Assignment - Fundamentals of Python

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

Ans:-

Introduction to Python
Python is a popular, easy-to-learn programming language used for a wide range of applications — from web development to data science, automation, AI, and more.

It was created by Guido van Rossum and first released in 1991. Python's main goal is code readability and simplicity, which makes it perfect for beginners and professionals alike.

Key Features of Python: -
1. Simple and Easy to Learn
Python has a clean and readable syntax, similar to English.
Example:
print("Hello, world!")

2. High-Level Language
Python takes care of low-level details like memory management.

You can focus on solving problems, not worrying about how the computer works internally.

3. Interpreted Language
Python code is executed line by line.
You don’t need to compile the code before running it — just write and run.
python script.py

4. Dynamically Typed
You don’t need to declare variable types.

```
x = 10       # integer
x = "hello"  # now it's a string
```

5. Object-Oriented
Supports classes and objects for reusable and organized code.

```
class Dog:
    def bark(self):
        print("Woof!")
```

6. Large Standard Library
Comes with built-in modules for handling files, math, dates, web, etc.

```
import math
print(math.sqrt(16))  # 4.0
```

7. Cross-Platform

Runs on Windows, macOS, Linux, and more — write once, run anywhere.

8. Extensible and Embeddable
You can use Python with other languages like C/C++.

9. Huge Community Support
Thousands of tutorials, libraries, and active developers make learning and problem-solving easy.

10. Popular in Many Fields
Used in:
Web development (Django, Flask)
Data Science (pandas, NumPy)
Machine Learning (TensorFlow, scikit-learn)
Automation (scripts)
Game Development (Pygame)

2. History and evolution of Python.

Ans: -

History and Evolution of Python
Origin

- Creator: Guido van Rossum
- Started: Late 1980s
- Released: 1991

- Inspired by: ABC language (a teaching language developed at CWI in the Netherlands)

Guido wanted a language that was:

- Easy to read and write
- Powerful but simple
- Open and accessible to all

The name "Python" was inspired by the British comedy group Monty Python, not the snake!

Evolution Timeline

1991 – Python 0.9.0

- First official release by Guido
- Included: functions, exception handling, modules, and core data types (list, dict, etc.)

1994 – Python 1.0

- Introduced: lambda, map(), filter(), reduce()
- Started gaining attention in the programming world

2000 – Python 2.0

- Major improvements:
- List comprehensions
- Garbage collection using reference counting
- Unicode support
- Widely adopted in web scripting and early web development

2008 – Python 3.0 (aka Python 3000)

- Not backward-compatible with Python 2
- Improved:
- Print function: print("Hello") instead of print "Hello"
- Better Unicode support
- Cleaner syntax and standard libraries
- Goal: Make Python more consistent and future-proof

Python 2 vs. Python 3

- Python 2 was supported until January 1, 2020
- Python 3 is the current and actively developed version

Recent Versions and Improvements

Python 3.6 (2016)

- f-strings for cleaner string formatting
- Type hints

Python 3.8 (2019)

- Walrus operator :=
- Improved performance and assignment expressions

Python 3.10 (2021)

- Pattern matching (match/case syntax)

Python 3.11 / 3.12 (2022–2023)

- Significant speed improvements

- Better error messages and typing features

Today: -

- Python is one of the most popular programming languages in the world
- Used in:
- Web development
- Machine learning
- Data analysis
- Automation
- Cybersecurity
- Game development
- Maintained by the Python Software Foundation (PSF)

3. Advantages of using Python over other programming languages.

Ans:-

Advantages of Python Over Other Languages:-

1. Easy to Learn and Read
   - Simple, clean, and English-like syntax
   - Great for beginners and professionals

Example:

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

2. Rapid Development
   - Write fewer lines of code compared to Java or C++

- Ideal for quick prototyping and testing ideas

3. Extensive Standard Library
   - Comes with built-in modules for handling files, web, math, dates, etc.
   - Saves time — no need to write everything from scratch

4. Massive Collection of Libraries and Frameworks
   - Libraries for almost everything:
   - Data Science: pandas, NumPy
   - Machine Learning: TensorFlow, scikit-learn
   - Web Development: Django, Flask
   - Automation: selenium, pyautogui

5. Cross-Platform Compatibility
   - Works on Windows, macOS, Linux, and others
   - Write once, run anywhere (with minor changes if needed)

6. Dynamic Typing
   - No need to declare variable types
   - More flexible and faster coding

```
x = 5
x = "now a string"
```

7. High Demand and Active Community

- One of the most in-demand languages in the job market
- Huge global community → easier to get help and find solutions

8. Supports Multiple Programming Paradigms
   - Object-oriented, procedural, and functional programming
   - Adaptable to different project styles

9. Great for Automation and Scripting
   - Perfect for automating repetitive tasks
   - Often used in DevOps and IT scripting

10. Used in Cutting-Edge Fields
    - Leading language for:
    - AI & Machine Learning
    - Data Science & Analytics
    - Robotics
    - Cybersecurity

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

Ans:-

Installing Python and Setting Up the Development Environment

1. Installing Python (Standard Method)

Step 1: Download Python

- Go to the official site: https://www.python.org
- Click on Downloads
- Choose the latest version for your operating system (Windows, macOS, Linux)

Step 2: Install Python

- Run the downloaded installer
- Important: Check the box “Add Python to PATH”

Click Install Now

Verify installation:
python --version
Option A: Setting Up with Anaconda (Best for Data Science)
What is Anaconda?
A Python distribution that includes Python, Jupyter Notebook, and popular libraries (pandas, NumPy, matplotlib, etc.)

Steps:
Download from https://www.anaconda.com

Install it (accept defaults)

Open Anaconda Navigator or Anaconda Prompt

Launch tools like Jupyter Notebook, Spyder, or create environments
conda --version
Great for: Data Science, Machine Learning, scientific computing

Option B: Using PyCharm (Best for Full Python Projects)
What is PyCharm?
A powerful IDE (Integrated Development Environment) by JetBrains

Steps:

- Download from https://www.jetbrains.com/pycharm
- Install the Community Edition (free)
- Open PyCharm and:
- Create a new project
- Select a Python interpreter (auto-detected or manually added)

Great for: App development, large projects, debugging
Option C: Using Visual Studio Code (VS Code)
What is VS Code?
A lightweight, fast, and highly customizable code editor from Microsoft

Steps:

- Download from https://code.visualstudio.com
- Install it
- Install the Python extension from the Extensions tab
- Open a .py file or folder
- Select Python interpreter (usually prompted)

Great for: Beginners, web development, general scripting
Creating a Virtual Environment (optional but recommended)

```
# Create a new environment
python -m venv myenv

# Activate it
# Windows:
myenv\Scripts\activate
# macOS/Linux:
source myenv/bin/activate
```

5. **Writing and executing your first Python program.**

Ans:-

Step 1: Install Python

If you haven't already installed Python:

1. Go to https://www.python.org/downloads/
2. Download and install the latest version for your OS (Windows, macOS, Linux).
3. During installation, make sure to check "Add Python to PATH" before clicking Install.

Step 2: Write Your First Python Program
You can use any text editor (like Notepad, VS Code, or even IDLE that comes with Python). For now, we'll use a simple example:
Example Code: hello.py
print("Hello, world!")
Save this file as hello.py.

Step 3: Run the Program
Option 1: Using Command Line / Terminal

1. Open Command Prompt (Windows) or Terminal (macOS/Linux).
2. Navigate to the directory where you saved hello.py. For example:

Run the program:
python hello.py
You should see:
Hello, world!
If you get an error like "python is not recognized", try using python3 instead.
What's Happening?
print() is a built-in Python function that outputs text to the screen.

"Hello, world!" is a string, a basic data type in Python.
Ready to Try More?

Try changing the message or adding math:
print("Hello again!")
print(2 + 3)
If you want, I can help you set up an interactive environment or write a more advanced first program (e.g. calculator, quiz, etc.).

6. Understanding Python's PEP 8 guidelines.
Ans:-
What is PEP 8?
PEP 8 stands for Python Enhancement Proposal 8, and it is the style guide for Python code. It provides recommendations on how to format Python code to improve readability and consistency across the Python community.

Key PEP 8 Guidelines:-
1. Indentation
   - Use 4 spaces per indentation level.
   - Never mix tabs and spaces.

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

2. Variable and Function Naming
   - Use lowercase_with_underscores for variables and functions.
   - Use UPPERCASE for constants.

- Use CamelCase for class names.

```
user_name = "Alice"
MAX_RETRIES = 5
def get_user_input():
    pass
class UserProfile:
    Pass
```

3. Line Length
   - Limit lines to 79 characters (or 72 for comments/docstrings).
   - Use parentheses or backslashes for line breaks.

```
# Good:
result = some_function(param1, param2, param3,
                       param4, param5)
# Not preferred:
result = some_function(param1, param2, param3, param4,
param5, param6, param7)
```

4. Blank Lines
   - Use 2 blank lines to separate top-level functions and classes.
   - Use 1 blank line to separate methods within a class.

5. Imports
   - Imports should be on separate lines and grouped in the following order:

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

```
import os
import sys

import requests

from my_app import my_module
```

6. Whitespace Usage
   - Avoid extra spaces inside parentheses, brackets, or before commas.

```
# Good:
x = (1 + 2) * (3 / 4)

# Bad:
x = ( 1 + 2 ) * ( 3 / 4 )
```

7. Comments and Docstrings
   - Use # for inline or block comments.

   - Use triple quotes (""") for docstrings in functions, classes, and modules.

```
# This function adds two numbers
def add(a, b):
```

```
    """Return the sum of a and b."""
    return a + b
```

8. Boolean and None Comparisons
   - Use is or is not for comparisons with None.
   - Use True/False for boolean expressions.

```
if value is None:
    print("Value is missing")
```

Why Follow PEP 8?

- Makes your code easier to read, maintain, and collaborate on.
- Helps you write Pythonic code that aligns with community standards.
- Tools like flake8, pylint, and editors like VS Code or PyCharm can automatically check and format PEP 8 compliance.

7. Writing readable and maintainable code.

Ans:-

1. Use Meaningful Variable and Function Names

Good:

```
user_age = 25
def calculate_total_price(items):
    ...
```

Bad:

```
x = 25
def func(a):
```

```
    ...
```
Tip: Name things based on what they do or what they represent.

2. Write Small, Focused Functions

Functions should do one thing only and do it well.

```
def calculate_average(scores):
    return sum(scores) / len(scores)
```

If a function is getting too long or doing too many things, split it up.

3. Add Comments and Docstrings

- Use inline comments for complex logic.
- Use docstrings to explain what functions/classes do.

```
def greet_user(name):
    """Display a personalized greeting message."""
    print(f"Hello, {name}!")
```

Comments should explain why something is done, not what (which should be clear from code itself).

4. Follow PEP 8 Style Guide

- Indent with 4 spaces

- Keep lines under 79 characters
- Leave space around operators
- Group imports properly

Use tools like:

- black – code formatter
- flake8 – style checker

5. Avoid Repetition (DRY Principle)
Don't Repeat Yourself – extract repeated code into reusable functions.

```
def get_user_input(prompt):
    return input(prompt).strip()
```

6. Use Constants for Fixed Values

```
MAX_RETRIES = 3
```

Use ALL_CAPS for constants and place them at the top of your file.

7. Handle Errors Gracefully
Use try...except blocks to make code robust and user-friendly.

```
try:
    age = int(input("Enter your age: "))
except ValueError:
    print("Please enter a valid number.")
```

8. Use Lists, Dicts, and Loops Effectively
Python gives you powerful tools—use them cleanly.

```
for name in ["Alice", "Bob", "Charlie"]:
    print(f"Hello, {name}")
```

9. Organize Code into Modules and Functions
Avoid writing everything in one file or function. Break your project into:

- Functions (small tasks)
- Modules (related functionality)
- Packages (project-level structure)

10. Write Unit Tests
Even simple tests improve maintainability.

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

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

Use frameworks like unittest or pytest.

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

Ans:-

1. Integers (int)
   - Whole numbers, positive or negative, without decimals.
   - Examples: 5, -42, 0

```
x = 10
```

2. Floats (float)
   - Numbers with decimal points.

- Examples: 3.14, -0.001, 2.0

```
pi = 3.14159
```

3. Strings (str)
   - Sequences of characters (text), enclosed in single or double quotes.
   - Examples: 'hello', "123", "John's book"

```
name = "Alice"
```

4. Lists (list)
   - Ordered, changeable (mutable) collections. Can hold mixed data types.
   - Defined with square brackets: []

```
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4.5, "five"]
```

5. Tuples (tuple)
   - Ordered but immutable (unchangeable) collections.
   - Defined with parentheses: ()

```
point = (4, 5)
colors = ("red", "green", "blue")
```

6. Dictionaries (dict)
   - Unordered collections of key-value pairs.
   - Keys must be unique and immutable.
   - Defined with curly braces: {}

person = {"name": "Alice", "age": 30, "city": "New York"}
Access by key:
print(person["name"]) # Output: Alice

7. Sets (set)

- Unordered, mutable collections of unique elements.
- Defined with curly braces: {} or set() constructor.

unique_numbers = {1, 2, 3, 3, 2} # Becomes {1, 2, 3}
Useful for eliminating duplicates.

9. Python variables and memory allocation.
Ans:-

1. What is a Variable in Python?

A variable is a name that refers to a value stored in memory. It acts as a label or reference rather than a container.
x = 10
Here, x refers to an integer object 10 in memory.

2. How Memory Allocation Works

Python handles memory allocation automatically using a system of reference counting and garbage collection.
a. Reference
When you do:
a = 5
An object (5) is created in memory.

a becomes a reference (a "name") pointing to that object.
b. Multiple References
If you do:
b = a
Now both a and b point to the same object 5.

c. Garbage Collection
If no variable references an object, Python will automatically remove it from memory.
a = 5
a = 6 # The object 5 is no longer referenced, so it may be garbage collected.

3. Immutable vs Mutable Types and Memory

Type	Mutable?	Example	Behavior in Memory
int	No	x = 10	Creates new object when changed
float	No	pi = 3.14	Creates new object when changed
str	No	name = "Amy"	Changing creates a new object
list	Yes	mylist = [1]	Same object updated in place
dict	Yes	d = {'a': 1}	Updates happen inside the same object
set	Yes	s = {1, 2}	Updated in place

Example:
x = [1, 2, 3]
y = x
y.append(4)

```
print(x)  # Output: [1, 2, 3, 4] – because x and y point to the same list
```

4. id() and is Operator

- id(variable) shows the memory address of the object.
- is checks if two variables point to the same object.

```
x = 1000
y = 1000
print(x is y)          # False (may be True for small ints < 256 due to caching)
print(id(x), id(y))    # Different IDs
```

5. Interning and Small Integer Cache

Python caches small integers and some strings to save memory.

```
a = 5
b = 5
print(a is b)  # True (same memory location)
```

But with:

```
a = 1000
b = 1000
print(a is b)  # False
```

10. Python operators: arithmetic, comparison, logical, bitwise.

Ans:-

1. Arithmetic Operators

Used for basic mathematical operations.

Operator	Description	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
//	Floor Division	a // b
%	Modulus (remainder)	a % b
**	Exponentiation	a ** b

2. Comparison (Relational) Operators
Used to compare values.

Operator	Description	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal	a >= b
<=	Less than or equal	a <= b

3. Logical Operators
Used to combine conditional statements.

Operator	Description	Example
and	True if both are true	a > 0 and b > 0

or	True if at least one is true	a > 0 or b > 0
not	Inverts the result (negation)	not a > 0

4. Bitwise Operators
Used to perform bit-level operations.

Operator	Description	Example	Explanation (binary)
&	AND	a & b	0101 & 0011 → 0001
`	`	OR	`a
^	XOR	a ^ b	0101 ^ 0011 → 0110
~	NOT	~a	~0101 → 1010 (inverted)
<<	Left shift	a << 1	Shifts bits to the left
>>	Right shift	a >> 1	Shifts bits to the right

11. Introduction to conditional statements: if, else, elif.
Ans:-
What Are Conditional Statements?
Conditional statements let your program make decisions—run certain blocks of code only if specific conditions are met.

1. if Statement
Executes a block of code if a condition is true.

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

2. else Statement
Runs if the if condition is false.

```
x = 3
if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")
```

3. elif (else if) Statement
Tests multiple conditions, in sequence.

```
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")
```

Basic Structure

```
if condition1:
    # code block if condition1 is true
elif condition2:
    # code block if condition2 is true
else:
    # code block if none of the above conditions are true
```

12. Nested if-else conditions.

Ans:-

Nested if-else Conditions in Python:-

A nested if-else is when you place one if or else statement inside another. This is used when decisions depend on multiple levels of conditions.

Syntax

```
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 1: Checking age and citizenship

```
age = 20
citizen = True

if age >= 18:
    if citizen:
        print("You are eligible to vote.")
    else:
        print("You must be a citizen to vote.")
else:
    print("You must be at least 18 years old to vote.")
```

Output: You are eligible to vote.

Example 2: Grading system

```
marks = 85
if marks >= 50:
    if marks >= 90:
        print("Grade: A")
    elif marks >= 75:
        print("Grade: B")
    else:
        print("Grade: C")
else:
    print("Fail")
```

Output: Grade: B

13. Introduction to for and while loops.

Ans:-

Why Use Loops?

Loops help you repeat code without writing it over and over. Python has two main types:

1. for Loop

Used when you know how many times to loop—like going through a list or counting numbers.

Basic Syntax:

```
for variable in sequence:
```

```
    # code to repeat
```

Example: Print numbers 1 to 5

```
for i in range(1, 6):
    print(i)
```

Example: Loop through a list

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

2. while Loop

Used when you want to keep looping until a condition is false.

Basic Syntax:

```
while condition:
    # code to repeat
```

Example: Count from 1 to 5

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

14. How loops work in Python.

Ans:-

How Loops Work in Python
Loops in Python are used to repeat a block of code multiple times. Here's how they work under the hood:

1. for Loop – Iteration Over a Sequence
The for loop goes through each item in a sequence (like a list, string, or range) one at a time.

Example:

```
for letter in "hi":
    print(letter)
```

Behind the scenes:

- Python sees the string "hi" as: ["h", "i"]
- First iteration: letter = "h" → prints "h"
- Second iteration: letter = "i" → prints "i"

With range():

```
for i in range(3):
    print(i)
```

- range(3) creates: [0, 1, 2]
- Loop runs 3 times, printing 0, 1, 2

2. while Loop – Looping Based on a Condition
A while loop runs as long as a condition is true.
Example:

```
x = 0
```

```
while x < 3:
    print(x)
    x += 1
```
🔍Behind the scenes:

- Loop starts with x = 0
- Checks: Is x < 3? Yes → prints 0
- Increases x to 1, checks again → continues
- Stops when x becomes 3 (condition is false)

break Example:

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

continue Example:

python

Copy code

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

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

Ans:-

Using Loops with Collections in Python

Collections like lists, tuples, sets, and dictionaries are often used with loops to process multiple values efficiently. Let's look at how to use for loops with each.

1. Looping Through a List
Lists are ordered and allow duplicates.

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

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

Output:

```
apple
banana
Cherry
```

You can also loop using indices:

```
for i in range(len(fruits)):
    print(f"Index {i} → {fruits[i]}")
```

2. Looping Through a Tuple
Tuples are similar to lists, but immutable.

```
dimensions = (10, 20, 30)
for d in dimensions:
    print(d)
```

Output:

```
10
20
```

3. Looping Through a Set
Sets are unordered and contain only unique values.

```
unique_numbers = {1, 2, 3, 2, 1}

for num in unique_numbers:
    print(num)
```

Output (order may vary):

```
1
2
3
```

4. Looping Through a Dictionary

➢ Looping through keys:

```
person = {"name": "Alice", "age": 25}
for key in person:
    print(key)
```

➢ Looping through values:

```
for value in person.values():
    print(value)
```

➢ Looping through key–value pairs:

```
for key, value in person.items():
    print(key, "→", value)
```

Output:

```
name → Alice
age → 25
```

Bonus: enumerate() with Lists and Tuples
Get both the index and the item:

```
names = ["Tom", "Jerry", "Spike"]

for index, name in enumerate(names):
    print(index, name)
```

Output:

```
0 Tom
1 Jerry
2 Spike
```

16. Understanding how generators work in Python.
Ans:-
Understanding How Generators Work in Python
Generators are a simple and memory-efficient way to produce a sequence of values, especially useful when dealing with large data.

What is a Generator?
A generator is a special type of iterator in Python that yields values one at a time using the yield keyword instead of returning them all at once.

Why Use Generators?
Saves memory — doesn’t store the entire result in memory

Ideal for large or infinite data streams
Lazy evaluation — computes each value only when needed

Creating a Generator Function

```
def count_up_to(n):
    i = 1
    while i <= n:
        yield i
        i += 1
```

This function does not return a list. It yields one value at a time.

Using the Generator

```
counter = count_up_to(3)

for num in counter:
    print(num)
```

Output:

1
2
3

How It Works Internally:-

- When count_up_to(3) is called, it does not run the function.

- It returns a generator object.
- On each iteration, Python resumes the function from where it left off at yield.

Example: Compare List vs Generator
# List: all values at once
nums_list = [x * 2 for x in range(1000000)] # High memory

# Generator: one value at a time
nums_gen = (x * 2 for x in range(1000000))   # Very efficient
You can loop through nums_gen just like a list:

```
for n in nums_gen:
    print(n)
    if n > 10:
        break
```

17. Difference between yield and return.
Ans:-
Difference Between yield and return in Python
Both yield and return are used in functions, but they behave very differently. Here's a clear comparison:

1. return
   - Ends the function immediately.
   - Sends back a single value (or None).
   - Function can't be resumed.

Example:

```
def get_number():
    return 5
print(get_number())  # Output: 5
```

2. yield

- Pauses the function and saves its state.
- Sends back a value without ending the function.
- Can resume from where it left off (used in generators).

Example:

```
def count_up_to(n):
    i = 1
    while i <= n:
        yield i
        i += 1

# Create a generator
gen = count_up_to(3)

for num in gen:
    print(num)
```

Output:

1
2
3

Example Side by Side

```
def with_return():
```

```
    return [1, 2, 3]

def with_yield():
    yield 1
    yield 2
    yield 3

print(with_return())      # Output: [1, 2, 3]
print(list(with_yield())) # Output: [1, 2, 3]
```

Use yield When:

- You're working with large or infinite sequences
- You want to process one item at a time
- You need to build custom iterators

18. Understanding iterators and creating custom iterators.

Ans:-

Understanding Iterators in Python:-

In Python, an iterator is any object that allows you to loop through a sequence, one item at a time.

What is an Iterator?

An iterator is an object that:

1. Implements the __iter__() method (returns the iterator object itself)

2. Implements the __next__() method (returns the next item, or raises StopIteration when done)

Built-in Iterators:-
Most collections like lists, tuples, and strings are iterables (they can be looped through), and their iterators are created behind the scenes in a loop.

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

it = iter(numbers)  # Get an iterator object

print(next(it))  # 1
print(next(it))  # 2
print(next(it))  # 3
# print(next(it))  # Raises StopIteration
```

Creating a Custom Iterator (Manually)
You can create your own iterator class by defining __iter__() and __next__():

Example: Count from 1 to n

```
class CountUpTo:
    def __init__(self, limit):
        self.limit = limit
        self.current = 1

    def __iter__(self):
```

```
        return self  # The object itself is the iterator

    def __next__(self):
        if self.current <= self.limit:
            num = self.current
            self.current += 1
            return num
        else:
            raise StopIteration
```
Using it:
```
counter = CountUpTo(3)

for num in counter:
    print(num)
```
Output:
1
2
3

19. Defining and calling functions in Python.
Ans:-
Defining and Calling Functions in Python
Functions in Python are blocks of reusable code that perform a specific task. They help make your code organized, readable, and reusable.

1. Defining a Function

Use the def keyword to define a function:-

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

- def – starts the function definition
- greet – name of the function
- () – parentheses (can include parameters)
- : – indicates the start of the function body
- Indented lines – the code block that runs when the function is called

2. Calling a Function

You call (run) a function by writing its name followed by parentheses:

```
greet()  # Output: Hello!
```

3. Function with Parameters

You can pass data to functions using parameters:

```
def greet(name):
    print("Hello,", name)
python
Copy code
greet("Alice")  # Output: Hello, Alice
```

4. Function with Return Value

Use return to send a result back from the function:

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

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

5. Function with Default Parameters
You can provide default values for parameters:

```
def greet(name="friend"):
    print("Hi,", name)

greet("Emma")     # Output: Hi, Emma
greet()           # Output: Hi, friend
```

20. Function arguments (positional, keyword, default).
Ans:-
Function Arguments in Python
When you define and call functions, you can pass values (arguments) in different ways: positional, keyword, and default arguments. Here's how each works:

1. Positional Arguments
- Arguments are matched to parameters based on their order.
- You must pass them in the correct order.

Example:

```
def greet(name, age):
    print(f"Hello, {name}. You are {age} years old.")
```

```
greet("Alice", 25)  # Correct
# greet(25, "Alice")  # Wrong order!
```

2. Keyword Arguments

- Specify arguments by parameter name.
- Order doesn’t matter.
- Makes code more readable.

Example:

```
greet(age=25, name="Alice")  # Same output as above
```

3. Default Arguments

- Assign a default value to a parameter.
- If the argument is not provided during the call, the default is used.

Example:

```
def greet(name, age=18):
    print(f"Hello, {name}. You are {age} years old.")

greet("Bob")          # age defaults to 18
greet("Eve", 30)      # age provided as 30
```

Combining All:-

- Positional arguments come first.
- Then keyword arguments.
- Default arguments are optional and used if no argument is passed.

Example:

```
def describe_pet(pet_name, animal_type='dog'):
    print(f"I have a {animal_type} named {pet_name}.")

describe_pet('Buddy')                  # Uses default 'dog'
describe_pet('Whiskers', 'cat')        # Overrides default
describe_pet(pet_name='Goldie')        # Keyword argument with default
describe_pet(animal_type='parrot', pet_name='Polly')  # Both keyword arguments
```

21. Scope of variables in Python.

Ans:-

Scope of Variables in Python

Scope refers to where in your code a variable can be accessed. Python has specific rules for variable visibility based on where a variable is declared.

1. Local Scope

Defined inside a function — accessible only within that function.

```
def greet():
    name = "Alice"  # Local variable
    print(name)

greet()
```

```
# print(name)  # ❌ Error: name is not defined outside
```

2. Global Scope

Defined outside all functions — accessible anywhere in the module.

```
x = 10  # Global variable
def show():
    print(x)
show()  # Output: 10
```

3. Enclosing Scope (Nested Functions)

A variable in an outer function, used inside an inner function.

```
def outer():
    msg = "Hello"

    def inner():
        print(msg)  # Enclosing variable

    inner()

outer()  # Output: Hello
```

4. Built-in Scope

Python’s built-in functions and keywords:

```
print(len("Python"))  # Output: 6
```

You can see all built-in names with:

```
import builtins
print(dir(builtins))
```

Modifying Global Variables from Inside a Function
Use the global keyword:

```
x = 5
def change():
    global x
    x = 10
change()
print(x)  # Output: 10
```

Modifying Enclosing Variables (nonlocal)
Use nonlocal in nested functions:

```
def outer():
    count = 0

    def inner():
        nonlocal count
        count += 1
        print(count)

    inner()

outer()  # Output: 1
```

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

Ans:-

Built-in Methods for Common Python Data Types

Python provides many built-in methods for working with data types like strings, lists, tuples, sets, and dictionaries. These methods make it easier to manipulate data without writing extra code.

1. String Methods (str)

Strings are sequences of characters.

Common Methods:

Method	Description	Example
.lower()	Converts to lowercase	"HELLO".lower() → 'hello'
.upper()	Converts to uppercase	"hi".upper() → 'HI'
.strip()	Removes whitespace	" hi ".strip() → 'hi'
.replace(a, b)	Replaces substring	"hello".replace("l", "x") → 'hexxo'
.split(sep)	Splits string into list	"a,b,c".split(",") → ['a','b','c']
.find(sub)	Returns first index of substring	"hello".find("e") → 1
.startswith(sub)	Checks if string starts with sub	"hello".startswith("he") → True
.endswith(sub)	Checks if string ends with sub	"hi!".endswith("!") → True

2. List Methods (list)

Lists are ordered and mutable sequences.

Common Methods:

Method	Description	Example
.append(x)	Adds item to end	[1, 2].append(3) → [1, 2, 3]
.extend([x, y])	Adds all items from another list	[1].extend([2,3]) → [1,2,3]
.insert(i, x)	Inserts at index i	[1, 2].insert(1, 9) → [1, 9, 2]
.remove(x)	Removes first occurrence of x	[1,2,1].remove(1) → [2,1]
.pop(i)	Removes and returns item at index	[1,2,3].pop(1) → 2
.sort()	Sorts list in place	[3,1,2].sort() → [1,2,3]
.reverse()	Reverses list in place	[1,2].reverse() → [2,1]
.index(x)	Returns index of first occurrence	[3,1,2].index(1) → 1
.count(x)	Counts number of times x appears	[1,1,2].count(1) → 2

3. Tuple Methods (tuple)

Tuples are like lists, but immutable.

Method	Description
.count(x)	Number of times x appears
.index(x)	Index of first occurrence of x

Example:

python
Copy code
t = (1, 2, 2)
print(t.count(2)) # → 2
print(t.index(2)) # → 1

4. Set Methods (set)
Sets are unordered collections of unique elements.

Method	Description
.add(x)	Adds an element
.remove(x)	Removes element (error if not found)
.discard(x)	Removes element (no error if missing)
.clear()	Removes all elements
.union(set2)	Combines two sets (all unique)
.intersection()	Common elements
.difference()	Items in one set but not the other

5. Dictionary Methods (dict)
Dictionaries store key–value pairs.

Method	Description
.keys()	Returns list-like view of keys
.values()	Returns view of values

.items() Returns view of (key, value) tuples
.get(key) Returns value or None if missing
.pop(key) Removes key and returns value
.update({}) Updates dict with another dict
.clear() Removes all items

Example:
person = {"name": "Alice", "age": 30}
print(person.get("age")) # → 30
print(person.keys()) # → dict_keys(['name', 'age'])

23. Understanding the role of break, continue, and pass in Python loops.
Ans:-
Understanding break, continue, and pass in Python Loops
These three keywords are used to control the flow inside loops (for or while). Each has a distinct purpose:

1. break → Stop the loop completely
   - Exits the loop immediately.
   - Loop ends even if the condition is still true or there are more items.

Example:
```
for i in range(5):
    if i == 3:
```

```
        break
    print(i)
```
Output:
0
1
2
When i == 3, break stops the loop.

2. continue → Skip this iteration, go to the next
- Skips the current loop cycle.
- Loop continues with the next item.

Example:
```
for i in range(5):
    if i == 3:
        continue
    print(i)
```
Output:
0
1
2
4
When i == 3, continue skips the print(i) and jumps to the next number.

3. pass → Do nothing (placeholder)
- Does literally nothing.

- Often used as a placeholder for future code.

Example:

```
for i in range(5):
    if i == 3:
        pass  # Placeholder — no action
    print(i)
```

Output:

```
0
1
2
3
4
```

pass lets the loop run normally; it's just a "do-nothing" statement.

24. Understanding how to access and manipulate strings.

Ans:-

Understanding How to Access and Manipulate Strings in Python

Strings in Python are sequences of characters, and you can access, slice, and manipulate them easily using built-in features and methods.

1. Accessing Characters

Strings are like lists of characters. You can access individual characters using indexing:

```
text = "Python"
print(text[0])   # P
print(text[-1])  # n (last character)
```

- Indexing starts at 0
- Negative indexes count from the end

2. Slicing Strings

You can extract parts (substrings) using slicing:

```
text = "Python"
print(text[0:2])   # Py
print(text[2:])    # thon
print(text[:3])    # Pyt
print(text[-3:])   # hon
```

Syntax: string[start:stop] (stop is exclusive)

3. Common String Methods

Method Description Example

.lower() Convert to lowercase "HELLO".lower() → 'hello'

.upper() Convert to uppercase "hello".upper() → 'HELLO'

.capitalize() Capitalize first letter "python".capitalize() → 'Python'

.strip() Remove leading/trailing whitespace " text ".strip() → 'text'

.replace(old, new)Replace substring "a-b-c".replace("-", "+") → 'a+b+c'

.find(sub) Find index of substring "hello".find("e") → 1
.split(sep) Split into list by separator "a,b,c".split(",") → ['a', 'b', 'c']
.join(list) Join list into string",".join(["a", "b"]) → 'a,b'

4. Looping Through a String

```
for char in "cat":
    print(char)
```

Output:

```
c
a
T
```

5. String Concatenation and Repetition

```
greeting = "Hello" + " " + "World"
print(greeting)  # Hello World
repeat = "ha" * 3
print(repeat)    # hahaha
```

6. String Immutability

Strings in Python cannot be changed directly.

```
word = "hello"
# word[0] = "H"  # This will raise an error
word = "H" + word[1:]  #  This works
print(word)  # Hello
```

7. Other Useful Functions
python
Copy code
len("Python")   # → 6
"py" in "Python"   # → True
"z" not in "Python" # → True

25. Basic operations: concatenation, repetition, string methods (upper(), lower(), etc.).
Ans:-
Basic String Operations in Python
Strings are one of the most used data types in Python. Here's how you can combine, repeat, and manipulate them using built-in methods.

1. Concatenation (Joining Strings)
Use + to combine strings:
first = "Hello"
second = "World"
result = first + " " + second
print(result) # Output: Hello World

2. Repetition
Use * to repeat a string:
laugh = "ha"
print(laugh * 3) # Output: hahaha

3. Common String Methods

Method	Description	Example
.upper()	Converts all characters to uppercase	"hello".upper() → 'HELLO'
.lower()	Converts all characters to lowercase	"HELLO".lower() → 'hello'
.capitalize()	Capitalizes the first letter	"python".capitalize() → 'Python'
.strip()	Removes spaces at both ends	" hi ".strip() → 'hi'
.replace()	Replaces parts of the string	"a-b-c".replace("-", "+") → 'a+b+c'
.title()	Capitalizes first letter of each word	"hello world".title() → 'Hello World'

4. Examples in Action

```
s = "  Python is fun!  "

print(s.strip())        # 'Python is fun!'
print(s.upper())        # '  PYTHON IS FUN!  '
print(s.lower())        # '  python is fun!  '
print(s.replace("fun", "awesome"))  # '  Python is awesome!  '
```

Quick Notes

- Strings are immutable: you can't change them in place, but you can create new ones.

- You can chain methods:

```
" hello ".strip().upper()  # Output: 'HELLO'
```

26. String slicing.

Ans:-

String Slicing in Python

String slicing lets you extract a portion of a string using a specific range of characters. It uses this syntax:

string[start:stop:step]

- start – index to begin slicing (inclusive)
- stop – index to stop (exclusive)
- step – how many characters to skip (optional)

Basic Slicing Examples

text = "Python"

Slice	Description	Output
text[0:2]	Characters at index 0 and 1	'Py'
text[2:5]	Characters 2, 3, 4	'tho'
text[:4]	From start to index 3	'Pyth'
text[3:]	From index 3 to the end	'hon'
text[:]	Entire string (copy)	'Python'
text[::2]	Every 2nd character	'Pto'

Negative Indexing

Negative indices count from the end:

text = "Python"

Slice Description Output
text[-1] Last character 'n'
text[-3:-1] From 3rd last to 2nd last character 'ho'
text[::-1] Reversed string 'nohtyP'

Examples
word = "Programming"

print(word[0:5]) # Progr
print(word[3:8]) # gram
print(word[:4]) # Prog
print(word[4:]) # ramming
print(word[::2]) # Pormig
print(word[::-1]) # gnimmargorP

Out-of-Bounds? No Problem
Slicing won't raise an error if the indices are out of bounds:
print("Hi"[0:10]) # Output: Hi

27. How functional programming works in Python.
Ans:-
How Functional Programming Works in Python
Python supports functional programming, a style that treats functions as first-class citizens — meaning you can assign them to variables, pass them as arguments, and return them from other functions.

Key Concepts of Functional Programming in Python

1. First-Class Functions

Functions can be stored in variables, passed, and returned.

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

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

2. Pure Functions

A pure function:

- Always gives the same output for the same input
- Has no side effects (no print, no modifying globals)

```
def add(a, b):
    return a + b  # Pure: only returns value
```

3. Higher-Order Functions

Functions that take other functions as arguments or return functions.

```
def apply_twice(func, x):
    return func(func(x))

def square(n):
    return n * n

print(apply_twice(square, 2))  # Output: 16
```

4. Lambda Functions (Anonymous Functions)
Short, one-line functions.

```
square = lambda x: x * x
print(square(5))  # Output: 25
```

Used in map(), filter(), and sorted().

5. Built-in Functional Tools
map(function, iterable)
Applies a function to each item.

```
nums = [1, 2, 3]
squared = list(map(lambda x: x**2, nums))  # [1, 4, 9]
```

filter(function, iterable)
Filters items for which the function returns True.

```
nums = [1, 2, 3, 4]
even = list(filter(lambda x: x % 2 == 0, nums))  # [2, 4]
```

reduce(function, iterable)
Applies a function cumulatively (must import it):

```
from functools import reduce

nums = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, nums)  # 24
```

6. Immutability and No Side Effects
Functional programming encourages avoiding changing data:

```
# Avoid
```

```
my_list = [1, 2]
my_list.append(3)  # Modifies list

# Prefer
new_list = my_list + [3]  # Returns new list
```

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

Ans:-

1. map()

Used to transform data by applying a function to each element in an iterable.

```
# Example: Squaring numbers in a list
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))
print(squared)  # Output: [1, 4, 9, 16, 25]
```

2. filter()

Used to filter elements in an iterable based on a condition.

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

3. reduce()

- Used to accumulate values from an iterable into a single result.

You need to import it from functools.
from functools import reduce

```
# Example: Multiplying all numbers in a list
numbers = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # Output: 24
```

Combined Example:
Process a list to:

1. Remove negative numbers.
2. Square the remaining ones.
3. Sum the result.

```
from functools import reduce
data = [-5, 3, -1, 2, 7]
# Step 1: Filter out negatives
positives = filter(lambda x: x >= 0, data)
# Step 2: Square them
squared = map(lambda x: x ** 2, positives)
# Step 3: Sum them
total = reduce(lambda x, y: x + y, squared)
print(total)  # Output: 62 (3² + 2² + 7²)
```

29. Introduction to closures and decorators.

Ans:-

Introduction to Closures and Decorators in Python:-

1. Closures

A closure is a function that retains access to variables from its enclosing scope even after that scope has finished executing.

Key Concept:-

A nested function remembers the values of variables in its enclosing function, even if that outer function has finished.

Example:-

```
def outer(msg):
    def inner():
        print(f"Message: {msg}")
    return inner

# Create a closure
greet = outer("Hello")
greet()  # Output: Message: Hello
```

Even though outer() has finished running, the inner function still remembers msg.

When is it useful?

Closures are often used:

- To create function factories
- In decorators (explained next)

- To encapsulate behavior with private data

2. Decorators
A decorator is a function that takes another function and extends or modifies its behavior without changing its source code.

Syntax:

```
@decorator_name
def function_name():
    pass
```

This is syntactic sugar for:

```
function_name = decorator_name(function_name)
```

Basic Decorator Example:

```
def my_decorator(func):
    def wrapper():
        print("Before the function runs")
        func()
        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

Decorator with Arguments

```
def repeat(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorator

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

greet("Alice")
```

Output:
Hello, Alice!
Hello, Alice!
Hello, Alice!

Connection Between Closures and Decorators
Decorators rely on closures to remember the function they wrap and any arguments passed to the decorator.