Python Fundamentals

(theory)

-Harsh Chauhan

Introduction to Python Theory:

1). Introduction to Python and its Features (simple, high-level, interpreted language).

- Python is a simple and easy-to-learn programming language with clear syntax similar to English.

- It is a high-level language, meaning it abstracts complex computer details, making coding easier.

- Python is an interpreted language, so code runs line by line without needing compilation, which helps in quick testing and debugging.

- It is free and open-source, allowing anyone to use and modify it without cost.

- Python supports multiple programming paradigms, especially object-oriented programming, enabling reusable and organized code.

- It has a large standard library with many built-in modules to perform various tasks without writing code from scratch.

- Python is dynamically typed, so variable types are determined at runtime, making programming more flexible.

- It is cross-platform and works on Windows, macOS, Linux, and more without needing code changes.

- Python's readable and indented code structure makes it beginner-friendly and easy to maintain.

- It has strong community support, providing many tutorials, resources, and third-party libraries for all kinds of projects.

2).History and evolution of Python.

- Python was created by Guido van Rossum in the late 1980s as a hobby project during his holiday in December 1989.

- It was designed to be a successor to the ABC programming language, with a focus on code readability and simplicity.

- The name "Python" comes from the British comedy series "Monty Python's Flying Circus," reflecting the creator's sense of fun.

- The first public version, Python 0.9.0, was released in February 1991, featuring core elements like functions, exception handling, classes, and basic data types.

- Python 1.0 was officially released in 1994, introducing important features such as lambda functions, map/filter/reduce, and the foundation for object-oriented programming.

- Python 2.0 came out in 2000, adding list comprehensions, garbage collection, Unicode support, and other language improvements.

- Python 3.0, released in 2008, was a major update that made the language more consistent and modern but was not backward compatible with Python 2.x.

- Guido van Rossum led Python's development as its "Benevolent Dictator for Life" until he stepped down in 2018.

- Today, Python continues to evolve with strong community support, widely used across web development, data science, AI, and more

3). Advantages of using Python over other programming languages.

- Easy to learn and use due to its simple and human-readable syntax.

- Reduces development time with fewer lines of code compared to languages like Java or C++.

- Vast standard libraries and frameworks support many tasks, such as web development, data science, and machine learning.

- Cross-platform compatibility allows Python programs to run on Windows, macOS, Linux, and more without changes.

- Supports multiple programming styles including object-oriented, procedural, and functional programming.

- Strong community support with abundant resources, tutorials, and third-party packages available.

- Highly extensible and can integrate with other languages like C, C++, and Java.

- Automatic memory management via garbage collection helps avoid memory leaks and errors.

- Flexible deployment options from desktop to cloud environments.
- Ideal for rapid development, prototyping, and iterative testing due to its interpreted nature.
- Open-source and free to use, reducing costs for individual developers and organizations.
- Widely adopted in emerging fields like artificial intelligence, machine learning, and data analytics.

4). Installing Python and setting up the development environment (Anaconda, PyCharm, or VS Code).

- Visit the official Python website (python.org) and download the latest Python installer for your operating system (Windows/macOS/Linux).

- Run the installer and make sure to check the "Add Python to PATH" option before clicking Install. This allows Python to be accessible from the command line.

- After installation, verify it by opening the command prompt (or terminal) and typing `python --version` to see the installed Python version.

- Download and install Visual Studio Code (VS Code) from its official website (code.visualstudio.com). It is a lightweight, free code editor.

- Open VS Code and install the Python extension by Microsoft from the Extensions marketplace (search for "Python" and install it).

- Open a new folder or workspace in VS Code where you want to create Python projects.

- To run Python code, open a new file with `.py` extension, write your Python script, and save it.

- Use the terminal inside VS Code (View -> Terminal) to run Python scripts by typing python filename.py.

- You can also use the Run and Debug feature in VS Code to execute and debug Python programs easily.

- Customize your Python interpreter in VS Code by clicking on the interpreter selection button at the bottom-left and selecting the installed Python path if there are multiple versions

5).Writing and executing your first Python program.

Here is a simple guide in points for writing and executing your first Python program:

- Open your code editor (like VS Code) or a Python interactive shell (IDLE).

- Create a new file and save it with a .py extension, for example, hello.py.

- Write your first Python code, for example:

```
print('Yoooooo!')
```

- Save the file after writing the code.

- To run the program, open the terminal or command prompt.

- Navigate to the folder where you saved your Python file using the cd command.

- Type python hello.py and press Enter to execute the program.

- You should see the output:

```
>>> print('Yoooooo!')
Yoooooo!
>>>
```

- Alternatively, in VS Code, you can run the program by right-clicking the file and selecting "Run Python File" or clicking the run button at the top right.

2. Programming Style

1). Understanding Python’s PEP 8 guidelines.

Key points about PEP 8 are:

- It provides coding conventions to write clean, readable, and consistent Python code.

- Created in 2001 by Guido van Rossum and others to improve Python code quality.

- Covers naming conventions, indentation (4 spaces per level), line length (max 79-88 characters), whitespace usage, and comments.

- Encourages readable code by defining how code blocks, functions, and classes should be formatted.

- Recommends using spaces over tabs and proper spacing around operators and commas.

- Suggests guidelines for inline comments, block comments, and docstrings for better code documentation. Helps improve collaboration and maintainability by making code look uniform regardless of who writes it.
- Following PEP 8 makes Python code easier to understand and reduces bugs caused by unclear structure.

2). Indentation, comments, and naming conventions in Python.

Indentation:

- Use 4 spaces per indentation level; spaces are preferred over tabs.
- Do not mix tabs and spaces; Python 3 disallows mixing them.
- Indentation defines code blocks (e.g., inside functions, loops, conditions), essential for Python syntax.
- Continuation lines can use hanging indent aligned with opening delimiter or an extra indentation level for clarity.

Comments:

- Use comments to explain the purpose of code, not obvious details.
- Inline comments should be separated by at least two spaces from code, start with a # and a single space.
- Block comments are full lines of comments usually preceding code blocks.
- Use docstrings (triple quotes """) to document modules, classes, and functions.

Naming Conventions:

- Use lowercase words separated by underscores for functions and variable names (e.g., my_variable).
- Use CapitalizedWords (PascalCase) for class names (e.g., MyClass).
- Constants are written in all uppercase with underscores (e.g., MAX_LIMIT).
- Avoid single-character names except for counters or iterators (e.g., i, n).
- Follow consistent and descriptive names for readability and maintainability.

3). Writing readable and maintainable code.

- Follow consistent naming conventions for variables, functions, classes, and constants as per PEP 8.
- Use meaningful and descriptive names that clearly communicate the purpose of the code element.
- Keep functions and classes small and focused on a single task to enhance clarity and reusability.
- Use comments and docstrings to explain why the code does something, not what it does, keeping comments concise and useful.
- Organize code logically with proper indentation and spacing for visual clarity.
- Avoid deep nesting by using early return statements and breaking complex logic into smaller functions.
- Write modular code by dividing large programs into smaller, manageable modules or files.
- Follow the DRY principle (Don't Repeat Yourself) to reduce redundancy by reusing functions and components.
- Use error handling and exceptions properly to make the code robust and easier to debug.

- Keep the code clean by removing unused variables, imports, and debugging statements before finalizing.

3. Core Python Concepts

1). Understanding data types: integers, floats, strings, lists, tuples, dictionaries, sets.

- Integers (int): Whole numbers without decimals, e.g., 5, -10, 100. Used for counting or indexing.

- Floats (float): Numbers with decimal points, e.g., 3.14, -0.001, 2.0. Used for precise measurements or calculations.

- Strings (str): Sequence of characters enclosed in quotes, e.g., "Hello", 'Python'. Used to represent text.

- Lists: Ordered, mutable collections enclosed in square brackets, e.g.,, ["apple", "banana"]. Lists can hold mixed data types.

- Tuples: Ordered, immutable collections enclosed in parentheses, e.g., (1, 2, 3), ("a", "b"). Used when data should not change.

- Dictionaries (dict): Unordered collections of key-value pairs enclosed in curly braces, e.g., {"name": "John", "age": 25}. Keys must be unique.

- Sets: Unordered collections of unique elements enclosed in curly braces, e.g., {1, 2, 3}. Used to store distinct items and perform set operations.

2). Python variables and memory allocation.

- In Python, a variable is a name or label that refers to an object stored in memory, not a fixed memory location like in some languages.

- When a variable is assigned a value, Python creates an object for that value in the heap memory and the variable stores a reference (pointer) to that object.

- Python uses dynamic typing, so the type of the object is determined at runtime and the same variable can be reassigned to objects of different types.

- Memory for objects is managed automatically by Python's private heap and a specialized allocator called pymalloc for small objects (under 256 bytes).

- The memory size of an object depends on its type and contents; for example, an integer might take more memory than in low-level languages because Python stores extra information with the object.

- Python performs automatic memory management using reference counting to track object usage, and garbage collection to clean up unreferenced objects.

- Immutable objects like strings and tuples cannot be changed after creation; any change creates a new object in memory.

- Mutable objects like lists and dictionaries can change their content without changing their identity, and their memory can grow or shrink dynamically as needed.

- This memory model simplifies programming but may use more memory than static languages, trading off performance for ease of use and flexibility.

3). Python operators: arithmetic, comparison, logical, bitwise.

Arithmetic Operators: Perform mathematical calculations.

- + Addition (e.g., 5 + 3 = 8)
- - Subtraction (e.g., 5 - 3 = 2)
- * Multiplication (e.g., 5 * 3 = 15)
- / Division (e.g., 5 / 3 = 1.666...)
- // Floor Division (e.g., 5 // 3 = 1)
- % Modulus (remainder) (e.g., 5 % 3 = 2)
- ** Exponentiation (e.g., 5 ** 3 = 125)

Comparison Operators: Compare values and return True or False.

- == Equal (5 == 3 is False)
- != Not equal (5 != 3 is True)
- > Greater than (5 > 3 is True)
- < Less than (5 < 3 is False)
- >= Greater than or equal (5 >= 3 is True)
- <= Less than or equal (5 <= 3 is False)

Logical Operators: Combine conditional statements.

- and Returns True if both conditions are true (e.g., True and False is False)
- or Returns True if at least one condition is true (e.g., True or False is True)
- not Negates the condition (e.g., not True is False)

Bitwise Operators: Work on bits of numbers.

- & AND (sets bit to 1 if both bits are 1)
- | OR (sets bit to 1 if one of the bits is 1)
- ^ XOR (sets bit to 1 if only one bit is 1)
- ~ NOT (inverts bits)
- << Left shift (shifts bits to the left)
- >> Right shift (shifts bits to the right)

4. Conditional Statements

1). Introduction to conditional statements: if, else, elif.

- if statement: Used to execute a block of code only if a specified condition is true.
  Syntax:

```
if condition:
    # code to execute if condition is true
```

- else statement: Used with if to execute a block of code when the if condition is false.
  Syntax:

```
if condition:
    # code if true
else:
    # code if false
```

- elif statement: Short for "else if," used to check multiple conditions sequentially after the initial if. Only the first true condition block executes.

Syntax:

```
if condition1:
    # code if condition1 is true
elif condition2:
    # code if condition2 is true
else:
    # code if no conditions are true
```

- These control structures allow the program to make decisions and branch logic based on conditions.
- Conditions typically involve comparison or logical operators and evaluate to True or False.
- Python evaluates conditions in order; once a true condition is found, the rest are skipped.

Example:

```
number = 0
if number > 0:
    print("Positive")
elif number < 0:
    print("Negative")
else:
    print("Zero")
```

-This will print "Zero" since the number is 0.

2). Nested if-else conditions.

- Nested if-else means placing an if-else statement inside another if or else block to check multiple layers of conditions.
- It allows more detailed decision-making by evaluating a secondary condition only if the first condition is true (or false).
- Syntax example:

python
```
if condition1:
    if condition2:
        # Executes if both condition1 and condition2 are true
    else:
        # Executes if condition1 is true but condition2 is false
else:
    # Executes if condition1 is false
```

- Example code:

python
```
num = 10
if num > 0:
    if num % 2 == 0:
        print("The number is positive and even.")
    else:
        print("The number is positive but odd.")
else:
    print("The number is not positive.")
```

- This structure helps in checking complex criteria step-by-step and managing different outcomes accordingly.

- Be careful with indentation as Python uses it to define the scope of each if-else block.

5. Looping (For, While)

1). Introduction to for and while loops.

For Loop:

- Used to iterate over a sequence (like a list, tuple, string, or range) a specified number of times.
  Syntax:

```
for variable in sequence:
   # code block to execute
```

- Iterates through each item in the sequence one by one.
  Example:

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

- Can also loop over numbers using `range()`, e.g., `for i in range(5):` loops from 0 to 4.

While Loop:

- Repeats a block of code as long as a specified condition is true.

Syntax:

```
while condition:
    # code block to execute
```

- Useful when the number of iterations is not known beforehand.
  Example:

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

- Make sure to update variables inside the loop to avoid infinite loops.

2). How loops work in Python.

Loops in Python work by repeatedly executing a block of code as long as a certain condition is true or until a sequence is fully processed.

- For loops iterate over elements of a sequence one by one (like items in a list or characters in a string). The loop variable takes each value in the sequence in order, and the code inside the loop runs once per item. After completing all items, the loop stops.
- While loops repeatedly execute a code block while a condition remains true. Before each iteration, the condition is checked; if true, the loop runs, otherwise it ends.
- The loop control keywords break and continue can be used to exit the loop early or skip the current iteration, respectively.
- When the loop condition becomes false (in while) or all items have been processed (in for), control passes to the code immediately after the loop body.
- Loops allow automation of repetitive tasks by executing the same statements multiple times without writing them repeatedly.
- Proper indentation defines the scope of the loop body in Python, making block structure clear and avoiding syntax errors.

3). Using loops with collections (lists, tuples, etc.).

- Python's for loop is ideal for iterating directly over elements in collections such as lists or tuples. You do not need to manually use indices.
  Example (loop through a list):

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

- This prints every fruit in the list one by one.
- Similarly, you can loop through a tuple directly:

```
thistuple = ("apple", "banana", "cherry")
for item in thistuple:
   print(item)
```

- You can also loop using indices with `range(len(collection))` if you need the index for some reason:

```
for i in range(len(fruits)):
   print(fruits[i])
```

- While loops can be used with collections by manually managing an index variable:

```
i = 0
while i < len(fruits):
    print(fruits[i])
    i += 1
```

- The enumerate() function can be used in for loops to get both index and value together:

```
for index, fruit in enumerate(fruits):
    print(index, fruit)
```

- Loops with collections enable efficient processing and manipulation of grouped data without writing repetitive code.

6. Generators and Iterators

1). Understanding how generators work in Python.

- A generator is a special type of function that returns an iterator object which produces a sequence of values one at a time, instead of returning them all at once.

- Generators use the yield keyword instead of return. When yield is called, it pauses the function and sends a value back to the caller, saving its state.

- Execution can be resumed later to produce the next value, making generators memory-efficient as they generate items on demand rather than storing them all.

- Generator functions do not run immediately when called; they return a generator object which can be iterated over with a loop or the next() function.

- Generator expressions are a concise way to create generators using a syntax similar to list comprehensions but with parentheses.

- They are useful for handling large datasets, streams, or infinite sequences where creating a full list in memory is costly.

2). Difference between yield and return.

- Purpose:
  - yield is used to create a generator, which can produce a sequence of values lazily, one at a time.
  - return is used to exit a function and send a value back to its caller immediately.

- Execution:
  - yield pauses the function execution, saving its state, and resumes from there when called again. It can produce multiple values over time.
  - return terminates the function execution completely; code after return is not executed.

- Number of times executed:
  - A yield statement can be executed multiple times as the generator is iterated.
  - A return statement executes only once per function call.

- Memory usage:
  - yield is memory efficient because it generates values on the fly without building the entire list in memory.

  - return typically returns a complete set of results, potentially using more memory.
- <u>Code behavior:</u>
  - Code after yield runs the next time the generator resumes.
  - Code after return never runs in that function call.

➔Example:

```
def gen():
    yield 1
    yield 2

def func():
    return 1
    return 2
```

➔gen() produces 1 and 2 over multiple calls, func() returns 1 and exits immediately.

➔In summary, yield is used for creating iterators/generators that produce a series of values lazily, while return immediately exits a function and returns a single value to the caller.

3). Understanding iterators and creating custom iterators.

Understanding iterators and creating custom iterators in Python:

- An iterator is an object in Python that lets you traverse through all elements of a collection (like lists, tuples, dictionaries, etc.) one element at a time.
- Iterators follow the iterator protocol, requiring two special methods:
  - __iter__() which returns the iterator object itself.
  - __next__() which returns the next item from the sequence and raises a StopIteration exception when no more items are available.
- You create an iterator by calling the built-in iter() function on an iterable (e.g., a list or tuple).
- The next() function is used to get successive elements from the iterator.
- After consuming all elements, calling next() will raise StopIteration to signal the end.

Creating a custom iterator involves defining a class with __iter__() and __next__() methods.

Example:

```
class MyNumbers:
    def __iter__(self):
        self.num = 1
        return self

    def __next__(self):
        if self.num <= 5:
            current = self.num
            self.num += 1
            return current
        else:
            raise StopIteration

my_iter = iter(MyNumbers())

for number in my_iter:
    print(number)
```

- This will output numbers from 1 to 5, demonstrating a simple custom iterator.
- Iterators are memory efficient as they yield one item at a time, making them suitable for large datasets or streams

7. Functions and Methods

1). Defining and calling functions in Python.

Here is a simple, point-wise guide to defining and calling functions in Python:

- Functions in Python are blocks of code designed to perform a specific task and promote code reuse.
- Define a function using the def keyword, followed by the function name and parentheses.
- Any input values (parameters) are placed inside the parentheses.
- The function body is written below the definition and must be indented.
- Optionally, use the return statement to send a value back to the caller.

Example – Defining a function:

python
```python
def greet():
    print("Hello from a function!")
```

Example – Calling a function:

python
```python
greet()  # Output: Hello from a function!
```

Function with parameters:

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

result = add(3, 5)  # result is 8
```

- Calling a function uses its name with parentheses, passing arguments if required.
- Functions help avoid code repetition and make code easier to manage.

2). Function arguments (positional, keyword, default).

Here is a simple, point-wise explanation of function arguments in Python:

- Positional arguments:
  - Values are matched to parameters based on their position (order) in the function call.

Example:

```
def add(a, b):
    return a + b
result = add(2, 3)   # a=2, b=3
```

- Keyword arguments:
  - Values are given by explicitly naming the parameter in the function call, regardless of their position.

Example:

```
def greet(name, msg):
    print(name, msg)
greet(name="Alice", msg="Hello")  # order doesn't matter
```

- Default arguments:
  - Parameters can have default values, which are used if no value is provided in the call.

Example:

```
def greet(name, msg="Hi"):
```

```
    print(name, msg)
greet("Bob")         # Output: Bob Hi (uses default msg)
greet("Bob", "Hey")  # Output: Bob Hey (overrides default)
```

- You can mix these, but positional arguments must always come before keyword arguments in the call.

3). Scope of variables in Python.

- Scope refers to the region of a program where a variable is recognized and can be accessed.
- Python follows the LEGB rule to resolve variable names in this order:
  - Local: Variables defined inside the current function.
  - Enclosing: Variables in enclosing (outer but non-global) functions in case of nested functions.
  - Global: Variables defined at the top level of a script or module, outside all functions.
  - Built-in: Names preassigned by Python (like print, len, etc.).
- A variable created inside a function is local and accessible only within that function.
- A variable declared outside any function is global and can be accessed throughout the module.
- Nested (enclosing) functions can use variables from their immediately surrounding function using the nonlocal keyword.
- You can use the global keyword inside a function to modify a global variable.
- There is no block scope in Python variables defined inside an if, for, or while block remain accessible outside that block within the same function or module.

4). Built-in methods for strings, lists, etc.

Common Built-in String Methods:

- upper(): Converts the entire string to uppercase.
- lower(): Converts the entire string to lowercase.
- capitalize(): Capitalizes the first character of the string.
- strip(): Removes leading and trailing whitespace.
- split(separator): Splits the string into a list based on the separator (default is space).
- join(iterable): Joins elements of an iterable into a single string with the string as a separator.
- replace(old, new): Replaces occurrences of a substring with another substring.
- find(substring): Returns the lowest index of the substring, or -1 if not found.
- startswith(prefix): Returns True if the string starts with the specified prefix.
- endswith(suffix): Returns True if the string ends with the specified suffix.
- count(substring): Counts occurrences of a substring in the string.

Common Built-in List Methods:

- append(item): Adds an item to the end of the list.
- insert(index, item): Inserts an item at the specified index.
- remove(item): Removes the first occurrence of an item.
- pop(index): Removes and returns the item at the given index (default last element).
- sort(): Sorts the list in ascending order.

- reverse(): Reverses the elements of the list.
- index(item): Returns the index of the first occurrence of the item.
- count(item): Counts how many times an item appears in the list.
- extend(iterable): Extends the list by appending elements from the iterable.

8. Control Statements (Break, Continue, Pass)

1). Understanding the role of break, continue, and pass in Python loops.

- break:
  - Terminates the loop immediately when executed.
  - Control moves to the first statement after the loop.
  - Useful for exiting a loop early when a condition is met, such as finding a target item.
  - Only breaks out of the innermost loop if nested loops exist.
- continue:
  - Skips the rest of the current loop iteration and moves to the next iteration.
  - Useful for bypassing specific cases without stopping the loop entirely.
  - Helps avoid deeply nested conditionals and keeps code cleaner.

- pass:
  - Does nothing; it's a placeholder where Python syntax requires a statement but no action is needed.
  - Often used in loops, functions, or conditionals during development as a temporary stand-in.

    - Allows the code to run without errors before the actual code is written

9. String Manipulation

1). Understanding how to access and manipulate strings.

Accessing Strings:

- You can access individual characters in a string using indexing:

```
text = "Hello"
print(text[0])  # Output: H
```

- Negative indices count from the end:

```
print(text[-1])  # Output: o
```

Manipulating Strings (using built-in methods):

- upper(): Converts string to uppercase.
- lower(): Converts string to lowercase.
- capitalize(): Capitalizes the first letter.
- strip(): Removes leading/trailing whitespace.
- replace(old, new): Replaces occurrences of a substring.
- find(sub): Finds the first index of a substring, returns -1 if not found.

- split(separator): Breaks string into a list based on a delimiter.
- join(list): Combines list elements into a string with a separator.
- startswith() / endswith(): Checks start/end of strings.
- format(): Inserts variables into strings for formatting.

2). Basic operations: concatenation, repetition, string methods (upper(), lower(), etc.).

Concatenation:

- Use the + operator to join two or more strings.

```
s1 = "Hello"
s2 = "World"
s3 = s1 + " " + s2  # "Hello World"
```

- You can also use join() to concatenate a list of strings efficiently.

```
words = ["Hello", "Python", "World"]
sentence = " ".join(words)  # "Hello Python World"
```

Repetition:

- Use the * operator to repeat strings multiple times.

```
repeated = "Hi! " * 3  # "Hi! Hi! Hi! "
```

Common String Methods:

- upper(): Converts all characters to uppercase.
- lower(): Converts all characters to lowercase.
- capitalize(): Capitalizes the first letter of the string.
- strip(): Removes leading and trailing whitespace.
- replace(old, new): Replaces occurrences of a substring.

- split(separator): Splits string into a list based on a separator.
- find(substring): Returns the index of the first occurrence of a substring or -1.
- startswith(prefix): Checks if string starts with a prefix.
- endswith(suffix): Checks if string ends with a suffix.

Example Code:

```
text = " hello world "
print(text.upper())      # " HELLO WORLD "
print(text.strip())      # "hello world"
print(text.replace("world", "Python"))  # " hello Python "
print(" ".join(["Python", "is", "fun"]))  # "Python is fun"
print("abc" * 3)         # "abcabcabc"
```

3). String slicing.

- String slicing extracts a portion (substring) from a string using the syntax:
- substring=*s*[start:end:step]
- start: index where slicing starts (inclusive). Default is 0 if omitted.
- end: index where slicing stops (exclusive). Default is end of the string if omitted.
- step: interval between characters. Default is 1. Can be negative to slice backwards.

Examples:

- Get substring from index 0 to 4 (5 characters):

```
s = "Hello, World!"
print(s[0:5])  # Output: Hello
```

- Slice from index 7 to the end:

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

- Slice from start to index 4:

```
print(s[:5])  # Output: Hello
```

- Get every second character between index 1 and 8:

```
print(s[1:8:2])  # Output: el,
```

- Reverse a string:

```
print(s[::-1])  # Output: !dlroW ,olleH
```

- Using negative indices for slicing:

```
print(s[-6:-1])  # Output: World
```

String slicing is useful for getting substrings, reversing strings, skipping characters, and more.

10. Advanced Python (map(), reduce(), filter(), Closures and Decorators)

1). How functional programming works in Python.

➔Functional programming in Python is a programming paradigm where computation is treated as the evaluation of mathematical functions without changing state or mutable data.

Key concepts in Python's functional programming include:

- Pure Functions: Functions that always produce the same output for the same input and do not cause side effects.
- Immutability: Data objects are not modified after creation; new data structures are created instead of altering existing ones.
- First-Class and Higher-Order Functions: Functions are treated as first-class objects that can be passed as arguments, returned from other functions, or assigned to variables.
- Recursion: Functional code often uses recursion instead of loops for iteration.
- Functions like map(), filter(), and reduce() allow applying functions to sequences.
- Lambda functions: Anonymous, concise functions useful for short operations passed as arguments.

➔Python supports functional programming concepts while also supporting imperative and object-oriented styles, allowing hybrid approaches.

2).Using map(), reduce(), and filter() functions for processing data.

Here is an explanation of how to use map(), reduce(), and filter() functions in Python for processing data:

- map():
  - Applies a given function to all items in an iterable (like a list) and returns an iterator with the results.
  - Syntax: map(function, iterable)

Example:

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

  -

- filter():
  - Filters items in an iterable based on a function that returns True or False. Returns only items that satisfy the condition.
  - Syntax: “”””””””””””””””””””””””””’?

Example:

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

- reduce() (from functools module):
  - Applies a rolling computation to sequential pairs of values in an iterable, reducing it to a single value.
  - Syntax: reduce(function, iterable)

Example:

```
from functools import reduce
numbers = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # Output: 24
```

- These functions support concise, readable, and efficient data transformations, especially when combined with lambda functions.
- Common use case examples: summing values, filtering subsets of data, applying transformations to lists.

3). Introduction to closures and decorators.

Closures:

- A closure is a nested (inner) function that remembers and has access to variables from its outer (enclosing) function, even after the outer function has finished execution.
- It allows the inner function to retain the state of the outer function's variables.
- Closures are useful for data hiding and creating function factories or decorators.

Example:

```
def outer(x):
    def inner(y):
        return x + y
    return inner

add5 = outer(5)
print(add5(3))  # Output: 8
```

- Here, inner is a closure that remembers x=5 from outer.

Decorators:

- A decorator is a special type of closure used to modify or enhance functions without changing their code.
- Decorators wrap a function, adding behavior before or after the wrapped function runs.

- They are applied using the @decorator_name syntax above the function to be decorated.

Example:

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

@decorator
def say_hello():
    print("Hello!")

say_hello()
```

- Output:

```
Before function call
Hello!
After function call
```

➔Closures and decorators are powerful tools in Python for flexible, reusable, and clean code.