(1) .Introduction to Python

1. Introduction to Python and its Features

Python is a **simple**, **high-level**, and **interpreted** programming language.

- **Simple:** Easy to read and write. The syntax is like English.
- **High-level:** You don't need to manage memory manually. Python does it automatically.
- **Interpreted:** Python runs your code line-by-line, which makes debugging easier.

Main Features of Python:

- Easy to learn and use
- Cross-platform (runs on Windows, Mac, Linux)
- Open-source and free to use
- Huge community support
- Large standard library (math, web, system, etc.)
- Used in web development, data science, machine learning, automation, game development, and more

2. History and Evolution of Python

- **Created by**: Guido van Rossum
- **Year**: First released in **1991**
- **Inspired by**: ABC language and named after "Monty Python's Flying Circus" (a comedy show)
- **Major Versions**:
    - Python 2.x – Older version, now discontinued
    - Python 3.x – Current version, actively maintained and improved

Python has grown rapidly and is now one of the most popular programming languages in the world.

3. Advantages of Using Python Over Other Languages

- **Readable and clear syntax**
- **Fewer lines of code** compared to Java or C++
- **Extensive libraries** (for AI, ML, Web, GUI, etc.)
- **Portable** – Write once, run anywhere
- **Good for beginners** and professionals
- **Versatile** – Used in many fields like data science, automation, web apps, games, etc.

4. Installing Python and Setting Up the Development Environment

A. Install Python:

- Download from https://www.python.org
- Install and check version with:

B. IDE Options:

1. **VS Code** (Recommended for beginners)
    - Lightweight and customizable
    - Add Python extension

2. **PyCharm**
    - Powerful Python IDE with many tools built-in

3. **Anaconda**
    - Best for data science
    - Comes with Python + Jupyter + many data libraries

5. Writing and Executing Your First Python Program

Step 1: Open any code editor (like VS Code)

Step 2: Write this simple program:

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

Step 3: Save it as `hello.py`

Step 4: Run the program:

- Open terminal/command prompt
- Navigate to the folder where the file is saved

(2). Programming Style

Understanding Python's PEP 8 Guidelines

PEP 8 stands for **Python Enhancement Proposal 8**, and it is the **official style guide** for writing clean and readable Python code.

Why use PEP 8?

- Makes your code look clean and professional
- Helps teams work better together
- Improves readability and reduces errors

Some key rules from PEP 8:

- Use **4 spaces** for indentation (not tabs)
- Keep lines under **79 characters**
- Add **blank lines** to separate code blocks
- Use **meaningful variable names**
- Add **spaces around operators** (`a = b + c`, not `a=b+c`)
- Write **comments** to explain "why", not just "what"

Indentation, Comments, and Naming Conventions

Indentation

Indentation defines code blocks in Python. No curly braces { } are used.

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

Comments

- **Single-line comment** uses #

```
# This is a comment
```

```
print("Hello")  # Inline comment
```

- **Multi-line comments** use triple quotes (though used rarely for comments)
  """
- This is a
- multi-line comment
- """

Naming Conventions

Type	Convention	Example
Variable	lowercase_words	`user_name`
Function	lowercase_words	`calculate_total()`
Class	PascalCase	`StudentData`
Constant	ALL_CAS	`PI = 3.14`
Private Variable	_underscore	`_hidden_value`

Writing Readable and Maintainable Code

Here are **tips** for clean and maintainable code:

1. **Follow PEP 8** for consistent style
2. **Use meaningful names** for variables and functions
   Bad: `a = 10`
   Good: `age = 10`
3. **Write comments** for complex logic
4. **Use functions** to break code into parts
5. **Avoid repetition** (DRY = Don't Repeat Yourself)
6. **Keep lines short** and avoid writing too much in one line
7. **Group related code** together and use blank lines to separate sections

Ex

```
def calculate_area(radius):
    """Calculate area of a circle"""
    PI = 3.1416
    return PI * radius * radius
# Main program
r = 5
area = calculate_area(r)
print("Area:", area)
```

(3). Core Python Concepts

Understanding Data Types in Python

Python has many built-in data types. Here are the most commonly used:

1. Integers (`int`)

Whole numbers — positive or negative.

```
a = 10     # integer
```

2. Floats (`float`)

Decimal numbers (floating-point numbers).

```
b = 3.14   # float
```

3. Strings (`str`)

Sequence of characters, text inside quotes.

```
name = "John"
```

4. **Lists (`list`)**

Ordered, changeable (mutable), allows duplicates.

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

5. Tuples (`tuple`)

Ordered, **unchangeable** (immutable), allows duplicates.

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

6. Dictionaries (`dict`)

Key-value pairs, unordered, mutable.

```
student = {"name": "Alice", "age": 20}
```

7. Sets (`set`)

Unordered, **no duplicates** allowed.

```
unique_numbers = {1, 2, 3, 4}
```

Python Variables and Memory Allocation

What is a variable?

A **variable** stores data in memory. It is a name pointing to a value.

x = 10

Here, x is a variable pointing to value 10.

- Python automatically decides the **data type** based on the value.
- Memory is allocated dynamically when the variable is created.

Example

```
name = "Alice"    # string
age = 21          # int
height = 5.8      # float
```

Python Operators

1. Arithmetic Operators

Used for 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
**	Exponent (power)	a ** b

2. Comparison (Relational) Operators

Used to compare values (returns `True` or `False`):

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 conditions:

Operator	Description	Example
`and`	True if both are true	`a > 5 and b < 10`

`or`	True if any one true	`a > 5 or b < 10`
`not`	Reverses condition	`not(a > 5)`

4. Bitwise Operators

Works on bits (used rarely in basic programming):

Operator	Name	Example
&	AND	a & b
`	`	OR
^	XOR	a ^ b
~	NOT	~a
<<	Left Shift	a << 1
>>	Right Shift	a >> 1

(4). Conditional Statements

`if` Statement:

Used to execute a block of code **only if** a specified condition is `True`.

`if-else` Statement:

Used to choose between **two blocks of code** — one runs if the condition is `True`, the other if it is `False`.

`if-elif-else` Statement:

Used when there are **multiple conditions** to check. It allows checking several expressions one after another.

Nested `if-else` Statement:

When one `if` or `else` block contains **another if-else inside it**. It is used for **multi-level decision-making**.

(5). Looping (For, While)

Introduction to `for` and `while` Loops

Loops are used to repeat a block of code multiple times.
In Python, there are mainly two types of loops:

`for` Loop:

Used to iterate over a **sequence** (like list, tuple, string, etc.).
It runs once for **each item** in the sequence.

`while` Loop:

Repeats a block of code **as long as the condition is true**.
It checks the condition **before every loop iteration**.

How Loops Work in Python

- A loop starts with a **condition**.
- If the condition is **True**, the loop executes the code block.
- The loop continues **until** the condition becomes **False**.
- You can control loops using keywords like **`break`** (to stop early) or **`continue`** (to skip to next iteration).

Using Loops with Collections

Python loops are often used with **collections** like:

- **Lists**
- **Tuples**
- **Strings**
- **Dictionaries**
- **Sets**

The loop goes through each item in the collection one-by-one.

(6). Generators and Iterators

Understanding How Generators Work in Python

- **Generators** are a way to create **iterators** in Python.
- They allow you to **generate values one at a time**, instead of storing all of them in memory.
- A generator **remembers its state** between function calls.
- You create a generator using a **function that contains `yield`** instead of `return`.

Difference Between `yield` and `return`

`return`	`yield`
Ends the function completely	Pauses the function and saves state
Returns **a single value**	Returns **a generator object**
Cannot be resumed	Can be resumed from where it left
Used in normal functions	Used in generator functions

Understanding Iterators

- An **iterator** is any object that can be looped over.
- It follows the **iterator protocol**, meaning it has `__iter__()` and `__next__()` methods.
- Built-in data types like **lists, tuples, sets, strings** are all iterable.

Creating Custom Iterators

- You can create your own iterator by defining a **class** with:
    - `__iter__()` – returns the iterator object
    - `__next__()` – returns the next
    - Custom iterators are useful when you want to define your **own logic** for generating sequence data.

(7). Functions and Methods

Defining and Calling Functions in Python

- A **function** is a block of reusable code that performs a specific task.
- Functions help in making the code **modular, readable, and reusable**.
- In Python, you define a function using the `def` keyword, and you call (run) it by using its name followed by parentheses.

Function Arguments in Python

Python functions can take different types of arguments:

1. **Positional Arguments**:
    - Passed in the correct **order** defined in the function.

2. **Keyword Arguments**:
    - Passed using **key = value** format.
    - Order doesn't matter when using keywords.

3. **Default Arguments**:
    - Have a default value if no value is provided during the call.

Python also supports variable-length arguments (`*args`, `**kwargs`), but that's more advanced.

Scope of Variables in Python

- The **scope** of a variable determines **where it can be accessed**.
- There are two main types of scope:

1. **Local Scope**:
    - Variables declared **inside a function**.
    - Can only be used within that function.

2. **Global Scope**:
    - Variables declared **outside any function**.
    - Can be accessed anywhere in the program.

Python also supports `nonlocal` for nested functions.

Built-in Methods for Strings, Lists, etc.

Python provides **built-in methods** to work easily with data types like:

- **Strings**:
  Useful for modifying or checking text.
  (e.g., converting case, finding characters, replacing text)

- **Lists**:
  Methods to add, remove, sort, or search items in a list.

- **Tuples, Dictionaries, Sets** also have their own useful methods.

These methods make it easy to **process and manipulate data** efficiently.

(8). Control Statements (Break, Continue, Pass)

Understanding the Role of `break`, `continue`, and `pass` in Python Loops

These three keywords are used to control the **flow of loops** (`for` and `while`):

`break`

- Used to **exit** the loop **immediately**, even if the loop condition is still true.
- It **stops the entire loop** and moves to the code after the loop.

`continue`

- Used to **skip the current iteration** and move to the **next iteration** of the loop.
- The rest of the loop body for that cycle is **not executed**.

`pass`

- Does **nothing** — it's a **placeholder**.
- Used when the code is **not written yet** but you want the program to run without error.
- Commonly used in loops, functions, or condition blocks where code will be added later.

(9). String Manipulation

Understanding How to Access and Manipulate Strings

- A **string** in Python is a **sequence of characters** enclosed in single (') or double (") quotes.

- Strings are **immutable**, meaning you cannot change characters directly, but you can create modified copies.

You can **access individual characters** in a string using **indexing**, and perform many operations to **manipulate** the text.

Basic Operations on Strings

Concatenation

- Joining two or more strings together to form one combined string.

Repetition

- Repeating the same string multiple times using an operator.

String Methods

Python provides several **built-in methods** to modify and work with strings:

- `upper()` – Converts to uppercase
- `lower()` – Converts to lowercase
- `strip()` – Removes spaces from beginning and end
- `replace()` – Replaces a part of the string
- `find()` – Finds the position of a character or substring
- `split()` – Splits the string into a list

String Slicing

- **Slicing** is used to extract a portion (substring) of a string.
- It is done using **start:stop** syntax inside square brackets.
- Python also supports **negative indexing** to slice from the end of the string.

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

How Functional Programming Works in Python

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

- Functions can be passed as arguments
- Functions can be returned from other functions
- You can write **clean, concise, and reusable** code using built-in functions

Python supports functional programming with:

- **Lambda functions** (anonymous functions)
  Higher-order functions like `map()`, `filter()`, `reduce()`
- **Closures** and **Decorators**

Using `map()`, `reduce()`, and `filter()`

These functions are used to **process collections (like lists)** in a functional way:

`map()`

Applies a given function to **each item** in a collection and returns a new collection with the results.

`filter()`

Filters items from a collection **based on a condition** (true or false) and returns only the matching items.

`reduce()`

Used to **reduce** a collection to a **single value** by repeatedly applying a function (from `functools` module).

Introduction to Closures

A **closure** is a function that:

- Is defined inside another function
- **Remembers** the variables from the outer function, even after that function has finished executing

Closures are useful when you want to **preserve a value** or **create a customized function** on the fly.

Introduction to Decorators

A **decorator** is a function that **modifies or extends the behavior** of another function, without changing its structure.

Decorators are often used for:

- Logging
- Access control
- Timing functions
- Adding extra functionality to existing code

They are written using the `@decorator_name` syntax and are widely used in **frameworks like Flask and Django**.