Python Fundamentals

Introduction to Python:-

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

Ans-

Introduction to Python

Python is a popular programming language that is:

- Easy to learn
- Simple to use
- Used for many things like making websites, apps, games, and data analysis.

It was created by Guido van Rossum and released in 1991.

Features of Python:-

1. Simple and Easy to Learn
    - o Python looks like plain English.
    - o You don't need to write a lot of code to get things done.
    - o Example:

    print ("Hello, World!")
2. High-Level Language
    - o You don't need to worry about how the computer handles memory or hardware.
    - o Python takes care of all that for you.
3. Interpreted Language
    - o Python runs your code line by line.
    - o You don't need to compile (convert) the whole code first.

o This makes it easy to find errors and test code quickly.

4. Free and Open Source

o You can download and use Python for free.

o You can even see or change how Python works.

5. Portable

o Python works on Windows, Mac, Linux – almost any system.

6. Large Library Support

o Python comes with many tools and functions you can use directly.

o Example: math, random, datetime, etc.

7. Used in Many Areas

o Web development (Django, Flask)

o Machine learning (TensorFlow, scikit-learn)

o Data science (Pandas, NumPy)

o Game development (Pygame)

2. History and evolution of Python.

=> History of Python:

1. Created by Guido van Rossum

- Year: Late 1980s (officially released in 1991)
- Why? Guido wanted a language that was:

o Easy to read

o Simple to write

o Powerful and flexible

2. Name Origin

- Not named after a snake!

It's named after the comedy group "Monty Python's Flying Circus".

Evolution of Python Versions:

Python 1.0 – Released in 1991

- First official version.
- Included basic features like: functions, exceptions, and core data types (lists, dictionaries.

Python 2.0 – Released in 2000

- Added new features like:
  - o Garbage collection
  - o Unicode support
- BUT: Not fully compatible with Python 3 (this caused confusion later)

Python 3.0 – Released in 2008

- A complete upgrade of the language.
- Not backward compatible with Python 2.
- Fixed many design flaws of Python 2.
- Now the main version used today.

Modern Updates (Python 3.x):

- New versions keep coming (e.g., 3.6, 3.8, 3.10, 3.11, 3.12…).
- Each version adds better performance and cool features like:
  - o f-strings (for easier printing)
  - o Type hinting
  - o Pattern matching
  - o Async programming

Why Python Became Popular

- Easy to learn and use
- Huge community and libraries
- Used in many fields (AI, web, automation, etc.)
- Clean and readable syntax (like English)

3. Advantages of using Python over other programming languages

=>

1. Easy to Read and Write

- Python has simple syntax, like plain English.
- This makes it easy for beginners to learn and use.

Example:

```
Print ("Hello, World!")
```

Compare this to Java:

```
public class HelloWorld {
public static void main (String [] args) {
System.out.println("Hello, World!");}
}
```

2. Interpreted Language

- No need to compile code; it runs line by line.
- Easier to debug and test.

3. Large Standard Library

- Python comes with many built-in modules for:
  - o File handling
  - o Web development
  - o Data processing
  - o Math operations
  - o Networking, etc.

4. Cross-Platform

- Python works on Windows, macOS, Linux, etc.
- Write code once and run it anywhere.

5. Versatile

- Used in many fields:
  - o Web Development (e.g., Django, Flask)

- Data Science (e.g., Pandas, NumPy)
- Machine Learning (e.g., TensorFlow, Scikit-learn)
- Automation/Scripting
- Game Development (e.g., Py game)

6. Strong Community Support

- Huge community = plenty of help, tutorials, and libraries.
- Active forums like Stack Overflow and GitHub.

7. Supports Object-Oriented and Functional Programming

- You can use Python for both OOP and functional styles.
- Flexible for different types of projects.

8. Integration Capabilities

- Easily integrates with:
  - C/C++ code
  - Java (via Jython)
  - .NET (via Iron Python)
  - Databases like MySQL, PostgreSQL, SQLite

9. Popular in Data Science and AI

- Preferred language for AI, ML, and data analytics.
- Major tools are built around Python.

10. Free and Open-Source

- Python is completely free to use and distribute.
- Open-source means anyone can improve it.

4.Understanding Python's PEP 8 guidelines

=>

PEP 8 stands for Python Enhancement Proposal 8. It is the style guide for writing clean, readable, and consistent Python code. It was created by Guido van Rossum (the creator of Python) and others to help developers write Python code in a standardized way.

Key Guidelines of PEP 8

1. Indentation

- Use 4 spaces per indentation level (not tabs).

```
def greet():
    print("Hello")
```

2. Maximum Line Length

- Keep lines under 79 characters.

```
# Good
name = "John"
# Bad
name = "John is a very talented programmer who works at a software
company"
```

3. Blank Lines

- Use blank lines to separate functions and classes, and larger blocks of code inside functions.

4. Imports

- Imports should be:

1. Standard libraries
2. Third-party libraries
3. Local application imports

(Each group separated by a blank line)

```
import os
import sys
import requests
import my_module
```

5. Spaces in Expressions

- Use spaces around operators, but not after opening or before closing parentheses.

```
# Good
x = (a + b) * c
# Bad
x=(a+b)*c
```

6. Function and Variable Naming

- Use lowercase_with_underscores for functions and variable names.

```
def calculate_area():
    pass
```

7. Class Naming

- Use CamelCase for class names.

```
class BankAccount:
    pass
```

8. Constants

- Constants should be written in UPPERCASE_WITH_UNDERSCORES.

```
MAX_SPEED = 100
```

9. Docstrings

- Use triple quotes """ to describe modules, functions, classes, and methods.

```
def add(x, y):
"""Add two numbers and return the result."""
return x + y
```

Why Follow PEP 8?

- Makes your code more readable
- Helps teams collaborate more easily
- Increases code quality and maintainability
- Helps in professional Python development

5. Indentation, comments, and naming conventions in Python.

=>

1. Indentation in Python

Python uses indentation (whitespace) to define code blocks, instead of curly braces {} like in other languages.

- Rules:
  - Use 4 spaces per indentation level (not tabs).
  - Required after:
    - o if, else, elif
    - o for, while
    - o def, class
    - o try, except, etc.

Example:

```
def greet(name):
if name:
print ("Hello", name)
else:
print ("Hello, stranger")
```

2. Comments in Python

Comments help explain your code. Python ignores them during execution.

Types:

- Single-line comment: Use #
- Multi-line (docstring): Use triple quotes """ """ or ''' '''

Examples:

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

```
This is a multi-line comment
or docstring used for documentation.
"""
def add (a, b):
"""Returns the sum of a and b."""
return a + b
```

Tips:

- Keep comments short and relevant.
- Don’t write obvious comments.

3. Naming Conventions

Use meaningful names and follow snake case or CamelCase

depending on the type.

Type Convention Example

```
Variable snake_case user_name, score
Function snake_case get_data()
Class CamelCase StudentProfile
Constant UPPER_CASE MAX_LIMIT
Module/Script snake_case.py my_script.py
```

Bad Examples:

```
a = 10 # too short
def fun (): # unclear
pass
```

Good Examples:

```
max_students = 30
def calculate_grade(score):
...
```

6. Writing readable and maintainable code.

1. Follow Naming Conventions

- Use meaningful names:

  total_price = quantity * unit_price

- Follow snake_case for variables and functions in Python:

  def calculate_total():

  ...

2. Keep Code Simple and Clear

- Write simple logic that's easy to understand. Avoid clever one-liners that are hard to read.

  # Good

  if user_age >= 18:

  print ("Eligible to vote")

  # Avoid

  print ("Eligible to vote" if user_age >= 18 else "")

3. Use Comments Wisely

- Explain why something is done, not what is done (if it's already clear).

  # Converting price to float to ensure correct calculation

  price = float(price_string)

4. Break Code into Functions

- Avoid long blocks of code; use functions to organize logic.

  def calculate_discount(price):

  ...

5. Consistent Indentation and Spacing

- Stick to 4 spaces per indentation level (Python standard).

  for i in range(5):

  print(i)

6. Avoid Repetition (DRY Principle)

- DRY = Don't Repeat Yourself

  def greet(name):

  print ("Hello, {name}!")

  greet("Alice")

  greet("Bob")

7. Use Constants for Fixed Values

- Makes the code more maintainable:

  TAX_RATE = 0.18

  final_price = price + (price * TAX_RATE)

8. Add Docstrings to Functions

- Help others (and your future self) understand your code:

  def add(a, b):

  """

  Returns the sum of a and b.

  """

  return a + b

9. Handle Errors Gracefully

- Use try-except blocks for error-prone code:

  try:

  result = int(user_input)

  except ValueError:

  print("Please enter a valid number.")

10. Use Version Control (like Git)

- Makes collaboration and rollback easier.

7. Understanding data types: integers, floats,

strings, lists, tuples, dictionaries, sets.

=>

1. Integers (int)

- What: Whole numbers (no decimal).
- Examples: 5, -10, 0

age = 25

2. Floats (float)

- What: Numbers with decimal points.
- Examples: 3.14, -0.5, 10.0

price = 19.99

3. Strings (str)

- What: Text (a sequence of characters).
- Examples: "Hello", 'Python', "123" (even though it looks like a

number)

name = "Alice"

4. Lists (list)

- What: Ordered, changeable (mutable) collection.
- Use: Can store multiple values (any type).
- Examples: [1, 2, 3], ["apple", "banana", "cherry"]

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

fruits [0] = "mango" # Lists can be changed

5. Tuples (tuple)

- What: Ordered but unchangeable (immutable) collection.
- Use: Similar to lists but cannot modify after creation.
- Examples: (1, 2, 3), ("red", "green", "blue")

colors = ("red", "green", "blue")

6. Dictionaries (dict)

- What: Key-value pairs, unordered.
- Use: Store data with unique keys.
- Examples:

person = {

"name": "Alice",

"age": 30,

"city": "New York"

}

print(person["name"]) # Output: Alice

7. Sets (set)

- What: Unordered collection with unique values.
- Use: Removes duplicates automatically.
- Examples:

  my_set = {1, 2, 3, 3, 4}

  print(my_set) # Output: {1, 2, 3, 4}

8. Python variables and memory allocation.

=>

What is a Variable in Python?

A variable is a name that refers to a value stored in memory.

x = 10

name = "Alice"

- x is a variable holding the integer value 10.
- name is a variable holding the string "Alice".

i. How Does Memory Allocation Work in Python?

When you create a variable:

1. Python creates an object (e.g., an integer, string, list, etc.).

2. It stores that object in memory.

3. The variable name is a reference (pointer) to the memory address where the object is stored.

Example:

a = 5

b = a

- Python creates an integer object 5 and stores it in memory.
- a refers to that memory location.
- Then b = a makes b point to the same object in memory.

ii. Object Identity (Memory Address)

You can check where a variable points in memory using the id()

function:

```
x = 100
print(id(x)) # Shows memory location of the object
```

If two variables point to the same object:

```
a = 5
b = 5
print(id(a) == id(b)) # True, both point to same object (because of
integer caching)
```

iii. Mutable vs Immutable Types and Memory

Type	Examples	Memory Behaviour
Immutable	int, float, string, tuple	Cannot change value once created
Mutable	list, dict, set	Can change contents without changing identity

```
x = [1, 2, 3]
y = x
y.append(4)
print(x) # [1, 2, 3, 4] – x also changes because y points to the same
list
```

iv. Python Memory Management

Python uses:

- Automatic Garbage Collection: Frees memory when an object is no longer referenced.
- Reference Counting: Tracks how many variables refer to an object.
- Interning: Reuses common objects (like small integers and short strings) to save memory.

9. Python operators: arithmetic, comparison,

logical, bitwise.

=>

1. Arithmetic Operators

Used to perform mathematical operations:

Operator Description Example Result

+ Addition 5 + 3 8

Operator Description Example Result

- Subtraction 5 - 3 2

* Multiplication 5 * 3 15

/ Division 5 / 2 2.5

// Floor Division 5 // 2 2

% Modulus (remainder) 5 % 2 1

** Exponentiation 2 ** 3 8

2. Comparison Operators

Used to compare two values:

Operator Description Example Result

== Equal to 5 == 5 True

!= Not equal to 5 != 3 True

> Greater than 5 > 3 True

< Less than 5 < 3 False

>= Greater than or equal to 5 >= 5 True

<= Less than or equal to 3 <= 5 True

3. Logical Operators

Used to combine conditional statements:

Operator Description Example Result

and True if both are true True and False False

or True if at least one is true True or False True

Operator Description Example Result

not Inverts the condition not True False

4. Bitwise Operators

Work on bits (binary representation):

Operator Description Example Binary

Example Result

& AND 5 & 3 0101 &

0011 0001 → 1

` ` OR `5 3`

^ XOR 5 ^ 3 0101 ^

0011 0110 → 6

~ NOT (invert

all bits) ~5 ~0101 →

1010

-6 (in 2's

complement)

<< Left shift 5 << 1 0101 →

1010 10

>> Right shift 5 >> 1 0101 →

0010 2

10. Introduction to conditional statements: if, else,

elif.

=>

Conditional statements in Python are used to make decisions in your program. They help your code take different actions based on different conditions.

1. if Statement

   The if statement checks a condition.

   If the condition is True, the code inside the if block runs.

   hon

   x = 10

   if x > 5:

   print("x is greater than 5")

   Explanation: Since x is 10, the condition x > 5 is True, so the message is printed.

2. else Statement

   The else block runs when the if condition is False.

   x = 3

   if x > 5:

   print("x is greater than 5")

   else:

   print("x is 5 or less")

   Explanation: x is 3, which is not greater than 5, so the else part runs.

3. elif (Else If) Statement

   Use elif to check multiple conditions in order.

   Python checks each condition from top to bottom and runs the first True one.

```
x = 5
if x > 5:
print("x is greater than 5")
elif x == 5:
print("x is equal to 5")
else:
print("x is less than 5")
```

Explanation:

- x > 5 is False
- x == 5 is True, so "x is equal to 5" is printed.
- else is skipped because one condition is already true.

11. Introduction to for and while loops.

=> Loops in Python

Loops are used to repeat a block of code multiple times. Python provides two main types of loops:

1. for Loop

   Used when you know how many times you want to repeat something (like going through a list or range).

Syntax:

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

📌 Example:

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

Output:

```
0
1
```

2

3

4

range (5) means it goes from 0 to 4 (not including 5).

2. while Loop

Used when you want to repeat something as long as a condition is true.

Syntax:

while condition:

code block

📌 Example:

i = 0

while i < 5:

print(i)

i += 1

Output:

0

1

2

3

4

Key Differences:

Feature	for Loop	while Loop
Use case	Known number of repetitions	Unknown or conditional

loops

Structure Iterates over a sequence Runs while condition is

True

Control Automatically updates

variable

You must manually

update

Loop Control Statements:

- break: Exit the loop early.
- continue: Skip the current iteration and go to the next.

12. How loops work in Python.

=>

Loops in Python are used to repeat a block of code multiple

times — useful when you want to perform the same task many times

without writing the same code again.

Two Main Types of Loops

1. for loop

Used to iterate over a sequence (like a list, string, range, etc.).

Syntax:

for variable in sequence:

code block

Example:

for i in range (5):

print(i)

Output:

0

1

2

3

4

range (5) generates numbers from 0 to 4.

2. while loop

Runs a block of code as long as a condition is True.

Syntax:

while condition:

code block

Example:

i = 0

while i < 5:

print(i)

i += 1

Output:

0

1

2

3

4

Loop Control Statements

Statement Description

break Exits the loop immediately

continue Skips the current iteration, moves to next

pass Does nothing; used as a placeholder

Example with break:

for i in range (5):

if i == 3:

break

print(i)

Output:

0

1

2

Example with continue:

for i in range (5):

if i == 3:

continue

print(i)

Output:

0

1

2

4

Infinite Loop (Be Careful!)

while True:

print ("This will run forever!")

Use Ctrl + C or close the program to stop it manually.

13. Using loops with collections (lists, tuples,

etc.).

=>

loops are often used to iterate through collections like lists,

tuples, sets, and dictionaries. Here's a simple explanation with

examples:

1. Looping through a List

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

Output:

apple

banana

cherry

2. Looping through a Tuple

```
colours = ("red", "green", "blue")
for colour in colours:
print(colour)
```

Output:

red

green

blue

3. Looping through a Set

Note: Sets do not maintain order.

```
animals = {"dog", "cat", "lion"}
for animal in animals:
print(animal)
```

4. Looping through a Dictionary

Looping through keys:

```
student = {"name": "Alice", "age": 22, "grade": "A"}
for key in student:
print(key)
```

Looping through values:

```
for value in student. Values ():
print(value)
```

Looping through key-value pairs:

for key, value in student. Items ():

print (key, ":", value)

5. Using range () with indexing

Sometimes, you want to loop using the index:

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

for i in range (Len(fruits)):

print (f" Index {i} has {fruits[i]}")

6. While loop with a list

numbers = [10, 20, 30]

i = 0

while i < Len(numbers):

print(numbers[i])

i += 1

14. Understanding how generators work in

Python.

=>

Understanding How Generators Work in Python

Generators are a special type of iterator in Python that let you generate values one at a time using the yield keyword, instead of creating a full list in memory.

1. What is a Generator?
   - A generator produces values lazily—only when needed.
   - Created using:

     o Generator functions (with yield)

     o Generator expressions (like list comprehensions, but with ()) 🛠️

Generator Function Example

def count_up_to(n):

```
    count = 1
    while count <= n:
    yield count
    count += 1
```

2 Using the Generator:

```
gen = count_up_to (5)
for Num in gen:
print (Num)
```

Output:

1

2

3

4

5

Behind the Scenes

- When you call count_up_to (5), it doesn't run the function immediately. It returns a generator object.
- Each call to next(gen) runs the function up to the next yield.
- When there are no more values, a Stop Iteration is raised.

3 Generator Expression Example

```
squares = (x * x for x in range (5))
for Val in squares:
print (Val)
```

Output:

0

1

4

9

16

4 Why Use Generators?

Advantage Description

Memory efficient Doesn’t store all values in memory

Faster startup Starts producing values without computing all

at once

Infinite

sequences Can represent infinite data streams safely

5 Difference from Return

```
def example ():
return 10 # ends the function
def generator ():
yield 10 # pauses the function and can resume later
```

15. Difference between yield and return.

=>

yield (Used in generators):

- Pauses the function and saves its state.
- Returns a value, but the function can be resumed later from where it left off.
- Used when you want to generate a sequence of values one by one (lazy evaluation).
- The function becomes a generator.

Example:

```
def count_up_to(n):
for i in range (1, n+1):
yield i
# Usage:
for Num in count_up_to(3):
print(Num)
```

Output:

1

2

3

return (Used in regular functions):

- Ends the function immediately.
- Returns a single value (or multiple values using a tuple).
- Once return is called, the function is done executing.

Example:

```
def add (a, b):

return a + b

# Usage:

result = add (3, 4)

print(result)
```

Output:

7

Summary of Differences:

Feature	yield	return
Behaviour	Pauses and resumes	Ends the function
Returns	A generator object	A final value
Use case	Iterating large or infinite data	Getting one final result
Memory usage	Efficient (lazy evaluation)	

16. Understanding iterators and creating custom

iterators.

=>

1. Understanding Iterators in Python

▶What is an Iterator?

An iterator is an object that allows you to loop through a collection

(like a list, tuple, etc.), one item at a time.

▶ How does it work?

An iterator must implement two methods:

- __iter__ () → Returns the iterator object itself.
- __next__ () → Returns the next value in the sequence. Raises

Stop Iteration when no more items.

▶ Built-in Iterable Examples:

```
nums = [1, 2, 3]

it = iter(nums) # Creates an iterator object

print(next(it)) # Output: 1

print(next(it)) # Output: 2

print(next(it)) # Output: 3

# next(it) now raises Stop Iteration
```

2. Creating Custom Iterators

▶ Step-by-step:

Let’s create a simple iterator that returns numbers from 1 to 5.

```
class CountToFive:

def __init__(self):

self. Num = 1 # Start value

def __iter__(self):

return self # The object itself is the iterator

def __next__(self):

if self. Num <= 5:
```

current = self. Num

self. Num += 1

return current

else:

raise Stop Iteration # No more items

▶ Using the custom iterator:

counter = CountToFive ()

for number in counter:

print(number)

Output:

1

2

3

4

5

17. Defining and calling functions in Python.

=>

Defining and Calling Functions in Python

In Python, functions are blocks of reusable code that perform a specific task. You can define your own functions using the def keyword and call (use) them by their name.

1. Defining a Function

Syntax:

def function_name(parameters):

code block

return result # optional

Example:

```
def greet(name):
print ("Hello", name)
```

2. Calling a Function

Just use the function's name and pass required arguments.

Example:

```
greet("Alice”) # Output: Hello Alice
```

3. Function with Return Value

Example:

```
def add (a, b):
return a + b
result = add (5, 3)
print ("Sum:", result) # Output: Sum: 8
```

4. Function Without Parameters

Example:

```
def say_hello ():
print ("Hello, world!")
say_hello () # Output: Hello, world!
```

5. Function with Default Parameters

Example:

```
def greet(name="User"):
print ("Hello", name)
greet () # Output: Hello User
greet("Alice”) # Output: Hello Alice
```

6. Function with Multiple Returns

Example:

```
def operations (a, b):
return a + b, a * b
sum_, product = operations (4, 5)
```

print ("Sum:", sum_, "Product:", product)

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

=>

1. Positional Arguments

These are the most common. Values are assigned to parameters in the order they appear.

def greet (name, age):

print (f"Hello {name}, you are {age} years old.")

greet ("Alice", 25) # Positional: name="Alice", age=25

2. Keyword Arguments

Here, you specify the parameter name during the function call. The order doesn't matter.

greet (age=30, name="Bob") # Keyword: name="Bob", age=30

3. Default Arguments

You can assign a default value to a parameter in the function definition. If no value is passed, the default is used.

def greet (name, age=18):

print (f"Hello {name}, you are {age} years old.")

greet("Charlie") # Uses default age=18

greet ("Daisy", 22) # Overrides default

19. Scope of variables in Python.

=>

LEGB Rule (Local, Enclosing, Global, Built-in)

Scope Description Example Location

Local Inside a function or block, accessible only within it

Variable declared

inside a function

Enclosing

In the local scope of an

enclosing function (nested

function case)

Variable in the outer

function of a nested

function

Global

Defined at the top-level of a

script or module, accessible

everywhere

Variable declared

outside functions

Built-in Predefined names in Python

(e.g., Len, print)

Python’s built-in

scope (no user

control)

Example of Scope in Action

```
x = "global"
def outer ():
x = "enclosing"
def inner ():
x = "local"
print ("Inner:", x)
inner ()
```

print ("Outer:", x)

outer ()

print ("Global:", x)

Output:

Inner: local

Outer: enclosing

Global: global

Global and nonlocal Keywords

- global – used to access or modify global variables from within a function.
    - nonlocal – used in nested functions to access variables from the enclosing (outer) function.

x = 10

def modify ():

global x

x = 20

modify ()

print(x) # Output: 20

def outer ():

x = 5

def inner ():

nonlocal x

x = 10

inner ()

print(x)

outer () # Output: 10

20. Built-in methods for strings, lists, etc.

=>

String Methods

Method Description

St. Upper () Converts string to uppercase

St. Lower () Converts string to lowercase

St. Title () Capitalizes each word

St. Strip () Removes leading/trailing whitespace

St. Replace (a, b) Replaces a with b in string

St. Split () Splits string into a list

str. find(x) Returns index of first occurrence of x

St. Count(x) Counts how many times x appears

Method Description

str. starts with (x) Checks if string starts with x

str. endswith(x) Checks if string ends with x

str. is alpha () Checks if all characters are letters

str. isdigit () Checks if all characters are digits

List Methods

Method Description

list. Append(x) Adds x to the end of the list

list. Extend(list) Adds all elements from list to current list

list. Insert (i, x) Inserts x at index i

list. Remove(x) Removes first occurrence of x

list. Pop(i) Removes and returns item at index i (default last)

list. Clear () Removes all elements from the list

list. Index(x) Returns index of first occurrence of x

list. Count(x) Counts number of times x appears

list. Sort () Sorts the list in ascending order

list. Reverse () Reverses the list

list. Copy () Returns a shallow copy of the list

Dictionary Methods

Method Description

dict.get(key) Returns value for key or None

Method Description

dict. keys () Returns a view of all keys

dict. values () Returns a view of all values

dict. items () Returns a view of key-value pairs

dict.

update(other_dict)

Adds key-value pairs from another

dictionary

dict.pop(key) Removes specified key and returns value

dict. clear () Removes all items

Set Methods

Method Description

set. Add(x) Adds element x to the set

set. Remove(x) Removes x; error if not found

set. Discard(x) Removes x if present (no error)

set. Pop () Removes and returns a random element

set. Clear () Removes all elements

set. Union(set2) Returns union of sets

set. Intersection(set2) Returns common elements

set. difference(set2) Returns elements in 1st set but not 2nd

21. Understanding the role of break, continue, and

pass in Python loops.

break – Exit the loop completely

- Use: When you want to stop the loop before it has iterated over all

elements.

- Effect: Terminates the current loop and moves to the next line after the loop.

Example:

for i in range (1, 6):

if i == 3:

break

print(i)

Output:

1

2

Loop stops when i == 3.

continue – Skip the current iteration

- Use: When you want to skip certain iterations but keep the loop running.
- Effect: Skips the rest of the code for that iteration and goes to the next loop cycle.

Example:

for i in range (1, 6):

if i == 3:

continue

print(i)

Output:

1

2

4

5

Skips printing 3 but continues the loop.

pass – Do nothing (placeholder)

- Use: When a statement is syntactically required but you don’t want any action to occur.
- Effect: Does nothing; just a placeholder for future code.

Example:

```
for I in range (1, 6):
if I == 3:
pass
print(I)
```

Output:

1

2

3

4

5

It allows the loop to run as usual; pass simply does nothing.

22. Understanding how to access and manipulate

strings.

=>

Accessing Strings

You can access characters in a string using indexing and slicing.

1. Indexing

Each character in a string has an index starting from 0.

```
text = "Python"
print (text [0]) # Output: P
print (text [3]) # Output: h
```

You can also use negative indexing:

```
print (text [-1]) # Output: n (last character)
```

print (text [-3]) # Output: h

2. Slicing

You can extract parts of a string using slicing:

string [start: end] → includes start index, excludes end index.

print (text [0:3]) # Output: Pyt

print (text [2:]) # Output: thon

print (text [:4]) # Output: Pyth

Manipulating Strings

1. String Concatenation

s1 = "Hello"

s2 = "World"

result = s1 + " " + s2

print(result) # Output: Hello World

2. String Repetition

print ("Hi! " * 3) # Output: Hi! Hi! Hi!

3. Changing Case

s = "Python"

print (s. upper ()) # Output: PYTHON

print (s. lower ()) # Output: python

print (s. capitalize ()) # Output: Python

4. Replacing Substrings

text = "I like Python"

print (text. Replace ("like", "love")) # Output: I love Python

5. Splitting and Joining

sentence = "Python is fun"

words = sentence. Split () # ['Python', 'is', 'fun']

print(words)

```
joined = "-". join(words) # 'Python-is-fun'
print(joined)
```

6. Strip Whitespace

```
text = " Hello "
print (text. Strip ()) # Removes both sides: "Hello"
print (text. Strip ()) # Removes left side
print (text. strip ()) # Removes right side
```

String Check Methods

These return True or False:

```
s = "hello123"
print (s. is alpha ()) # False (contains numbers)
print (s. isdigit ()) # False
print (s. isalnum ()) # True
print (s. starts with("he")) # True
print (s. endswith ("3")) # True
```

Strings Are Immutable

You cannot change a string directly:

```
s = "hello"
# s[0] = "H" ❌ This will cause an error
```

To "change" it, you must create a new string:

```
s = "hello"
s = "H" + s [1:]
print(s) # Output: Hello
```

23. Basic operations: concatenation, repetition,

string methods(upper(), lower(), etc.).

=>

1. Concatenation (+)

- Joins two or more strings together.

```
a = "Hello"
b = "World"
result = a + " " + b
print(result) # Output: Hello World
```

2. Repetition (*)

- Repeats a string multiple time.

```
text = "Hi! "
result = text * 3
print(result) # Output: Hi! Hi! Hi!
```

3. String Methods

upper () – Converts to uppercase

```
name = "python"
print (name. Upper ()) # Output: PYTHON
```

lower () – Converts to lowercase

```
name = "PYTHON"
print (name. Lower ()) # Output: python
```

title () – Capitalizes the first letter of each word

```
sentence = "hello world"
print (sentence. Title ()) # Output: Hello World
```

strip () – Removes spaces from beginning and end

```
data = " hello "
print (data. Strip ()) # Output: "hello"
```

replace () – Replaces part of the string

```
text = "I like Java"
print (text. Replace ("Java", "Python")) # Output: I like Python
```

split () – Splits string into a list

line = "apple, banana, mango"

print (line. Split (",")) # Output: ['apple', 'banana', 'mango']

find () – Finds the index of the first occurrence

text = "hello"

print (text. Find("e")) # Output: 1

Example Putting It All Together:

greeting = "hello"

name = "Alice"

message = (greeting + " " + name + "!"). upper() * 2

print(message)

Output:

HELLO ALICE! HELLO ALICE!

24. String slicing.

=>

String Slicing in Python

String slicing means extracting a portion (substring) of a string using its index values.

Syntax:

string[start: stop:step]

- start: index to start from (inclusive)
- stop: index to stop (exclusive)
- step: how many characters to skip (optional)

Examples

text = "Python Programming"

Expression Result Explanation

text [0:6] 'Python' Characters from index 0 to 5

text [6:] 'Programming' From index 6 to end

text [:6] 'Python' From start to index 5

text [:2] 'Pto rgamn' Every 2nd character

text [: -1] 'gnimmargorPnohtyP' Reversed string

text [-1] 'g' Last character

text [-3:] 'ing' Last 3 characters

Important Points

- Indexing starts from 0
- Negative indices count from the end
- Slicing never raises an error even if the index is out of range

25. How functional programming works in Python.

=>

Key Concepts of Functional Programming in Python

1. Pure Functions

Functions that:

o Always return the same output for the same input.

o Do not change any state or data outside the function.

def add (a, b):

return a + b

2. First-Class and Higher-Order Functions

o Functions can be passed as arguments, returned from other

functions, and assigned to variables.

def square(x):

return x * x

def apply_func (f, value):

return f(value)

result = apply_func (square, 5) # Output: 25

3. Lambda Functions (Anonymous Functions)

Small one-line functions using lambda.

square = lambda x: x * x

print (square (4)) # Output: 16

4. Map, Filter, and Reduce

o map (): Applies a function to every item in an iterable.

o filter (): Filters items in an iterable based on a condition.

o reduce (): Applies a function cumulatively to the items

(needs fun tools).

nums = [1, 2, 3, 4]

from fun tools import reduce

print (list (map (lambda x: x * 2, nums))) # [2, 4, 6, 8]

print (list (filter (lambda x: x % 2 == 0, nums))) # [2, 4]

print (reduce (lambda x, y: x + y, nums)) # 10

5. Recursion Instead of Loops

Functional programming prefers recursion over iteration (though Python

has limits on recursion depth).

def factorial(n):

if n == 0:

return 1

return n * factorial (n - 1)

6. Immutability

Avoid changing objects. Use tuples instead of lists, or create new values

instead of modifying existing ones.

When to Use Functional Programming in Python?

- When you want clean, concise, and reusable code.
- When working with data transformations (e.g., in data analysis, pipelines).
- When using parallelism/concurrency (pure functions make it safer).

26. Using map (), reduce (), and filter() functions for

processing data.

=>

1. map (function, iterable)

- Purpose: Applies a function to each item in an iterable.
- Returns: A map object (which is an iterator).

🔹 Example:

```
numbers = [1, 2, 3, 4]

squared = list (map (lambda x: x**2, numbers))

print(squared) # Output: [1, 4, 9, 16]
```

2. filter (function, iterable)

- Purpose: Filters items based on a condition (function returns True or False).
- Returns: A filter object (which is an iterator).

🔹 Example:

```
numbers = [1, 2, 3, 4, 5, 6]

even_numbers = list (filter (lambda x: x % 2 == 0, numbers))

print(even_numbers) # Output: [2, 4, 6]
```

3. reduce (function, iterable)

- Purpose: Applies a function cumulatively to reduce the iterable to a single value.
- Requires: from fun tools import reduce

🔹 Example:

```
from fun tools import reduce

numbers = [1, 2, 3, 4]

product = reduce (lambda x, y: x * y, numbers)

print(product) # Output: 24
```

Summary Table

Function Use case Output

map () Transform each element New transformed list

filter () Select elements conditionally Filtered list

reduce () Combine elements to one value Single aggregated value

27. Introduction to closures and decorators.

=>

Introduction to Closures and Decorators in Python

- Closures

Definition:

A closure is a function object that remembers values in enclosing

scopes even if they are not in memory.

How It Works:

- A nested function remembers the values from its enclosing function's scope.
- The outer function returns the inner function.

Example:

```
def outer(x):

def inner(y):

return x + y

return inner

add_five = outer (5)

print(add_five (10)) # Output: 15
```

Explanation:

- x is remembered by the inner () function even after outer () has

finished executing.

- add_five (10) means x=5 and y=10.

- Decorators

Definition:

A decorator is a function that modifies the behaviour of another function without changing its code.

Used For:

- Logging
- Authentication
- Performance timing
- Access control, etc.

Basic Syntax:

```
@decorator_function
def actual_function ():
pass
```

This is the same as:

```
actual_function = decorator_function(actual_function)
```

Example:

```
def my_decorator(func):
def wrapper ():
print ("Before the function runs")
fun ()
print ("After the function runs")
return wrapper
@my_decorator
def say_hello ():
print("Hello!")
say_hello ()
```

Output:

```
Before the function runs
Hello!
```

After the function runs

How Closures Help in Decorators:

- Decorators use closures to keep a reference to the original function.
- The wrapper function (inside the decorator) is a closure that has access to the original function.