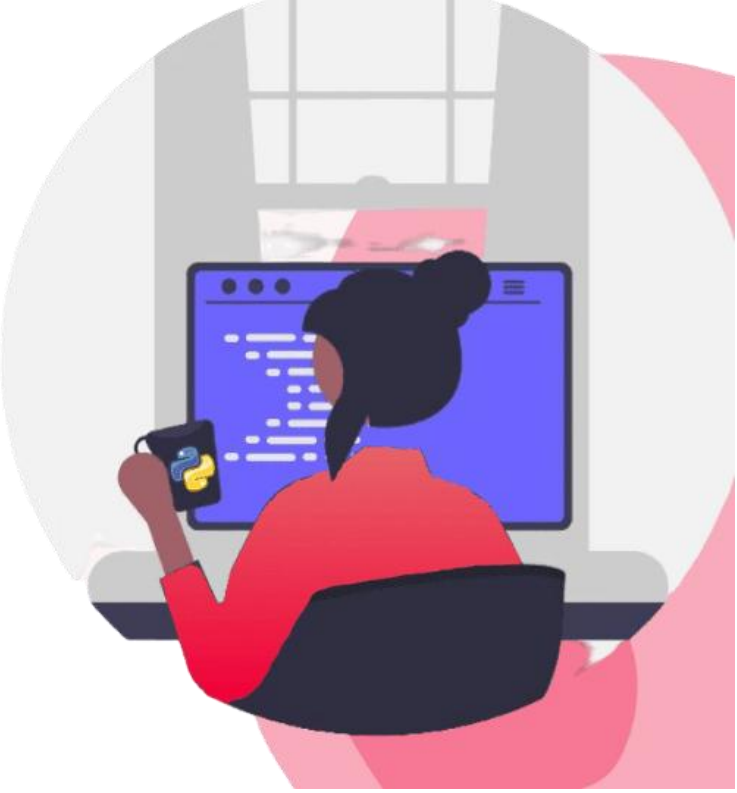
THE ULTIMATE PYTHON HANDBOOK

CONTENTS

INTRODUCTION TO PYTHON



1. Introduction to Python

1.1 What is Python?

- Python is a high-level, general-purpose, interpreted programming language.
- Developed by **Guido van Rossum** and first released in **1991**.
- Known for simple syntax and readability, making it ideal for beginners and advanced users.
- Widely used in **data science**, **machine learning**, **deep learning**, **web development**, **automation**, **scripting**, and **software development**.

2. Key Features of Python

2.1 Easy to Read and Write

- Clean and simple syntax similar to English.
- Reduces development time and makes code maintenance easier.

2.2 Interpreted Language

- Code runs line-by-line.
- No need for compilation.
- Helps in quick development and debugging.

2.3 Dynamically Typed

- No need to declare variable types.
- Type is decided at runtime.
- Increases flexibility during coding.

2.4 Portability

- Python code can run on multiple operating systems:
    - Windows
    - macOS
    - Linux

2.5 Large Standard Library

- Comes with built-in modules for:
    - File handling
    - Networking
    - Regular expressions
    - Data structures

2.6 Extensive Third-Party Libraries

- Essential for AI and ML:
    - **NumPy** – numerical computing
    - **Pandas** – data analysis
    - **Matplotlib / Seaborn** – visualization
    - **Scikit-learn** – machine learning
    - **TensorFlow / PyTorch / Keras** – deep learning

2.7 Object-Oriented and Functional

- Supports classes, objects, inheritance.
- Also supports functional style (lambda, map, filter).

---

3. Why Python for AI, Machine Learning, and Deep Learning?

3.1 Rich Ecosystem of Libraries

- Specialized libraries accelerate model building and training.
- Strong community support and frequent updates.

3.2 Easy to Prototype

- Simple syntax allows fast experimentation.
- Important in ML/AI where models need iterative improvement.

3.3 Strong Community and Support

- Huge tutorials, documentation, GitHub projects, and forums.
- Easier problem solving when stuck.

3.4 Integration Capabilities

- Can integrate with:
    - C/C++ for high performance
    - Java
    - Web services
    - Databases

---

4. Applications of Python in AI/ML

4.1 Data Preprocessing

- Cleaning, transforming, and preparing data using NumPy and Pandas.

4.2 Machine Learning

- Use scikit-learn for classification, regression, clustering.

4.3 Deep Learning

- Build neural networks using TensorFlow, PyTorch, or Keras.

4.4 Natural Language Processing

- Libraries like NLTK, spaCy, Hugging Face Transformers.

4.5 Computer Vision

- OpenCV, TensorFlow, PyTorch for image classification and detection.

---

5. Basic Python Syntax Example

Example

```
# Simple Python program
x = 10
y = 20
sum = x + y
print("Sum =", sum) # Output: 30
```

Python Modules

1. Introduction to Modules

1.1 What is a Module?

- A **module** in Python is a file containing Python code such as functions, classes, or variables.
- Helps in organizing code into smaller, reusable pieces.
- File extension of a module is .py.

1.2 Purpose of Using Modules

- Reduces code duplication.
- Improves code readability and maintainability.
- Allows separation of logic into meaningful components.
- Supports reusability across projects.

---

2. Types of Modules

2.1 Built-in Modules

- Provided by Python automatically.
- No installation needed.
- Examples:
    - **math**
    - **os**
    - **sys**
    - **random**
    - **datetime**

2.2 User-Defined Modules

- Created by programmers.
- Usually written to organize project-specific logic.
- Any .py file can become a module.

2.3 Third-Party Modules

- Installed using package managers like **pip**.
- Useful in advanced fields like **AI/ML**.
- Examples:
    - **numpy**
    - **pandas**
    - **matplotlib**
    - **scikit-learn**
    - **tensorflow**
    - **torch**

---

3. Importing Modules

3.1 Basic Import

```
import math
print(math.sqrt(25))
```

3.2 Import with Alias

```
import numpy as np
```

- Useful for shorter code.

3.3 Import Specific Functions

```
from math import sqrt, pi
```

3.4 Import Everything

```
from math import *
```

- Not recommended due to namespace pollution.

---

4. Creating a User-Defined Module

Step 1: Create a Python file

my_module.py

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

def greet(name):
    return "Hello " + name
```

Step 2: Import and use it

```
import my_module

print(my_module.add(10, 20))
print(my_module.greet("Ravi"))
```

---

5. Module Search Path

5.1 How Python Finds Modules

Python looks for modules in:

1. Current working directory
2. System paths
3. Site-packages (installed libraries)

5.2 Check Module Search Path

```
import sys
print(sys.path)
```

---

6. Special Module: __main__

- Allows a file to act as both:
    - A script (runnable)
    - A module (importable)

- Example:

```
if __name__ == "__main__":
    print("This code runs when executed directly")
```

---

7. Important Modules for AI/ML

7.1 NumPy

- Numerical computing
- Arrays, linear algebra, mathematical functions

7.2 Pandas

- Data analysis and manipulation
- DataFrames, cleaning, and preprocessing

7.3 Matplotlib

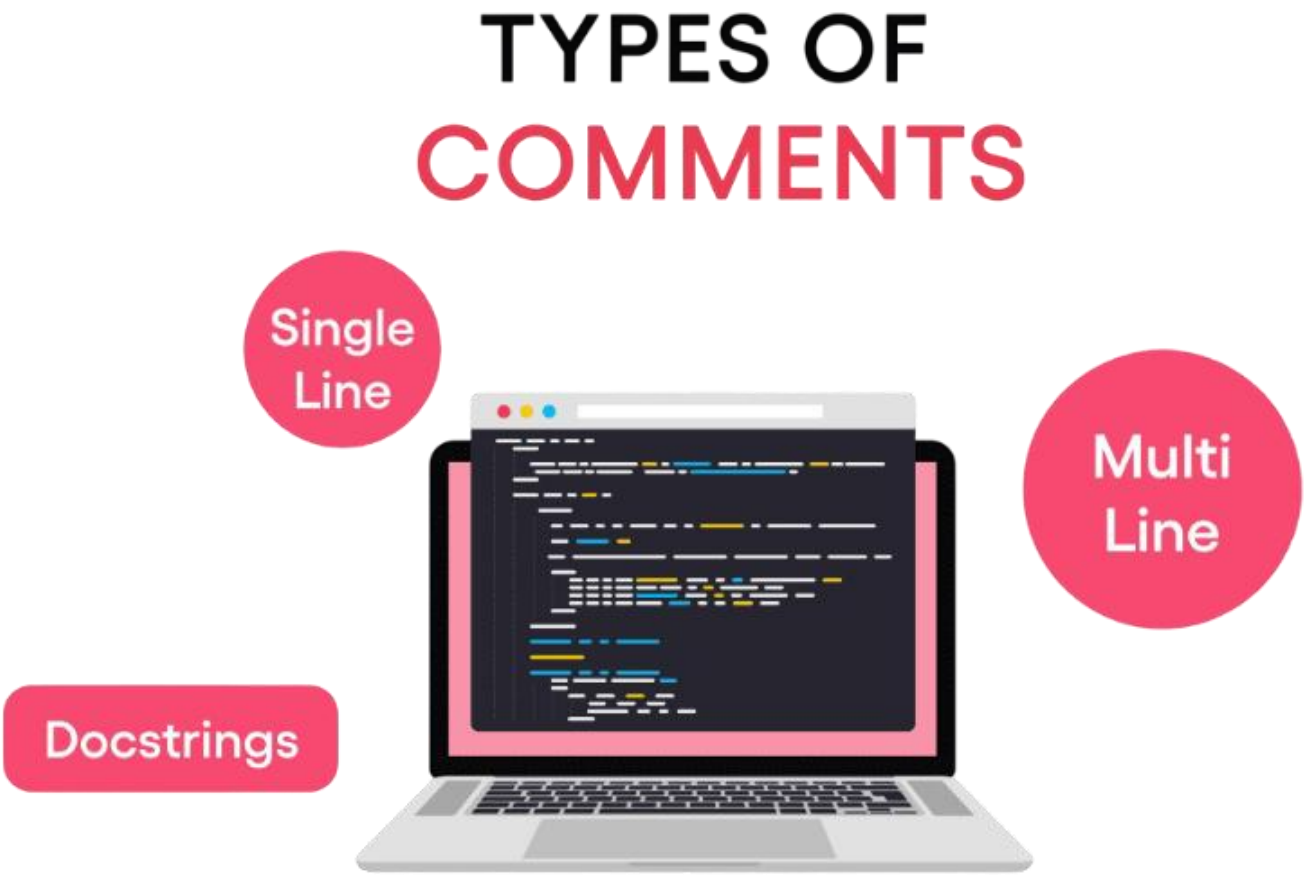
- Data visualization
- Graphs, charts, plots

7.4 Scikit-learn

- Traditional machine learning algorithms
- Classification, regression, clustering

7.5 TensorFlow / PyTorch

- Deep learning
- Neural networks, training models



1. Introduction to Comments

1.1 What is a Comment?

- A comment is a line or text in Python that is **ignored by the interpreter**.
- Used for explaining code, logic, or purpose of a function.
- Makes code readable and maintainable.

1.2 Why Comments Are Important

- Improves understanding of code blocks.
- Helps other developers read and maintain code.
- Useful for documenting functions, classes, and complex logic.
- Essential in AI/ML projects where data processing and models involve many steps.

---

2. Types of Comments in Python

2.1 Single-Line Comment

- Begins with the # symbol.
- Everything after # on that line is treated as a comment.

Syntax

```
# This is a single-line comment
x = 10  # Inline comment
```

2.2 Multi-Line Comment

Python does not have a dedicated multi-line comment syntax.

But multi-line comments can be created using:

- Triple single quotes (''')
- Triple double quotes (""")

Syntax

```
'''
This is a multi-line comment
used to describe a block of code
'''
```

or

```
"""
Multiple lines of explanation
written inside triple quotes
"""
```

---

3. Docstrings (Documentation Strings)

3.1 What is a Docstring?

- A docstring is a string placed inside functions, classes, or modules.
- Used to describe their purpose.
- Accessible using .__doc__.
- Important for professional coding and AI/ML projects.

Syntax

```
def add(a, b):
    """
    This function adds two numbers and returns the result.
    """
    return a + b
```

Accessing Docstring

```
print(add.__doc__)
```

---

4. Commenting Best Practices

4.1 Use Comments for

- Complex algorithms
- Data preprocessing steps
- Model training logic
- Explaining function purpose
- Clarifying tricky parts of the code

4.2 Avoid Comments for

- Obvious code lines
- Redundant explanations
- Over-commenting that clutters code

4.3 Write Clear and Simple Comments

- Use short and meaningful sentences.
- Keep comments updated if code changes.

---

5. Examples

5.1 Single-Line Example

```
# Reading dataset
data = pd.read_csv("data.csv")
```

5.2 Multi-Line Example

```
"""
Training a linear regression model
Steps:
1. Clean data
2. Split into training and test sets
3. Train model
4. Evaluate model
"""
```

5.3 Docstring Example

def normalize(data):

```
def greet(name):
    """
    This function returns a greeting message.
    Parameter:
        name: the name of the person
    Returns:
        A greeting string
    """
    return "Hello, " + name
```

Usage

```
print(greet("Ravi"))
print(greet.__doc__)
```

Output

```
Hello, Ravi
This function returns a greeting message.
    Parameter:
        name: the name of the person
    Returns:
        A greeting string
```

PIP IN PYTHON

1. Introduction to pip

1.1 What is pip?

- pip stands for **Package Installer for Python**.
- It is the default tool used to install and manage Python libraries.
- Comes pre-installed with most Python versions (Python 3.4+).

1.2 Why pip is Important?

- Allows you to install third-party packages needed for:
  - Data Science
  - Machine Learning
  - Deep Learning
  - Web Frameworks
  - Automation
- Essential libraries like NumPy, Pandas, TensorFlow, PyTorch are installed through pip.

---

2. Check pip Version

Command

```
pip --version
```

---

3. Installing a Package Using pip

Basic Installation

```
pip install package_name
```

Example

```
pip install numpy
pip install pandas
pip install scikit-learn
```

---

4. Upgrading a Package

Command

```
pip install --upgrade package_name
```

Example

```
pip install --upgrade numpy
```

5. Uninstalling a Package

Command

```
pip uninstall package_name
```

---

6. Listing Installed Packages

Command

```
pip list
```

---

7. Installing a Specific Version

Command

```
pip install package_name==version_number
```

Example

```
pip install tensorflow==2.13.0
```

---

8. Requirements File

8.1 What is requirements.txt?

- A text file listing all required packages for a project.
- Used for sharing or deploying applications.

8.2 Install All Packages from requirements.txt

```
pip install -r requirements.txt
```

---

9. Downloading Without Installing (Optional)

Command

```
pip download package_name
```

---

10. Pip for AI/ML Setup (Common Libraries)

Commands

```
pip install numpy
pip install pandas
pip install matplotlib
pip install seaborn
pip install scikit-learn
pip install tensorflow
pip install torch torchvision torchaudio
```

These are the core libraries used in AI, machine learning, and deep learning.

REPL IN PYTHON

1. Introduction to REPL

1.1 What is REPL?

- REPL stands for **Read–Eval–Print Loop**.
- It is an **interactive Python environment** where you can type Python commands and get immediate output.
- Useful for testing small code snippets, debugging, and learning Python quickly.

1.2 Why Use REPL?

- Immediate feedback.
- No need to create a Python file.
- Good for beginners and AI/ML developers to quickly test logic and small calculations.

---

2. How REPL Works

The name REPL comes from its workflow:

1. **Read** → Takes user input (Python command).
2. **Eval** → Evaluates the command.
3. **Print** → Displays the result.
4. **Loop** → Repeats the cycle.

---

3. How to Open Python REPL

3.1 On Windows

Open Command Prompt or PowerShell:

```
python
```

or

```
py
```

3.2 On Linux / macOS

```
python3
```

Once it opens, you'll see:

>>>

This is the Python REPL prompt.

4. Using REPL

4.1 Simple Calculations

```
>>> 5 + 3
8
```

4.2 Variable Testing

```
>>> x = 10
>>> x * 2
20
```

4.3 Using Functions

```
>>> def add(a, b):
...     return a + b
...
>>> add(5, 3)
8
```

4.4 Importing Modules

```
>>> import math
>>> math.sqrt(25)
5.0
```

---

5. Exiting REPL

Use any of the following:

```
exit()
quit()
Ctrl + Z (Windows) then Enter
Ctrl + D (Linux/Mac)
```

---

6. Advantages of REPL

- Fast execution without file creation.
- Easy debugging and testing.
- Good for trying AI/ML functions quickly.
- Ideal for learning Python basics.

---

7. Limitations of REPL

- Not suitable for large code.
- No auto-save unless you manually copy/paste.

- Hard to manage complex multi-file projects.

---

8. Mini Example

```
>>> import numpy as np
>>> arr = np.array([1, 2, 3])
>>> arr * 2
array([2, 4, 6])
```

PYTHON VARIABLES

1. Introduction to Variables

1.1 What is a Variable?

- A variable is a **named memory location** used to store data.
- In Python, you do **not** declare the type of a variable; it is decided at runtime.
- Variables help store values that can be used and modified throughout a program.

1.2 Characteristics of Python Variables

- Dynamically typed
- Flexible and easy to use
- Can store any type of data (int, float, string, list, etc.)
- Memory automatically managed by Python

---

2. Variable Declaration

2.1 Syntax

```
variable_name = value
```

2.2 Examples

```
x = 10
name = "Ravi"
pi = 3.14
is_valid = True
```

---

3. Python is Dynamically Typed

3.1 Meaning

- You can assign any type of value to a variable.
- Type can change during execution.

Example

```
x = 10        # integer
x = "Hello"   # now string
```

---

4. Variable Naming Rules

4.1 Allowed

- Must start with a letter (a-z or A-Z) or underscore (_)
- Can contain letters, digits, or underscores

4.2 Not Allowed

- Cannot start with a number
- No spaces allowed
- Special characters like @, #, $, %, etc. not allowed

4.3 Case-Sensitive

- age, Age, and AGE are different variables.

---

5. Valid and Invalid Examples

5.1 Valid

```
x
total_amount
num1
_data
name35
```

5.2 Invalid

```
1value        (cannot start with number)
total amount  (spaces not allowed)
my-name       (hyphen not allowed)
```

---

6. Multiple Variable Assignment

6.1 Same Value to Multiple Variables

```
a = b = c = 10
```

6.2 Different Values to Multiple Variables

```
x, y, z = 1, 2, 3
```

---

7. Type Checking

Use `type`() to check variable type.

Example

```
x = 10
print(type(x))      # <class 'int'>
```

---

8. Variable Reassignment

```
data = 100
data = 3.14
data = "Python"
```

- Values and types can change as needed.

---

9. Global and Local Variables

9.1 Local Variable

- Declared inside a function
- Accessible only within that function

Example

```
def test():
    x = 5   # local variable
```

9.2 Global Variable

- Declared outside functions
- Can be accessed anywhere

Example

```
x = 10   # global variable

def show():
    print(x)
```

9.3 Using global Keyword

```
count = 0

def update():
    global count
    count = count + 1
```

DATA TYPES

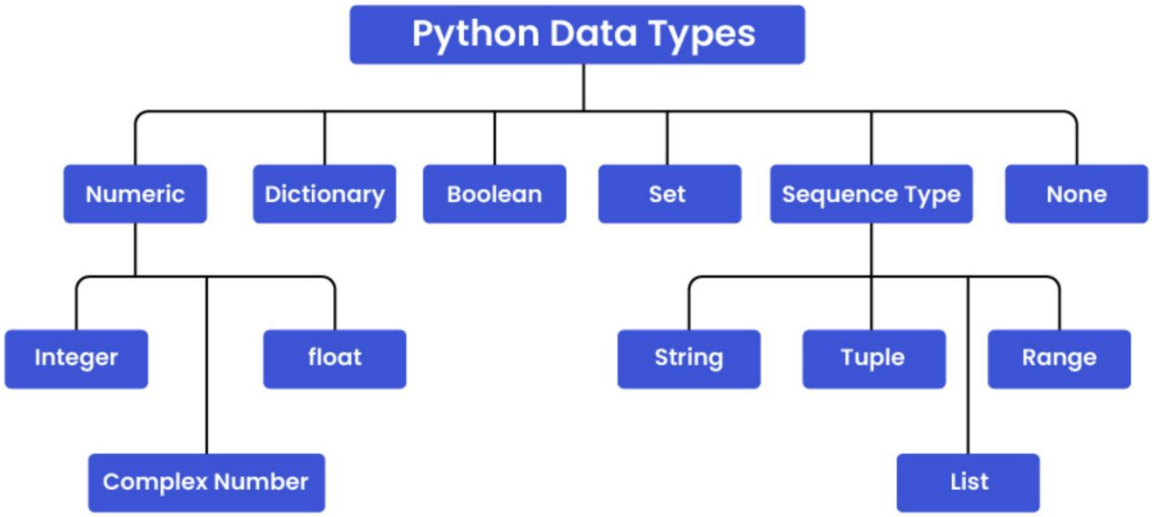
1. Introduction to Data Types

1.1 Definition

- A data type represents the kind of value a variable can store.
- Python automatically assigns data types at runtime (dynamic typing).
- Data types help Python interpret how to process data.

1.2 Importance in AI/ML

- Required for mathematical operations.
- Helps in handling datasets, arrays, tensors, text, and labels.
- Libraries like NumPy, Pandas, and TensorFlow rely on specific data types for performance.



---

2. Numeric Data Types

2.1 Integer (int)

- Stores whole numbers without decimals.
- Can handle very large numbers.
- Example:

```
x = 10
```

2.2 Floating Point (float)

- Stores decimal or fractional numbers.
- Used in mathematical and scientific calculations.
- Example:

```
pi = 3.14
```

2.3 Complex Number (complex)

- Stores numbers in the form a + bj.
- Mainly used in scientific and engineering tasks.
- Example:

```
z = 2 + 5j
```

---

3. String Data Type

3.1 Definition

- A sequence of characters enclosed in quotes (single, double, or triple).
- Strings support indexing, slicing, and various built-in methods.

3.2 Example

```
name = "Python"
```

3.3 Use in AI/ML

- Important for text processing, NLP tasks, and dataset labels.

---

4. Boolean Data Type

4.1 Definition

- Stores only two values: True or False.
- Result of comparison operations.

4.2 Example

```
is_valid = True
```

4.3 Use in Programs

- Used in decision making (if-else).
- Essential for controlling loops and conditions.

---

5. Sequence Data Types

5.1 List

- Ordered and mutable collection.
- Can store mixed data types.
- Example:

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

5.2 Tuple

- Ordered and immutable collection.
- Faster than lists.
- Example:

```
point = (4, 5)
```

5.3 Range

- Represents a sequence of numbers.
- Often used in loops.
- Example:

```
r = range(5)
```

---

6. Mapping Data Type

6.1 Dictionary (dict)

- Stores data in key–value pairs.
- Keys must be unique.
- Example:

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

---

7. Set Data Types

7.1 Set

- Unordered collection of unique items.
- Automatically removes duplicates.
- Example:

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

7.2 Frozen Set

- Immutable version of a set.
- Example:

```
fs = frozenset([1, 2, 3])
```

---

8. None Type

8.1 Definition

- Represents no value or empty value.
- Used as a placeholder.

8.2 Example

```
result = None
```

---

9. Checking Data Types

9.1 Using type() Function

- Returns the type of a variable.
- Example:

```
x = 10
print(type(x))     # <class 'int'>
```

---

10. Type Conversion (Type Casting)

10.1 Definition

- Converts one data type into another.

Examples

```
int("10")       # string to int
float(5)        # int to float
str(45)         # number to string
list("abc")     # string to list
```

10.2 Importance in AI/ML

- Required for preprocessing datasets.
- Ensures correct input format for models.

---

11. Data Types Used in AI/ML Libraries

11.1 NumPy

- Uses ndarray with fixed numeric types (int32, float32, float64).

11.2 Pandas

- Uses data types like:
  - **int64**
  - **float64**
  - **object (string)**
  - **category**
  - **datetime64**

11.3 TensorFlow / PyTorch

- Tensors use types like:
  - **float32**
  - **float64**
  - **int32**
  - **int64**

TYPE CASTING

1. Introduction to Type Casting

1.1 Definition

- Type casting means converting a value from one data type to another.
- Python supports **explicit type casting** using built-in functions.
- Useful for ensuring correct data types during calculations and data processing.

1.2 Importance in AI/ML

- Required when cleaning and preprocessing data.
- Ensures compatibility with NumPy arrays, Pandas DataFrames, and ML models.
- Prevents type errors during mathematical operations.

---

2. Types of Type Casting

2.1 Explicit Type Casting (Manual Conversion)

- Performed using built-in functions.
- Programmer decides how to convert the value.

2.2 Implicit Type Casting (Automatic Conversion)

- Python automatically converts one type to another during operations.
- Usually happens in numeric operations.

---

3. Explicit Type Casting

3.1 Convert to Integer (int)

- Removes decimal part.
- Converts numeric strings to integers.

Example

```
int(10.9)       # output: 10
int("25")       # output: 25
```

3.2 Convert to Float (float)

- Converts integers or numeric strings to floating values.

Example

```
float(5)        # output: 5.0
float("3.14")   # output: 3.14
```

3.3 Convert to String (str)

- Converts numbers or other types to strings.

Example

```
str(50)        # output: "50"
str(3.14)      # output: "3.14"
```

3.4 Convert to List (list)

- Converts iterable types (string, tuple) into lists.

Example

```
list("abc")       # output: ['a','b','c']
list((1, 2, 3))   # output: [1, 2, 3]
```

3.5 Convert to Tuple (tuple)

- Converts iterable types into tuples.

Example

```
tuple([1, 2, 3])     # output: (1, 2, 3)
```

3.6 Convert to Set (set)

- Removes duplicates automatically.

Example

```
set([1, 2, 2, 3])    # output: {1, 2, 3}
```

---

4. Implicit Type Casting

4.1 Python Automatically Converts Types

- Happens when different numeric types interact.
- Done to prevent data loss.

4.2 Common Cases

Example: int → float

```
x = 10      # int
y = 3.5     # float
z = x + y   # result becomes float
```

Example: small → bigger type

```
result = 5 + 5.0     # result is float
```

---

5. Errors in Type Casting

5.1 Invalid String Conversion

- A non-numeric string cannot be converted to int or float.

Example

```
int("abc")   # Error
```

5.2 Converting Incompatible Types

```
int([1, 2, 3])    # Error
```

---

6. Checking Data Type After Casting

Example

```
x = float(10)
print(type(x))      # <class 'float'>
```

INPUT() FUNCTION

1. Introduction to the input() Function

1.1 Definition

- input() is a built-in Python function used to **take input from the user**.
- Always returns the input as a **string**.
- Commonly used in interactive programs, data collection, and user-driven logic.

1.2 Importance in AI/ML Learning

- Helps create simple data-entry programs.
- Useful for taking hyperparameters, filenames, or user choices in scripts.

---

2. Basic Syntax

2.1 Syntax Format

```
input(prompt)
```

2.2 Explanation

- prompt → message shown to user.
- Returns user input as a string.

---

3. Basic Usage

3.1 Without Prompt

```
name = input()
```

3.2 With Prompt Message

```
name = input("Enter your name: ")
```

---

4. Input is Always String

4.1 Explanation

- Whatever the user types is stored as a string.
- Even numbers are taken as strings.

4.2 Example

```
x = input("Enter number: ")
print(type(x))     # <class 'str'>
```

---

5. Converting Input to Other Data Types

5.1 Convert to Integer

```
age = int(input("Enter age: "))
```

5.2 Convert to Float

```
price = float(input("Enter price: "))
```

5.3 Convert to Boolean (Manual Logic)

```
flag = input("Enter yes/no: ") == "yes"
```

5.4 Convert to List

```
values = input("Enter numbers: ").split()
```

---

6. Input with Split() for Multiple Values

6.1 Taking Multiple Values

```
a, b = input("Enter two numbers: ").split()
```

6.2 Convert Values to Integers

```
a, b = map(int, input("Enter two numbers: ").split())
```

---

7. Practical Examples

7.1 Simple Greeting Program

```
name = input("Enter your name: ")
print("Hello", name)
```

7.2 Sum of Two Numbers

```
x = int(input("Enter first number: "))
y = int(input("Enter second number: "))
print("Sum =", x + y)
```

STRINGS IN PYTHON

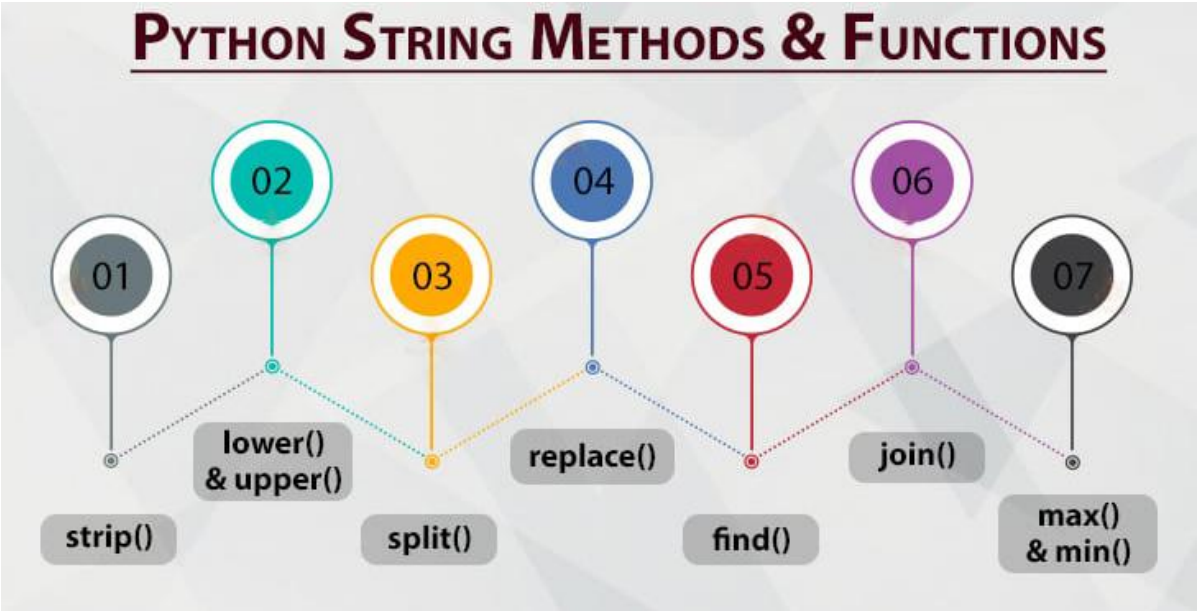
1. Introduction to Strings

1.1 Definition

- A string is a **sequence of characters** enclosed in quotes.
- Can use **single**, **double**, or **triple** quotes.
- Strings are **immutable** (cannot be changed after creation).

1.2 Importance in AI/ML

- Used in dataset labels, file paths, text processing, NLP, and logs.



---

2. Creating Strings

2.1 Using Single or Double Quotes

```
s1 = 'Hello'
s2 = "Python"
```

2.2 Using Triple Quotes

- Used for multi-line strings.

```
s = """This is
a multi-line string."""
```

3. String Indexing and Slicing

3.1 Indexing

- Access characters using index values.
- Index starts at 0.

Example

```
s = "Python"
s[0]   # 'P'
s[3]   # 'h'
```

3.2 Negative Indexing

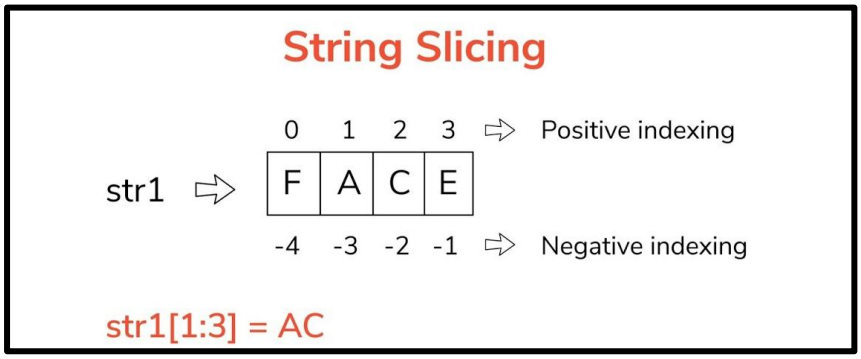
```
s[-1]    # 'n'
s[-3]    # 'h'
```

3.3 Slicing

- Extract part of a string.

Example

```
s[1:4]    # 'yth'
s[:3]     # 'Pyt'
s[2:]     # 'thon'
```



---

4. String Properties

4.1 Immutable

- Cannot modify characters directly.

Example

```
s = "Hello"
s[0] = "Y"      # Error
```

4.2 Concatenation

```
a = "Hello"
b = "World"
c = a + " " + b
```

4.3 Repetition

```
s = "Hi" * 3    # "HiHiHi"
```

---

5. String Functions (Built-in Methods)

5.1 Length

```
len("Python")     # 6
```

5.2 Case Conversion

s.upper()

s.lower()

s.title()

s.capitalize()

5.3 Searching

```
s.find("on")     # returns index
s.count("a")
```

5.4 Replace

```
s.replace("old","new")
```

5.5 Split and Join

```
s.split()              # string to list
" ".join(["a","b"])    # list to string
```

5.6 Strip

```
s.strip()       # removes spaces
```

---

6. String Formatting

6.1 Using f-strings

```
name = "Ravi"
age = 20
f"My name is {name} and I am {age}"
```

6.2 Using format()

```
"Name: {}, Age: {}".format(name, age)
```

6.3 Using % Operator

```
"Value = %d" % 10
```

---

7. Escape Characters

7.1 Common Escape Sequences

- \n → new line
- \t → tab
- \" → double quote
- \\ → backslash

Example

```
s = "Hello\nWorld"
```

---

8. String Comparison

8.1 Using Relational Operators

```
"apple" == "apple"
"abc" < "bcd"
```

---

9. Looping Through String

9.1 Using for Loop

```
for ch in "Python":
    print(ch)
```

LIST & TUPLES

1. Introduction

1.1 Definition

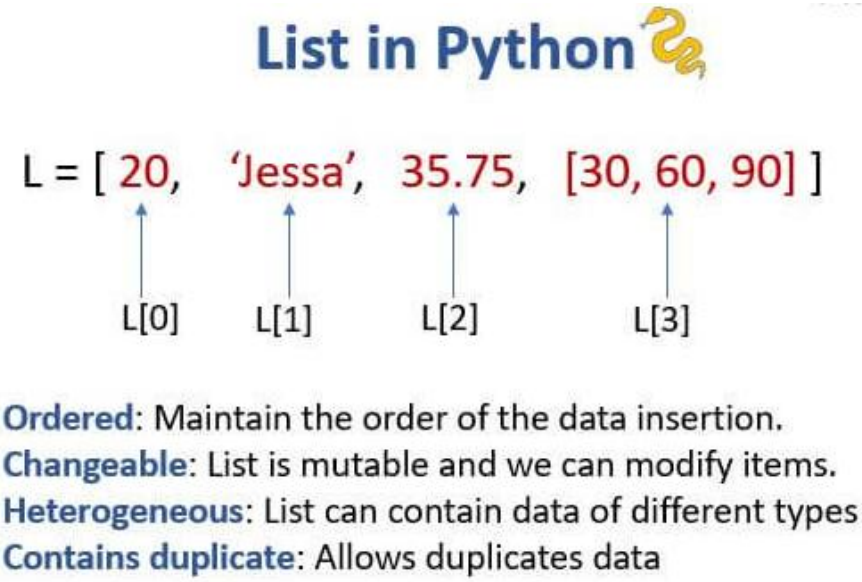
- **List** and **Tuple** are Python sequence data types used to store multiple items.
- Both can store mixed data types such as integers, floats, strings, etc.

1.2 Key Difference

- **List** is mutable (can be changed).
- **Tuple** is immutable (cannot be changed once created).

---

2. List



2.1 Definition

- A list is an **ordered**, **mutable** collection of items.
- Elements are enclosed in **square brackets [ ]**.

2.2 Creating a List

```
numbers = [1, 2, 3, 4]
mixed = [10, "Python", 3.14, True]
```

2.3 Accessing List Elements

- Using indexing:

```
numbers[0]      # 1
numbers[-1]     # 4
```

2.4 Slicing

```
numbers[1:3]    # [2, 3]
numbers[:2]     # [1, 2]
```

2.5 Modifying a List

- Lists are editable.

Example

```
numbers[0] = 100
numbers.append(5)
numbers.remove(3)
```

2.6 List Functions and Methods

- append() – add item
- insert() – insert at position
- remove() – removes item
- pop() – removes last or given index
- sort() – sort list
- reverse() – reverse list
- len(list) – length of list

2.7 Looping Through List

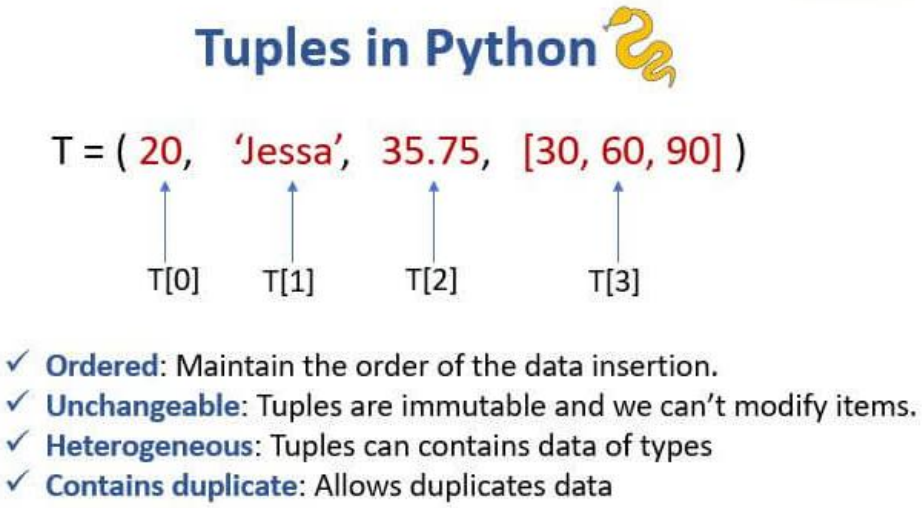
```
for item in numbers:
    print(item)
```

2.8 Nested Lists

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

---

4. Tuple



3.1 Definition

- A tuple is an **ordered**, **immutable** collection of items.
- Elements enclosed in **parentheses ( )**.

3.2 Creating a Tuple

```
t1 = (1, 2, 3)
t2 = ("Python", 3.14, True)
```

3.3 Accessing Tuple Elements

```
t1[0]     # 1
t1[-1]    # 3
```

3.4 Slicing

```
t1[1:3]   # (2, 3)
```

3.5 Tuple Immutability

- Cannot change or delete elements.
- Any attempt raises an error:

```
t1[0] = 10     # Error
```

3.6 Tuple Methods

- count() – count occurrences
- index() – find index of item

3.7 Tuple Advantages

- Faster than lists
- More memory efficient
- Data integrity (fixed values)

3.8 Tuple Packing and Unpacking

```
t = 1, 2, 3          # packing
a, b, c = t          # unpacking
```

---

4. Difference Between List and Tuple

Feature	List	Tuple
Mutability	Mutable	Immutable
Syntax	[ ]	( )
Speed	Slower	Faster
Memory Usage	More	Less
Modification	Allowed	Not allowed
Use Case	Dynamic data	Fixed data

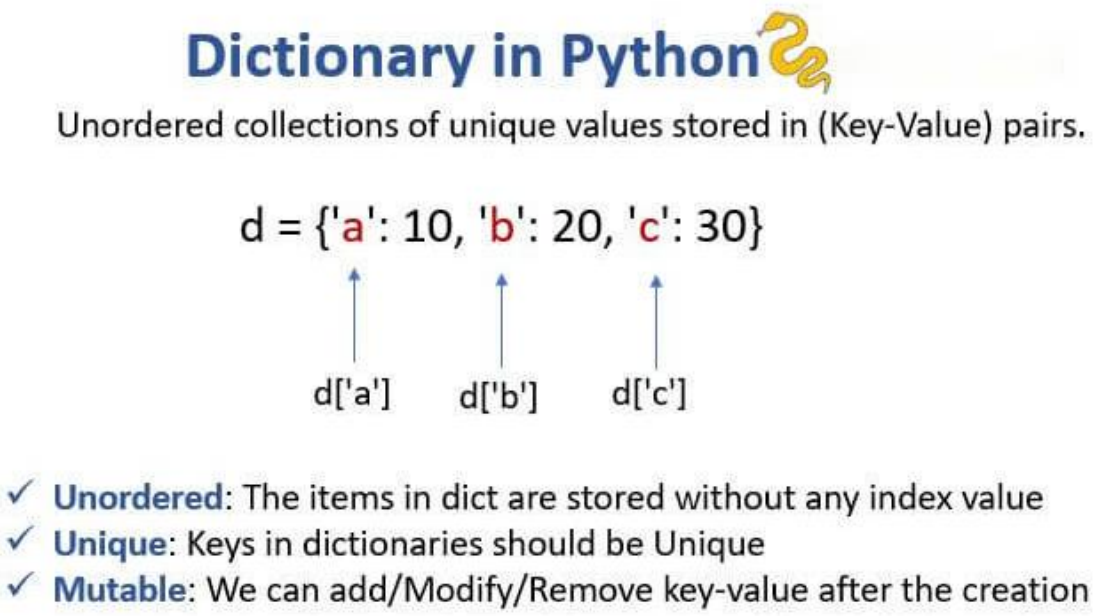
---

5. Example Program

```
# List example
fruits = ["apple", "banana", "mango"]
fruits.append("orange")

# Tuple example
coordinates = (10, 20)
print(coordinates[0])
```

DICTIONARY & SET

1. Dictionary



1.1 Definition

- A dictionary is a **mapping data type** that stores data in **key–value pairs**.
- Keys must be **unique**.
- Enclosed in **curly braces { }**.
- Mutable (can be changed).

---

2. Creating a Dictionary

2.1 Basic Syntax

```
student = {"name": "Ravi", "age": 20, "marks": 85}
```

2.2 Empty Dictionary

```
data = {}
```

---

3. Accessing Dictionary Values

3.1 Using Keys

```
student["name"]
```

3.2 Using get()

```
student.get("marks")
```

---

4. Adding and Updating Values

4.1 Add New Key–Value Pair

```
student["city"] = "Delhi"
```

4.2 Update Existing Value

```
student["age"] = 21
```

---

5. Removing Items

5.1 pop()

```
student.pop("age")
```

5.2 popitem()

- Removes last inserted item.

```
student.popitem()
```

5.3 del

```
del student["name"]
```

---

6. Dictionary Methods

- keys() – returns list of keys
- values() – returns list of values
- items() – returns key–value pairs
- update() – merges two dictionaries
- clear() – removes all items

```
student.keys()
student.items()
```

---

7. Looping in Dictionary

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

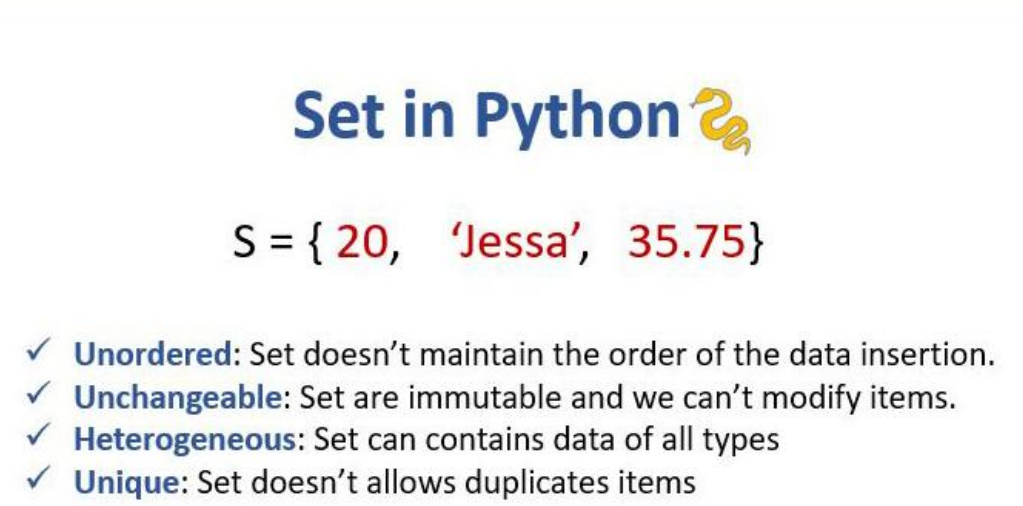
---

8. Nested Dictionary

```
records = {
    "student1": {"name": "Ravi", "age": 20},
    "student2": {"name": "Neha", "age": 22}
}
```

---

9. Sets



9.1 Definition

- A set is an **unordered**, **mutable**, **unique-element** collection.
- Automatically removes duplicate values.
- Enclosed in **curly braces { }**.

---

10. Creating a Set

```
s = {1, 2, 3, 3}
# Output: {1, 2, 3}
```

10.1 Empty Set

```
s = set()
```

---

11. Adding and Removing Items

11.1 add()

```
s.add(4)
```

11.2 remove()

```
s.remove(2)
```

11.3 discard()

```
s.discard(10)   # no error if item missing
```

11.4 pop()

- Removes random item.

12. Set Operations

12.1 Union

```
s1.union(s2)
```

12.2 Intersection

```
s1.intersection(s2)
```

12.3 Difference

```
s1.difference(s2)
```

12.4 Symmetric Difference

```
s1.symmetric_difference(s2)
```

CONDITIONAL EXPRESSIONS

1. Introduction to Conditional Expressions

1.1 Definition

- A **conditional expression** (also called **ternary operator**) allows writing **if–else** logic in a **single line**.
- Provides a short way to assign values based on a condition.

1.2 Purpose

- Makes code concise and readable.
- Useful for quick decisions inside assignments or function arguments.

---

2. Syntax of Conditional Expression

2.1 General Syntax

```
value_if_true if condition else value_if_false
```

2.2 Explanation

- If the condition is true → expression returns value_if_true.
- If the condition is false → expression returns value_if_false.

---

3. Basic Examples

3.1 Simple Example

```
x = 10
result = "Even" if x % 2 == 0 else "Odd"
```

3.2 Numeric Decision

```
a = 5
b = 8
max_num = a if a > b else b
```

---

4. Using Conditional Expression Inside Print

```
age = 18
print("Adult" if age >= 18 else "Minor")
```

---

5. Conditional Expression with Function Calls

```
temp = 25
response = "High" if temp > 30 else "Normal"
```

---

6. Nested Conditional Expressions

6.1 Syntax

```
value1 if condition1 else value2 if condition2 else value3
```

6.2 Example

```
marks = 85
grade = "A" if marks >= 90 else "B" if marks >= 80 else "C"
```

---

7. Conditional Expression vs if–else Statement

7.1 When to Use Conditional Expression

- For short and simple decisions.
- When assigning values based on a condition.
- When writing compact logic.

7.2 When Not to Use

- Complex logic.
- Multiple statements.
- When readability suffers.

---

8. Another Example Program

```
num = int(input("Enter number: "))
result = "Positive" if num > 0 else "Negative" if num < 0 else
"Zero"
print(result)
```

IF-ELSE STATEMENTS

1. Introduction to Conditional Statements

1.1 Definition

- Conditional statements allow decision-making in Python.
- Based on a condition, different blocks of code are executed.

1.2 Importance

- Helps control program flow.
- Essential for logic building, AI decision rules, and data filtering.

---

2. if Statement

2.1 Definition

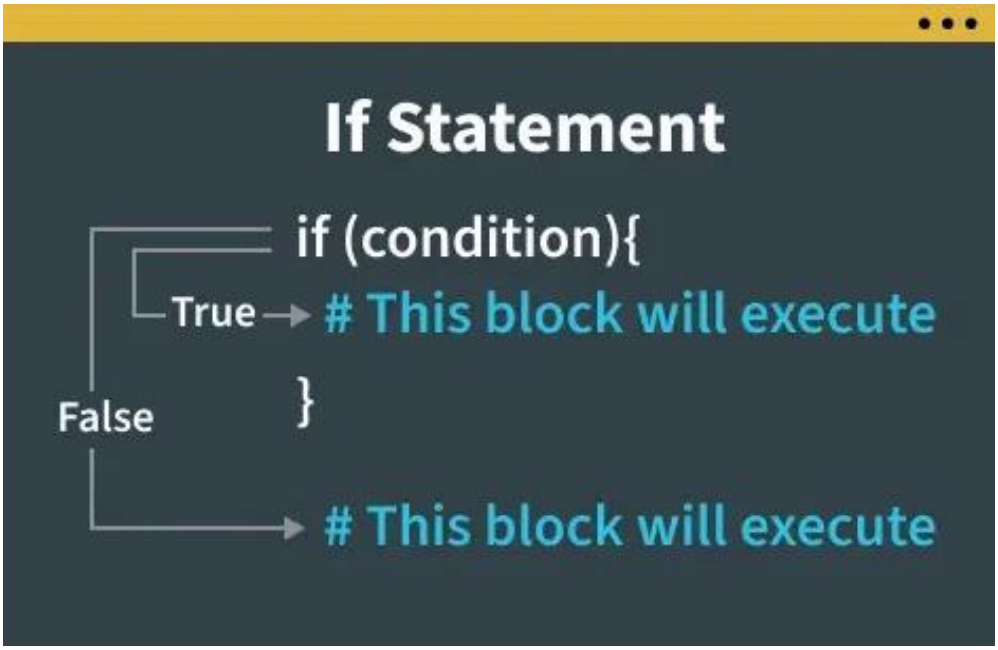
- The if statement executes a block of code only if the condition is true.

2.2 Syntax

```
if condition:
    statement
```

2.3 Example

```
age = 20
if age >= 18:
    print("Adult")
```



---

3. if–else Statement

3.1 Definition

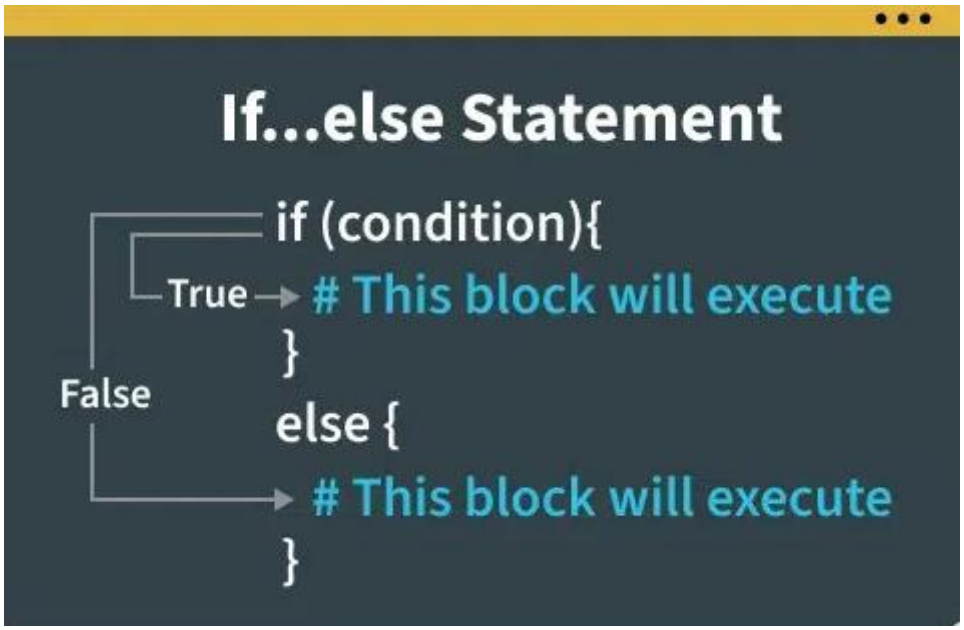
- The else block executes when the if condition is false.

3.2 Syntax

```
if condition:
    statement1
else:
    statement2
```

3.3 Example

```
num = 5
if num > 0:
    print("Positive")
else:
    print("Negative")
```



---

4. elif Statement

4.1 Definition

- elif stands for “else if”.
- Used when multiple conditions need to be checked.
- Executes the first true condition.

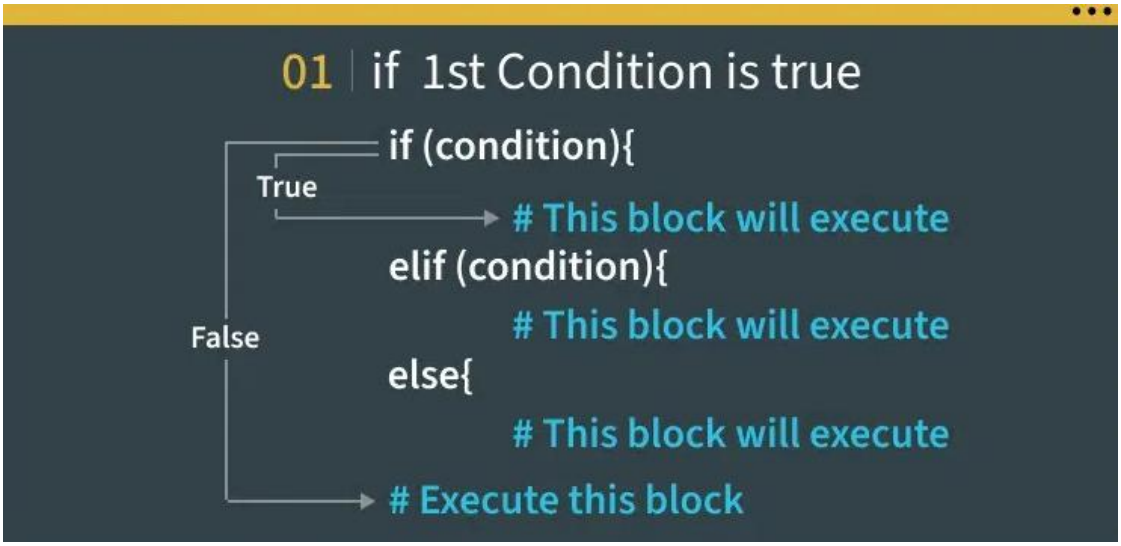
4.2 Syntax

```
if condition1:
    statement1
elif condition2:
    statement2
else:
    statement3
```

---

5. Example with if–elif–else

```
marks = 85

if marks >= 90:
    print("Grade A")
elif marks >= 80:
    print("Grade B")
elif marks >= 70:
    print("Grade C")
else:
    print("Grade D")
```



---

6. Multiple elif Conditions

6.1 Example

```
num = 0

if num > 0:
    print("Positive")
elif num < 0:
    print("Negative")
else:
    print("Zero")
```

---

7. Using Conditions with Logical Operators

7.1 AND Operator

```
age = 25
if age >= 18 and age <= 60:
    print("Eligible")
```

7.2 OR Operator

```
x = 10
if x == 10 or x == 20:
    print("Match found")
```

7.3 NOT Operator

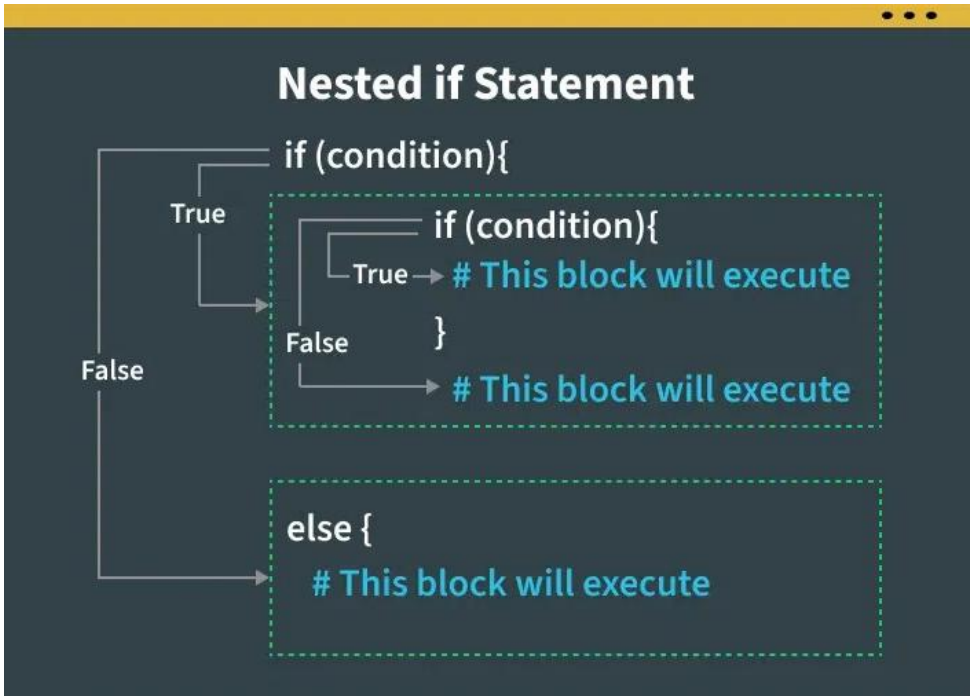
```
flag = False
if not flag:
    print("Condition True")
```

---

8. Nested if Statements

8.1 Example

```
num = 10
if num > 0:
    if num % 2 == 0:
        print("Positive Even")
```



---

10. Common Mistakes

10.1 Missing Indentation

```
if x > 10:
print("Wrong")    # Error
```

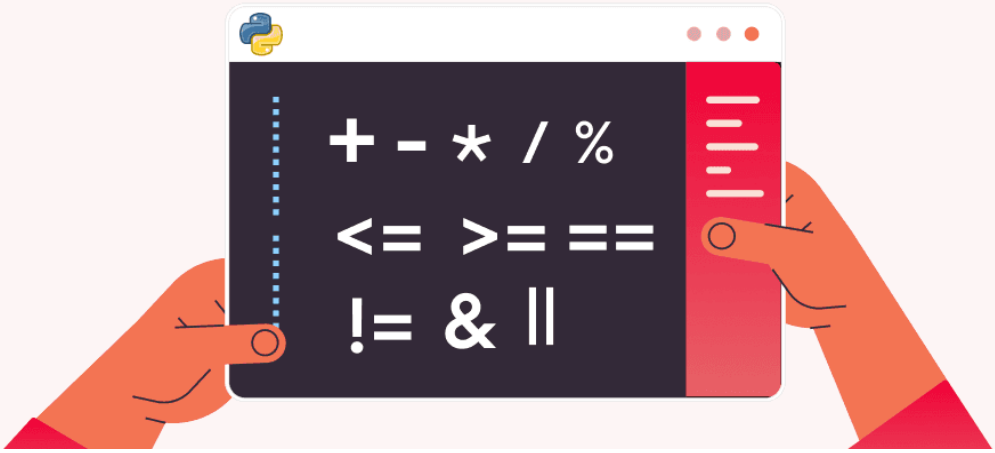
10.2 Using elif Without if

- Not allowed.

10.3 Wrong Condition Checking

- Equal operator is ==, not =.

OPERATORS IN PYTHON



1. Introduction to Operators

1.1 Definition

- Operators are special symbols used to perform operations on variables and values.
- Python supports various types of operators for arithmetic, comparison, logical operations, etc.

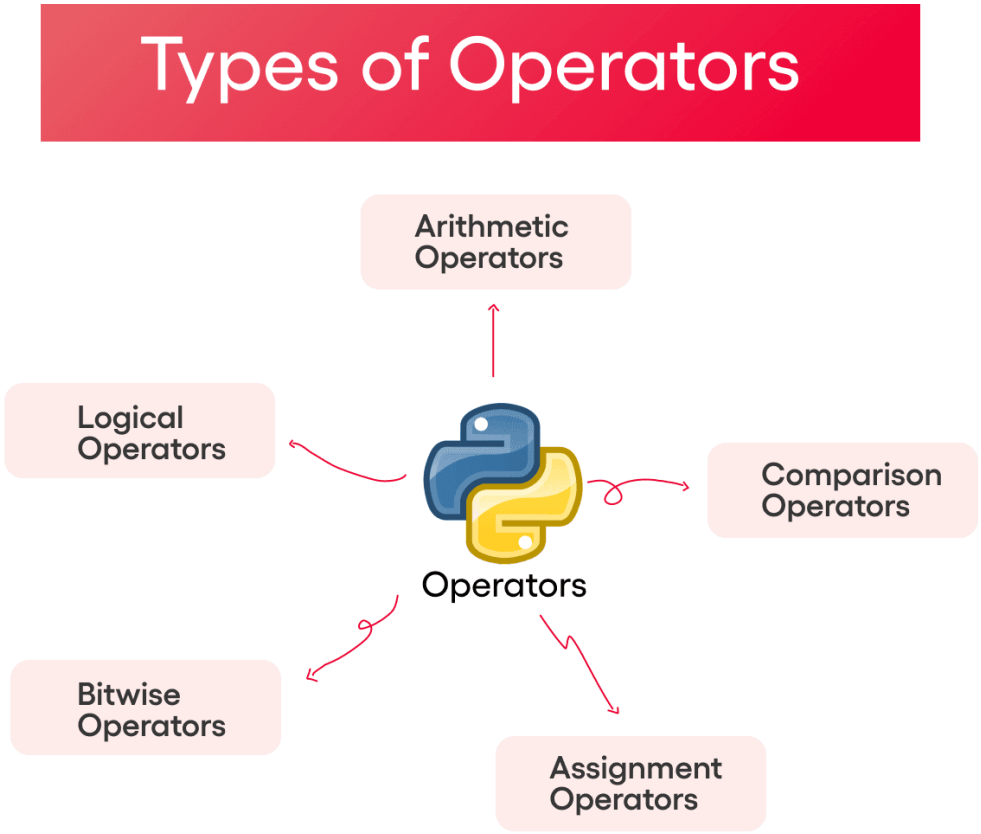
1.2 Importance

- Essential for calculations, decision-making, comparisons, and data manipulation.
- Used heavily in AI/ML for mathematical operations on data.

---

2. Types of Operators in Python

- Arithmetic Operators
- Comparison (Relational) Operators
- Logical Operators
- Assignment Operators
- Bitwise Operators
- Membership Operators
- Identity Operators



3. Arithmetic Operators

3.1 List of Arithmetic Operators

- + → Addition
- - → Subtraction
- * → Multiplication
- / → Division (float result)
- // → Floor division
- % → Modulus (remainder)
- ** → Exponent (power)

3.2 Examples

```
x + y
x - y
x * y
x / y
x // y
x % y
x ** y
```

---

4. Comparison (Relational) Operators

4.1 Purpose

- Used to compare two values.
- Result is always **True or False**.

4.2 List of Comparison Operators

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

4.3 Example

```
x == y
x > y
x != y
```

---

5. Logical Operators

5.1 Purpose

- Used to combine multiple conditions.

5.2 Types

- and → True if both conditions true
- or → True if any condition true
- not → Inverts the condition

5.3 Example

```
x > 10 and x < 20
x == 5 or y == 10
not (x > 0)
```

6. Assignment Operators

6.1 Purpose

- Used to assign values to variables.

6.2 List of Assignment Operators

- =
- +=
- -=
- *=
- /=
- //=
- %=
- **=

6.3 Example

```
x = 10
x += 5    # x = x + 5
x *= 2
```

---

7. Bitwise Operators

7.1 Purpose

- Operate on binary (bit-level) data.

7.2 List of Bitwise Operators

- & → AND
- | → OR
- ^ → XOR
- ~ → NOT
- << → Left shift
- >> → Right shift

7.3 Example

```
x & y
x | y
x ^ y
x << 2
```

---

8. Membership Operators

8.1 Purpose

- Check membership in sequences (list, string, tuple).

8.2 Operators

- in
- not in

8.3 Example

```
3 in [1, 2, 3]
"a" not in "Python"
```

---

9. Identity Operators

9.1 Purpose

- Compare memory locations of two objects.

9.2 Operators

- is
- is not

9.3 Example

```
x is y
x is not y
```

---

10. Operators in AI/ML Applications

10.1 Arithmetic Operators

- Used for mathematical operations in NumPy arrays and tensors.

10.2 Comparison Operators

- Used for filtering datasets (e.g., Pandas conditions).

10.3 Logical Operators

- Used in decision rules and data preprocessing.

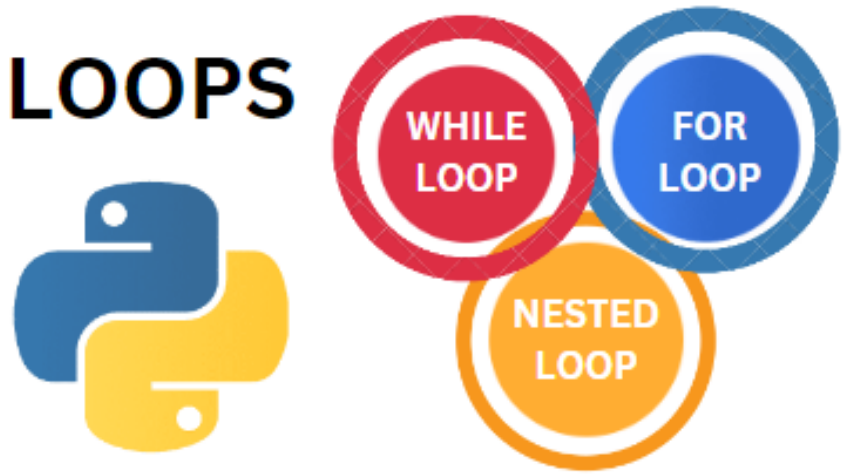
10.4 Assignment Operators

- Used in training loops: updating weights, counters.

---

11. Example Program

```
x = 15
y = 10

if x > y and x % 5 == 0:
    print("x is greater and divisible by 5")
else:
    print("Condition failed")
```



1. Introduction to Loops

1.1 Definition

- A loop is used to execute a block of code repeatedly.
- Reduces code repetition.
- Essential for data processing, iteration, and algorithm implementation.

1.2 Importance in AI/ML

- Used for iterating over datasets, training batches, and model parameters.
- Backbone of training loops in deep learning.

---

2. Types of Loops in Python

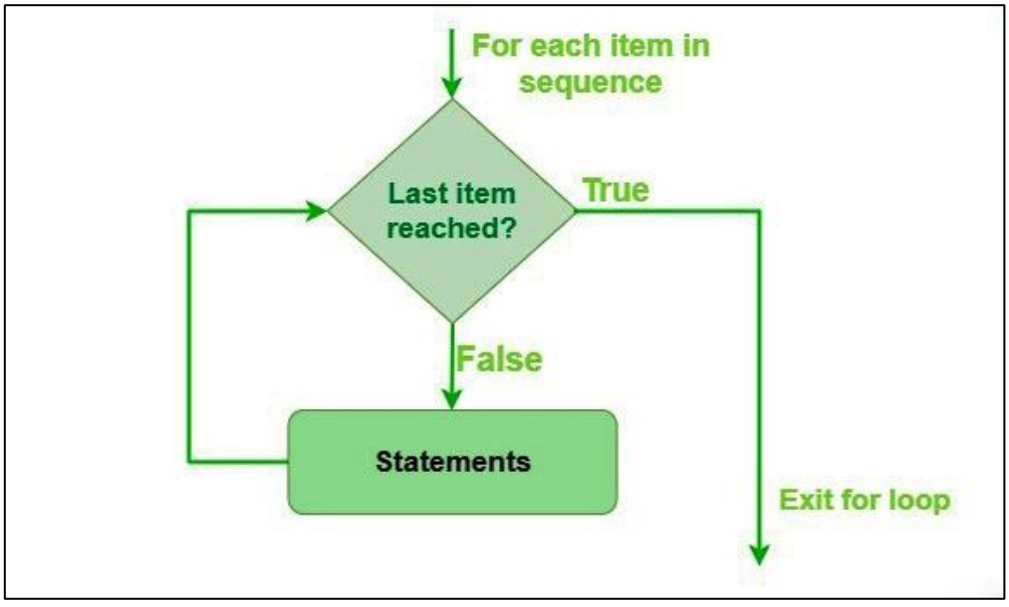
1. for loop
2. while loop
3. Loop control statements (break, continue, pass)

---

3. for Loop

3.1 Definition

- Used to iterate over sequences like lists, strings, tuples, ranges, etc.



3.2 Syntax

```
for variable in sequence:
    statement
```

3.3 Example

```
for i in [1, 2, 3]:
    print(i)
```

3.4 Using range() in for Loop

```
for i in range(5):
    print(i)   # prints 0 to 4
```

3.5 Iterating Over Strings

```
for ch in "Python":
    print(ch)
```

3.6 Nested for Loop

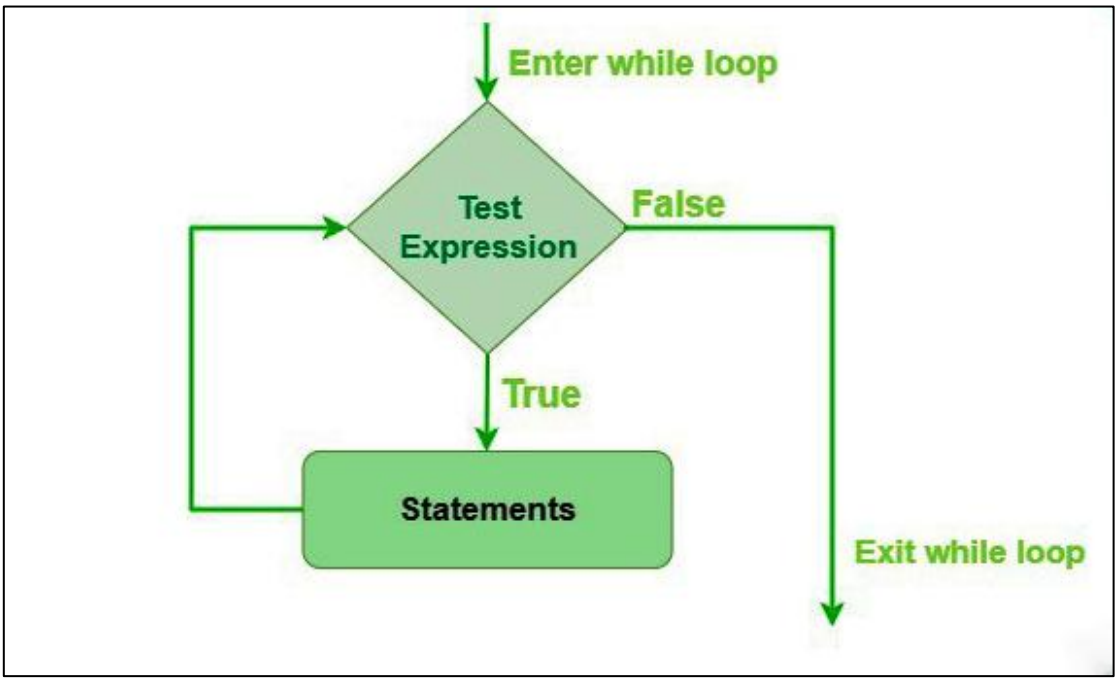
```
for i in range(3):
    for j in range(2):
        print(i, j)
```

---

4. while Loop

4.1 Definition

- Executes as long as a condition remains true.



4.2 Syntax

```
while condition:
    statement
```

4.3 Example

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

4.4 Infinite Loop (Avoid)

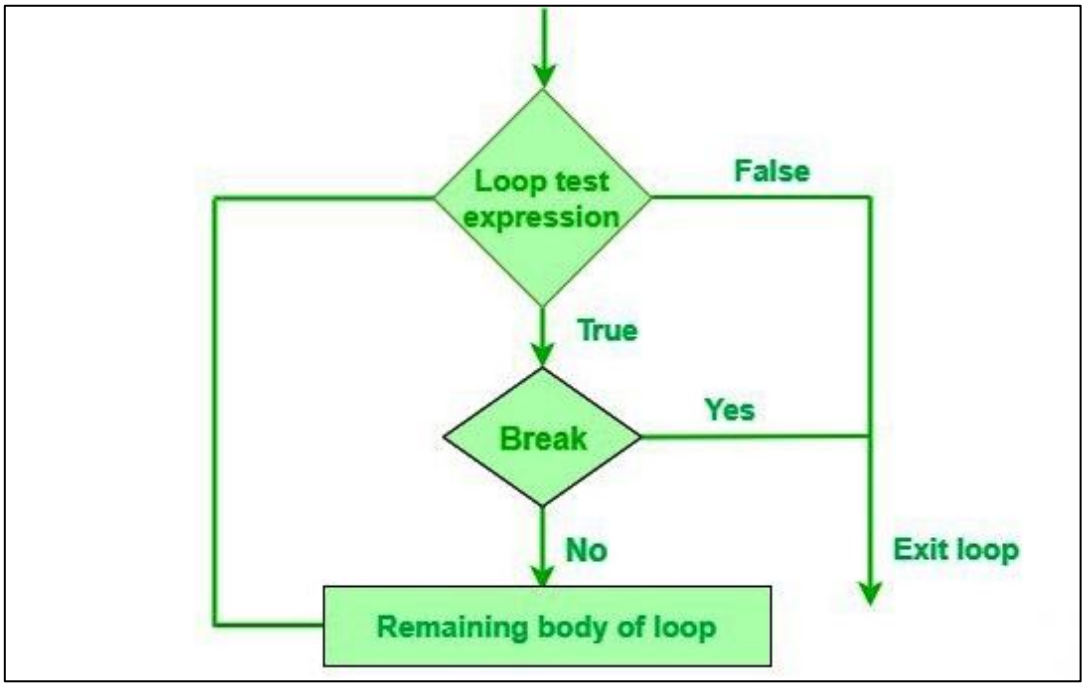
```
while True:
    print("Runs forever")
```

---

5. Loop Control Statements

5.1 break Statement

- Stops the loop immediately.

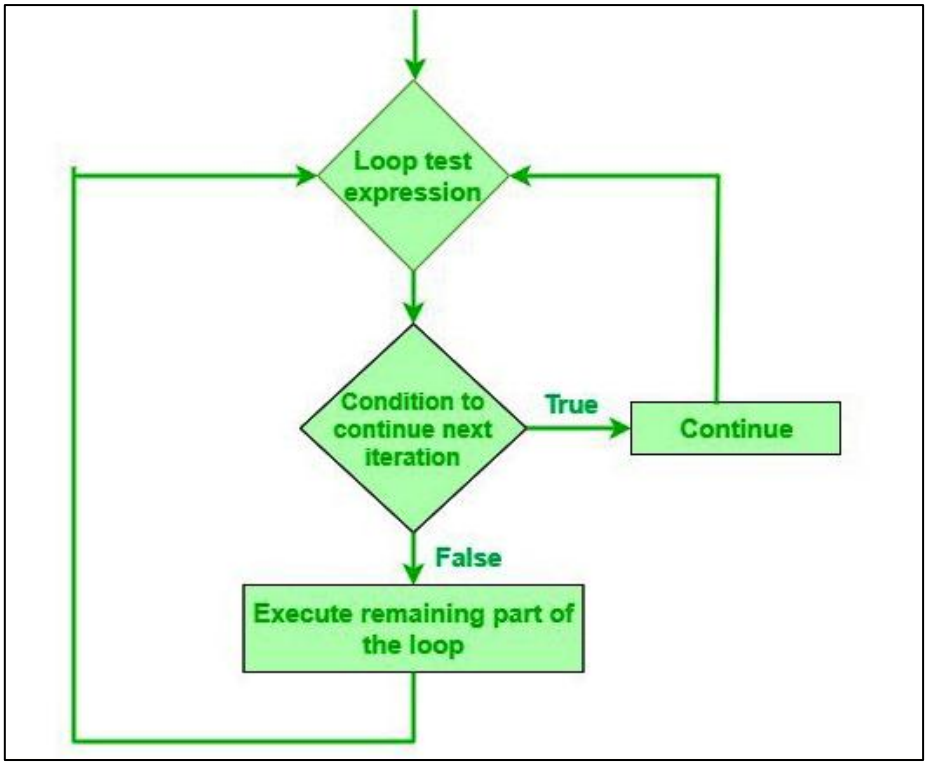


Example

```
for i in range(10):
    if i == 5:
        break
```

5.2 continue Statement

- Skips the current iteration.



Example

```
for i in range(5):
    if i == 2:
        continue
    print(i)
```

5.3 pass Statement

- Placeholder for empty code block.

Example

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

---

6. else with Loops

6.1 Definition

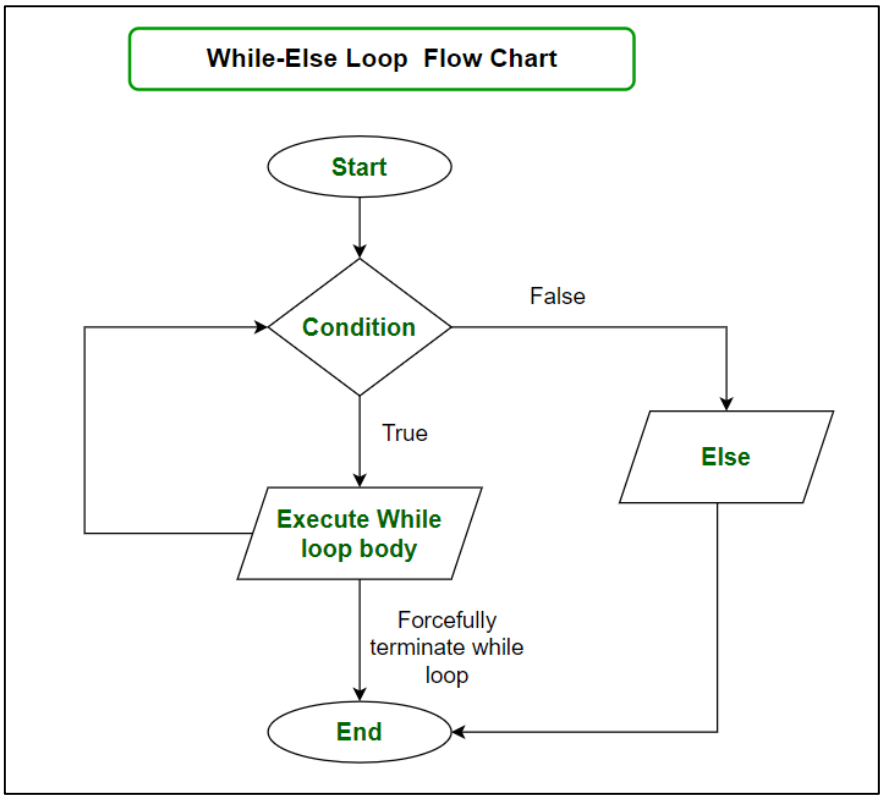
- The else block executes when the loop completes normally (no break).

6.2 Example with for Loop

```
for i in range(3):
    print(i)
else:
    print("Loop finished")
```

6.3 Example with while Loop

```
i = 1
while i <= 3:
    print(i)
    i += 1
else:
    print("Completed")
```

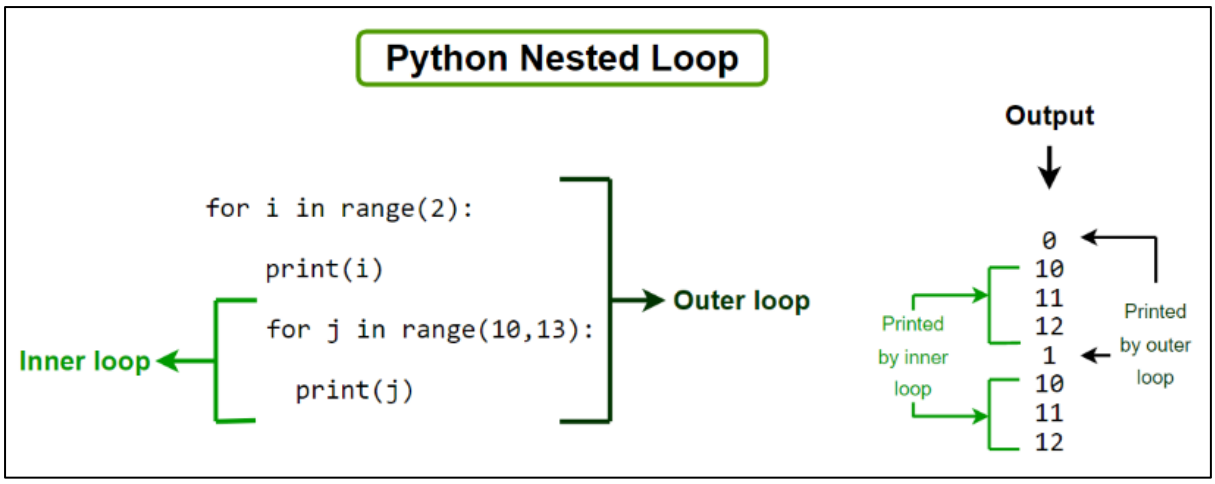


---

7. Nested Loops

7.1 Definition

- A loop inside another loop.



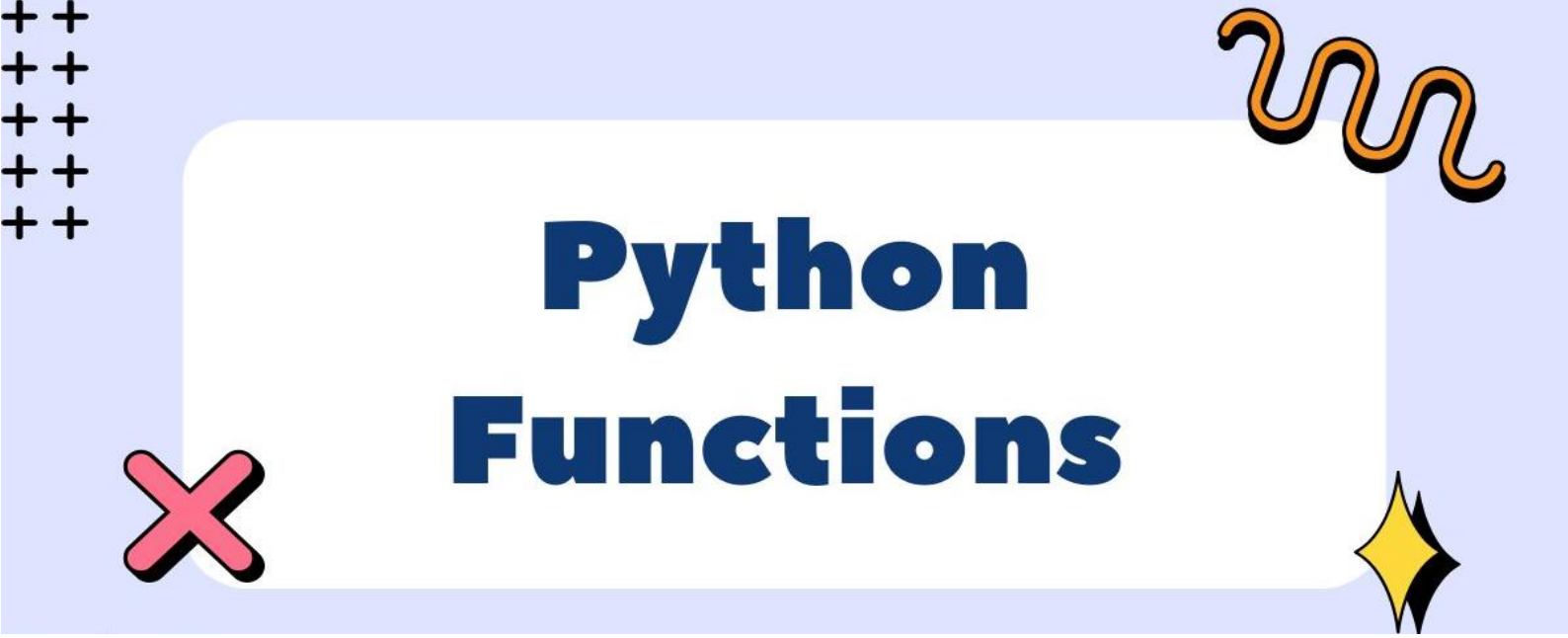
7.2 Example

```
for i in range(2):
    for j in range(3):
        print(i, j)
```

---

8. Example Program

```
for i in range(1, 6):
    print("Square of", i, "is", i*i)
```



FUNCTION & RECURSION

1. Functions in Python

1.1 Definition

- A function is a block of reusable code that performs a specific task.
- Improves code reusability, clarity, and modularity.

1.2 Importance

- Reduces code repetition.
- Makes programs organized.
- Essential in AI/ML for data preprocessing, model building, evaluation, etc.

Built-in Functions

- **Definition**
  - Functions already provided by Python.
  - Can be used directly without creating them.

  Examples
  - print()
  - input()
  - len()
  - sum()
  - type()
  - range()

Importance

- Saves time and avoids rewriting common operations.

---

2. Defining a Function

User-Defined Functions

- **Definition**
  - Functions created by the programmer using the def keyword.

2.1 Syntax

```
def function_name(parameters):
    statements
```

2.2 Example

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

---

3. Calling a Function

3.1 Example

```
greet()
```

---

4. Function with Parameters

4.1 Single Parameter

```
def square(x):
    return x * x
```

4.2 Multiple Parameters

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

---

5. Return Statement

5.1 Definition

- return sends a value back to the caller.

5.2 Example

```
def get_name():
    return "Python"
```

---

6. Default Parameters

6.1 Definition

- A default value is used if no argument is passed.

6.2 Example

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

7. Keyword Arguments

7.1 Definition

- Parameters passed with names.

7.2 Example

```
def details(name, age):
    print(name, age)

details(age=20, name="Ravi")
```

---

8. Scope of Variables

8.1 Local Variable

- Declared inside a function.

8.2 Global Variable

- Declared outside the function.

8.3 global Keyword

```
x = 10
def update():
    global x
    x = 20
```

---

9. Anonymous Functions (Lambda)

9.1 Definition

- A small, single-expression function.

9.2 Syntax

```
lambda arguments: expression
```

9.3 Example

```
square = lambda x: x*x
```

---

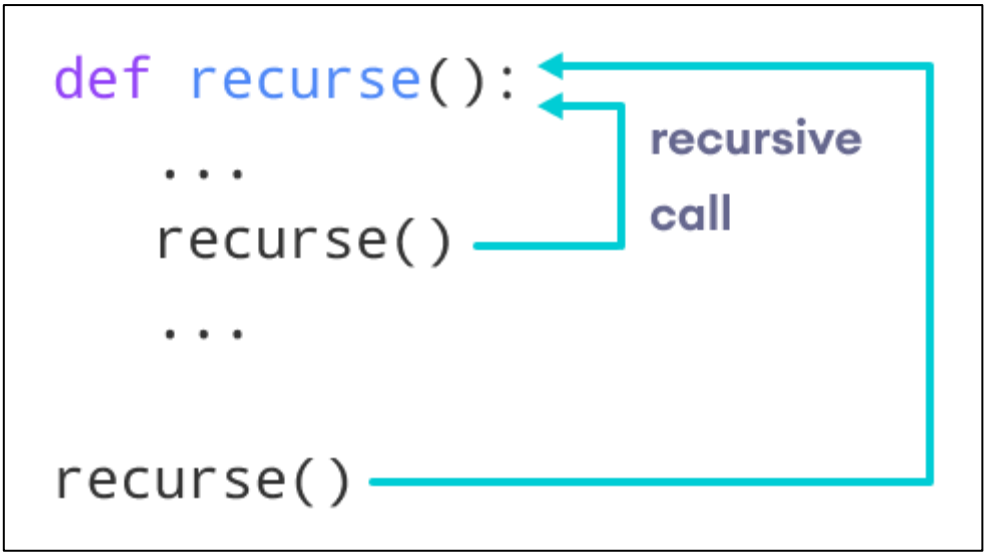
10. Recursion in Python

10.1 Definition

- Recursion occurs when a function calls itself.
- A problem is broken into smaller subproblems.
- Must include a **base case** to stop recursion.

10.2 Importance

- Used in mathematical problems, tree traversal, backtracking algorithms.
- Important for algorithms used in AI search techniques.



11. Structure of a Recursive Function

11.1 General Pattern

```
def function():
    if base_condition:
        return value
    else:
        return function(smaller_input)
```

---

12. Example: Factorial Using Recursion

```
def fact(n):
    if n == 0:
        return 1            # base case
    else:
        return n * fact(n-1)
```

---

13. Example: Fibonacci Using Recursion

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

---

14. Advantages of Recursion

- Code becomes cleaner and shorter.
- Good for problems naturally defined recursively.
- Useful for tree and graph algorithms.

---

15. Disadvantages of Recursion

- Uses more memory (stack frames).
- Slower than loops.
- Risk of infinite recursion if base case missing.

---

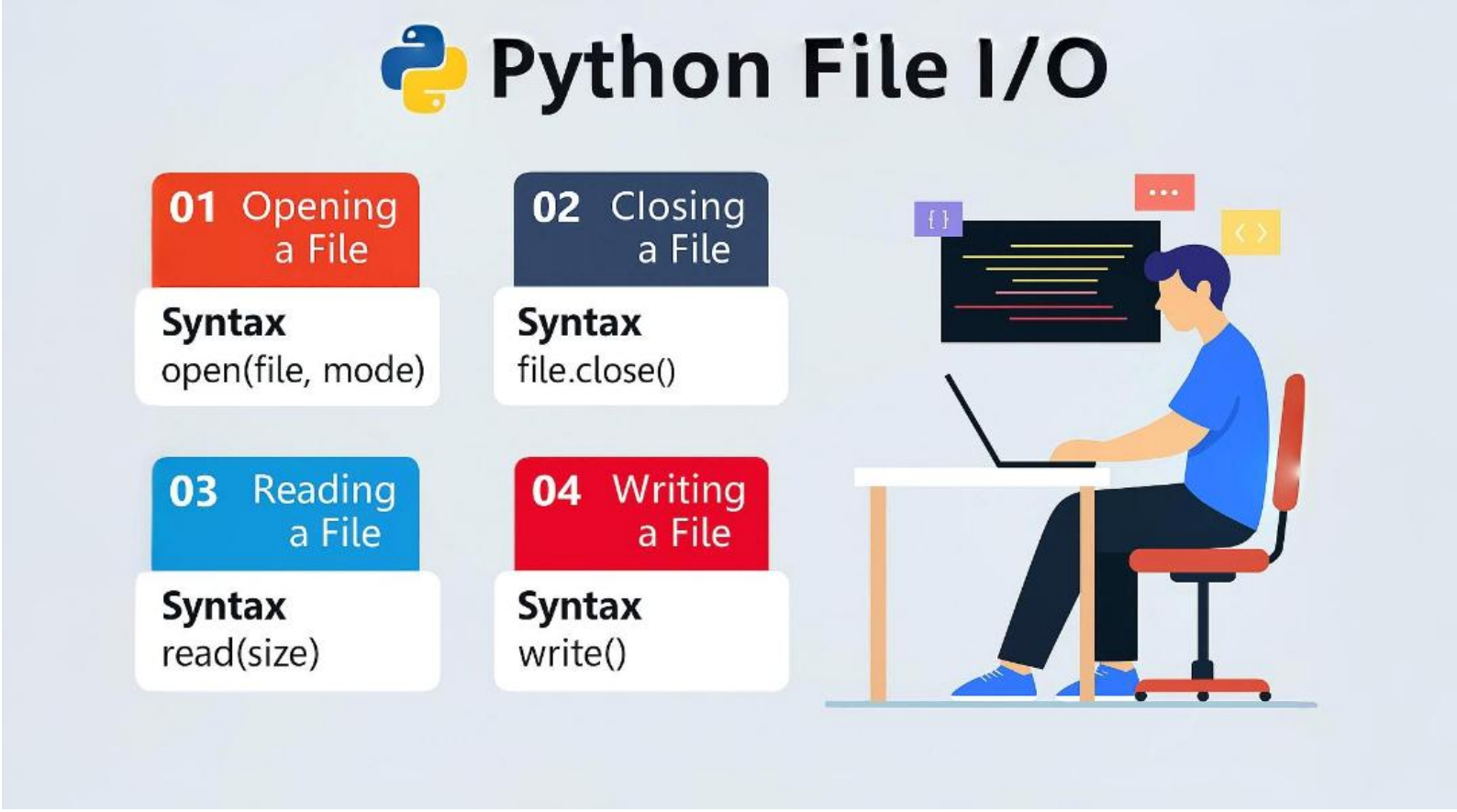
16. Recursion Limit

16.1 Checking Recursion Limit

```
import sys
sys.getrecursionlimit()
```

16.2 Setting Recursion Limit

```
sys.setrecursionlimit(2000)
```



1. Introduction to File I/O

1.1 Definition

- File Input/Output (I/O) refers to reading data from files and writing data to files.
- Python provides built-in functions to handle text files and binary files.

1.2 Importance

- Used for storing datasets, logs, model outputs, and configuration files.
- Essential in AI/ML for reading CSV, text, JSON files.

---

2. Opening a File

2.1 Syntax

```
file = open("filename", "mode")
```

2.2 File Modes

Mode	Meaning
"r"	Read (default)
"w"	Write (overwrite existing)
"a"	Append (add to end)
"x"	Create new file (error if exists)
"b"	Binary mode
"t"	Text mode (default)

"r+"	Read + write
"w+"	Write + read

---

3. Reading From a File

3.1 read() – Read Entire File

```
file = open("data.txt", "r")
content = file.read()
```

3.2 readline() – Read One Line

```
line = file.readline()
```

3.3 readlines() – Read All Lines as List

```
lines = file.readlines()
```

---

4. Writing to a File

4.1 write() – Write Text

```
file = open("data.txt", "w")
file.write("Hello Python")
```

4.2 writelines() – Write Multiple Lines

```
lines = ["A\n", "B\n", "C\n"]
file.writelines(lines)
```

---

5. Closing a File

5.1 close() Method

```
file.close()
```

5.2 Importance

- Releases system resources.
- Ensures data is fully written to disk.

---

6. Using with Statement (Best Practice)

6.1 Syntax

```
with open("file.txt", "r") as f:
    data = f.read()
```

6.2 Advantages

- Closes file automatically.
- Prevents file damage and memory leaks.

---

7. Checking File Existence (Optional)

```
import os
os.path.exists("file.txt")
```

---

8. Working With Binary Files

8.1 Read Binary

```
with open("image.jpg", "rb") as f:
    data = f.read()
```

8.2 Write Binary

```
with open("copy.jpg", "wb") as f:
    f.write(data)
```

---

9. Example: Complete File Read–Write Program

```
# Write to a file
with open("sample.txt", "w") as f:
    f.write("Hello Python Programming")

# Read the same file
with open("sample.txt", "r") as f:
    print(f.read())
```

TYPES OF FILES

1. Introduction to File Types

1.1 Definition

- Files are used to store data permanently on disk.
- Python can work with different types of files depending on data format.

1.2 Main Categories

1. **Text Files**
2. **Binary Files**

---

2. Text Files

2.1 Definition

- Store data in plain readable text.
- Characters encoded using formats like ASCII or UTF-8.
- Extension usually .txt, .csv, .json, .html, .py.

2.2 Examples

- **"notes.txt"**
- **"data.csv"**
- **"config.json"**
- **"script.py"**

2.3 How Python Opens Text Files

```
open("file.txt", "r")
open("file.txt", "w")
```

2.4 Use Cases

- Logs
- Configuration files
- Dataset files (CSV)
- Code files
- Documentation

---

3. Binary Files

3.1 Definition

- Store data in binary (0s and 1s).
- Cannot be read directly as text.
- Extension types .jpg, .png, .mp3, .mp4, .exe, .dat.

3.2 Examples

- **"image.jpg"**
- **"video.mp4"**
- **"audio.mp3"**
- **"model.dat"**

3.3 How Python Opens Binary Files

```
open("file.jpg", "rb")
open("file.jpg", "wb")
```

3.4 Use Cases

- Images
- Audio/Video
- Machine Learning model files
- Pickle files
- Compressed files

---

4. Structured Data Files (Special Category)

(Technically text or binary, but used specially in data science.)

4.1 CSV Files

- Text-based table data.

```
import csv
```

4.2 JSON Files

- Human-readable structured data.

```
import json
```

4.3 XML Files

- Markup-based structured text.

4.4 YAML Files

- Configuration files used in ML projects.

---

5. Python Script Files

5.1 .py Files

- Contain Python code.
- Executed by Python interpreter.

---

6. Pickle Files (Binary)

6.1 Definition

- Store Python objects in binary form.
- Extension .pkl or .pickle.

6.2 Example

```
import pickle
pickle.dump(obj, open("model.pkl", "wb"))
```

7. Difference Between Text and Binary Files

Feature	Text Files	Binary Files
Readability	Human-readable	Not readable
Format	Characters	Raw bytes
Opening Mode	"r", "w"	"rb", "wb"
Errors	More tolerant	Very sensitive
Usage	Logs, CSV, JSON	Images, audio, models

1. Introduction to OOP

1.1 Definition

- Object-Oriented Programming (OOP) is a programming paradigm based on objects and classes.
- Focuses on reusability, modularity, and organization of code.

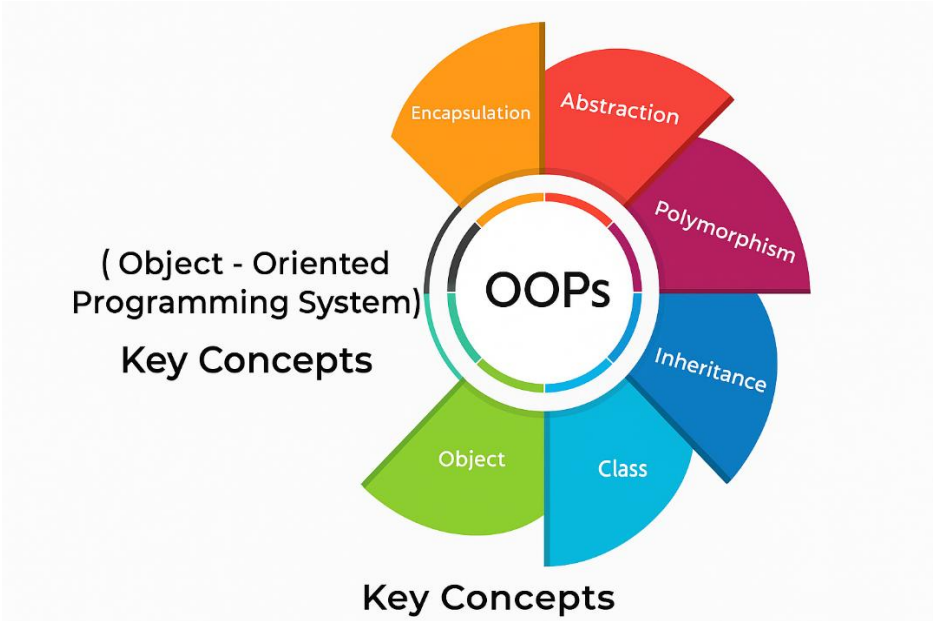
1.2 Importance

- Makes complex programs easier to manage.
- Commonly used in AI/ML frameworks (TensorFlow, PyTorch).
- Encourages code reuse through inheritance and abstraction.

---

2. Key Concepts of OOP

1. Class
2. Object
3. Encapsulation
4. Inheritance
5. Polymorphism
6. Abstraction



---

3. Class and Object

3.1 Class

- A variable defined inside a class but outside any method.
- Shared by all objects of the class.

Usage

- Store constant or common values.

Example

```
class Employee:
    company = "ABC Corp"    # class attribute
```

Accessing

```
print(Employee.company)
print(emp1.company)
```

3.2 Object

- An object is an instance of a class.
- Represents a real-world entity.
- Contains attributes (data) and methods (functions) defined in its class.

3.3 Characteristics

- Has state (data)
- Has behavior (methods)
- Occupies memory

Example

```
class Car:
    pass

c1 = Car()   # c1 is an object
```

---

4. Constructor (init Method)

4.1 Definition

- Special method that automatically runs when an object is created.
- Used to initialize object attributes.

4.2 Example

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

s = Student("Ravi", 20)
print(s.name)
print(s.age)
```

Output

```
Ravi
20
```

---

5. Attributes and Methods

5.1 Attributes

- Variables inside a class (data).

5.2 Methods

- Functions inside a class (behavior).

Example

```
class Car:
    def __init__(self, brand):
        self.brand = brand

    def show(self):
        print(self.brand)

c = Car("Toyota")
c.show()
```

Output

```
Toyota
```

---

6. Encapsulation

6.1 Definition

- Binding data (variables) and methods together.
- Protecting internal data from direct access.

6.2 Private Variables

- Declared using __var.

Example

```
class Test:
    __x = 10

print(Test._Test__x)   # private variable access using name-mangling
```

Output
10

---

7. Inheritance

7.1 Definition

- One class (child) inherits features from another class (parent).
- Promotes code reuse.

7.2 Syntax

```
class Child(Parent):
    pass
```

7.3 Example

```
class Animal:
    def sound(self):
        print("Some sound")

class Dog(Animal):
    def sound(self):
        print("Bark")

d = Dog()
d.sound()
```

Output

```
Bark
```

---

8. Polymorphism

8.1 Definition

- Same function name behaves differently for different objects.
- Achieved by method overriding.

8.2 Example

```
class Cat:
    def sound(self):
        print("Meow")

class Dog:
    def sound(self):
        print("Bark")

c = Cat()
d = Dog()

c.sound()
d.sound()
```

Output

```
Meow
Bark
```

---

9. Abstraction

9.1 Definition

- Hiding internal details and showing only essential features.

9.2 Using Abstract Classes

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side * self.side

s = Square(5)
print(s.area())
```

Output

```
25
```

---

10. self Keyword

10.1 Definition

- Refers to the current object.
- Used to access attributes and methods inside the class.

Example

```
    def display(self):
        print(self.name)
```

11. Modeling a Problem in OOP

11.1 Definition

- Translating a real-world problem into classes, objects, attributes, and methods.

11.2 Steps in Modeling

1. **Identify entities** → Convert them into classes
2. **Identify properties** → Convert them into attributes
3. **Identify actions/behaviors** → Convert them into methods
4. **Create objects** → Represent real instances
5. **Define relationships** → Use inheritance, composition, etc.

11.3 Example: Student Management System

- **Classes**: Student, Teacher, Course
- **Attributes**: name, age, roll_no
- **Methods**: enroll(), attend(), get_marks()
- **Objects**: s1, s2 (students)

12.Instance Attribute

12.1 Definition

- Attributes created **for each object separately**.
- Defined inside the constructor (__init__()).

12.2 Example

```
class Employee:
    def __init__(self, name, salary):
        self.name = name           # instance attribute
        self.salary = salary
```

12.3 Characteristics

- Each object gets its own copy.
- Changing one does not affect others.

---

13. self Parameter

13.1 Definition

- Refers to the **current object** calling the method.
- Used to access instance attributes and methods.

13.2 Why self is Needed

- Distinguishes between class attributes and object attributes.
- Automatically passed during method call.

13.3 Example

```
class Test:
    def show(self):
        print("Inside method using", self)

t = Test()
t.show()
```

Output

```
Inside method using <__main__.Test object at 0x000...>
```

---

14. Static Method

14.1 Definition

- A method inside a class that does **not access instance or class attributes**.
- Behaves like a normal function but belongs to the class namespace.

14.2 Declared Using

```
@staticmethod
```

14.3 Use Cases

- Utility functions
- Helper methods
- Operations not related to instance data

14.4 Example

```
class Math:
    @staticmethod
    def add(a, b):
        return a + b

print(Math.add(5, 7))
```

Output

```
12
```

---

15. init() Constructor

15.1 Definition

- Special method that automatically runs when an object is created.
- Used to initialize object attributes.

- Known as the “constructor”.

15.2 Syntax

```
def __init__(self, parameters):
    statements
```

15.3 Example

```
class Student:
    def __init__(self, name, roll):
        self.name = name
        self.roll = roll

s = Student("Ravi", 101)
print(s.name)
print(s.roll)
```

Output

```
Ravi
101
```

15.4 Characteristics

- Must include **self** as first parameter.
- Automatically called.
- Used for setting initial state of object.

Complete Example Demonstrating All Concepts

```
class Student:
    # Class Attribute
    school = "ABC School"
    # Constructor (Instance Attributes)
    def __init__(self, name, age):
        self.name = name
        self.age = age
    # Instance Method (uses self)
    def show(self):
        print("Name:", self.name)
        print("Age:", self.age)
    # Static Method
    @staticmethod
    def info():
        print("This is a Student class")
# Creating Object
s1 = Student("Ravi", 20)
s1.show()           # instance method
Student.info()      # static method
print(Student.school)  # class attribute
```

Inheritance in Python

1. Introduction to Inheritance

1.1 Definition

- Inheritance allows one class (**child class**) to acquire the properties and methods of another class (**parent class**).
- Promotes **code reuse**, **extensibility**, and **organization**.

1.2 Importance

- Avoids code duplication.
- Helps build hierarchical class structures.
- Essential in frameworks like TensorFlow, PyTorch (layer inheritance).

---

2. Terminology

2.1 Parent Class (Base Class / Superclass)

- The class whose attributes and methods are inherited.

2.2 Child Class (Derived Class / Subclass)

- The class that inherits from the parent class.

2.3 Syntax

```
class Child(Parent):
    pass
```

---

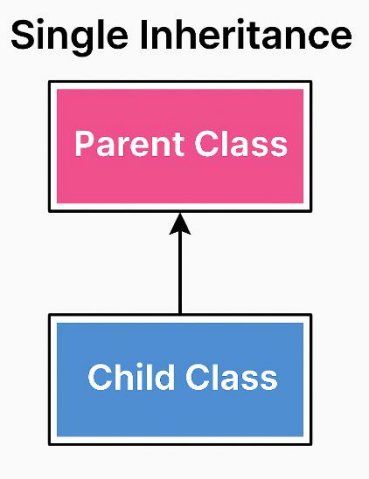
3. Types of Inheritance in Python

1. **Single Inheritance**
2. **Multiple Inheritance**

3. **Multilevel Inheritance**
4. **Hierarchical Inheritance**
5. **Hybrid Inheritance**

---

4. Single Inheritance



4.1 Definition

- One parent class and one child class.

4.2 Example

```
class Animal:
    def sound(self):
        print("Some sound")

class Dog(Animal):
    pass

d = Dog()
d.sound()
```

Output

```
Some sound
```

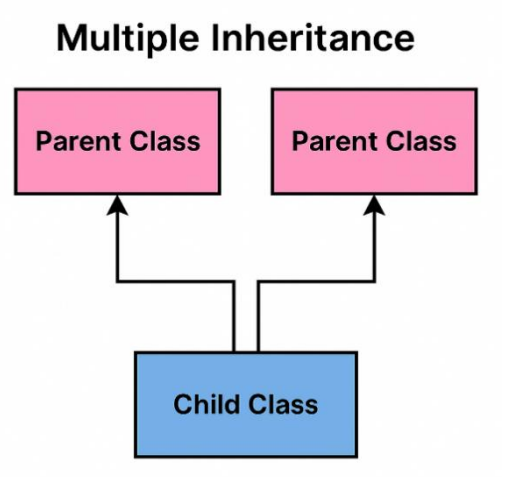
---

5. Multiple Inheritance

5.1 Definition

- Child class inherits from **multiple parent classes**.

5.2 Example




```
class A:
    def showA(self):
        print("A")

class B:
    def showB(self):
        print("B")

class C(A, B):
    pass

obj = C()
```

```
obj.showA()
obj.showB()
```

Output

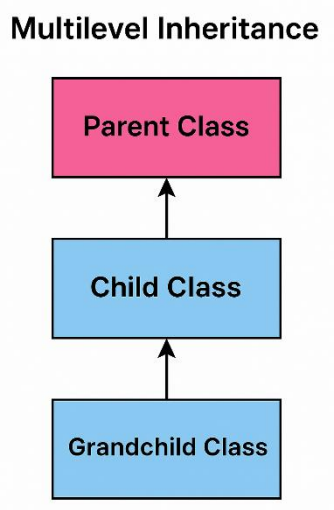
```
A
B
```

---

6. Multilevel Inheritance

6.1 Definition

- A class inherits from a child class, forming a chain (grandparent → parent → child).



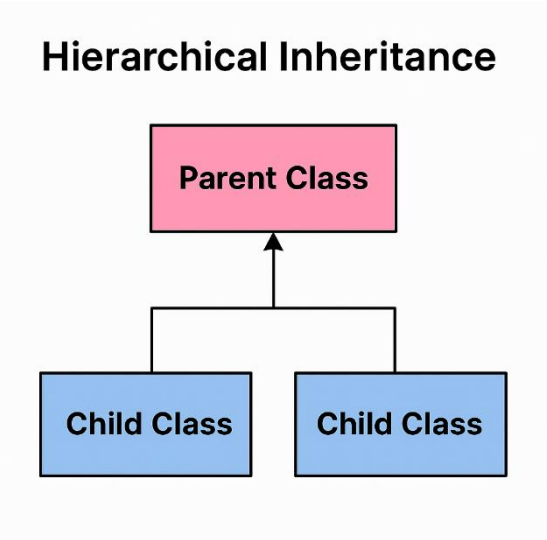
6.2 Example

```
class A:
    def showA(self):
        print("Class A")

class B(A):
    pass

class C(B):
    pass

obj = C()
obj.showA()
```

Output

```
Class A
```

7. Hierarchical Inheritance



7.1 Definition

- One parent class with multiple child classes.

7.2 Example

```
class Animal:
    def sound(self):
        print("Animal sound")

class Dog(Animal):
    pass

class Cat(Animal):
    pass

d = Dog()
```

```
c = Cat()

d.sound()
c.sound()
```

Output

```
Animal sound
Animal sound
```

---

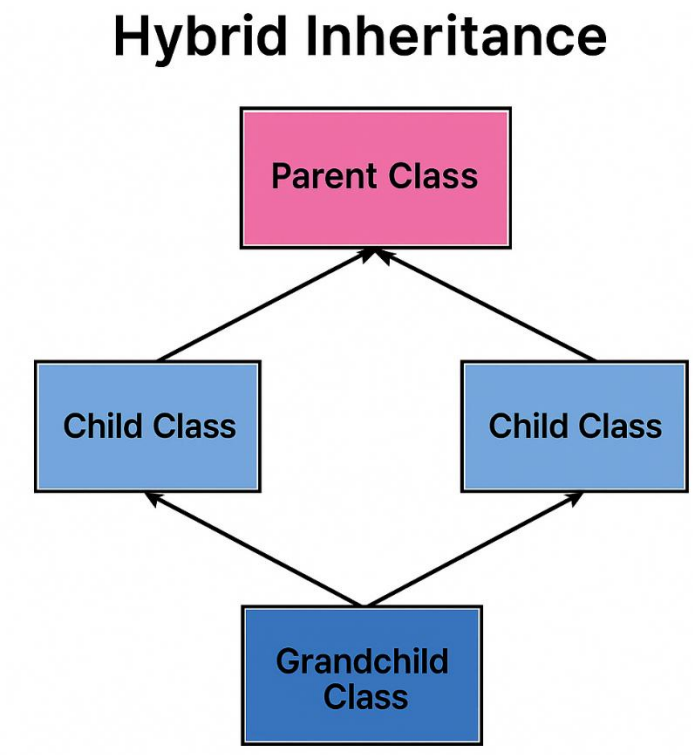
8. Hybrid Inheritance

8.1 Definition

- Hybrid inheritance combines **multiple types of inheritance** in Python.
- Python supports hybrid inheritance because it allows **multiple and multilevel inheritance**.
- It is used to create **complex class relationships**.
- Python uses **MRO (Method Resolution Order)** to resolve method conflicts.
- Helps in **code reusability** by using features of multiple classes.

8.2 Example

```
# Parent Class 1
class A:
    def showA(self):
        print("Class A")

# Parent Class 2
class B:
    def showB(self):
        print("Class B")

# Child of A (Multilevel part)
class C(A):
    def showC(self):
        print("Class C")

# Hybrid Class (inherits from B
class D(B, C):
    def showD(self):
        print("Class D")

# Object
obj = D()
obj.showA()
obj.showB()
```



```
obj.showC()
obj.showD()
```

Output
Class A
Class B
Class C
Class D

---

9. Overriding Methods

9.1 Definition

- Child class can redefine a method already defined in the parent class.

9.2 Example

```
class Animal:
    def sound(self):
        print("General sound")

class Dog(Animal):
    def sound(self):
        print("Bark")

d = Dog()
d.sound()
```

Output
Bark

---

10. super() Keyword

10.1 Purpose

- **super is used to call the parent class methods inside the child class.**
- It helps access overridden methods of the parent class.
- It is mainly used in **method overriding** to reuse parent class code.
- super() also works inside **constructors (__init__)** to call the parent constructor.
- It supports **multiple inheritance** by following Python's **MRO (Method Resolution Order)**.
- It reduces code duplication by allowing child classes to reuse parent class functionality.
- It makes the program more organized and easier to maintain.

Example

```
class Animal:
    def sound(self):
        print("General sound")

class Dog(Animal):
    def sound(self):
        super().sound()   # parent class method call
        print("Bark")
```

Output

General sound

Bark

10.2 Example (Calling Parent Constructor)

```
class A:
    def __init__(self):
        print("A constructor")

class B(A):
    def __init__(self):
        super().__init__()
        print("B constructor")

b = B()
```

Ouptut

```
A constructor
B constructor
```

---

11. Constructor in Inheritance

11.1 Child Calls Parent Automatically?

- If child has **no constructor**, parent constructor is called automatically.
- If child has its own constructor, parent constructor must be called manually using super().

---

12. Advantages of Inheritance

12.1 Benefits

- Code reusability
- Reduces redundancy
- Encourages modularity
- Easier maintenance

- Supports hierarchical modeling

✓ Complete Example Demonstrating Inheritance

```
class Person:
    def __init__(self, name):
        self.name = name

    def info(self):
        print("Name:", self.name)

class Student(Person):
    def __init__(self, name, roll):
        super().__init__(name)
        self.roll = roll

    def show(self):
        print("Roll:", self.roll)

s1 = Student("Ravi", 101)
s1.info()
s1.show()
```

MORE ON OOPS

1. @classmethod

1.1 Definition

- A @classmethod is a method that takes **class** as its first argument instead of object.
- Uses cls instead of self.

1.2 Purpose

- Access and modify **class attributes**.
- Create **alternative constructors**.

1.3 Syntax

```
@classmethod
def method_name(cls):
    statements
```

1.4 Example

```
class Student:
    school = "ABC School"

    @classmethod
    def change_school(cls, new_name):
        cls.school = new_name

Student.change_school("XYZ School")
print(Student.school)
```

Output

```
XYZ School
```

---

2. @property Decorator

2.1 Definition

- Converts a method into a **read-only attribute**.
- Allows controlled access to private variables.

2.2 Purpose

- Encapsulation
- Validation of data
- Access private attributes safely

2.3 Syntax

```
@property
def name(self):
    return self._name
```

2.4 Example

```
class Employee:
    def __init__(self, salary):
        self._salary = salary

    @property
    def salary(self):
        return self._salary

e = Employee(50000)
print(e.salary)
```

Output

50000

---

3. Getters and Setters

3.1 Definition

- **Getter** → Method to get the value of private attributes.
- **Setter** → Method to set/update the private attribute with validation.

3.2 Why Needed

- Protects data integrity.
- Prevents invalid values.

3.3 Syntax Using @property

3.3.1 Getter

```
@property
def age(self):
    return self._age
```

3.3.2 Setter

```
@age.setter
def age(self, value):
    if value > 0:
        self._age = value
    else:
        print("Invalid age")
```

3.4 Complete Example

```
class BankAccount:
    def __init__(self, balance):
        self._balance = balance

    @property
    def balance(self):
        return self._balance

    @balance.setter
    def balance(self, amount):
        if amount >= 0:
            self._balance = amount
        else:
            print("Balance cannot be negative")

account = BankAccount(1000)

print(account.balance)   # getter

account.balance = 1500   # setter
print(account.balance)

account.balance = -200   # invalid
```

Output

1000
1500
Balance cannot be negative

---

4. Operator Overloading

4.1 Definition

- Python allows operators like +, -, *, == to work on user-defined objects.
- Achieved by defining **special methods** (magic methods).

4.2 Purpose

- Customize behavior of operators for objects.
- Used in AI/ML libraries for tensor operations (PyTorch, NumPy).

---

5. Common Magic Methods for Operator Overloading

5.1 Addition

```
    __add__(self, other)
```

5.2 Subtraction

```
__sub__(self, other)
```

5.3 Multiplication

```
__mul__(self, other)
```

5.4 String Representation

```
__str__(self)
```

5.5 Equality

```
__eq__(self, other)
```

5.6 Greater Than / Less Than

```
__gt__(self, other)
__lt__(self, other)
```

---

6. Example of Operator Overloading

6.1 Adding Two Objects

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

p1 = Point(2, 3)
p2 = Point(4, 1)

p3 = p1 + p2

print(p3.x, p3.y)
```

Output

```
6 4
```

---

7. Full Example Covering Everything

```
class Product:

    # class attribute
    store = "Online Store"

    # constructor
    def __init__(self, price):
        self._price = price

    # class method
    @classmethod
    def change_store(cls, new_name):
        cls.store = new_name

    # getter
    @property
    def price(self):
        return self._price

    # setter
    @price.setter
    def price(self, value):
        if value >= 0:
            self._price = value
        else:
            print("Invalid price")

    # operator overloading
    def __add__(self, other):
        return Product(self.price + other.price)


# creating objects
p1 = Product(300)
p2 = Product(500)

# using + operator
p3 = p1 + p2

print(p1.price)
print(p2.price)
print(p3.price)

# changing class attribute
Product.change_store("SuperMart")
print(Product.store)
```

Output

```
300
500
800
SuperMart
```

ADVANCED CONCEPTS

1. Walrus Operator (:=)

1.1 Definition

- Introduced in Python 3.8.
- Assigns a value **inside an expression**.
- Also called **Assignment Expression Operator**.

1.2 Purpose

- Reduces code repetition.
- Useful inside loops and conditions.

1.3 Example

```
if (n := len("Python")) > 3:
    print(n)
```

Output

```
6
```

---

2. Type Definitions in Python

2.1 Definition

- Python allows defining variable and function types using **type hints**.
- Not enforced at runtime; improves readability.

2.2 Variable Type Definition

```
age: int = 20
name: str = "Ravi"
```

2.3 Function Type Definition

```
def add(a: int, b: int) -> int:
    return a + b

result = add(10, 15)
print(result)
```

Output

```
25
```

3. Advanced Type Hints

3.1 Union Type

```
x: int | str
```

Example

```
x: int | str

x = 10
print(x)

x = "Hello"
print(x)
```

Output

```
10
Hello
```

3.2 Optional (value can be None)

```
from typing import Optional
age: Optional[int] = None
```

Example

```
from typing import Optional

age: Optional[int] = None
print(age)

age = 25
print(age)
```

Output

```
None
25
```

3.3 Advanced Python Tips:

List, Tuple, Dict Types

```
from typing import List, Dict, Tuple
names: List[str]
student: Dict[str, int]
```

Example

```
from typing import List, Dict, Tuple

names: List[str] = ["Ravi", "Aman", "Neha"]
student: Dict[str, int] = {"Ravi": 20, "Aman": 22}

print(names)
print(student)
```

Output

```
['Ravi', 'Aman', 'Neha']
{'Ravi': 20, 'Aman': 22}
```

3.4 Any Type

```
from typing import Any
data: Any
```

Example

```
from typing import Any

data: Any = 10
print(data)

data = "Hello"
print(data)

data = [1, 2, 3]
print(data)
```

Output

```
10
Hello
[1, 2, 3]
```

3.5 Type Alias

```
UserID = int
```

Example

```
UserID = int

uid: UserID = 101
print(uid)
```

Output

```
101
```

3.6 Callable Type

```
from typing import Callable
func: Callable[[int, int], int]
```

3.7 TypedDict

```
from typing import TypedDict

class Student(TypedDict):
    name: str
    age: int
```

Example

```
from typing import Callable

func: Callable[[int, int], int]

def add(a: int, b: int) -> int:
    return a + b

func = add
print(func(5, 7))
```

Output

```
12
```

---

4. match–case (Structural Pattern Matching)

4.1 Definition

- Introduced in Python 3.10.
- Works like switch-case with pattern matching.

4.2 Syntax

```
match value:
    case pattern1:
        ...
    case pattern2:
        ...
```

4.3 Example

```
day = 3

match day:
    case 1:
        print("Monday")
    case 2:
        print("Tuesday")
    case 3:
        print("Wednesday")
    case 4:
        print("Thursday")
    case 5:
        print("Friday")
    case 6:
        print("Saturday")
    case 7:
        print("Sunday")
    case _:
        print("Invalid day")
```

Output

```
Wednesday
```

---

5. Dictionary Merge & Update Operators

5.1 Merge Operator (|)

- Creates a new merged dictionary.

Syntax

```
d1 | d2
```

Example

```
d1 = {"name": "Ravi", "age": 20}
d2 = {"city": "Delhi", "age": 21}

merged = d1 | d2
print(merged)
```

Output

```
{'name': 'Ravi', 'age': 21, 'city': 'Delhi'}
```

5.2 Update Operator (|=)

- Updates existing dictionary.

Syntax

```
d1 |= d2
```

Example

```
d1 = {"name": "Ravi", "age": 20}
d2 = {"city": "Delhi", "age": 21}

d1 |= d2   # d1 gets updated
print(d1)
```

Output

```
{'name': 'Ravi', 'age': 21, 'city': 'Delhi'}
```

---

6. Exception Handling in Python

6.1 Definition

- Mechanism to handle runtime errors.

6.2 try–except Syntax

```
try:
    code
except ExceptionType:
    handle_error
```

Example

```
try:
    x = int(input("Enter a number: "))
    result = 10 / x
    print("Result =", result)
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Invalid input. Please enter a number.")
```

6.3 Multiple Except Blocks

```
try:
    ...
except ValueError:
    ...
except ZeroDivisionError:
    ...
```

Example

```
try:
    a = int(input("Enter value: "))
    b = int(input("Enter value: "))
    print(a / b)
except ZeroDivisionError:
    print("Division by zero not allowed")
except ValueError:
    print("Please enter valid integers")
except Exception as e:
    print("Unknown Error:", e)
```

---

7. Raising Exceptions

7.1 Purpose

- Forcefully create an error when needed.

7.2 Syntax

```
raise ValueError("Invalid input")
```

Example

```
def set_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative")
    return age

try:
    set_age(-5)
except ValueError as e:
    print("Exception Raised:", e)
```

---

8. try with else Clause

8.1 Definition

- Executes **only if no exception occurs**.

8.2 Syntax

```
try:
    code
except:
    handler
else:
    run_if_no_error
```

Example

```
try:
    num = int(input("Enter number: "))
except ValueError:
    print("Invalid input")
else:
    print("Square =", num * num)
```

---

9. try with finally Clause

9.1 Definition

- Block that executes **always**, even if error occurs.

9.2 Use Case

- Closing files, releasing resources.

9.3 Syntax

```
try:
    code
finally:
    cleanup
```

Example

```
try:
    file = open("demo.txt", "r")
    content = file.read()
    print(content)
except FileNotFoundError:
    print("File not found")
finally:
    print("Closing program...")
```

---

10. if name == 'main'

10.1 Definition

- Special block that runs only when the file is executed directly.
- Prevents code from running during import.

10.2 Example

```
def main():
    print("Running script")

if __name__ == "__main__":
    main()
```

Output

```
Running script
```

---

11. The global Keyword

11.1 Definition

- Used to modify a global variable inside a function.

11.2 Example

```
x = 10

def update():
    global x
    x = 20

update()
print(x)
```

Output

```
20
```

---

12. enumerate() Function

12.1 Definition

- Returns index and value while looping.

12.2 Syntax

```
for index, value in enumerate(sequence):
```

12.3 Example

```
for i, name in enumerate(["A","B"]):
    print(i, name)
```

Output

```
0 A
1 B
```

---

13. List Comprehensions

13.1 Definition

- Short syntax to create lists using loops and conditions.

13.2 Basic Syntax

```
[expression for item in iterable]
```

13.3 With Condition

```
[x for x in range(10) if x % 2 == 0]
```

Output

```
[0, 2, 4, 6, 8]
```

13.4 Example

```
squares = [n*n for n in range(5)]
```

Output

```
[0, 1, 4, 9, 16]
```

MORE CONCEPTS

1. Virtual Environment in Python

1.1 Definition

- A virtual environment is an isolated Python environment.
- Allows installing libraries for one project without affecting others.

1.2 Purpose

- Avoid version conflicts.
- Create separate environments for separate projects.
- Essential in AI/ML projects for specific library versions.

---

2. Creating and Activating Virtual Environment

2.1 Installation (if needed)

```
pip install virtualenv
```

2.2 Create Virtual Environment

```
python -m venv myenv
```

2.3 Activate Environment

Windows

```
myenv\Scripts\activate
```

Mac/Linux

```
source myenv/bin/activate
```

2.4 Deactivate

```
deactivate
```

---

3. pip freeze Command

3.1 Definition

- Lists all installed packages with exact versions.

3.2 Usage

```
pip freeze
```

3.3 Export To File

```
pip freeze > requirements.txt
```

3.4 Install From File

```
pip install -r requirements.txt
```

4. Lambda Functions

4.1 Definition

- Small, anonymous functions created using lambda.
- Limited to one expression.

4.2 Syntax

```
lambda arguments: expression
```

4.3 Examples

Square

```
square = lambda x: x * x
print(square(5))
```

Output

```
25
```

Add

```
add = lambda a, b: a + b
print(add(3, 7))
```

Output

```
10
```

---

5. join() Method (Strings)

5.1 Definition

- Converts a list of strings into a single string.
- Joins elements using a separator.

5.2 Syntax

```
separator.join(iterable)
```

5.3 Example

```
names = ["Ravi", "Neha", "Aman"]
result = ", ".join(names)
```

Output:

```
"Ravi, Neha, Aman"
```

---

6. format() Method (Strings)

6.1 Definition

- Formats strings using placeholders {}.

6.2 Syntax

```
"{}".format(value)
```

6.3 Examples

Positional

```
"Name: {}, Age: {}".format("Ravi", 20)
```

Keyword

```
"Name: {n}, Age: {a}".format(n="Ravi", a=20)
```

f-String (Advanced)

```
name = "Ravi"
f"Hello {name}"
```

---

7. map(), filter(), reduce()

7.1 map()

Definition

- Applies a function to each element of an iterable.

Syntax

```
"Name: {n}, Age: {a}".format(n="Ravi", a=20)
```

Example

```
numbers = [1, 2, 3]
result = list(map(lambda x: x*2, numbers))
```

Output:

[2, 4, 6]

7.2 filter()

Definition

- Filters elements based on a condition.

Syntax

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

Example

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

Output:

[2, 4]

7.3 reduce()

Definition

- Reduces a sequence to a single value.
- Comes from functools.

Syntax

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

Example

```
from functools import reduce
nums = [1, 2, 3, 4]
result = reduce(lambda a, b: a + b, nums)
print(result)
```

Output:

10

---

8. Practical AI/ML Use Cases

8.1 Virtual Environment

- Different ML projects need different library versions.

8.2 pip freeze

- Export environment for deployment and reproducibility.

8.3 Lambda

- Used in data preprocessing and sorting.

8.4 map/filter

- Apply transformations to datasets.

8.5 reduce

- Summarize data (sum, product, accumulation).

1. LIST (Dynamic Array)

1.1 Definition

- Ordered, mutable, dynamic sequence of items.
- Allows duplicates and mixed data types.

1.2 Syntax

```
list_name = [item1, item2, item3]
```

1.3 Example

```
nums = [10, 20, 30]
nums.append(40)
nums[1] = 25
print(nums)
```

Output

[10, 25, 30, 40]

---

2. TUPLE (Immutable Sequence)

2.1 Definition

- Ordered, immutable, fixed-size sequence.

2.2 Syntax

```
t = (1, 2, 3)
```

2.3 Example

```
t = (5, 10, 15)
print(t[1])     # accessing
```

Output

10

---

3. SET (Unique Unordered Collection)

3.1 Definition

- Unordered collection of **unique** elements.
- No duplicates, no indexing.

3.2 Syntax

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

3.3 Example

```
s = {1, 2, 3, 3}
s.add(4)
print(s)
```

Output

{1, 2, 3, 4}

---

4. DICTIONARY (HashMap)

4.1 Definition

- Stores **key–value pairs**.
- Implemented using **hash tables**.

4.2 Syntax

```
d = {"key": "value"}
```

4.3 Example

```
student = {"name": "Ravi", "marks": 90}
student["age"] = 20
print(student["name"])
```

Output

Ravi

---

5. STACK (LIFO: Last In First Out)

5.1 Definition

- Last inserted element is removed first.

5.2 Implementation Using List

```
stack = []
stack.append("A")   # push
stack.append("B")
print(stack.pop())  # pop
```

Output

B

5.3 Implementation Using deque (recommended)

```
from collections import deque
stack = deque()
stack.append(10)
stack.append(20)
print(stack.pop())
```

Output

20

---

6. QUEUE (FIFO: First In First Out)

6.1 Definition

- First inserted element is removed first.

6.2 Using deque

```
from collections import deque

queue = deque()
queue.append(10)       # enqueue
queue.append(20)
print(queue.popleft()) # dequeue
```

Output

10

---

7. LINKED LIST

7.1 Definition

- Data stored in nodes.
- Each node contains:
  - data
  - next pointer

7.2 Node Class Example

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

7.3 Creating Linked List & Adding Nodes

```
class LinkedList:
    def __init__(self):
        self.head = None

    def insert_begin(self, data):
        new = Node(data)
        new.next = self.head
        self.head = new

    def display(self):
        temp = self.head
        while temp:
            print(temp.data, end=" -> ")
            temp = temp.next

# Usage
ll = LinkedList()
ll.insert_begin(30)
ll.insert_begin(20)
ll.insert_begin(10)
ll.display()
```

Output

```
10 -> 20 -> 30 ->
```

---

8. TREE

8.1 Definition

- Hierarchical structure with a **root node** and children.

8.2 Binary Tree Node Example

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```

8.3 Example: Creating a Simple Tree

```
root = Node(10)
root.left = Node(5)
root.right = Node(20)
print(root.left.data)
```

Output

```
5
```

---

9. GRAPH

9.1 Definition

- Set of **vertices** connected by **edges**.

9.2 Representation Using Dictionary (Adjacency List)

```
graph = {
    "A": ["B", "C"],
    "B": ["A", "D"],
    "C": ["A"],
    "D": ["B"]
}

print(graph["A"])
```

Output

```
['B', 'C']
```

---

10. HEAP (Priority Queue)

10.1 Definition

- Complete binary tree used for priority tasks.
- Python uses **min-heap** by default.

10.2 Example Using heapq

```
import heapq

heap = []
heapq.heappush(heap, 30)
heapq.heappush(heap, 10)
heapq.heappush(heap, 20)

print(heapq.heappop(heap))
```

Output

```
10
```

---

11. ADVANCED COLLECTIONS

11.1 deque

- Fast stack & queue.

Example:

```
from collections import deque
d = deque([1,2,3])
d.appendleft(0)
print(d)
```

11.2 Counter

```
from collections import Counter
print(Counter("banana"))
```

Output

```
Counter({'a': 3, 'n': 2, 'b': 1})
```

11.3 defaultdict

```
from collections import defaultdict
d = defaultdict(int)
d["count"] += 1
print(d)
```

11.4 namedtuple

```
from collections import namedtuple
Point = namedtuple("Point", "x y")
p = Point(3, 4)
print(p.x, p.y)
```

Output

```
3 4
```

12. Time Complexity Overview

Structure	Access	Search	Insert	Delete
List	O(1)	O(n)	O(n)	O(n)
Tuple	O(1)	O(n)	N/A	N/A
Set	N/A	O(1)	O(1)	O(1)
Dict	N/A	O(1)	O(1)	O(1)

ALGORITHMS

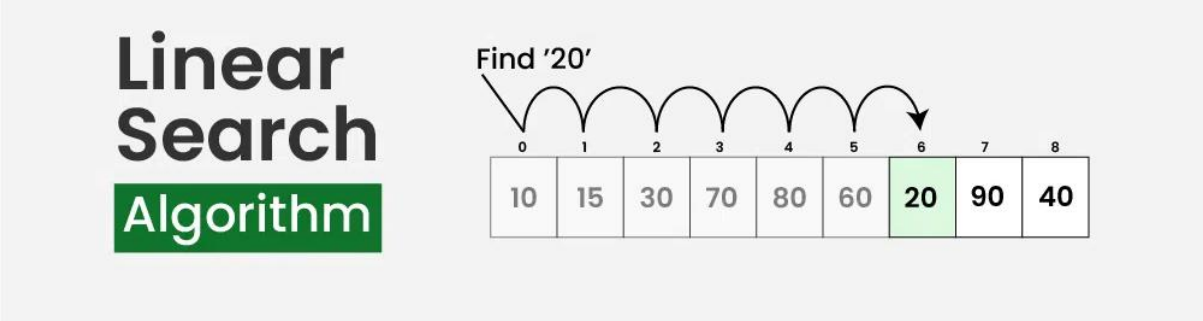
1.Algorithms

1.1 SEARCHING ALGORITHMS

A) Linear Search

- Checks each element **one by one** in a list.
- Works on **sorted or unsorted** lists.
- Start from index **0** and move forward.
- Compare each element with the **target**.
- If match found → return index.
- If list ends → element **not found**.

Time Complexity: O(n)



Example

```
def linear_search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1

print(linear_search([5,10,15,20], 15))
```

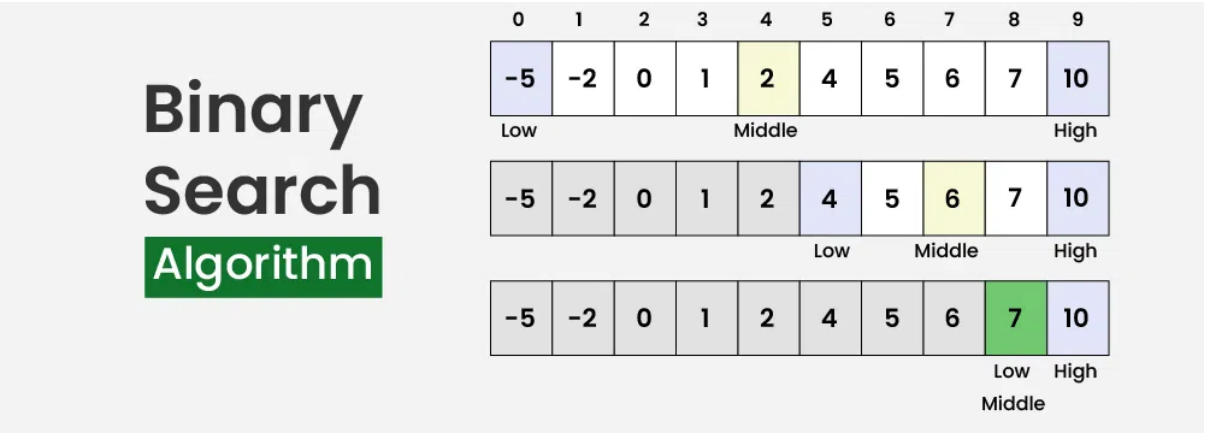
Output:

2

B) Binary Search

- Works only on a **sorted list**.
- Use **low**, **high**, and **mid** indexes.
- Compare target with **list[mid]**.
- If target < mid → move **left**.
- If target > mid → move **right**.
- Repeat until found or low > high.

Time Complexity: O(log n)



Example

```
def binary_search(arr, x):
    low, high = 0, len(arr)-1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
    return -1

print(binary_search([10,20,30,40,50], 30))
```

Output:

2

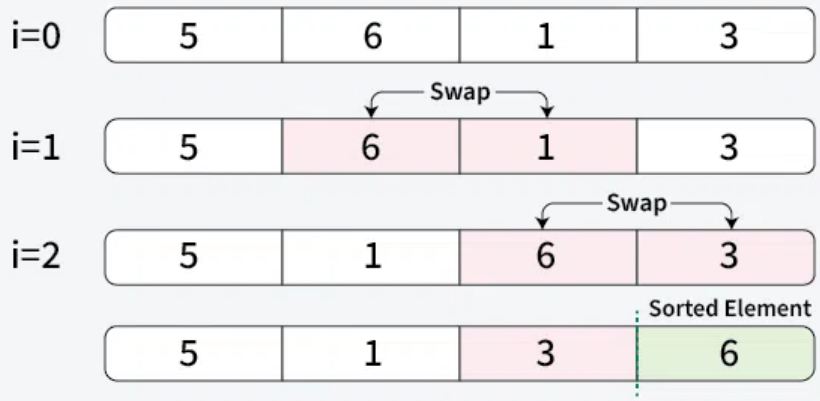
1.2 SORTING ALGORITHMS

A) Bubble Sort

- Repeatedly compares **adjacent elements**.
- **Swaps** them if they are in the wrong order.
- Largest element "**bubbles up**" to the end in each pass.
- Process repeats for remaining unsorted part.
- Works on arrays/lists of any type.
- Simple but **slow for large data**.

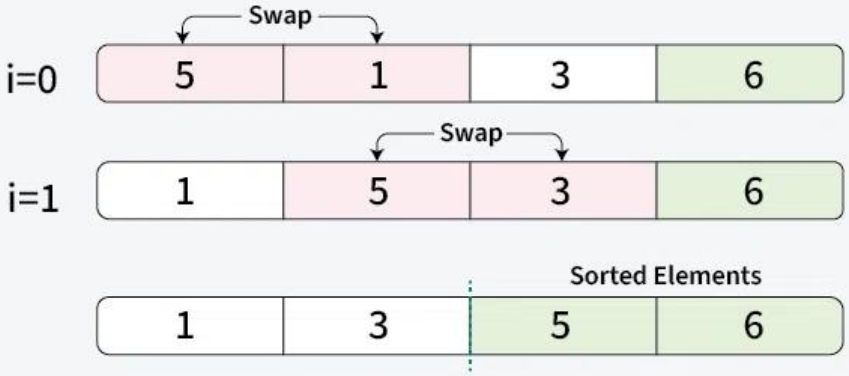
Time: $O(n^2)$

01 Step | Placing the 1st largest element at its correct position



Bubble sort

02 Step | Placing 2nd largest element at its correct position



Bubble sort

03 Step | Placing 3rd largest element at its correct position

Bubble sort

Example

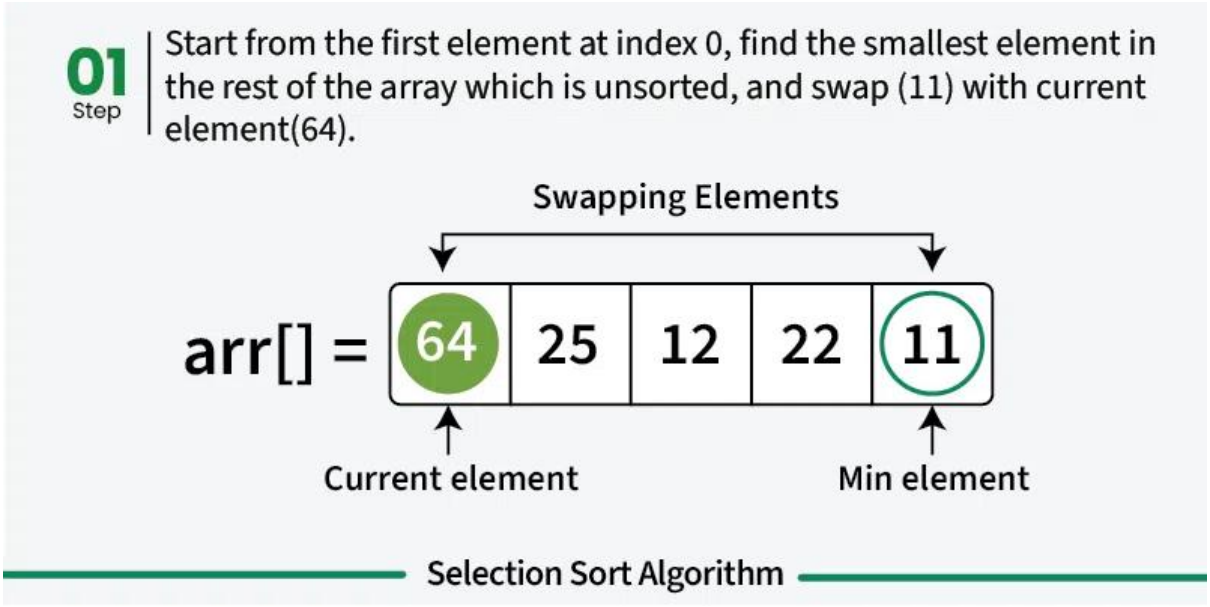
```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n-1):
        for j in range(n-1-i):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

arr = [5,3,8,2]
bubble_sort(arr)
print(arr)
```

Output:
[2, 3, 5, 8]

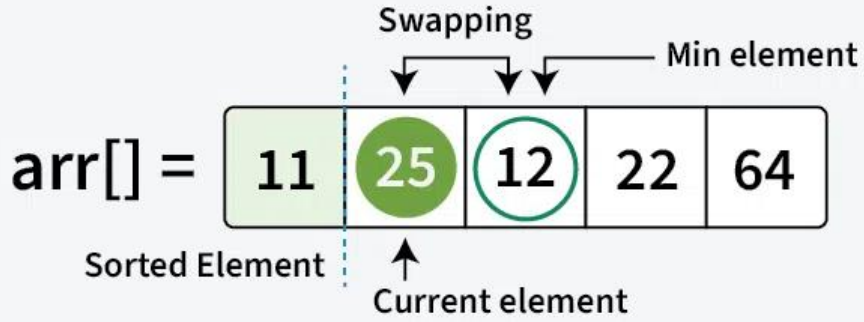
B) Selection Sort

- Divides list into **sorted** and **unsorted** parts.
- Finds the **minimum element** from the unsorted part.
- Swaps it with the **first unsorted position**.
- Moves boundary of sorted part **one step right**.
- Repeats until entire list is sorted.
- Simple but **slow for large lists**.

Time: $O(n^2)$

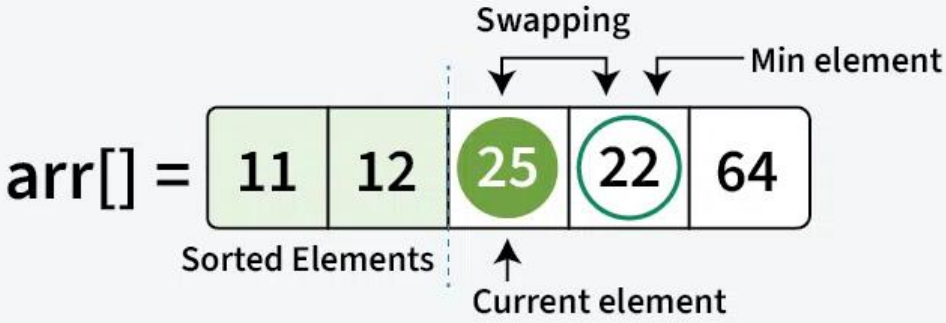


02 Step | Move to the next element at index 1 (25). Find the smallest in unsorted subarray, and swap (12) with current element (25).



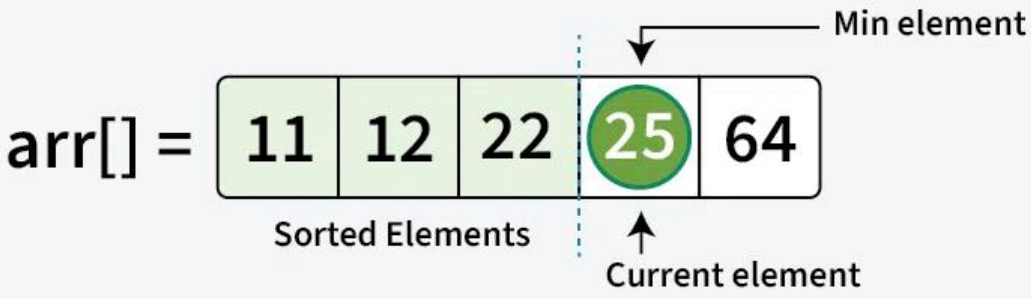
Selection Sort Algorithm

03 Step | Move to element at index 2 (25). Find the minimum element from unsorted subarray, Swap (22) with current element (25).

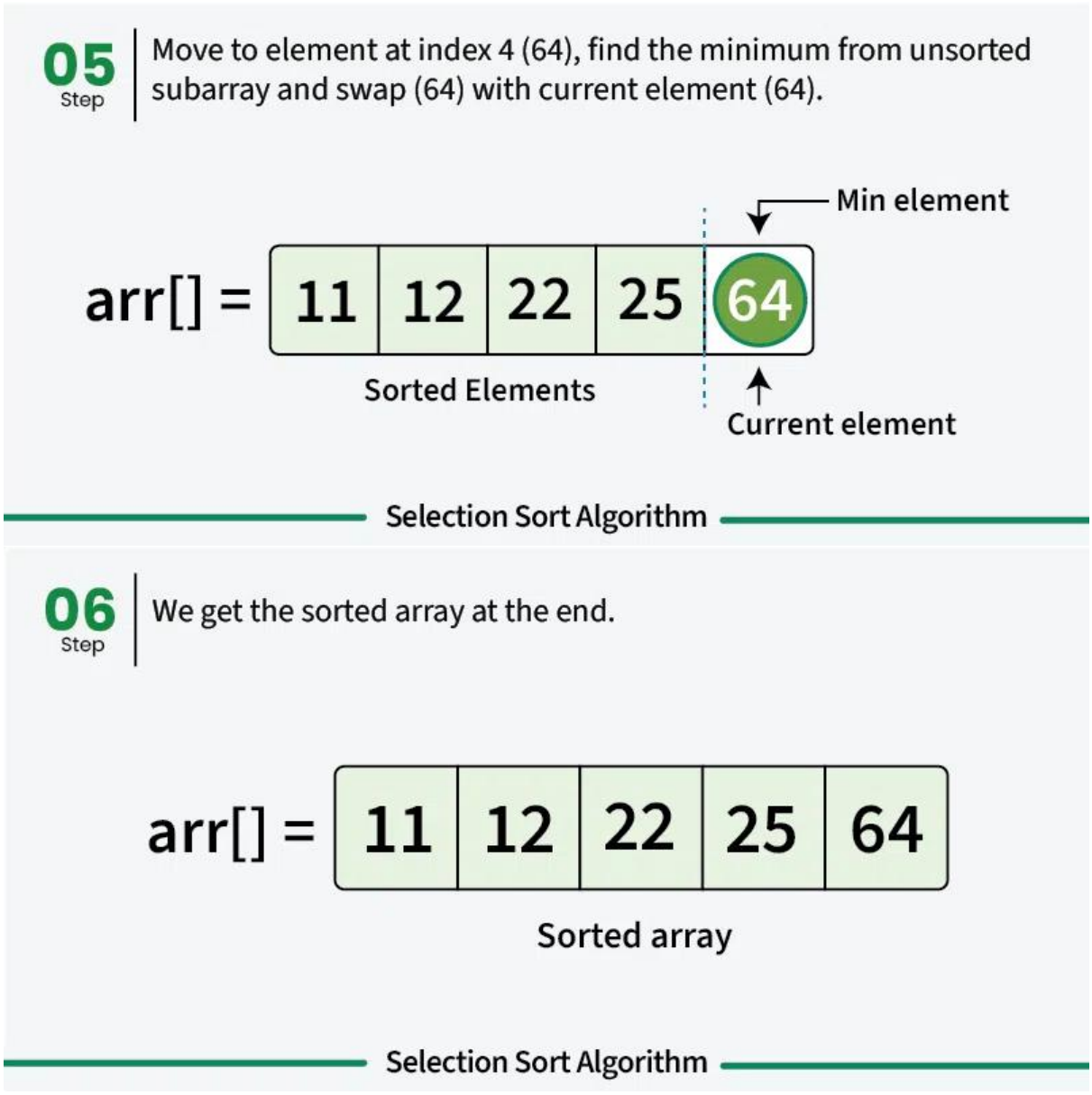


Selection Sort Algorithm

04 Step | Move to element at index 3 (25), find the minimum from unsorted subarray and swap (25) with current element (25).



Selection Sort Algorithm



Example

```
def selection_sort(arr):
    for i in range(len(arr)):
        min_i = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_i]:
                min_i = j
        arr[i], arr[min_i] = arr[min_i], arr[i]

arr = [29,10,14,37]
selection_sort(arr)
print(arr)
```

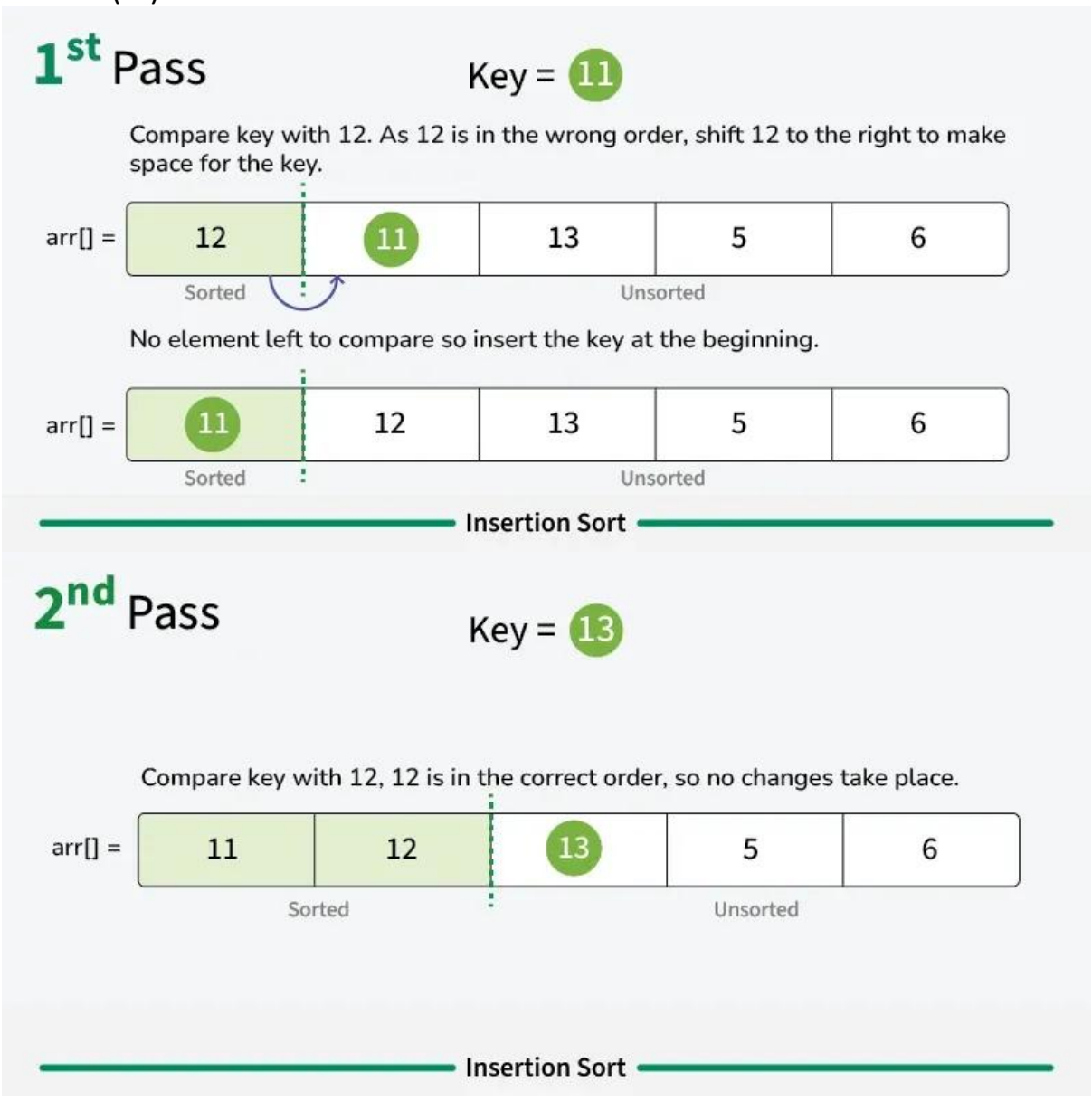
Output:

[10, 14, 29, 37]

C) Insertion Sort

- Builds the sorted list **one element at a time**.
- Takes an element and **inserts** it into its correct position in the sorted part.
- Shifts larger elements **one position right** to make space.
- Works well for **small or nearly sorted** lists.
- Sorting happens **in-place** (no extra memory).
- Simple and easy to implement.

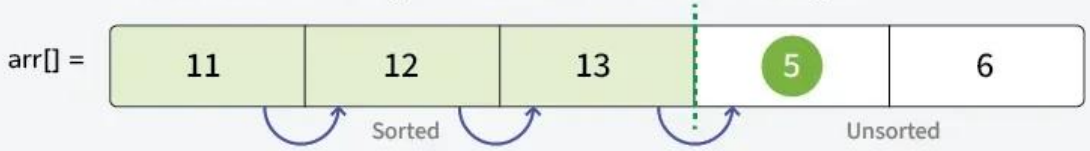
Time: $O(n^2)$



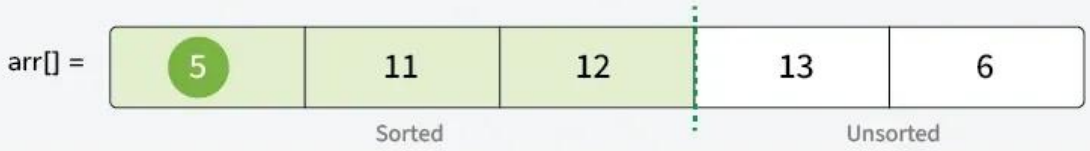
3rd Pass

Key = 5

Compare key with all the elements in the sorted subarray starting with 13, if the element is in the wrong order, shift that element to the right.



No element left to compare, so insert key at the beginning.

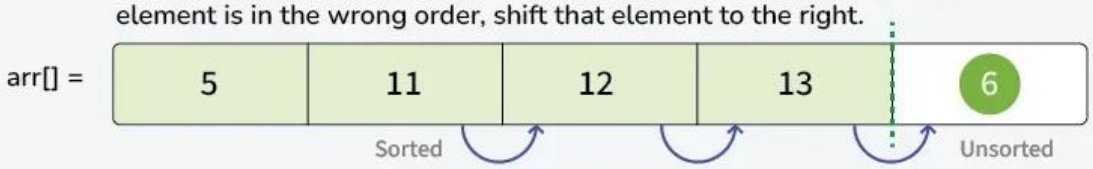


Insertion Sort

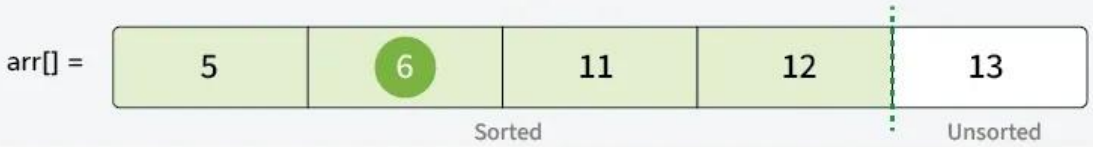
4th Pass

Key = 6

Compare key with all the elements in the sorted subarray starting with 13, if the element is in the wrong order, shift that element to the right.



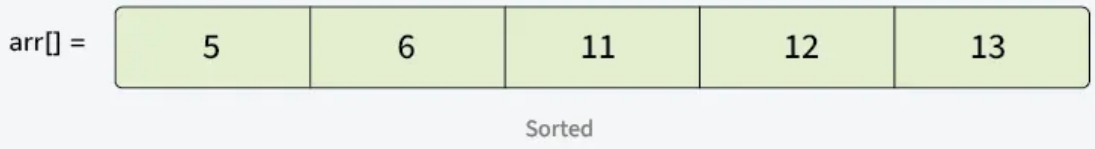
5 is in the correct order, so shifting stops. Insert key after 5.



Insertion Sort

Sorted Array

The sorted part contains the whole array. Means that the whole array is sorted.



Insertion Sort

Example

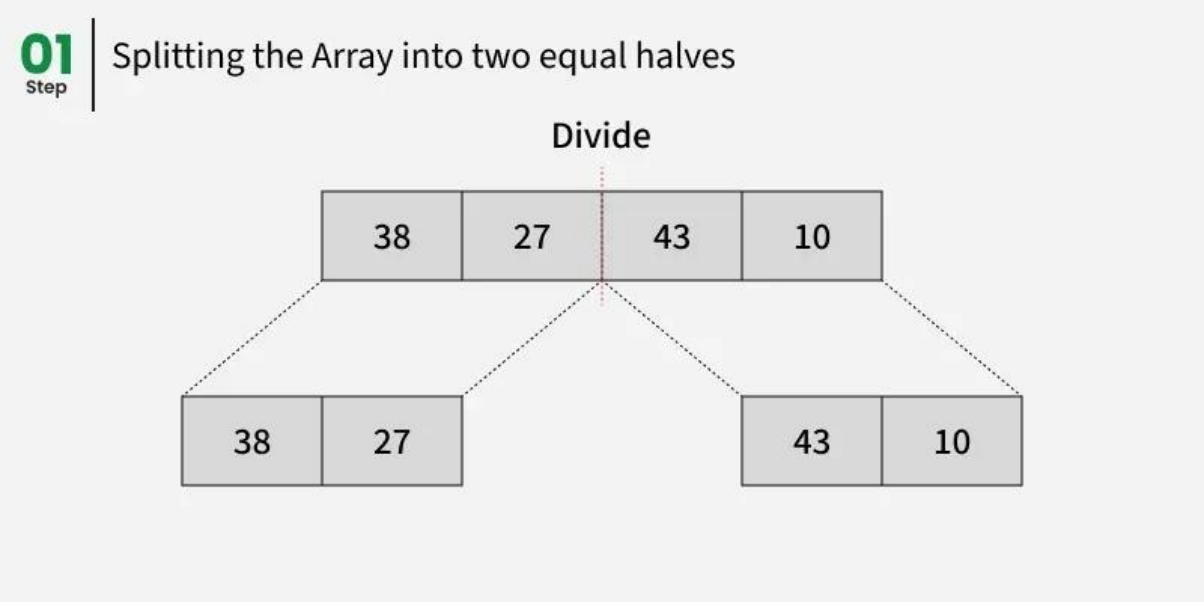
```
def selection_sort(arr):
    for i in range(len(arr)):
        min_i = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_i]:
                min_i = j
        arr[i], arr[min_i] = arr[min_i], arr[i]

arr = [29,10,14,37]
selection_sort(arr)
print(arr)
```

Output:
[1, 4, 5, 9]

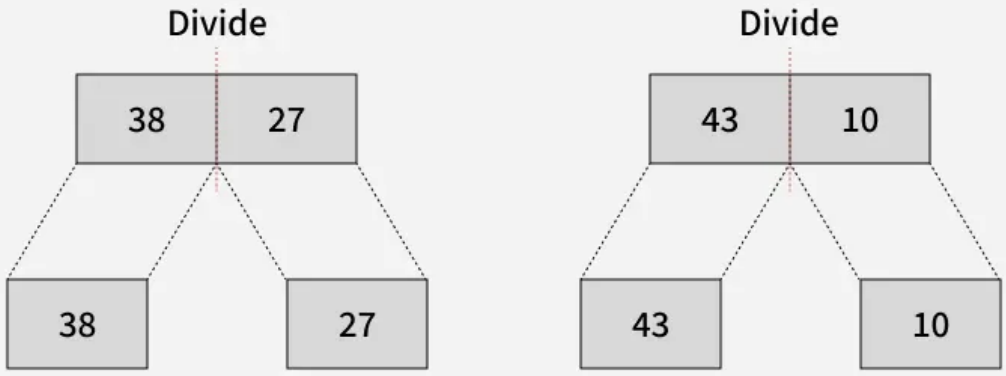
D) Merge Sort

- Uses **Divide and Conquer** technique.
- Splits the list into **two halves** repeatedly.
- Sorts each half **recursively**.
- **Merges** the two sorted halves into one sorted list.
- Not in-place (needs **extra memory**).
- Very efficient for large datasets.

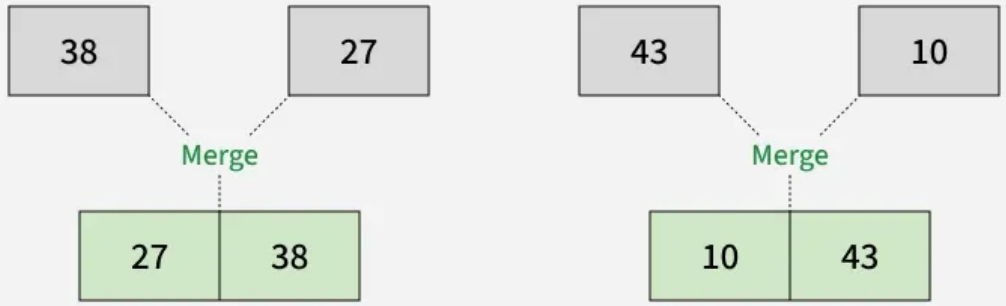
Time: O(n log n)



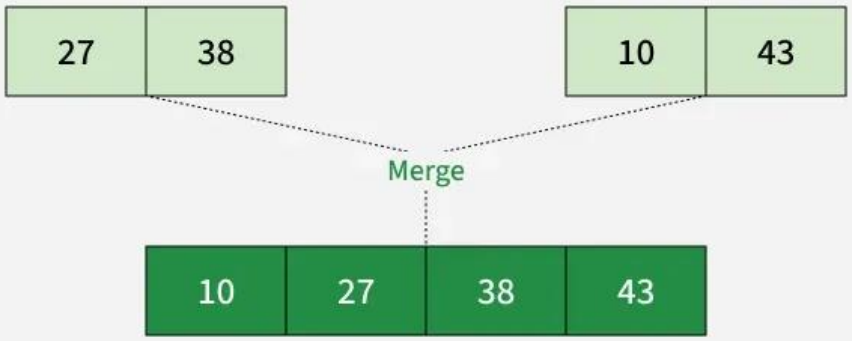
02 Step | Splitting the subarrays into two halves



03 Step | Merging unit length cells into sorted subarrays



04 Step | Merging sorted subarrays into the sorted array



Example

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result

print(merge_sort([8,3,1,7,0]))
```

Output:
[0, 1, 3, 7, 8]

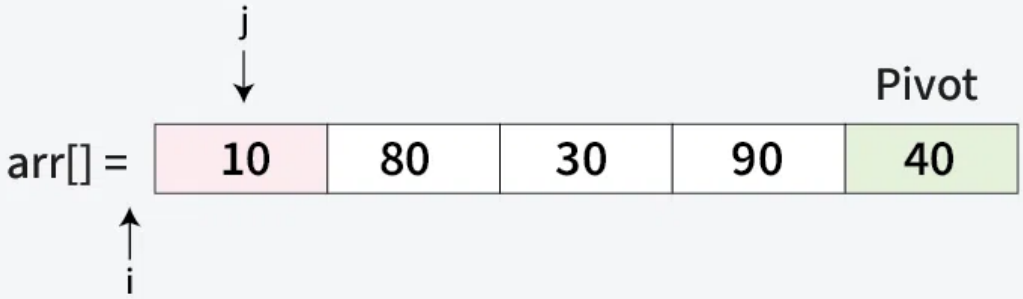
E) Quick Sort

- Uses **Divide and Conquer** technique.
- Selects a **pivot element** from the list.
- Partitions list into **elements smaller** and **larger** than pivot.
- Recursively applies quick sort to both partitions.
- Sorting happens **in-place** (usually).
- Very fast in practice, but worst-case slow if pivot chosen badly.

Time:

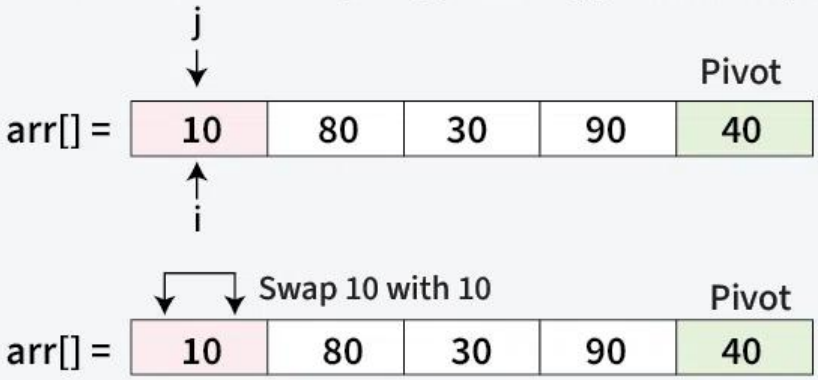
- Best: O(n log n)
- Worst: $O(n^2)$

01 Step | Pivot Selection: The last element arr[4] = 40 is chosen as the pivot.
Initial Pointers: i = -1 and j = 0.



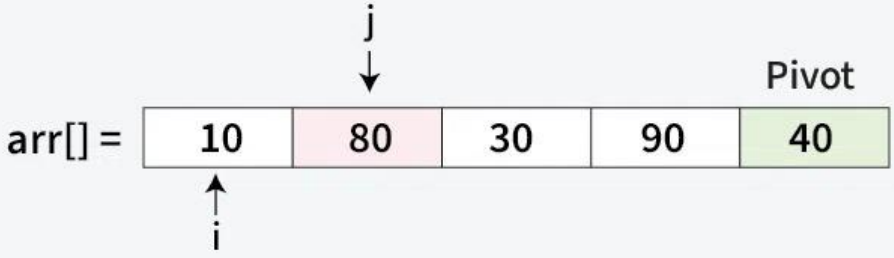
Quick sort

02 Step | Since, arr[j] < pivot (10<40)
Increment i to 0 and swap arr[i] with arr[j]. Increment j by 1



Quick sort

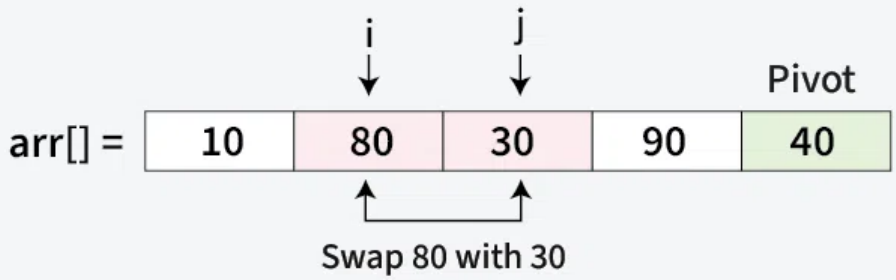
03 Step | Since, arr[j] > pivot (80>40)
No swap needed. Increment j by 1



Quick sort

04 Step

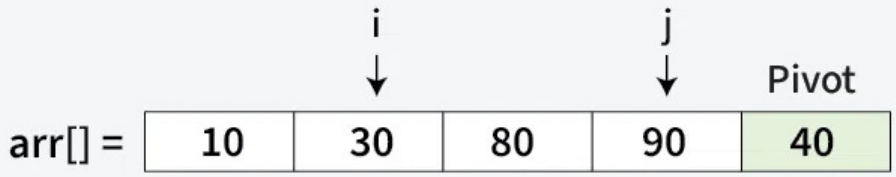
Since, arr[j] < pivot (30<40)
Increment i by 1 and swap arr[i] with arr[j]. Increment j by 1



Quick sort

05 Step

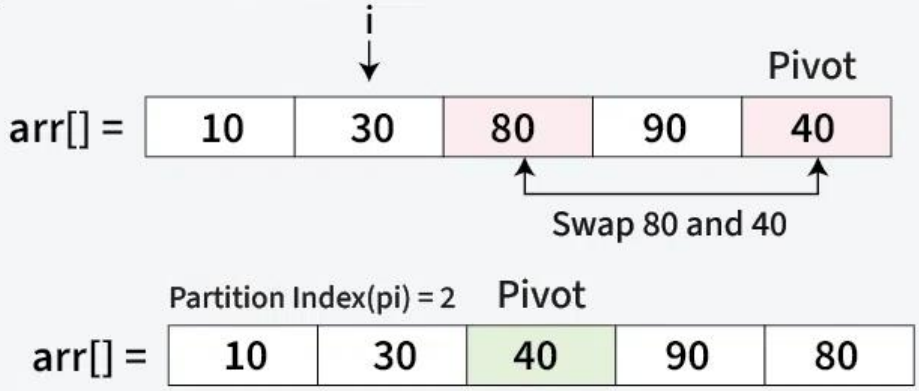
Since, arr[j] > pivot (90>40)
No swap needed. Increment j by 1



Quick sort

06 Step

Since traversal of j has ended. Now move pivot to its correct position, Swap arr[i + 1] = arr[2] with arr[4] = 40.



Quick sort

Example

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[0]
    left = [x for x in arr[1:] if x < pivot]
    right = [x for x in arr[1:] if x >= pivot]
    return quick_sort(left) + [pivot] + quick_sort(right)

print(quick_sort([4,1,3,9,7]))
```

Output:

[1, 3, 4, 7, 9]

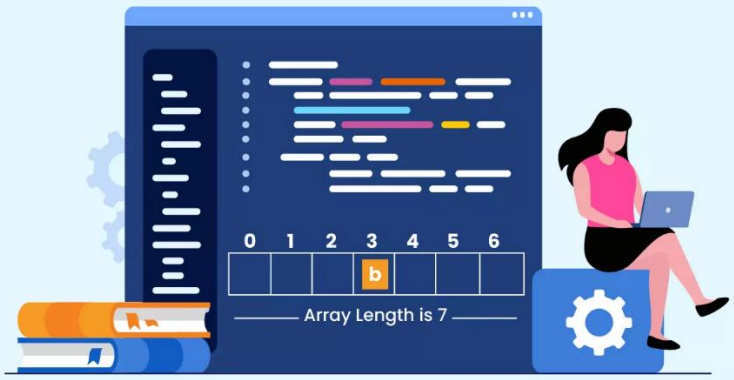
---

4. TIME COMPLEXITY (BIG-O)

Algorithm	**Best**	**Worst**
Linear Search	O(1)	O(n)
Binary Search	O(1)	O(log n)
Bubble Sort	O(n)	$O(n^2)$
Merge Sort	O(n log n)	O(n log n)
Quick Sort	O(n log n)	$O(n^2)$

Python Array

Upgrade your Python skills by learning about Python Arrays for efficient data handling.



PYTHON ARRAY

1. Introduction to Arrays

1.1 Definition

- An array is a data structure that stores **multiple values of the same data type** in a single variable.
- Python does not have a built-in array like C/C++; it provides arrays using the **array module**.
- Arrays are more memory-efficient than lists when storing large numbers of numeric values.

1.2 Why Use Arrays?

- To store large numeric data in less memory.
- Faster processing compared to lists for numeric operations.
- Useful in scientific, mathematical, and data-processing applications.

---

2. Creating Arrays in Python

2.1 Importing the Array Module

```
import array
```

2.2 Syntax to Create an Array

```
array.array(typecode, [elements])
```

2.3 Typecodes (Common Ones)

- 'i' → integer
- 'f' → float
- 'd' → double
- 'u' → Unicode character

---

3. Basic Array Operations

3.1 Accessing Elements

- Use index positions

```
a[0]
```

3.2 Updating Elements

```
a[2] = 50
```

3.3 Traversing an Array

```
for x in a:
    print(x)
```

---

4. Important Array Methods

4.1 append()

- Adds an element at the end

```
a.append(40)
```

4.2 insert()

- Inserts element at a specific index

```
a.insert(1, 20)
```

4.3 remove()

- Removes first occurrence of an element

```
a.remove(20)
```

4.4 pop()

- Removes element at an index (default: last)

```
a.pop()
```

4.5 index()

- Returns the index of an element

```
a.index(10)
```

4.6 reverse()

- Reverses the array

```
a.reverse()
```

---

5. Example Program

```
import array

a = array.array('i', [10, 20, 30, 40])

print("Array elements:")
for x in a:
    print(x)

a.append(50)
print("After append:", a)
```

---

6. Advantages of Arrays

- More memory-efficient for numeric data.
- Faster performance than lists for mathematical operations.
- Stores homogeneous data (same type).

---

7. Limitations of Arrays

- Can store **only one data type**.
- Not as flexible as lists.
- Need to import the array module.