**1. Programming Fundamentals & Translation**

**What Is Programming?**

**Programming** is the process of writing detailed instructions (**code**) for a computer to follow to execute specific tasks. Since computers only understand **binary** (0s and 1s), we use **high-level languages** like Python and Java, which are much easier for humans to read and write.

**Compiler vs. Interpreter (The Translators)**

High-level code must be translated into machine code using one of two methods:

| Tool | How It Works | Key Difference |
| --- | --- | --- |
| **Compiler** | Translates the **entire code** into an executable machine file **all at once** before execution. | **Faster execution** but debugging is often tougher, as errors are reported after the whole process. |
| **Interpreter** | Translates and executes the code **line-by-line** as the program runs. | **Easier debugging** because execution stops precisely where an error occurs (e.g., **Python**). |

**Language Levels:**

* **High-Level Languages** (e.g., Python, Java): Closer to human language, easier to write, but generally slower.
* **Low-Level Languages** (e.g., Assembly): Closer to machine code, harder to write, but execute much faster.

**2. Introduction to Python Language and Careers**

**Python** is a highly popular, **high-level, interpreted** language created by **Guido van Rossum** in 1991. It is famous for its **simplicity and readability**, using straightforward syntax like a+b for addition.

Python is a versatile tool, supporting **functional** and **object-oriented programming** styles, making it scalable and powerful enough for major professional tasks in:

* Data Science & Analytics
* Artificial Intelligence (AI) and Machine Learning (ML)
* Web Development
* Automation and Scripting

| Career Role | Focus Area | Salary Potential (LPA) |
| --- | --- | --- |
| **Data Scientist** | AI modeling, in-depth data analysis | High range (10 - 30) |
| **ML Engineer** | Building core AI models, utilizing tools like TensorFlow | Very high range (12 - 35) |
| **Cyber Security Specialist** | Network security and automated scripting | Mid range (10 - 20) |
| **DevOps Engineer** | Automating software deployment and infrastructure | Mid-to-high range (10 - 25) |

**3. Programming Fundamentals: Variables & Data Types**

**Variables and Comments**

A **Variable** is a named **container** used to store data. Think of it as a labeled box that holds a value.

**Naming Rules:**

1. Must not start with a number (e.g., 1variable is invalid).
2. Cannot contain spaces.
3. Only the underscore (\_) is allowed as a special symbol.

To display the data, we use the variable name inside the print() function.

**Comments** are notes within the code that are completely ignored by the interpreter. They are essential for explaining complex logic or temporarily disabling code.

* **Single-line comment:** Starts with #.
* **Multi-line comment:** Enclosed by triple quotes: '''...''' or """...""".

**Data Types: Defining the Data**

The **data type** defines what kind of value a variable holds.

| Type Category | Type | Description | Key Examples |
| --- | --- | --- | --- |
| **Numeric** | int | Whole numbers (integers). | 100,−5 |
|  | float | Numbers with a decimal point. | 3.14,−10.5 |
|  | complex | Numbers with an imaginary part (less common). | 3+4j |
| **Logical** | bool | Represents two states: **True** or **False**. | Used for decision-making. |
| **Null** | None | Represents the **absence of a value**. | Used for uninitialized variables. |
| **Sequence** | String | Ordered sequence of characters (text). **Immutable** (cannot be changed). | "Hello World" |
|  | List | Ordered collection of mixed data. **Mutable** (can be changed). | ['apple', 1, 3.14] |
|  | Tuple | Ordered collection of mixed data. **Immutable** (cannot be changed). | (1, 2, 'c') |
| **Mapping** | Dictionary | Unordered collection of unique **key-value pairs**. **Mutable**. | {'name': 'Bob', 'age': 25} |
| **Set** | set | Unordered collection of **unique** elements. | Used for removing duplicates. |

**4. Operators and Control Flow**

**Operators: Performing Actions**

**Operators** are symbols that perform mathematical or logical actions on data (**operands**).

|  |  |  |
| --- | --- | --- |
|  | | |
| Operator Type | Purpose | Examples |
| **Arithmetic** | Math operations: +, -, \*, /, **floor division** (//), **remainder** (%), **exponent** (\*\*). | 5+3 |
| **Comparison** | Compares values: == (equal), != (not equal), >, <, >=, <=. | x>5 → True/False |
| **Logical** | Combines Boolean results: **and**, **or**, **not**. | age > 18 and knows\_code |
| **Assignment** | Assigns or updates values: =, +=, -=, etc. | x=10, x+=2 |
| **Identity** | Checks if variables point to the **same memory location**: is, is not. | A is B |
| **Membership** | Checks if an element is in a sequence: in, not in. | 'a' in 'apple' |

**Precedence:** Operations follow the **PEMDAS** rule (Parentheses, Exponents, Multiplication/Division, Addition/Subtraction).

**Control Statements (Conditional Logic and Loops)**

Control statements manage the program's flow, enabling it to execute code conditionally or repeatedly.

**Conditional Statements (Decisions):**

* **if:** Executes a block of code if the condition is **True**.
* **elif (Else If):** Checks an additional condition if the preceding if was **False**.
* **else:** Executes a block if all preceding conditions were **False**.

**Loops (Repetition):**

* **while loop:** Repeats a block of code **as long as** its condition remains **True**.
* **for loop:** Iterates a fixed number of times over items in a **sequence** (like a list or string).

**Loop Control:**

* **break:** Immediately **stops** and exits the entire loop.
* **continue:** Skips the rest of the code in the **current** loop iteration and moves to the next one.
* **pass:** A placeholder that does **nothing** but is used to maintain valid syntax (e.g., in a planned but empty function).

**5. Advanced Data Structures and Functions**

**Strings (Text)**

A **String** is a sequence of characters, created using single, double, or triple quotes.

* **Immutable:** You cannot change a string after it's created.
* **Indexing:** Characters are accessed by position, starting at 0 from the left (positive) or −1 from the right (negative).

**Common String Methods:**

* **len():** Gets the length.
* **+ and \*:** Concatenates (joins) or repeats strings.
* **lower(), upper():** Changes case.
* **find(), replace():** Searches for or substitutes substrings.
* **split(), join():** Converts a string to a list or a list back to a string, respectively.

**Lists (Ordered, Mutable Collections)**

A **List** is an ordered, dynamic, and **mutable** collection of elements, created with square brackets []. It can store mixed data types (**heterogeneous**).

* **Common Methods:** append (add to end), insert (add at index), remove, pop (remove and return), sort, and reverse.
* **Copying:** Use the copy() method to create a truly independent copy, avoiding issues where two variables point to the same data (**aliasing**).

**Dictionaries (Key-Value Pairs)**

A **Dictionary** stores data as **Key: Value** pairs, created with curly braces {}. It is **mutable** and designed for fast lookup.

* **Keys** must be **unique** and immutable (e.g., strings or numbers). Values can be anything.
* **Access:** Values are retrieved using their unique key, e.g., dictionary['key'].
* **Common Methods:** get() (safer access), keys(), values(), and items() (get pairs).

**Functions (Reusable Code)**

A **Function** is a named, reusable block of code defined using the **def** keyword. They improve code organization, reusability, and maintenance.

* **Parameters** are placeholders in the definition; **Arguments** are the actual values passed when the function is **called**.
* The **return** statement sends a value back to the caller and exits the function.
* **Local variables** exist only inside the function; **Global variables** exist outside and are accessible everywhere.
* **Lambda Functions:** Small, anonymous (unnamed), single-expression functions often used for quick tasks.

**6. Object-Oriented Programming (OOP)**

**OOP** is a powerful programming style that models real-world objects by grouping related data (**attributes** or properties) and functions (**methods** or behaviours) into **Objects**. It helps solve issues like code duplication and poor maintainability.

| OOP Concept | Simple Explanation | Implementation in Python |
| --- | --- | --- |
| **Class** | The **blueprint** or template for creating objects. | Defined with the class keyword. |
| **Object** | A specific **instance** created from a Class blueprint. | my\_car = Car() |
| **self** | A reference to the object itself inside its methods. | The required first parameter in instance methods. |
| **Constructor** | The special \_\_init\_\_ method, which automatically sets up initial attributes when an object is created. | def \_\_init\_\_(self, brand): |
| **Inheritance** | A Child Class takes all the properties and methods from a Parent Class, promoting code reuse. | class SportsCar(Car): |
| **Abstraction** | Hiding complex implementation and only showing the essential features to the user. | Often achieved using **Abstract Base Classes**. |

**Advantages of OOP:-**

* Modularity : Code is modulator and easier to debug and maintain .
* Reusability : Classes and methods can be reused in multiple programs.
* Scalability : Easier to scale and extend
* Security: Encapsulation hides the data ,ensuring security

**7. Exception Handling and File Operations**

**Exception Handling (Error Management)**

**Errors** disrupt program flow. **Exceptions** are runtime errors (like division by zero) that can be **caught and handled** to prevent the program from crashing.

The core structure for handling exceptions is the **try...except** block:

Python

try:

# Code where an error might occur

result = a / b

except ZeroDivisionError:

# Code to run IF a specific error happens

print("Cannot divide by zero!")

finally:

# Code that ALWAYS runs, regardless of success or error (good for cleanup)

print("Operation finished.")

You can catch multiple specific exceptions or use raise Exception() to create **Custom Exceptions** based on your own logic (e.g., password too short).

**File Handling (Permanent Data Storage)**

**File Handling** is the process of reading data from or writing data to permanent storage (files).

**Basic Steps:** 1. **Open** the file. 2. **Perform** read/write operations. 3. **Close** the file.

| Mode | Purpose | Key Behavior |
| --- | --- | --- |
| **'r'** | Read-only | File must already exist. |
| **'w'** | Write | **Overwrites** existing file or creates a new one. |
| **'a'** | Append | Adds data to the **end** of the file or creates a new one. |
| **'r+'**, **'w+'**, **'a+'** | Combine reading and writing functions. | Depends on base mode (r, w, or a). |

Export to Sheets

**Best Practice:** Use the **with open(...) as file:** statement. It automatically ensures the file is closed, even if errors occur, preventing resource leaks.

**Operations:**

* file.read(): Reads the entire file content.
* file.write(data): Writes data to the file

18. Coding Stadandars

Coding standards are a set of rules and best practices that define how code should be written, organized, and documented to ensure consistency, readability, maintainability, and quality across a project or team.

• Naming Conventions

a. Variables and Functions: Names should be written in snake\_case (all lowercase with words separated by underscores).

Example: total\_amount, calculate\_sum.

b. Classes and Exceptions: Class names should follow PascalCase (each word starts with a capital letter, no underscores).

Example: StudentRecord, PaymentProcessor, FileNotFoundError.

c. Constants: Constants should be written in UPPERCASE letters, with words separated by underscores.

Example: PI, MAX\_LIMIT, DEFAULT\_TIMEOUT.

d. Private and Protected Members: Members meant for internal use should start with an underscore (\_). This convention warns developers not to access them directly.

If stronger privacy is needed, use double underscores (\_\_), which invoke name mangling.

Example: \_balance (protected), \_\_password (private).

e. Modules and Packages: File names and package names should be in lowercase. Underscores can be used if the name is long or contains multiple words.

Example: math\_utils, stringtools.

• Docstrings & Comments

Docstring: Describes what a function, class, or module does.

Written using triple quotes """ ... """.

Example:

def add(a, b):

"""Return the sum of two numbers a and b."""

return a + b

Inline comments: Explain non-obvious code logic.

Use # before comment, keep them short and meaningful.

Example:

result = factorial(n) # Recursive call

• Type of Testing

a. Unit Testing: Tests small pieces (functions, classes) individually.

b. Integration Testing: Tests how modules work together.

c. System Testing: Tests the entire application end-to-end.

d. Acceptance Testing: Validates against business requirements.

e. Regression Testing: Ensures new changes don’t break existing features.

f. Smoke & Sanity Testing: Quick checks before detailed testing.

• PEP 8 (Python Enhancement Proposal 8)

a. Indentation: 4 spaces (no tabs).

b. Line length: max 79 characters.

c. Imports: standard - third-party - local.

d. Spaces around operators: a = b + c not a=b+c.

e. Readable variable names, avoid single letters (except in loops).

• SOLID Principles

S – Single Responsibility: One class = one responsibility.

O – Open/Closed: Open for extension, closed for modification.

L – Liskov Substitution: Subclasses should replace parent classes without breaking code.

I – Interface Segregation: Many small interfaces better than one large one.

D – Dependency Inversion: Depend on abstractions, not concrete implementations.

• DRY Principle (Don’t Repeat Yourself)

Avoid duplicating code; use functions, classes, or modules.

Example: Instead of writing the same calculation in multiple places → put it in one function and call it.

# 