to access class-level attributes.

28. Can you explain encapsulation in Python with an example?

- Encapsulation involves restricting direct access to certain attributes or methods. Example:

```
class MyClass:
    def __init__(self):
        self.__private = 10 # Private variable
```

29. What is polymorphism in Python? Can you give an example?

- Polymorphism allows different classes to define methods with the same name
- but different implementations. Example:

```
class Dog:
    def speak(self):
        print("Woof")
class Cat:
    def speak(self):
        print("Meow")
```

Modules & Packages

30. How do you import modules in Python? Can you give an example?

- To import a module in Python, use the import keyword. Example:

```
import math
print(math.sqrt(16))
```

31. What is the difference between import and from ... import in Python?

- import imports the entire module,
- while from ... import imports specific parts from the module. Example:

```
import math
from math import sqrt
```

32. What is the purpose of __name__ == "__main__" in Python scripts?
    - __name__ == "__main__" ensures that a Python script runs only
    - when executed directly, not when imported as a module.

33. How would you install an external package in Python using pip?
    - To install an external package in Python, use:

```
pip install package_name
```

34. How do you organize Python projects with packages and modules?
    - Python projects are organized into packages (directories with __init__.py) and modules (Python files). Example:

```
my_project/
    module1.py
    module2.py
    __init__.py
```

File Handling

35. How do you open and read a file in Python?
    - To open and read a file in Python, use the open() function and read() . Example:

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

36. Can you explain how to write to a file in Python?

- To write to a file in Python, use the open() function with mode ”w” or ”a” . Example:

```
with open(”file.txt”, ”w”) as file:
    file.write(”Hello, World!”)
```

37. What is the purpose of the with statement in file handling?

- The with statement automatically handles opening and closing files,
- ensuring proper cleanup after file operations.

38. How would you handle file not found or other I/O errors in Python?

- To handle file not found or other I/O errors, use try...except . Example:

```
try:
    with open(”file.txt”, ”r”) as file:
        content = file.read()
except FileNotFoundError:
    print(”File not found”)
```

Python Data Structures

39. How do you reverse a list in Python?

- To reverse a list in Python, use reverse() or slicing. Example:

```
my_list = [1, 2, 3]
my_list.reverse() # In-place
# or
reversed_list = my_list[::-1] # Creates a new reversed list
```

40. What is list comprehension, and how does it work? Can you give an example?

- List comprehension creates a new list by applying an expression to each element in an existing iterable. Example:

```
squares = [x**2 for x in range(5)]
```

41. What is a set in Python, and how does it differ from a list?

- A set in Python is an unordered collection of unique elements.
- Unlike a list, a set does not allow duplicates and is unordered.

42. How do you merge two dictionaries in Python?

- To merge two dictionaries, use the update() method or the | operator in Python 3.9+. Example:

```
dict1 = {"a": 1}
dict2 = {"b": 2}
dict1.update(dict2)
# or
merged = dict1 | dict2
```

43. What is a generator in Python, and how is it different from a list?

- A generator is an iterator that yields values lazily, one at a time. It differs from a list because it doesn't store the entire sequence in memory, making it more memory efficient. Example:

```
def my_gen():
    for i in range(3):
        yield i
```

Advanced Python Concepts

44. What is a decorator in Python, and how do you use it?

- A decorator in Python is a function
- that modifies the behavior of another function. Use @decorator_name above the
- function. Example:

```
def my_decorator(func):
    def wrapper():
        print("Before function")
        func()
        print("After function")
    return wrapper

@my_decorator
def say_hello():
    print("Hello")
```

45. Can you explain what a Python iterator is and how it works?
    - A Python iterator is an object that implements the __iter__() and __next__() methods.
    - It allows iteration over a collection one element at a time.

46. What is a context manager in Python, and when would you use one?
    - A context manager in Python is used to manage resources (e.g., files).
    - It is typically used with the with statement. Example:

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

47. How do you handle multiple exceptions in a single try block in Python?
    - To handle multiple exceptions in a single try block, use multiple except clauses. Example:

```
try:
    # code that may throw exceptions
except (TypeError, ValueError) as e:
    print(f"Error: {e}")
```

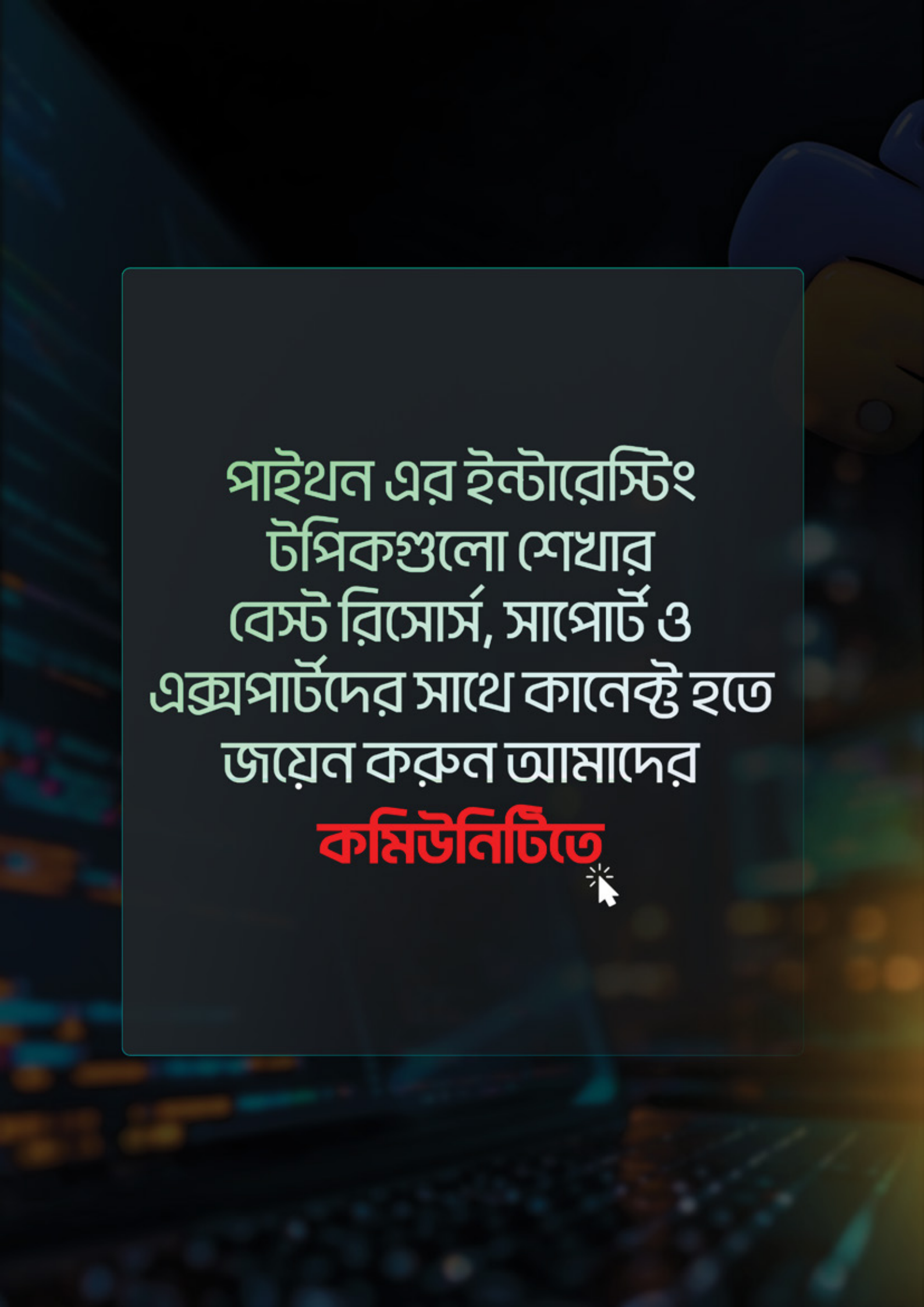
48. What is the difference between deep copy and shallow copy in Python?
    - Shallow copy copies the object, but not the nested objects,
    - while deep copy creates a completely independent copy, including nested objects.
    - Use copy() for shallow and copy.deepcopy() for deep copy.

49. Can you explain Python's Global Interpreter Lock (GIL)?

- GIL (Global Interpreter Lock) in Python is a mutex
- that allows only one thread to execute Python bytecodes at a time,
- limiting multi-threading performance in CPU-bound tasks.

50. How do you manage memory in Python, and what are some memory management techniques?

- Memory management in Python is handled by automatic garbage collection.
- Techniques include using del, weak references, and manual memory management with the gc module.


পাইথন এর ইন্টারেস্টিং
টপিকগুলো শেখার
বেস্ট রিসোর্স, সাপোর্ট ও
এক্সপার্টদের সাথে কানেক্ট হতে
জয়েন করুন আমাদের
কমিউনিটিতে

Intermediate Question and Answer

1. How can you add an item to a dictionary?
    - Use the syntax dict[key] = value to add an item to a dictionary.
2. What are Python modules and packages?
    - A module is a single file containing Python code (functions, classes, etc.).
    - A package is a collection of modules grouped together in a directory with a special __init__.py file.
3. Explain the self keyword in Python.
    - self is used in instance methods to refer to the current object. It allows access to the object's attributes and methods.
4. What is the purpose of the pass statement in Python?
    - pass is a placeholder used to indicate that nothing should happen, typically used in empty classes or functions.
5. How do you define a function in Python?
    - Use the def keyword followed by the function name and parameters, then the function body. Example:
      def my_function():
6. What is the difference between append() and extend() in Python?
    - append() adds a single item to the end of a list.
    - extend() adds each element of an iterable to the list.
7. What is the significance of __init__ method in Python?
    - __init__ is the constructor method for initializing a new object. It's automatically called when an object is created.

8. What are lambda functions in Python?

- Lambda functions are anonymous functions defined using the lambda keyword. They are typically used for short, throwaway functions.

9. What is a Python list comprehension?

- List comprehension is a concise way to create lists using a single line of code, often involving loops and conditionals.

10. How does Python handle variable scope?

- Python uses the LEGB rule (Local, Enclosing, Global, Built-in) to resolve variable scope. It checks for a variable in the local, enclosing, global, and built-in namespaces in that order.

11. Explain Python's garbage collection.

- Python uses automatic memory management with reference counting and a cyclic garbage collector to reclaim memory from objects no longer in use.

12. What is a Python set? How is it different from a list?

- A set is an unordered collection of unique items.
- A list is ordered and can contain duplicate values.

13. What are the advantages of using Python's with statement?

- The with statement simplifies resource management (like file handling) and ensures that
- resources are properly cleaned up after use (e.g., closing files).

14. How can you merge two dictionaries in Python?

- Use the update() method or the | operator (in Python 3.9+) to merge dictionaries.

15. What are namedtuples in Python?
    - Namedtuples are subclasses of tuples that allow you to access elements using named attributes instead of indices.
16. How do you create a class in Python?
    - Use the class keyword followed by the class name and a colon. Example: class MyClass:
17. Explain the difference between instance variables and class variables.
    - Instance variables are unique to each instance of the class.
    - Class variables are shared across all instances of the class.
18. What is slicing in Python?
    - Slicing allows you to extract a portion of a sequence (like a list or string) using a specified start, stop, and step.
19. What is the difference between map() and filter() ?
    - map() applies a function to each item in an iterable and returns an iterator of the results.
    - filter() applies a function to each item and returns an iterator of items for which the function returns True .
20. How do you check if a variable is a string in Python?
    - Use the isinstance() function to check if a variable is of type str . Example: isinstance(x, str) .
21. What is the purpose of super() in Python?
    - super() is used to call a method from a parent class in a derived class, often used in inheritance to extend or override functionality.
22. How can you handle multiple exceptions in Python?
    - Use multiple except blocks or a tuple of exception types in a single except block to handle multiple exceptions.

23. What is the __str__() method used for?
    - The __str__() method defines how an object is represented as a string when passed to print() or str() .
24. How would you remove duplicates from a list in Python?
    - Convert the list to a set (which removes duplicates) and then back to a list, or use a list comprehension with a conditional check.
25. How do you implement inheritance in Python?
    - Inheritance is implemented by creating a new class that derives from a base class, allowing the subclass to inherit methods and properties.
26. Explain method overloading in Python.
    - Python does not support traditional method overloading (same method name, different parameters). Instead, you can use default arguments or variable-length arguments.
27. What is the purpose of the del keyword in Python?
    - The del keyword is used to delete variables, items from lists, or even entire objects.
28. What are Python's enumerate() and zip() functions used for?
    - enumerate() adds a counter to an iterable, returning pairs of index and value.
    - zip() pairs elements from two or more iterables into tuples.
29. How do you get the current date and time in Python?
    - Use the datetime module to get the current date and time. Example: datetime.datetime.now() .

30. What is Python's any() and all() functions?

- any() returns True if at least one element in an iterable is True .
- all() returns True if all elements in an iterable are True .

31. How do you reverse a string in Python?

- You can reverse a string using slicing: string[::-1] .

32. What is a try-except-else block in Python?

- The try block contains code that may raise an exception. The except block handles the
- exception, and the else block runs if no exception occurs.

33. What is Python's assert statement used for?

- The assert statement tests a condition, and if the condition is False , it raises an AssertionError .

34. What is the difference between str() and repr() in Python?

- str() is used to return a readable, user-friendly string representation of an object.
- repr() returns a more formal or unambiguous string representation, often suitable for debugging.

35. How can you create an empty set in Python?

- Use set() to create an empty set.

36. How do you copy a list in Python?

- Use the copy() method or slicing (list[:]) to create a shallow copy of a list.

37. What is a staticmethod in Python?

- A staticmethod is a method that doesn't access or modify the instance or class. It can be called on the class or the instance.

38. What is a classmethod in Python?
    - A classmethod is a method that takes the class as the first argument (cls) and can modify class-level variables.
39. Explain the difference between __eq__() and __ne__() methods.
    - __eq__() is used to compare equality of two objects.
    - __ne__() is used to compare inequality of two objects.
40. How do you convert a list of tuples to a dictionary in Python?
    - Use the dict() function to convert a list of tuples (where each tuple contains a key-value pair) into a dictionary.
41. What is metaprogramming in Python?
    - Metaprogramming refers to writing programs that manipulate or modify other programs or themselves. In Python, this can involve manipulating classes, functions, or code dynamically (e.g., using decorators, __getattr__ , __setattr__).
42. Explain the difference between __new__() and __init__() methods.
    - __new__() is responsible for creating a new instance of a class.
    - __init__() is responsible for initializing the newly created instance.
43. What is the use of __call__() method in Python?
    - The __call__() method allows an object to be called like a function. This is useful for
    - making an object behave like a callable entity.

44. How can you implement a singleton pattern in Python?
    - The singleton pattern ensures a class has only one instance. This can be implemented by
    - overriding __new__() to ensure the same instance is returned every time.

45. What are Python's asyncio and await used for?
    - asyncio is used for writing asynchronous code, while await is used to pause the execution
    - of an async function until the result is ready, making code non-blocking.

46. How would you create a thread in Python?
    - You can create a thread by importing the threading module and using
    - threading.Thread(target=function) to start a thread.

47. What is the Global Interpreter Lock (GIL)?
    - The GIL is a mechanism that prevents multiple native threads from executing Python
    - bytecodes at once. It allows only one thread to execute at a time, limiting concurrency.

48. What is Python's multiprocessing module?
    - The multiprocessing module allows you to create multiple processes, each with its own Python interpreter and memory space, sidestepping the GIL and enabling true parallelism.

49. What are Python coroutines?
    - Coroutines are special functions defined with async def that can be paused and resumed, typically used in asynchronous programming to allow non-blocking operations.

50. Explain Python's memory view object.

- The memoryview object allows you to access the memory of an object (like byte data) without making a copy, improving performance when handling large data.

51. What is the difference between shallow copy and deep copy for objects?

- Shallow copy copies the object but not the nested objects, which are referenced.
- Deep copy creates a new object and also copies all nested objects, ensuring full independence from the original.

52. How would you optimize Python code for speed?

- You can optimize Python code by using built-in functions, avoiding unnecessary loops, using libraries like NumPy for numerical operations, and profiling with cProfile to find bottlenecks.

53. What is the purpose of Python's collections module?

- The collections module provides specialized container datatypes like Counter , deque , defaultdict , and OrderedDict , offering more functionality than built-in types.

54. What is the __del__() method used for?

- The __del__() method is a destructor called when an object is about to be destroyed. It can be used to release resources like closing files or network connections.

55. What is the yield keyword in Python used for?

- The yield keyword is used in a function to turn it into a generator, allowing the function to return values one at a time and pause execution between calls.

56. How do you handle database operations in Python?
    - You can use libraries like sqlite3 for SQLite databases or SQLAlchemy for more complex database operations, including ORM support.

57. Explain the difference between async and await in Python.
    - async is used to define an asynchronous function, while await is used to pause the function execution until the awaited task (like I/O) is completed.

58. What is the purpose of inspect module in Python?
    - The inspect module provides introspection tools to get information about live objects like modules, classes, functions, and their arguments.

59. What are closures in Python?
    - Closures occur when a nested function retains access to variables from the outer function even after the outer function has finished execution.

60. How does Python implement method resolution order (MRO)?
    - MRO is the order in which Python looks for methods in the class hierarchy, specifically in multiple inheritance. It follows the C3 linearization algorithm to determine the method lookup order.

</WEB
DEVELOPMENT with
python, Django & React

Batch 11

ক্যারিয়ার পাথে এনরোল করতে স্ক্যান করুন