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

- Python is a simple, high-level, interpreted language.

Features of Python:

- Simple & Easy → Easy to read and write, similar to English.
- High-Level → You don't need to worry about memory management like in C.
- Interpreted → Runs line by line, so debugging is easier.
- Object-Oriented → Supports classes, objects, and OOP principles.
- Cross-platform → Runs on Windows, Linux, macOS.
- Rich Libraries → Comes with lots of built-in libraries (math, datetime, os, etc.).
- Open Source → Free to use and develop.

2. History and evolution of Python.

- 1980s: Python was conceived by Guido van Rossum in the Netherlands.
- 1991: First version Python 1.0 released.
- 2000: Python 2.0 released with many improvements.
- 2008: Python 3.0 launched — not backward compatible with Python 2.
- Present: Python 3.x is widely used (latest stable version around Python 3.12 in 2025).
- Python is now managed by the Python Software Foundation (PSF).

3. Advantages of using Python over other programming languages.

- Beginner-friendly → Easier than Java, C, or C++.
- Versatile → Used for web development, AI/ML, data science, automation, etc.
- Huge Community → Tons of tutorials, support, and libraries.
- Cross-platform → Write once, run anywhere.
- Rapid Development → Faster coding compared to C/Java.

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

- Install Python (official site: python.org) → Download and install.
- Anaconda → Best for data science, comes with Jupyter Notebook.
- PyCharm → Professional IDE for Python.
- VS Code → Lightweight and widely used, install Python extension.

5. Writing and executing your first Python program.

- print("Hello, World!")

6. Understanding Python's PEP 8 guidelines.

- Python's official style guide, PEP 8 (Python Enhancement Proposal 8), defines conventions for writing clean and consistent Python code. It covers several aspects such as naming conventions, indentation, line length, and more.
- Indentation: Use 4 spaces per indentation level (avoid using tabs).
- Maximum Line Length: Limit all lines to a maximum of 79 characters (72 for docstrings).
- Naming Conventions Python has specific conventions for naming variables, functions, classes, and modules. These conventions help make code more readable and consistent.
- Classes: Use CamelCase (first letter capitalized, no underscores)

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

1. Indentation in Python

- Python does not use {} brackets like C/Java.
- Instead, indentation (spaces) defines code blocks.
- Standard rule (PEP 8): Use 4 spaces per indentation level (avoid tabs).

2. Comments in Python

- Comments help explain code. They are ignored by the interpreter.
- Types of comments

  Single line : # this is comment

  Multiline : “”” this Is comment “”””

3. Naming convention

Python has specific conventions for naming variables, functions, classes, and modules. These conventions help make code more readable and consistent.

8. Writing readable and maintainable code.

- Readable code → Easy to understand for you and others.
- Maintainable code → Easy to update, debug, and reuse in the future.

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

1. Integers (int)

- Whole numbers (positive or negative).
- No decimal point.

age = 21

2. Floats (float)

- Numbers with decimal points.

pi = 3.14

3. Strings (str)

- Sequence of characters (text).
- Written inside single ' ', double " ", or triple quotes """ """.

name = "Dharmesh"

4. Lists (list)

- Ordered, mutable (changeable) collection.
- Can hold mixed data types.

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

5. Tuples (tuple)

- Ordered, immutable (unchangeable) collection.
- Faster than lists.

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

6. Dictionaries (dict)

- Key-Value pairs (like a real dictionary).
- Keys must be unique & immutable.

```
student = {
   "name": "Dharmesh",
   "age": 21,
   "course": "Python"
}
```

7. Sets (set)

- Unordered, mutable, no duplicate elements.

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

10. Python variables and memory allocation.

1. What is a Variable in Python?

- A **variable** is just a **name (label)** that refers to a value stored in memory.
- In Python, you don't need to declare a variable type (like in C/Java).
- Python decides the type automatically at runtime.

2. Memory allocation

1. Static Memory Allocation (Compile-time)

- Done for fixed-size data like function names, variable names, and constants.
- Handled by the Python Memory Manager automatically.
- Example:

```
PI = 3.14   # memory is allocated once for constant value
```

2. Dynamic Memory Allocation (Run-time)

- Most variables in Python are allocated dynamically when the program runs.
- Objects like int, float, list, dict, etc. are stored in heap memory.
- Variables are just references (pointers) to these objects.

Example:

```
x = 10
y = [1, 2, 3]
```

- 10 is stored in heap → x points to it.
- A list [1,2,3] is created in heap → y points to it.

3. Small Object Pooling (Optimization)

- Python reuses memory for small integers (-5 to 256) and strings (interning).
- Example:

```
a = 100
b = 100
print(id(a), id(b))  # Same memory address
```

4. Garbage Collection

- If no variable references an object, memory is freed automatically by Python's Garbage Collector.

```
x = 50
x = None  # old 50 object has no reference → collected
```

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

1.Arithmetic Operators

- + → Addition
- - → Subtraction
- * → Multiplication
- / → Division
- // → Floor Division
- % → Modulus (remainder)
- ** → Exponentiation

2. Comparison Operators

- == → Equal to
- != → Not equal to
- > → Greater than
- < → Less than
- >= → Greater than or equal to
- <= → Less than or equal to

3. Logical Operators

- and → True if both conditions are True
- or → True if at least one condition is True
- not → Negates the condition

4. Bitwise Operators

- & → Bitwise AND
- | → Bitwise OR
- ^ → Bitwise XOR
- ~ → Bitwise NOT (inverts bits)
- << → Left shift
- >> → Right shift

12. Introduction to conditional statements: if, else, elif.

1. if Statement

- Used to test a condition.
- If the condition is True, the code block runs.

2. else Statement

- Used with if.
- Runs when the if condition is False.

3. elif Statement

- Short for else if.
- Used to check multiple conditions one by one.
- Only the first True condition will execute.

13. Nested if-else conditions.

- A nested if-else means placing one if or if-else inside another if or else block.
- Useful when you need to check multiple levels of conditions.

Key Points

- Inner if runs only if outer if condition is True.
- You can nest multiple levels, but keep it readable.
- Avoid too deep nesting → consider logical operators or functions instead.

14. Introduction to for and while loops.

- Loops allow you to repeat a block of code multiple times.
- Python has two main types of loops: for and while.

1. for Loop

- Used to iterate over a sequence (like a list, tuple, string, or range).
- Executes the code block for each item in the sequence.
- Best when you know how many times you want to repeat.

2. while Loop

- Repeats a block of code as long as a condition is True.
- Best when you don't know the number of iterations in advance.

3. Key Points

- Loops help avoid repetitive code.
- Can use break to exit a loop early.
- Can use continue to skip the current iteration.
- Both loops can have an else block that executes if the loop finishes normally.

15. How loops work in Python.

- oops are used to repeat a block of code until a certain condition is met or until all items in a sequence are processed.

1. for Loop

- Python's for loop iterates over a sequence (list, tuple, string, range, etc.).
- For each item in the sequence:
    1. The loop variable takes the value of the current item.

2. The code block inside the loop executes.
3. Moves to the next item until the sequence ends.

Flow:

Start → Take first item → Execute code → Next item → Repeat → End

2. while Loop

- Python's while loop executes as long as a condition is True.
- The steps:
  1. Check the condition.
  2. If True → execute the code block.
  3. Recheck the condition.
  4. Repeat until the condition becomes False.

Flow:

Start → Check condition → True? → Execute code → Repeat → Condition False → End

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

a) List

- Ordered, mutable collection.

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

b) Tuple

- Ordered, immutable collection.

```
numbers = (1, 2, 3)
for num in numbers:
   print(num)
```

c) Set

- Unordered collection, no duplicates.

```
colors = {"red", "green", "blue"}
for color in colors:
   print(color)
```

d) Dictionary

- Iterate over keys, values, or key-value pairs.

```
student = {"name": "Dharmesh", "age": 21}
```

```
for key in student:
    print(key, student[key])
```

- Or:

```
for key, value in student.items():
    print(key, value)
```

2. Using while Loop

- Can also iterate over collections using an index (for ordered collections like lists/tuples).

```
fruits = ["apple", "banana", "mango"]
i = 0
while i < len(fruits):
    print(fruits[i])
    i += 1
```

17. Understanding how generators work in Python.

- generators are a type of iterable, like lists or tuples, but unlike lists, they do not store all the values in memory at once. Instead, a generator produces items one at a time, only when they are needed. This makes generators more memory-efficient when working with large datasets or streams of data. Generators can be created using:

1. Generator functions – A function that uses the yield keyword.

2. Generator expressions – A concise syntax for creating a generator

1. Generator Functions: A generator function is a function that contains one or more yield statements. When called, it returns a generator object but does not execute the function immediately. Each time the generator is iterated over, the function runs until it hits the next yield statement.

Syntax:

```
def generator_function():

yield value1

yield value2

# More yield statements can follow…
```

Generators do not generate all items at once and do not store them in memory. This makes them more memory-efficient for large datasets. Real world example

2. Generator Expressions: A generator expression is similar to a list comprehension, but instead of creating a list, it creates a generator object. The syntax is almost identical to list comprehensions, but with parentheses () instead of square brackets [].

Syntax : (expression for item in iterable)

18. Difference between yield and return.

Feature	return	yield
Function type	Normal function	Generator function
Behavior	Ends function execution	Pauses function, resumes later
Values produced	One value only	Multiple values (one at a time)
Memory usage	Stores all values if multiple returns needed	Generates values lazily (memory efficient)
Usage	Used in regular computations	Used for generators, streams, large data

19. Understanding iterators and creating custom iterators.

1. What is an Iterator?

- An iterator is an object in Python that allows you to traverse through a sequence of elements one by one.
- Technically, an iterator must implement two methods:
    1. __iter__() → returns the iterator object itself.
    2. __next__() → returns the next element in the sequence.
        - Raises StopIteration when no elements are left.

2. Creating a Custom Iterator

We can create our own iterator class by defining __iter__() and __next__().

Example:

```
class CountDown:
    def __init__(self, start):
        self.current = start

    def __iter__(self):   # returns the iterator object
        return self

    def __next__(self):   # defines how to get next value
        if self.current <= 0:
            raise StopIteration
        value = self.current
        self.current -= 1
        return value

# Using custom iterator
for num in CountDown(5):
    print(num)
```

20. Defining and calling functions in Python.

1. What is a Function?

- A **function** is a **block of reusable code** that performs a specific task.
- Helps **avoid repetition** and **organize code**.

2. Defining a Function

- Use the **def** keyword followed by the function name and parentheses ().
- The code block inside the function is **indented**.

Syntax:

```
def function_name(parameters):
    """Optional docstring explaining the function"""
    # code block
    return value  # optional
```

Notes:

- Parameters (or arguments) are optional.
- return is optional; if omitted, the function returns None.

3. Calling a Function

- To execute a function, use its name followed by parentheses.
- Pass arguments if the function requires them.

Syntax:

```
function_name(arguments)
```

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

1**. Positional Arguments**

- Values are passed in order.
- The first value goes to the first parameter, the second to the second, and so on.

Example:

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

print(add(5, 10))  # 5 goes to 'a', 10 goes to 'b'
```

2. Keyword Arguments

- You specify the parameter name when passing values.
- The order doesn’t matter, since names are used.

Example:

```
def introduce(name, age):
   print(f"My name is {name} and I am {age} years old.")

introduce(age=21, name="Dharmesh")
```

3. Default Arguments

- You can assign default values to parameters.
- If no value is passed, the default is used.

Example:

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

greet("Dharmesh")  # uses given value
greet()            # uses default value "Guest"
```

22. Scope of variables in Python.

1. Local Scope

- Variables declared inside a function are local.
- They can be used only within that function.

```
def my_func():
   x = 10  # local variable
   print(x)

my_func()
# print(x)  # ✕ Error: x not defined outside
```

2. Global Scope

- Variables defined outside all functions.
- Accessible everywhere inside the module.
- Can be modified inside a function using the global keyword.

```
x = 100  # global variable

def func():
   global x
   x = 200
```

```
    print("Inside:", x)

func()
print("Outside:", x)
```

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

1. String Methods

Strings in Python have many useful methods:

- Case Conversion
  - upper() → converts to uppercase.
  - lower() → converts to lowercase.
  - title() → capitalizes each word.
  - capitalize() → capitalizes first letter.
- Searching & Checking
  - find(sub) → returns index of substring (or -1).
  - startswith(sub) → checks if string starts with substring.
  - endswith(sub) → checks if string ends with substring.
  - isdigit() → checks if all characters are digits.
  - isalpha() → checks if all characters are letters.
  - isspace() → checks if string has only spaces.
- Manipulation
  - replace(old, new) → replaces substring.
  - strip() → removes spaces (or chars) from both ends.
  - split(delimiter) → splits string into list.
  - join(list) → joins list items into string.

2. List Methods

Lists are mutable and support many operations:

- Adding & Removing
  - append(x) → adds item at the end.
  - insert(i, x) → inserts at index.
  - extend(list) → adds multiple items.
  - remove(x) → removes first occurrence.
  - pop(i) → removes item at index (default last).
  - clear() → removes all elements.
- Searching & Counting
  - index(x) → returns index of first occurrence.
  - count(x) → counts occurrences.
- Sorting & Reversing
  - sort() → sorts list in place.
  - reverse() → reverses list order.
- Copying
  - copy() → returns shallow copy.

3. Tuple Methods

Tuples are immutable, so methods are fewer:

- count(x) → counts occurrences.
- index(x) → returns index of first occurrence.

4. Set Methods

Sets are unordered and don't allow duplicates:

- add(x) → adds an element.
- remove(x) → removes element (error if not found).
- discard(x) → removes element (no error if not found).
- pop() → removes random element.
- clear() → removes all elements.
- union(other) → returns union.
- intersection(other) → returns intersection.
- difference(other) → returns difference.

5. Dictionary Methods

Dictionaries store key-value pairs:

- Accessing
  - get(key, default) → returns value or default.
  - keys() → returns all keys.
  - values() → returns all values.
  - items() → returns key-value pairs.
- Updating
  - update(dict) → adds/updates entries.
  - pop(key) → removes key and returns value.
  - popitem() → removes last inserted pair.
  - clear() → removes all items.

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

1. break Statement

- Role: Immediately exits the loop, even if the loop condition is still True.
- After break, control moves to the first statement after the loop.

Use Case → Stop looping when a certain condition is met.

```
for i in range(1, 6):
    if i == 3:
        break
    print(i)
```

2. continue Statement

- Role: Skips the current iteration and moves to the next one.
- The loop itself does not stop, only that iteration is skipped.

Use Case → Ignore unwanted values during looping.

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)
```

3. pass Statement

- Role: A do-nothing statement.
- Acts as a placeholder when a statement is syntactically required but no action is needed.

Use Case → Keep empty code blocks without errors.

```
for i in range(1, 6):
    if i == 3:
        pass  # does nothing
    print(i)
```

25. Understanding how to access and manipulate strings.

1. Accessing Strings

Strings in Python are sequences of characters. You can access them like lists.

a) Indexing

- Each character has a position (index).
- Indexing starts at 0 for the first character.

```
text = "Python"
print(text[0])   # P
print(text[5])   # n
print(text[-1])  # n (negative index → from end)
```

b) Slicing

- Extract part of a string using [start:end:step].

```
text = "Python"
print(text[0:4])   # Pyth (index 0 to 3)
print(text[:4])    # Pyth (default start=0)
print(text[2:])    # thon (from index 2 to end)
print(text[::2])   # Pto (step = 2)
```

2. Manipulating Strings

- Concatenation (Joining Strings)

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

- Repetition

```
word = "Hi"
print(word * 3)  # HiHiHi
```

- Changing Case

```
text = "python"
print(text.upper())   # PYTHON
print(text.lower())   # python
print(text.title())   # Python
print(text.capitalize())  # Python
```

- Stripping Spaces

```
msg = "   hello   "
print(msg.strip())   # "hello" (removes both sides)
print(msg.lstrip())  # "hello   " (left only)
print(msg.rstrip())  # "   hello" (right only)
```

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

1. Concatenation (Joining Strings)

- You can join two or more strings using the + operator.

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

2. Repetition (Repeating Strings)

- You can repeat a string multiple times using the * operator.

```
word = "Hi "
print(word * 3)   # Hi Hi Hi
```

3. Common String Methods

Method	Description	Example
upper()	Converts all characters to uppercase	"python".upper() → "PYTHON"

Method	Description	Example
lower()	Converts all characters to lowercase	"PyThOn".lower() → "python"
title()	Capitalizes first letter of each word	"hello world".title() → "Hello World"
capitalize()	Capitalizes first letter of the string	"python".capitalize() → "Python"
strip()	Removes leading & trailing spaces	" hello ".strip() → "hello"
replace(old, new)	Replaces part of a string	"I love Java".replace("Java", "Python") → "I love Python"
split(delimiter)	Splits string into list	"a,b,c".split(",") → ['a', 'b', 'c']

27. String slicing.

String Slicing

Slicing means extracting a portion of a string using the syntax:

string[start:end:step]

- start → index where slice begins (default = 0)
- end → index where slice stops (not included)
- step → jump between characters (default = 1)

1. Basic Slicing

```
text = "Python"
print(text[0:4])   # Pyth  (index 0 to 3)
print(text[:4])    # Pyth  (start defaults to 0)
print(text[2:])    # thon  (end defaults to last)
print(text[:])     # Python (entire string)
```

2. Negative Index Slicing

- Negative indices count from the end (-1 = last char).

```
text = "Python"
print(text[-3:])   # hon  (last 3 chars)
print(text[:-3])   # Pyt  (all except last 3)
```

3. Step in Slicing

- Controls the gap between characters.

```
text = "Python"
print(text[::2])   # Pto   (every 2nd character)
print(text[1::2])  # yhn   (every 2nd char from index 1)
```

4. Reversing a String

- Use a step of -1.

```
text = "Python"
print(text[::-1])   # nohtyP
```

28. How functional programming works in Python.

What is Functional Programming (FP)?

Functional programming is a **programming paradigm** that treats **functions as first-class citizens**. This means:

1. Functions can be stored in variables.
2. Functions can be passed as arguments to other functions.
3. Functions can return other functions.

In Python, FP is supported alongside **object-oriented** and **procedural** styles.

1. **First-Class Functions**

- Functions can be assigned to variables and called through them.

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

say_hello = greet   # function assigned to variable
print(say_hello("Python"))  # Hello, Python
```

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

1. map()

📌 Purpose: Applies a function to each element of an iterable and returns a new iterable (map object).

Syntax:

```
map(function, iterable)
```

Example:

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

2. filter()

Purpose: Filters elements from an iterable based on a condition (function that returns True/False).

Syntax:

```
filter(function, iterable)
```

Example:

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

3. reduce()

Purpose: Reduces an iterable to a single cumulative value by applying a function repeatedly.
Available in functools module.

Syntax:

```
from functools import reduce
reduce(function, iterable, initializer)
```

Example:

```
from functools import reduce

nums = [1, 2, 3, 4, 5]
sum_all = reduce(lambda x, y: x + y, nums)
print(sum_all)   # 15
```

30. Introduction to closures and decorators.

1. Closures in Python

Definition:

A **closure** is a function that **remembers variables** from its enclosing scope, even if the outer function has finished executing.

How it works:

- You define a function inside another function.
- The inner function uses variables from the outer function.

- The inner function is returned, and it “remembers” those variables.

2. Decorators in Python

Definition:

A **decorator** is a function that takes another function as input, adds some extra functionality to it, and returns a new function **without changing the original function code**.

How it works:

- Functions are **first-class objects** (they can be passed around).
- A decorator usually uses a **closure** inside.