Computer Programming I Lecture Notes

Piya Limcharoen

Version October 23, 2025at 7:54am

Contents

Copyright Notice

This lecture note was authored by Piya Limcharoen. Artificial Intelligence (AI) was utilized as a tool for language enhancement and grammatical correction; however, all intellectual property, original concepts, and the structure of this work belong exclusively to the author.

Terms of Use

This document is provided for educational purposes only and is intended for the exclusive use of students currently enrolled in this course.

The reproduction, distribution, public display, or transmission of this material, in whole or in part, in any form or by any means—electronic, mechanical, photocopying, recording, or otherwise—is strictly prohibited without the prior written permission of the author. Students may not share, upload, or distribute these notes outside of the official class environment.

Warning

Any unauthorized use of this material will be considered academic misconduct and will be subject to university disciplinary procedures. Furthermore, such an act may constitute an infringement of copyright law and is subject to legal action.

Topic 1:

File Handling for Program Persistence

Topic 1: File Handling for Program Persistence

1.1 Transient vs. Persistent Programs

Most of the programs we have encountered so far are **transient**[1]. A transient program executes for a limited duration and typically produces some output during its execution, but once it terminates, all data stored in memory is lost. If the program is run again, it starts from an initial (empty) state as if it had never been executed before.

Example: A simple Python script that takes two numbers as input, adds them, prints the result, and then exits is a transient program. When the script finishes, the input values and the computed result are not stored anywhere for later use.

In contrast, **persistent**[2] programs are designed to preserve data beyond the duration of their execution. Such programs typically store information in permanent storage devices (like hard drives or solid-state drives) and can restore their previous state when restarted.

Example: An operating system is a persistent program—it runs continuously while the computer is on, managing resources and maintaining logs. Similarly, a web server (such as Apache or Nginx) is persistent, running indefinitely and storing access logs and configuration data to ensure continuity even after restarts.

[1] *Transient* means temporary or short-lived; something that lasts only for a brief period of time.

[2] *Persistent* means continuing to exist or remain active for a prolonged time; in computing, it refers to data or programs that maintain their state between executions.

Persistence in programs is often achieved through **file handling**, where data is written to and read from files stored on disk. Files serve as a simple yet powerful mechanism for data storage.

1.2 Text and Binary Files

Files are generally categorized into two types: **text files** and **binary files**.

- **Text files** store data in human-readable form, using standard character encodings such as ASCII or UTF-8. Examples include `.txt`, `.csv`, and `.py` files. Text files are convenient for simple data storage but may be inefficient for complex data structures.
- **Binary files**, on the other hand, store data in a format readable only by programs. They can represent any kind of information—images, videos, compiled code, or serialized objects—without converting it to text. Examples include `.jpg`, `.exe`, and `.dat` files. Binary storage is more compact and faster for reading/writing large or structured data.

In Python, data persistence can be achieved in several ways, depending on the format of the stored data.

- For **text-based persistence**, programs can store data in human-readable files such as plain text (`.txt`), comma-separated values (`.csv`), or JavaScript Object Notation (`.json`) files. These formats are easy to inspect and edit manually. For instance, Python's built-in `open()` function can be used to read and write text files, and modules like `csv` and `json` simplify working with structured data.

- For **binary persistence**, Python provides the `pickle` module, which allows programs to serialize (convert) complex Python objects—such as lists, dictionaries, or custom classes—into a binary format that can be written to a file. The same module can later deserialize (restore) the objects from the file back into memory. This mechanism enables programs to save their internal state and resume from that state later, effectively bridging the gap between transient and persistent behavior.

1.3 What is a File?

In computing, a **file** is a named collection of related data stored on a secondary storage device such as a hard drive, SSD, or USB drive. Files serve as the primary means of storing and retrieving data outside a program's runtime memory, allowing information to persist even after the program or computer is shut down.

Files can contain any form of data—text, numbers, images, audio, video, or program instructions—and are organized and managed by the operating system through a **file system**. Each file is identified by a unique name and often an extension (e.g., `.txt`, `.csv`, `.exe`) that indicates its type or intended use.

The process of interacting with files is known as **File I/O (Input/Output)**:

- **Input (I)** refers to reading data from an external source into a program. For example, loading configuration data from a text file or importing saved game progress.
- **Output (O)** refers to writing data from a program to an external destination, such as saving a report, writing logs, or exporting user preferences.

File I/O enables communication between a program's transient memory (RAM) and persistent storage (disk), making it essential for data retention and long-term storage.

A typical file-handling operation follows three main steps:

1. **Opening** a file — establishes a connection between the program and the file. In Python, this is done using the `open()` function, specifying both the file name and the mode (e.g., read, write, or append).
2. **Reading from or Writing to** the file — depending on the mode, the program either retrieves existing data (input) or sends new data to the file (output). Python provides methods such as `read()`, `readline()`, and `write()` for these operations.
3. **Closing** the file — releases system resources and ensures that all written data is properly saved. In Python, this is typically done using the `close()` method or automatically managed with a `with` statement for safer handling.

Understanding file operations and I/O is fundamental to developing persistent programs, as it allows data to exist beyond a single program's execution and supports long-term information management.

1.4 Working with Filenames and Paths

Correctly managing file paths is crucial for writing robust programs that work across different operating systems (Windows, macOS, Linux). Improper handling of paths can lead to errors such as "file not found" or "invalid path." Python's `pathlib` module provides an object-oriented and platform-independent way to work with file paths safely and efficiently.

1.4.1 Absolute vs. Relative Paths

- **Absolute Path**: A full path that specifies a file's location from the root of the file system. It uniquely identifies the file's position in the directory structure.
  - Windows: `C:\Users\Ku\Documents\report.txt`
  - macOS/Linux: `/home/ku/documents/report.txt`
- **Relative Path**: A path that specifies a file's location relative to the current working directory (CWD). It is shorter and more portable but depends on where the program is executed.
  - Example: `data/sales.csv` or `../images/logo.png`

1.4.2 Path Handling in Python

1.4.2.1 Getting the Current Working and Home Directories

```python
from pathlib import Path

# Get the Current Working Directory (CWD)
cwd = Path.cwd()
print(f"Current Working Directory: {cwd}")

# Get the user's home directory
home_dir = Path.home()
print(f"Home Directory: {home_dir}")
```

Sample Output (on macOS):

```
Current Working Directory: /Users/youruser/projects/my_app
Home Directory: /Users/youruser
```

1.4.2.2 Creating Platform-Independent Paths

The **pathlib** module lets you build paths using the **/** operator, which automatically applies the correct separator for the host operating system ("\" on Windows, "/" on Unix-like systems).

```python
from pathlib import Path

# Build a path to a file in a subdirectory of the CWD
data_file_path = Path.cwd() / "data" / "users.csv"
print(f"Constructed Path: {data_file_path}")

# You can also join Path objects with other Path objects or strings
reports_dir = Path("reports")
monthly_report = reports_dir / "january" / "sales.txt"
print(f"Report Path: {monthly_report}")
```

Note: The `/` operator only works with `Path` objects. Attempting to use it with normal strings causes a `TypeError`, as shown below.

```
>>> 'spam' / 'bacon'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'str'
>>>
```

Explanation:

Strings in Python do not support the division operator (`/`). The `Path` class overloads this operator to simplify path concatenation in a way that is operating-system–independent.

Using plain strings to create paths manually (e.g., `"C:\folder\file.txt"`) is error-prone and non-portable, because separators differ between operating systems. Always use `pathlib.Path()` to ensure compatibility and safety.

1.4.2.3 Deconstructing Path Components

Once you have a `Path` object, you can easily extract its individual parts. This is useful for getting a filename, its extension, or its parent folder without manual string manipulation.

```python
from pathlib import Path

p = Path.cwd() / "data" / "users.csv"

print(f"Original Path: {p}")
print(f"Parent Directory: {p.parent}")
print(f"File Name: {p.name}")
print(f"File Stem (name without extension): {p.stem}")
print(f"File Suffix (extension): {p.suffix}")
```

Sample Output:

```
Original Path: /Users/youruser/projects/my_app/data/users.csv
Parent Directory: /Users/youruser/projects/my_app/data
File Name: users.csv
File Stem (name without extension): users
File Suffix (extension): .csv
```

1.4.2.4 Checking Path Validity

Before reading or writing a file, it's good practice to check whether the path exists and whether it refers to a file or directory.

```python
from pathlib import Path

path = Path("data/users.csv")

# Check if the path exists
print(f"Does '{path}' exist? {path.exists()}")

# Check if it's a file
print(f"Is '{path}' a file? {path.is_file()}")

# Check if it's a directory
print(f"Is '{path}' a directory? {path.is_dir()}")
```

1.4.2.5 Creating Directories

Often, you need to ensure a directory exists before you can write a file into it. `Path.mkdir()` is the modern way to create directories.

```python
from pathlib import Path

# Define a path for a new directory to store output files
output_dir = Path.cwd() / "output"
print(f"Attempting to create directory: {output_dir}")

# Create the directory
# parents=True: Creates parent directories if they don't exist.
# exist_ok=True: Does not raise an error if the directory already exists.
output_dir.mkdir(parents=True, exist_ok=True)

print(f"Does '{output_dir}' now exist? {output_dir.exists()}")
```

Explanation: Using `mkdir(parents=True, exist_ok=True)` is a robust pattern that safely creates a directory structure without crashing if the script is run more than once.

1.4.2.6 Listing Directory Contents

While **`pathlib`** is ideal for constructing and managing paths, changing the current working directory (CWD) is performed using the **`os`** module. After changing the directory, you can use **`Path.iterdir()`** to list all files and subdirectories in the new location.

```
import os
from pathlib import Path

print(f"Original CWD: {Path.cwd()}")

# Define the target directory path
target_dir = Path.home() / "Documents"

# Change the current working directory if it exists
if target_dir.is_dir():
    os.chdir(target_dir)
    print(f"New CWD: {Path.cwd()}")

    # List contents of the new current directory
    print("\nContents of the new directory:")
    for item in Path.cwd().iterdir():
        if item.is_dir():
            print(f"[DIR]  {item.name}")
        else:
            print(f"       {item.name}")
```

```
21 else:
22     print(f"Directory '{target_dir}' does not exist.")
```

1.5 Text Files vs. Binary Files

Files on a computer are stored as sequences of bytes. How a program interprets these bytes depends on the file type.

- **Text Files (.txt, .py, .csv, .html)**: The bytes in the file represent characters, following an encoding scheme like UTF-8 or ASCII. When you open a text file in Python (e.g., with mode `'r'`), Python automatically decodes the bytes into strings for you. These files are human-readable.
- **Binary Files (.jpg, .mp3, .exe, .pkl)**: The bytes represent data that is not plain text, such as image pixels, audio samples, or serialized Python objects. To work with these files, you must open them in a binary mode (e.g., `'rb'` or `'wb'`), which gives you direct access to the raw bytes without any decoding.

1.5.1 Expanding on File Modes

When you open a file with `open()`, the **mode** string specifies your intent—reading, writing, or both—and whether you are handling text or binary data. Using the correct mode is crucial for preventing errors and data corruption.

The mode can be a combination of one character from the first group and one from the second:

1. **Operation Type**: `'r'` (read), `'w'` (write), `'a'` (append), `'x'` (exclusive creation)
2. **Data Type**: `'t'` (text, default), `'b'` (binary)
3. **Modifier**: `'+'` (update - read and write)

Key Points:

- **Truncation:** Be careful with `'w'` and `'w+'` modes, as they will **erase all existing content** in a file upon opening.
- **Binary Mode (`b`):** This is essential for non-text files. Use it for images (`.jpg`), executables (`.exe`), and serialized data from `pickle` (`.pkl`). Attempting to open these files in text mode will result in errors or corrupted data.
- **Exclusive Creation (`x`):** This mode is a safe way to create a new file without accidentally overwriting an existing one.

Here is a comprehensive table of common file modes:

Mode	Description	Behavior If File Exists	Behavior If File Doesn't Exist
`'r'`	Read Text (Default)	Reads from the start.	Raises `FileNotFoundError`.
`'w'`	Write Text	**Truncates** (empties) the file.	**Creates** a new file.
`'a'`	Append Text	Appends to the end of the file.	**Creates** a new file.
`'x'`	Exclusive Creation	Raises `FileExistsError`.	**Creates** a new file.
`'rb'`	Read Binary	Reads from the start.	Raises `FileNotFoundError`.
`'wb'`	Write Binary	**Truncates** the file.	**Creates** a new file.
`'r+'`	Read and Write	Pointer is at the start.	Raises `FileNotFoundError`.
`'w+'`	Write and Read	**Truncates** the file first.	**Creates** a new file.

1.6 Reading from Text Files

1.6.1 Reading Files Using `open()` and `close()`

In Python, files can be opened and closed manually using the built-in **`open()`** and **`close()`** functions. Python's file handling model, using **`open()`** and **`close()`**, is conceptually similar to the **`fopen()`** and **`fclose()`** functions found in the C programming language. However, Python offers a more modern and safer approach with the with statement.

```
# Open a file for reading (equivalent to fopen in C)
f = open('poem.txt', 'r', encoding='utf-8')

# Read the entire content
content = f.read()
print(content)

# Close the file to free system resources (like fclose in C)
f.close()
```

Output:

```
Roses are red,
Violets are blue.
```

Explanation:

- `open()` returns a `file object`, which acts as a handle to the file.
- The second argument specifies the mode:
  - `'r'` – read (file must exist)
  - `'w'` – write (creates or overwrites a file)
  - `'a'` – append (writes at the end of the file)
- `f.close()` releases system resources and ensures that all data is saved.

Important: If you forget to call `close()`, the file remains open until Python's garbage collector closes it automatically. This may cause memory leaks, file locks, or incomplete writes.

Note on encoding: If you omit the `encoding` parameter (e.g., `open('poem.txt', 'r')`), Python uses the system default encoding, which varies by platform:

- On Windows, the default is often `'cp1252'`.
- On macOS and Linux, the default is usually `'utf-8'`.

Using the default may cause errors when reading files created on a different system or containing special characters. Therefore, it is best practice to explicitly specify `encoding='utf-8'` or another appropriate encoding.

1.6.2 Using `with open()` — Context Manager Style

A safer and more modern approach is to use a **`with`** statement, which automatically closes the file after the block ends, even if an error occurs.

```
1 with open('poem.txt', 'r', encoding='utf-8') as f:
2     content = f.read()
3     print(content)
4 # File is automatically closed when exiting the 'with' block
```

Advantages:

- Eliminates the need to call **`close()`** manually.
- Prevents data loss and resource leaks if an exception occurs.
- Makes the code shorter, cleaner, and more Pythonic.

1.6.3 Reading Entire File or Line by Line

Assume a file `poem.txt` contains:

```
Roses are red,
Violets are blue.
```

```
# Reading the entire file into one string
with open('poem.txt', 'r', encoding='utf-8') as f:
    content = f.read()
    # content is 'Roses are red,\nViolets are blue.'

# Reading the file line by line into a list
with open('poem.txt', 'r', encoding='utf-8') as f:
    lines = f.readlines()
    # lines is ['Roses are red,\n', 'Violets are blue.']
```

1.6.4 Parsing Lines with `split()`

Each line in a text file may contain multiple pieces of data. Use `.strip()` to remove whitespace (like `\n`) and `.split()` to separate the line into a list of strings.

```
data_line = "item:price:quantity\n"
parts = data_line.strip().split(':')
# parts is now ['item', 'price', 'quantity']
```

1.6.5 Reading and Nested Splitting

Some files contain multiple records per line, requiring nested splitting. For example, `inventory.txt` contains:

```
apple,1.20,10;banana,0.80,5;cherry,2.50,12
```

Each product is separated by a semicolon (`;`) and each field by a comma (`,`).

```
with open('inventory.txt', 'r', encoding='utf-8') as f:
    line = f.readline().strip()
    # Split line into product entries
    products = line.split(';')
    for product in products:
        name, price, qty = product.split(',')
        print(f"Name: {name}, Price: {price}, Quantity: {qty}")
```

Sample Output:

```
Name: apple, Price: 1.20, Quantity: 10
Name: banana, Price: 0.80, Quantity: 5
Name: cherry, Price: 2.50, Quantity: 12
```

Explanation: This is called **nested splitting**: first split by records (`;`), then by fields (`,`). It is commonly used for structured text data.

1.6.6 Reading Structured Text Files (CSV/TSV)

CSV (Comma-Separated Values) and TSV (Tab-Separated Values) are widely used formats for tabular data. While you can parse them manually with `split()`, Python's built-in `csv` module provides a safer and more robust approach, handling quotes, delimiters, and special characters automatically.

Example CSV File: `students.csv`

```
Name,Major,GPA,GraduationYear
Alice,Computer Science,3.9,2024
Bob,Physics,3.5,2023
Charlie,Mathematics,3.8,2025
```

```python
import csv

# Reading CSV file
with open('students.csv', 'r', newline='', encoding='utf-8') as f:
    reader = csv.reader(f)

    # Read header
    header = next(reader)
    print(f"Header: {header}")

    # Read each row
    for row in reader:
        print(f"Name: {row[0]}, Major: {row[1]}, GPA: {row[2]}, Graduation Year: {row
          [3]}")
```

Sample Output:

```
Header: ['Name', 'Major', 'GPA', 'GraduationYear']
Name: Alice, Major: Computer Science, GPA: 3.9, Graduation Year: 2024
Name: Bob, Major: Physics, GPA: 3.5, Graduation Year: 2023
Name: Charlie, Major: Mathematics, GPA: 3.8, Graduation Year: 2025
```

Example TSV File: employees.tsv

```
ID  Name    Department  Salary
101 John    Accounting  55000
102 Mary    HR  60000
103 Peter   IT  70000
```

```python
import csv

# Reading TSV file (tab-delimited)
with open('employees.tsv', 'r', newline='', encoding='utf-8') as f:
    reader = csv.reader(f, delimiter='\t')

    # Read header
    header = next(reader)
    print(f"Header: {header}")

    # Read each row
    for row in reader:
        print(f"ID: {row[0]}, Name: {row[1]}, Department: {row[2]}, Salary: {row[3]}")
```

Sample Output:

```
Header: ['ID', 'Name', 'Department', 'Salary']
ID: 101, Name: John, Department: Accounting, Salary: 55000
ID: 102, Name: Mary, Department: HR, Salary: 60000
ID: 103, Name: Peter, Department: IT, Salary: 70000
```

Notes:

- For TSV files, set `delimiter='\t'` when creating the reader object.
- Always specify `encoding='utf-8'` to ensure cross-platform compatibility and avoid `UnicodeDecodeError`.
- The `csv` module automatically handles quoted fields and embedded delimiters, which makes it safer than manually splitting lines.

1.7 Writing to Text Files

To write to a text file in Python, you can use mode `'w'` to create or overwrite a file, or `'a'` to append to it. Files can be handled either manually with `open()`/`close()` or using the `with` statement for automatic closure.

1.7.1 File Object Attributes

When a file is opened, Python returns a `file object` with useful attributes:

- `name` – the name of the file.
- `mode` – the mode in which the file was opened (`'r'`, `'w'`, `'a'`, etc.).
- `closed` – Boolean, indicates whether the file is closed.

```
1 f = open('report.txt', 'w')
2 print(f"File Name: {f.name}")
3 print(f"File Mode: {f.mode}")
4 print(f"Is Closed? {f.closed}")
5
6 f.write("Hello, world!\n")
7 f.write("Hello, KU!\n")
8 f.close()
9 print(f"Is Closed after close()? {f.closed}")
```

Output:

```
File Name: report.txt
File Mode: w
Is Closed? False
Is Closed after close()? True
```

Contents of `report.txt`

```
Hello, world!
Hello, KU!
```

1.7.2 Manual Writing with `open()` and `close()`

You can write to a file manually, similar to C-style `fopen`/`fclose`:

```
# Open a file for writing
f = open('report.txt', 'w', encoding='utf-8')
f.write("This is the first line.\n")
f.write("This line is written manually.\n")
f.close()
```

Contents of `report.txt`

```
This is the first line.
This line is written manually.
```

Note: Specifying `encoding='utf-8'` ensures correct handling of Unicode characters.

1.7.3 Writing from a List with `writelines()`

The **`writelines()`** method writes all strings from a list to a file. You must include newline characters yourself.

```
lines_to_write = ["First item\n", "Second item\n", "Third item\n"]
with open('items.txt', 'w', encoding='utf-8') as f:
    f.writelines(lines_to_write)
```

Contents of `items.txt`

```
First item
Second item
Third item
```

Note: Both **`write()`** and **`writelines()`** do not automatically add newline characters, so each string must include a newline character if you want a line break in the output.

1.7.4 Writing Nested Data Structures Manually

Before using CSV or TSV modules, you can write nested lists or dictionaries manually by iterating over them:

```
nested_dict = {
    "Alice": {"Major": "CS", "GPA": 3.9},
    "Bob": {"Major": "Physics", "GPA": 3.5}
}

with open('students_manual.txt', 'w', encoding='utf-8') as f:
    for name, info in nested_dict.items():
        line = f"{name},{info['Major']},{info['GPA']}\n"
        f.write(line)
```

Contents of `students_manual.txt`:

```
Alice,CS,3.9
Bob,Physics,3.5
```

1.7.5 Writing Structured Text Files (CSV/TSV) with `writerow()`

You can write rows to a CSV file using **`csv.writer`** and its **`writerow()`** method. Each row is provided as a list of values corresponding to the columns.

```
import csv

with open('names.csv', 'w', newline='', encoding='utf-8') as csvfile:
    writer = csv.writer(csvfile)

    # Write the header row
    writer.writerow(['first_name', 'last_name'])

    # Write data rows
    writer.writerow(['Baked', 'Beans'])
    writer.writerow(['Lovely', 'Spam'])
    writer.writerow(['Wonderful', 'Spam'])
```

Contents of `names.csv`:

```
first_name,last_name
Baked,Beans
Lovely,Spam
Wonderful,Spam
```

Explanation:

- `writerow()` writes a single row as a list of values.
- The first row can be used as a header, followed by data rows.
- This approach is simpler when your data is already organized as lists rather than dictionaries.

1.7.6 Writing Structured Text Files (CSV/TSV) with `writerows()`

For structured data, Python's **`csv`** module simplifies writing CSV or TSV files.

```
import csv

# Data to write (list of lists)
student_data = [
    ["Name", "Major", "GPA"],
    ["Charlie", "Literature", 3.7],
    ["David", "Engineering", 3.8]
]

# Writing to a CSV file
with open('new_students.csv', 'w', newline='', encoding='utf-8') as f:
    writer = csv.writer(f)
    writer.writerows(student_data)

# Writing to a TSV file (tab-delimited)
with open('new_students.tsv', 'w', newline='', encoding='utf-8') as f:
    writer = csv.writer(f, delimiter='\t')
    writer.writerows(student_data)

print("CSV and TSV files have been created.")
```

Contents of `new_students.csv`:

```
Name,Major,GPA
Charlie,Literature,3.7
David,Engineering,3.8
```

Contents of `new_students.tsv`:

```
Name    Major   GPA
Charlie Literature  3.7
David   Engineering 3.8
```

Explanation:

- `writerows()` writes multiple rows at once; each row is a list of values.
- `delimiter='\t'` specifies tab separation for TSV files.
- Using `newline=''` avoids adding extra blank lines on Windows.
- Always specify `encoding='utf-8'` for cross-platform consistency.

1.8 Working with Binary Files: Serialization with `pickle`

1.8.1 What is Serialization?

Serialization is the process of converting a Python object (like a list or dictionary) into a byte stream that can be stored in a file. This allows you to save complex data structures and load them back later, preserving their state. Python's standard library for this is **`pickle`**.

1.8.2 Saving Objects with `pickle.dump()`

To save an object, open the file in binary write mode (**`'wb'`**).

```
1 import pickle
2
3 # A complex Python object
4 user_prefs = {"font_size": 12, "theme": "dark", "bookmarks": [101, 205, 350]}
5
6 with open('preferences.pkl', 'wb') as f:
7     pickle.dump(user_prefs, f)
8
9 print("Preferences object saved to preferences.pkl")
```

Security Warning: The **`pickle`** module is not secure. Never unpickle data received from an untrusted source, as it can be crafted to execute malicious code.

1.8.3 Loading Objects with `pickle.load()`

To load the object back, open the file in binary read mode (`'rb'`).

```
import pickle

with open('preferences.pkl', 'rb') as f:
    loaded_prefs = pickle.load(f)

print("--- Loaded Preferences ---")
print(loaded_prefs)
print(f"Theme: {loaded_prefs['theme']}")
```

Output:

```
--- Loaded Preferences ---
{'font_size': 12, 'theme': 'dark', 'bookmarks': [101, 205, 350]}
Theme: dark
```

Topic 2:

Exception and Exception Handling

Topic 2: Exception and Exception Handling

2.1 Why Do We Need Exception Handling?

Before diving into the technical details, let's think about why exception handling is so crucial. Imagine your program is a highly organized workshop.

- **Normal Flow:** Your skilled workers (the code) are assembling products smoothly.
- **An Exception:** Suddenly, a machine breaks down (a runtime error occurs). Without a plan, the entire assembly line halts, work is left unfinished, and chaos ensues. Your whole workshop (the program) shuts down.
- **Exception Handling ('try...except'):** This is your workshop's emergency plan. When a machine breaks, an alarm sounds (`try` detects an error) and a specialized repair crew (`except`) immediately handles the broken machine. The main production might pause, but the workshop itself doesn't shut down. The repair crew can fix the problem or log it for later, allowing other parts of the workshop to continue running.
- **Cleanup ('finally'):** This is the safety protocol that runs no matter what—like turning off the main power to the broken machine's area. This cleanup happens whether the machine was fixed or not, ensuring the workshop is left in a safe state.

Exception handling transforms your program from a fragile process that breaks at the first sign of trouble into a robust system that can anticipate, manage, and recover from unexpected problems.

2.2 What is an Exception?

In programming, an **exception** is an error event that occurs during the execution of a program that disrupts its normal flow. Unlike syntax errors, which are detected by the interpreter before the program is run, exceptions occur at runtime. When the Python interpreter encounters an error it cannot handle, it "raises" an exception.

If the exception is not handled by the program, the program will terminate and display a traceback message, which provides information about where the error occurred.

Common built-in exceptions in Python include:

- `TypeError`: Raised when an operation or function is applied to an object of an inappropriate type.
- `ValueError`: Raised when an operation or function receives an argument that has the right type but an inappropriate value.
- `ZeroDivisionError`: Raised when the second argument of a division or modulo operation is zero.
- `FileNotFoundError`: Raised when a file or directory is requested but doesn't exist.
- `IndexError`: Raised when a sequence subscript is out of range.
- `KeyError`: Raised when a dictionary key is not found.

2.2.1 Examples of Unhandled Exceptions

Example 1: `ZeroDivisionError`

```
# This code will cause a ZeroDivisionError
numerator = 10
denominator = 0
result = numerator / denominator
print(result)
```

Output (Traceback):

```
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
ZeroDivisionError: division by zero
```

Example 2: ValueError

```
# This code will cause a ValueError
age_str = "twenty"
age_int = int(age_str) # Cannot convert 'twenty' to an integer
print(age_int)
```

Output (Traceback):

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: invalid literal for int() with base 10: 'twenty'
```

Example 3: IndexError

```
# This code will cause an IndexError
my_list = [10, 20, 30]
print(my_list[3]) # Index 3 is out of bounds for a list of size 3
```

Output (Traceback):

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
IndexError: list index out of range
```

2.3 The `try...except` Block

To prevent a program from crashing due to an exception, you can use **exception handling**. The core mechanism for this in Python is the `try...except` block.

The basic idea is to place the code that might raise an exception inside the `try` block. If an exception occurs, the rest of the `try` block is skipped, and the code inside the corresponding `except` block is executed. If no exception occurs, the `except` block is ignored.

2.3.1 Examples of `try...except`

Example 1: Handling a `ZeroDivisionError`

```
numerator = 10
denominator = 0

try:
    result = numerator / denominator
    print("This will not be printed.")
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")

print("The program continues to run.")
```

Output:

```
Error: Cannot divide by zero.
The program continues to run.
```

Example 2: Handling a `ValueError`

```python
try:
    user_input = input("Enter a number: ")
    number = int(user_input)
    print(f"You entered the number {number}.")
except ValueError:
    print("Invalid input. Please enter a valid integer.")
```

Sample Output (with invalid input):

```
Enter a number: abc
Invalid input. Please enter a valid integer.
```

Example 3: Handling a `FileNotFoundError`

```
try:
    with open("non_existent_file.txt", "r") as f:
        content = f.read()
        print(content)
except FileNotFoundError:
    print("Error: The file could not be found.")
```

Output:

```
Error: The file could not be found.
```

2.4 Handling Specific Exceptions

It is good practice to handle specific exceptions rather than using a generic `except` block. This allows your program to respond appropriately to different types of errors. You can have multiple `except` blocks to handle various exceptions, or handle multiple exceptions in a single block.

2.4.1 Examples of Handling Specific Exceptions

Example 1: Multiple `except` Blocks

```
1  def process_data(data, index):
2      try:
3          value = int(data[index])
4          result = 100 / value
5          print(f"The result is: {result}")
6      except IndexError:
7          print("Error: Index is out of bounds.")
8      except ValueError:
9          print("Error: The data at the given index is not a valid integer.")
10     except ZeroDivisionError:
11         print("Error: The value at the given index is zero, cannot divide.")
12
13 my_list = ["10", "20", "0", "abc"]
14 process_data(my_list, 2)  # ZeroDivisionError
15 process_data(my_list, 3)  # ValueError
16 process_data(my_list, 5)  # IndexError
```

Output:

```
Error: The value at the given index is zero, cannot divide.
Error: The data at the given index is not a valid integer.
Error: Index is out of bounds.
```

Example 2: Grouping Multiple Exceptions

```python
def calculate_average(numbers):
    try:
        total = sum(numbers)
        count = len(numbers)
        average = total / count
        print(f"The average is {average}.")
    except (TypeError, ZeroDivisionError) as e:
        print(f"Could not calculate average. Error: {e}")

calculate_average([10, 20, 'thirty']) # Raises TypeError
calculate_average([])                 # Raises ZeroDivisionError
```

Output:

```
Could not calculate average. Error: unsupported operand type(s) for +: 'int' and 'str'
Could not calculate average. Error: division by zero
```

Example 3: Catching the Base `Exception` Class

```python
import sys
try:
    # A risky operation
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
    result = 10 / i
except Exception as e:
    # This will catch any exception that inherits from Exception
    print(f"An unexpected error occurred: {e}")
    print(f"Error type: {type(e).__name__}")
```

Sample Output (if myfile.txt contains '0'):

```
An unexpected error occurred: division by zero
Error type: ZeroDivisionError
```

2.5 The `else` and `finally` Clauses

The `try...except` block can be extended with optional `else` and `finally` clauses.

- **`else` Clause**: The code inside the `else` block is executed **only if no exceptions are raised** in the `try` block.
- **`finally` Clause**: The code inside the `finally` block is **always executed**, regardless of whether an exception occurred or not. It is ideal for cleanup actions.

2.5.1 Examples of `else` and `finally`

Example 1: Basic Usage

```python
def divide_numbers(a, b):
    try:
        print("Attempting division...")
        result = a / b
    except ZeroDivisionError:
        print("Caught a ZeroDivisionError!")
    else:
        print(f"Division successful! Result is {result}")
    finally:
        print("This 'finally' block is always executed.")

divide_numbers(10, 2); print("-" * 20)  # No exception
divide_numbers(10, 0) # An exception occurs
```

Output:

```
Attempting division...
Division successful! Result is 5.0
This 'finally' block is always executed.
--------------------
Attempting division...
Caught a ZeroDivisionError!
This 'finally' block is always executed.
```

Example 2: File Handling with `finally`

```
f = None # Initialize f to ensure it exists in the finally block
try:
    f = open("important_data.txt", "w")
    f.write("This is a critical message.")
    # Simulate an error after writing
    # int("abc") # Uncomment this line to see finally run after an error
except Exception as e:
    print(f"An error occurred: {e}")
else:
    print("File written successfully.")
finally:
    if f: # Check if the file was opened
        f.close()
        print("File has been closed.")
```

Output:

```
File written successfully.
File has been closed.
```

Example 3: Return Value from `finally` A `return` statement in a `finally` block will override any `return` in the `try` or `except` blocks.

```
1  def check_status():
2      try:
3          print("Trying to return 'Success'")
4          return "Success from try"
5      except:
6          return "Failure from except"
7      finally:
8          print("Finally block is executed, returning 'Complete'")
9          return "Complete from finally"
10
11 print(check_status())
```

Output:

```
1 Trying to return 'Success'
2 Finally block is executed, returning 'Complete'
3 Complete from finally
```

2.6 Raising Exceptions

You can manually raise an exception using the **`raise`** keyword. This is useful for signaling an error condition based on your program's logic.

2.6.1 Examples of Raising Exceptions

Example 1: Raising a `ValueError`

```
1 def set_age(age):
2     if age < 0 or age > 120:
3         raise ValueError("Age must be between 0 and 120.")
4     print(f"Age set to {age}")
5 
6 try:
7     set_age(150)
8 except ValueError as e:
9     print(f"Error: {e}")
```

Output:

```
1 Error: Age must be between 0 and 120.
```

Example 2: Raising a TypeError

```
1  def add_to_list(items, item):
2      if not isinstance(items, list):
3          raise TypeError("First argument must be a list.")
4      items.append(item)
5      return items
6
7  my_tuple = (1, 2)
8  try:
9      add_to_list(my_tuple, 3)
10 except TypeError as e:
11     print(f"Error: {e}")
```

Output:

```
1  Error: First argument must be a list.
```

Example 3: Re-raising an Exception

Sometimes you may want to catch an exception, perform an action (like logging), and then re-raise it to let a higher-level handler deal with it.

```
def process_file(filename):
    try:
        with open(filename, 'r') as f:
            print(f.read())
    except FileNotFoundError as e:
        print(f"Logging error: Could not find {filename}.")
        raise # Re-raise the caught exception

try:
    process_file("data.txt")
except FileNotFoundError:
    print("Error handler: The program cannot continue without the file.")
```

Output:

```
Logging error: Could not find data.txt.
Error handler: The program cannot continue without the file.
```

2.7 The `assert` Statement

The **`assert`** statement is a debugging aid that tests a condition. If the condition is true, it does nothing, and the program continues. If the condition is false, it raises an **`AssertionError`** with an optional message.

Assertions are meant for internal self-checks and should not be used for handling expected runtime errors (like invalid user input), as assertions can be globally disabled.

Syntax:

```
assert condition, "Optional error message if condition is false"
```

2.7.1 Examples of using `assert`

Example 1: Checking a Value

```
# This assertion will pass silently
x = 10
assert x > 0, "x should be positive"
print("Assertion passed.")

# This assertion will fail and raise an AssertionError
y = -5
assert y > 0, "y should be positive"
print("This line will not be executed.")
```

Output:

```
Assertion passed.
Traceback (most recent call last):
  File "<stdin>", line 7, in <module>
AssertionError: y should be positive
```

Example 2: Sanity Check in a Function

```python
def calculate_discount(price, discount_percent):
    assert 0 <= discount_percent <= 100, "Discount must be between 0 and 100"
    return price * (1 - discount_percent / 100)

# This works fine
print(calculate_discount(50, 20))

# This will raise an AssertionError
print(calculate_discount(50, 110))
```

Output:

```
40.0
Traceback (most recent call last):
  File "<stdin>", line 8, in <module>
  File "<stdin>", line 2, in calculate_discount
AssertionError: Discount must be between 0 and 100
```

Example 3: Checking a Type

```python
def process_list(items):
    assert isinstance(items, list), "Input must be a list"
    for item in items:
        print(item)

# This works fine
process_list([1, 2, 3])

# This will raise an AssertionError
process_list((4, 5, 6)) # A tuple, not a list
```

Output:

```
1
2
3
Traceback (most recent call last):
  File "<stdin>", line 9, in <module>
  File "<stdin>", line 2, in process_list
AssertionError: Input must be a list
```

2.8 `assert` vs. `raise`: When to Use Each

It's common for beginners to confuse when to use an **`assert`** statement versus raising an exception with **`raise`**. The distinction is critical for writing correct and maintainable code. Here is a direct comparison:

Feature	**`assert` Statement**	**`raise` Statement**
Purpose	**Debugging.** Used to check for internal conditions that should *never* happen. Catches programmer errors.	**Error Handling.** Used to signal an expected, possible error condition that can occur at runtime (e.g., bad user input, file not found).
Audience	For the developer. It says, "I believe this condition to be true, and if it's not, my program has a bug."	For the user or caller of the code. It says, "You have provided invalid data or an unrecoverable situation has occurred."
When it's Active	Can be globally **disabled** in production by running Python with the **`-O`** (optimize) flag. They should not be relied upon for application logic.	Is **always active** and is a core part of the program's control flow and logic.
Exception Type	Always raises an **`AssertionError`**.	Can raise any type of exception, including built-in ones (**`ValueError`**, **`TypeError`**) or custom exceptions.

Golden Rule: Use **`assert`** to check for bugs in your own code. Use **`raise`** to signal errors to the person using your code.

2.9 Creating Custom Exceptions

For application-specific problems, you can create custom exceptions by defining a new class that inherits from the base `Exception` class. This makes your code more readable and your error handling more specific.

2.9.1 Examples of Custom Exceptions

Example 1: `InsufficientFundsError`

```
class InsufficientFundsError(Exception):
    """Raised when a withdrawal is attempted with insufficient funds."""
    def __init__(self, balance, amount):
        message = f"Attempted to withdraw {amount} with a balance of only {balance}"
        super().__init__(message)

def withdraw(balance, amount):
    if amount > balance:
        raise InsufficientFundsError(balance, amount)
    return balance - amount

try:
    withdraw(balance=100, amount=500)
except InsufficientFundsError as e:
    print(f"Transaction failed: {e}")
```

Output:

```
Transaction failed: Attempted to withdraw 500 with a balance of only 100
```

Example 2: InvalidEmailError

```python
class InvalidEmailError(ValueError):
    """Raised when an email format is invalid."""
    pass # Inherits behavior from ValueError

def send_email(email, message):
    if "@" not in email:
        raise InvalidEmailError(f"'{email}' is not a valid email address.")
    print(f"Email sent to {email}.")

try:
    send_email("not-an-email", "Hello!")
except InvalidEmailError as e:
    print(f"Error: {e}")
```

Output:

```
Error: 'not-an-email' is not a valid email address.
```

Example 3: PasswordTooShortError

```python
class PasswordTooShortError(Exception):
    def __init__(self, length, min_length=8):
        message = f"Password is too short. Required: {min_length}, Provided: {length}"
        super().__init__(message)

def set_password(password):
    if len(password) < 8:
        raise PasswordTooShortError(len(password))
    print("Password has been set successfully.")

try:
    set_password("12345")
except PasswordTooShortError as e:
    print(f"Could not set password. Reason: {e}")
```

Output:

```
Could not set password. Reason: Password is too short. Required: 8, Provided: 5
```

2.10 Summary: Best Practices for Exception Handling

To write clean, robust, and professional Python code, follow these key principles for exception handling:

- **Be Specific, Not General.** Always catch the most specific exception you can. Avoid bare `except:` blocks, as they can hide real bugs.

```python
# Good: Specific and clear
try:
    value = int("a")
except ValueError:
    print("Invalid number format.")

# Bad: Hides all errors, even ones you didn't expect
try:
    # ... code ...
except: # This could hide a TypeError, NameError, etc.
    print("Something went wrong.")
```

- **Keep `try` Blocks Small.** Only wrap the specific lines of code that you expect might raise an exception inside the **`try`** block. This makes your code easier to read and debug.
- **Use `finally` for Cleanup.** For actions that *must* be completed—like closing a file or a network connection—place the code in a **`finally`** block to guarantee its execution.
- **Don't Use Exceptions for Normal Control Flow.** Exceptions are for exceptional, unexpected events. If you can easily check for a condition with an **`if`** statement, that is more efficient and clearer than using a **`try...except`** block.
- **Create Custom Exceptions for Your Application.** When you need to signal an error that is specific to your program's domain (e.g., **`InsufficientFundsError`**), create a custom exception class. This makes your program's error conditions explicit and self-documenting.

Topic 3:

Introduction to Object-Oriented Programming

Topic 3: Introduction to Object-Oriented Programming

3.1 Overview of Object-Oriented Programming

Object-Oriented Programming (OOP) became the dominant programming paradigm during the 1990s, and its principles continue to be fundamental in modern software development. The OOP approach focuses on organizing code around objects that model real-world entities, making programs easier to design, maintain, and extend.

- Many popular programming languages today are built entirely around, or strongly support, object-oriented concepts. These include languages such as **Python**, **Java**, **C++**, **JavaScript** (both client-side and Node.js), **C#**, **Swift**, and **Go** (which supports limited forms of OOP through structs and interfaces).
- In contrast, some older or lower-level languages, such as **C** and **Assembly**, do not natively support object-oriented features. In these languages, developers typically use procedural or structured programming styles instead.

Whether fully or partially supported, OOP has influenced nearly every major programming language, shaping the way modern developers think about and design software systems.

3.1.1 Object-Oriented Programming Concepts

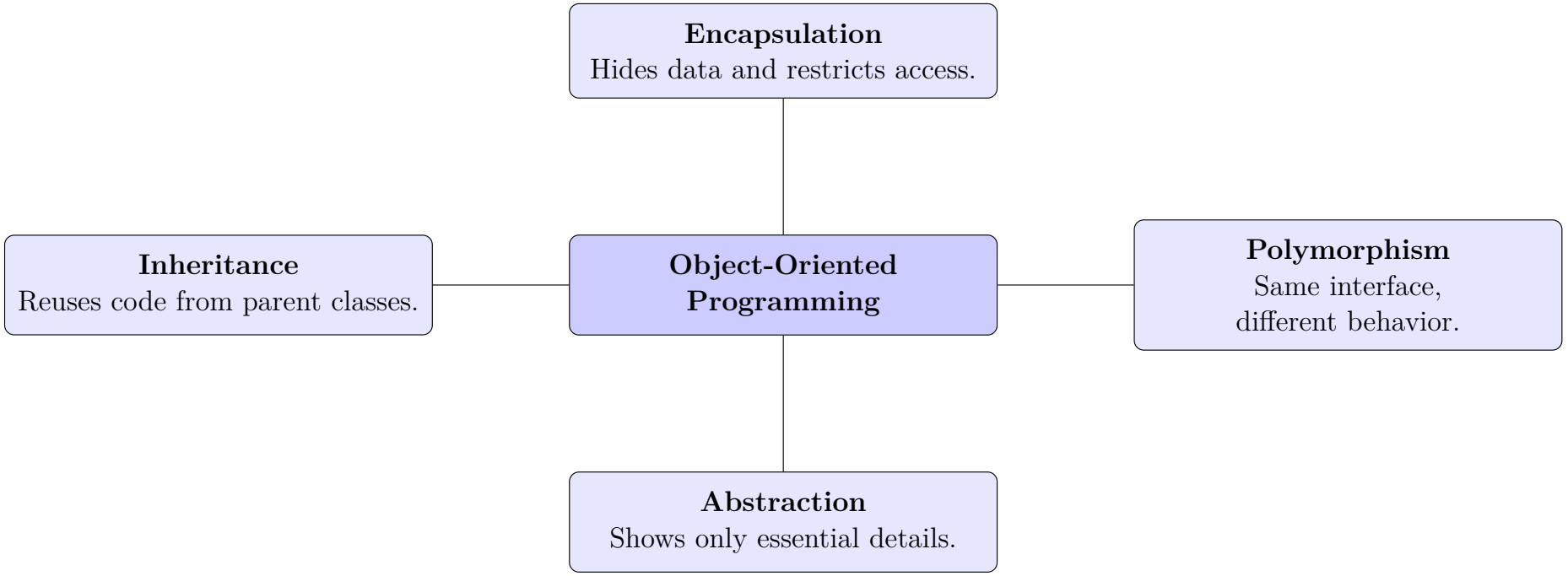

Figure 1: Object-Oriented Programming Concepts

3.2 What is Object-Oriented Programming (OOP)?

Object-Oriented Programming (OOP) is a programming paradigm centered around the concept of "objects." A paradigm[3] is a fundamental style or way of thinking about building software.

In OOP, we structure a program by creating objects that model real-world things, bundling related data and behaviors together.

This approach was developed to manage the growing complexity of large software systems. Before **OOP**, the dominant paradigm was **procedural programming**.

[3]**Paradigm** (noun): a typical example or pattern of something; a model. Also refers to a distinct set of concepts or thought patterns in a scientific discipline or other epistemological context. (Oxford Languages)

- **Procedural Programming**: This style consists of writing a sequence of steps or procedures (functions) that operate on data. Data (variables) and procedures (functions) are kept separate. For small programs, this is simple and effective.

  Example: Imagine building a car on an assembly line. You have a pile of parts (the data) and a set of instructions (the procedures). A worker takes a part, follows an instruction, and moves to the next step. The parts and the instructions are separate.

- **Object-Oriented Programming**: This style involves creating self-contained objects where data and the functions that operate on that data are bundled together. This promotes better organization and reusability.

  Example: Instead of an assembly line, imagine you have modular, pre-built components like a complete engine or a pre-wired dashboard (the objects). Each component manages its own internal complexity (its data and methods). To build the car, you simply connect these components. It's easier to reason about, test, and replace a single engine component than to re-evaluate the entire assembly line.

OOP helps us write code that is more modular, flexible, and easier to maintain and debug, especially for complex applications.

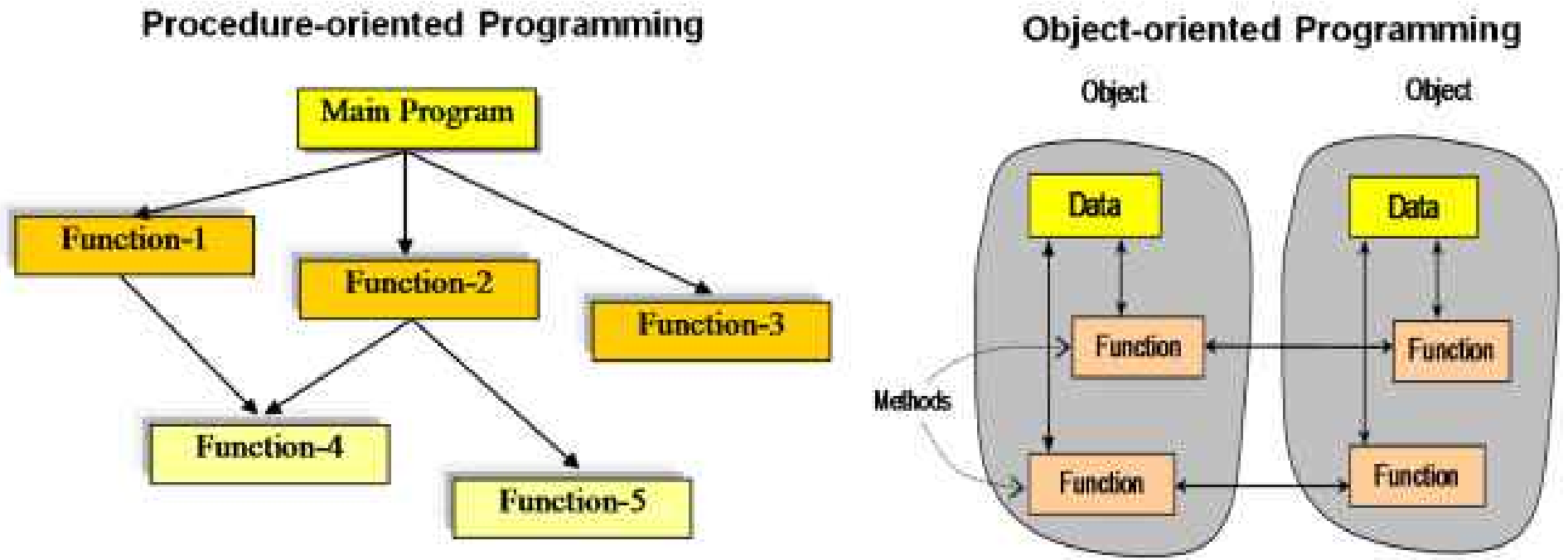

Figure 2: procedural programming vs object-oriented programming

Source: getmason.io

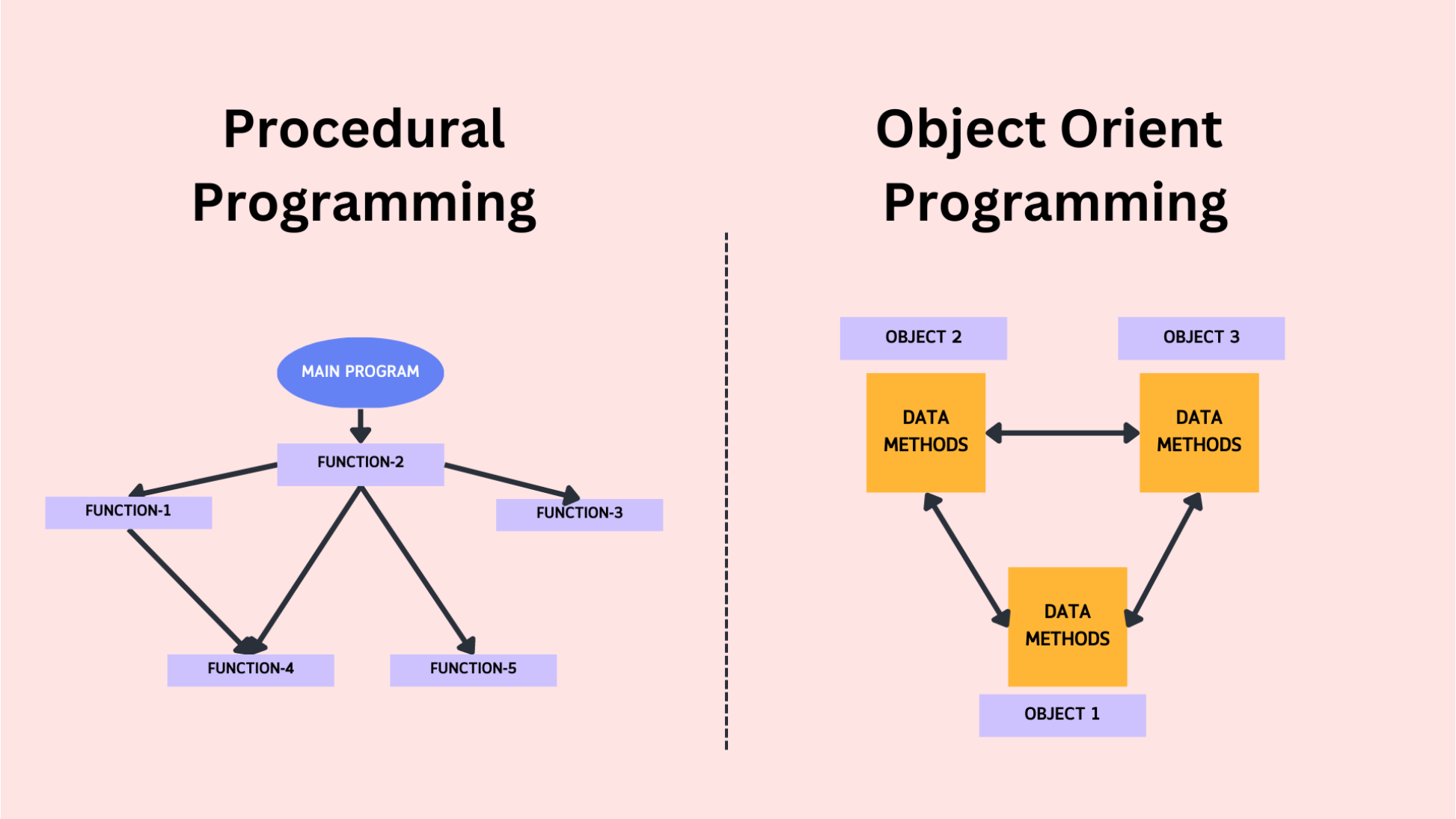

Figure 3: procedural programming vs object-oriented programming

Source: logicmojo.com

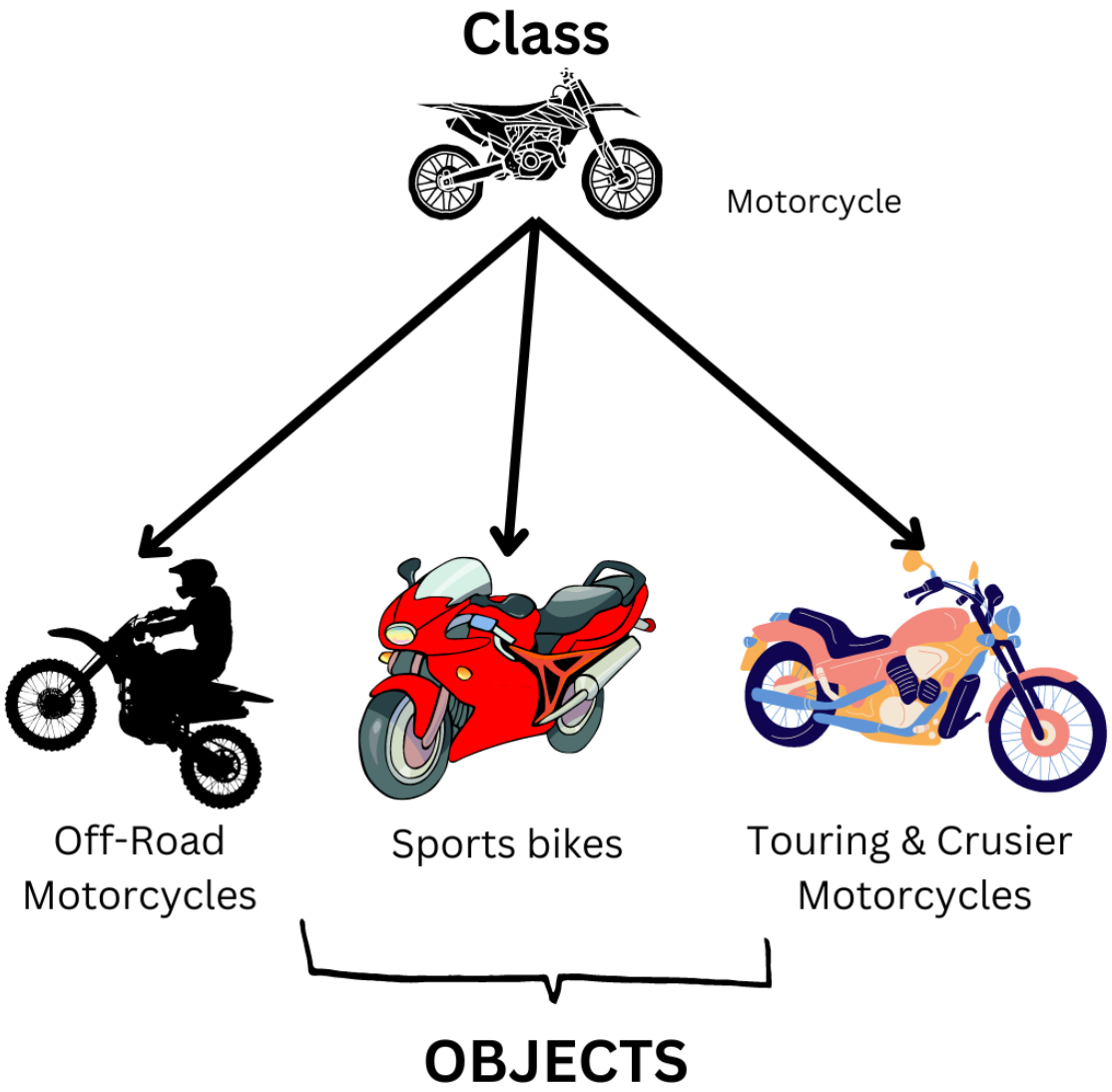

Figure 4: Concept of objects in object-oriented programming

Source: logicmojo.com

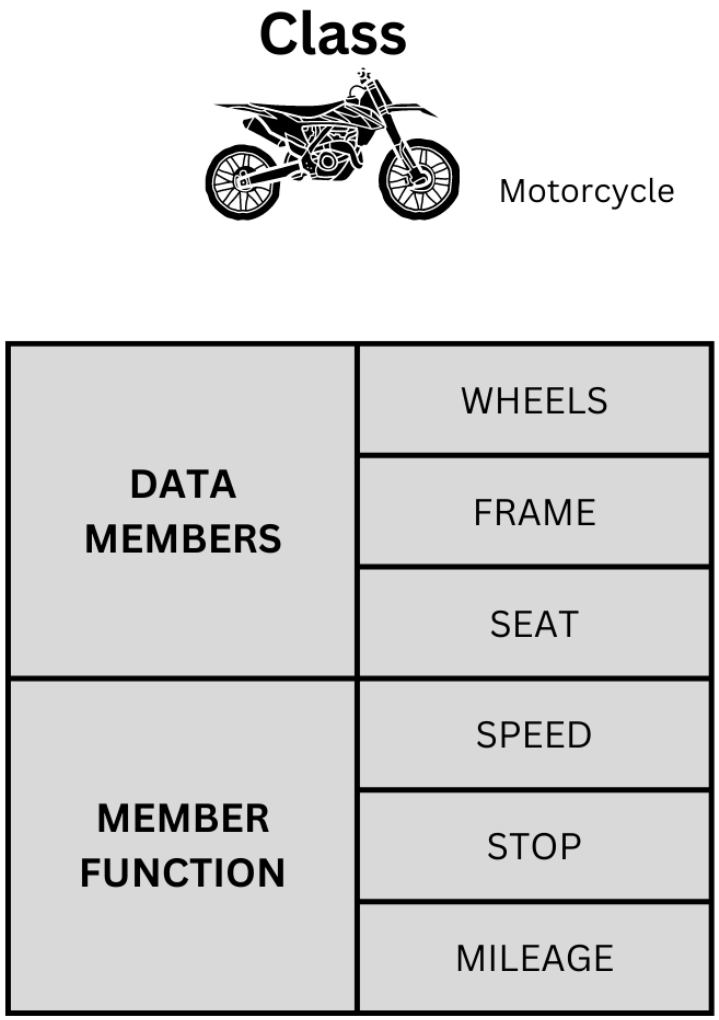

Figure 5: Concept of objects in object-oriented programming

Source: logicmojo.com

3.3 Classes and Objects

The two most fundamental concepts in OOP are the **class** and the **object**. By analogy, they are like a blueprint (class) and a house (object).

- **Class**: A class is a **blueprint**, template, or recipe for creating objects. It defines a common set of attributes (the data an object will hold) and methods (the behaviors an object can perform). The class itself is not the object; it's the plan for making objects.
- **Object**: An object is a concrete **instance** of a class. It is the actual thing built from the blueprint, with its own unique state (i.e., its own values for its attributes). You can create many distinct objects from a single class.

3.3.1 The `__init__` Method and the `self` Keyword

When we define a class in Python, two special elements are crucial:

- `__init__()`: This is a special method called the **constructor**. It is automatically called whenever a new object of the class is created. Its primary job is to initialize the attributes of the object.
- `self`: This is a special parameter that represents the instance of the object itself. Inside a class's methods, `self` is used to access the object's own attributes and other methods. When you call a method like `my_object.my_method(arg1)`, Python automatically passes the object `my_object` as the first argument, `self`.

3.3.2 Example: The `Motorcycle` Class (Attributes (data) Only)

Let's start with a very simple class that only defines data members (attributes) but no methods. This helps illustrate the idea that a class can simply describe what data an object holds.

```
# A simple class with only attributes.
class Motorcycle:
    def __init__(self, wheels, frame, seat):
        self.wheels = wheels
        self.frame = frame
        self.seat = seat

# Create two Motorcycle objects (instances)
bike1 = Motorcycle(2, "Steel", "Leather")
bike2 = Motorcycle(3, "Carbon Fiber", "Plastic")

# Access attributes
print("=== Motorcycle 1 ===")
print("Wheels:", bike1.wheels)
print("Frame:", bike1.frame)
print("Seat:", bike1.seat)

print("\n=== Motorcycle 2 ===")
print("Wheels:", bike2.wheels)
print("Frame:", bike2.frame)
print("Seat:", bike2.seat)
```

Output:

```
=== Motorcycle 1 ===
Wheels: 2
Frame: Steel
Seat: Leather

=== Motorcycle 2 ===
Wheels: 3
Frame: Carbon Fiber
Seat: Plastic
```

This example defines the structure of a motorcycle but no behavior yet. The next example adds methods to make the class more functional.

3.3.3 Example: The `Motorcycle` Class (With Methods)

Now, let's extend the **`Motorcycle`** class to include behaviors (methods). These methods let the object perform actions related to its state.

```
# A more complete Motorcycle class with attributes and methods.
class Motorcycle:
    def __init__(self, wheels, frame, seat):
        self.wheels = wheels
        self.frame = frame
        self.seat = seat
        self.speed = 0
        self.mileage = 0

    def start(self):
        print("Motorcycle is starting...")

    def accelerate(self, amount):
        self.speed += amount
        print(f"Accelerating... Current speed: {self.speed} km/h")

    def stop(self):
        self.speed = 0
        print("Motorcycle has stopped.")

    def add_mileage(self, distance):
        self.mileage += distance
```

```
        print(f"Added {distance} km. Total mileage: {self.mileage} km")

# Create and interact with a Motorcycle object.
bike = Motorcycle(2, "Aluminum", "Leather")
bike.start()
bike.accelerate(60)
bike.add_mileage(15)
bike.stop()
```

Output:

```
Motorcycle is starting...
Accelerating... Current speed: 60 km/h
Added 15 km. Total mileage: 15 km
Motorcycle has stopped.
```

3.3.4 Example: The Student Class

```python
# The class 'Student' is the blueprint.
class Student:
    # The constructor initializes each new Student object.
    def __init__(self, student_id, name, major):
        # self.student_id is an attribute of the instance.
        # student_id is the parameter passed during creation.
        self.student_id = student_id
        self.name = name
        self.major = major
        self.credits = 0  # Default value for all new students

    # A method defines a behavior. 'self' gives it access to instance attributes.
    def complete_course(self, course_credits):
        print(f"{self.name} is completing a course.")
        self.credits += course_credits

    def get_info(self):
        return f"ID: {self.student_id}, Name: {self.name}, Major: {self.major}, Credits
          : {self.credits}"

# Create two distinct objects (instances) from the Student class.
student1 = Student("S001", "Alice", "Computer Science")
student2 = Student("S002", "Bob", "Physics")

```

```
24 # Each object has its own data.
25 student1.complete_course(3)
26 student2.complete_course(4)
27 student2.complete_course(4)
28
29 print(student1.get_info())
30 print(student2.get_info())
```

Output:

```
1 Alice is completing a course.
2 Bob is completing a course.
3 Bob is completing a course.
4 ID: S001, Name: Alice, Major: Computer Science, Credits: 3
5 ID: S002, Name: Bob, Major: Physics, Credits: 8
```

3.4 The Four Pillars of OOP

OOP is built upon four fundamental principles that enable its power and flexibility.

Encapsulation — Abstraction — Inheritance — Polymorphism

The following are four key principles of Object-Oriented Programming:

- **Encapsulation**: The practice of bundling data (attributes) and methods (functions) into a single unit, while restricting direct access to some of an object's components.
- **Abstraction**: The process of hiding complex implementation details and exposing only the essential features needed to use an object effectively.
- **Inheritance**: A mechanism that allows one class to acquire the properties and behaviors of another, enabling code reuse and establishing hierarchical relationships.
- **Polymorphism**: The ability of different objects to respond to the same method call in different ways, allowing a single interface to represent multiple underlying forms.

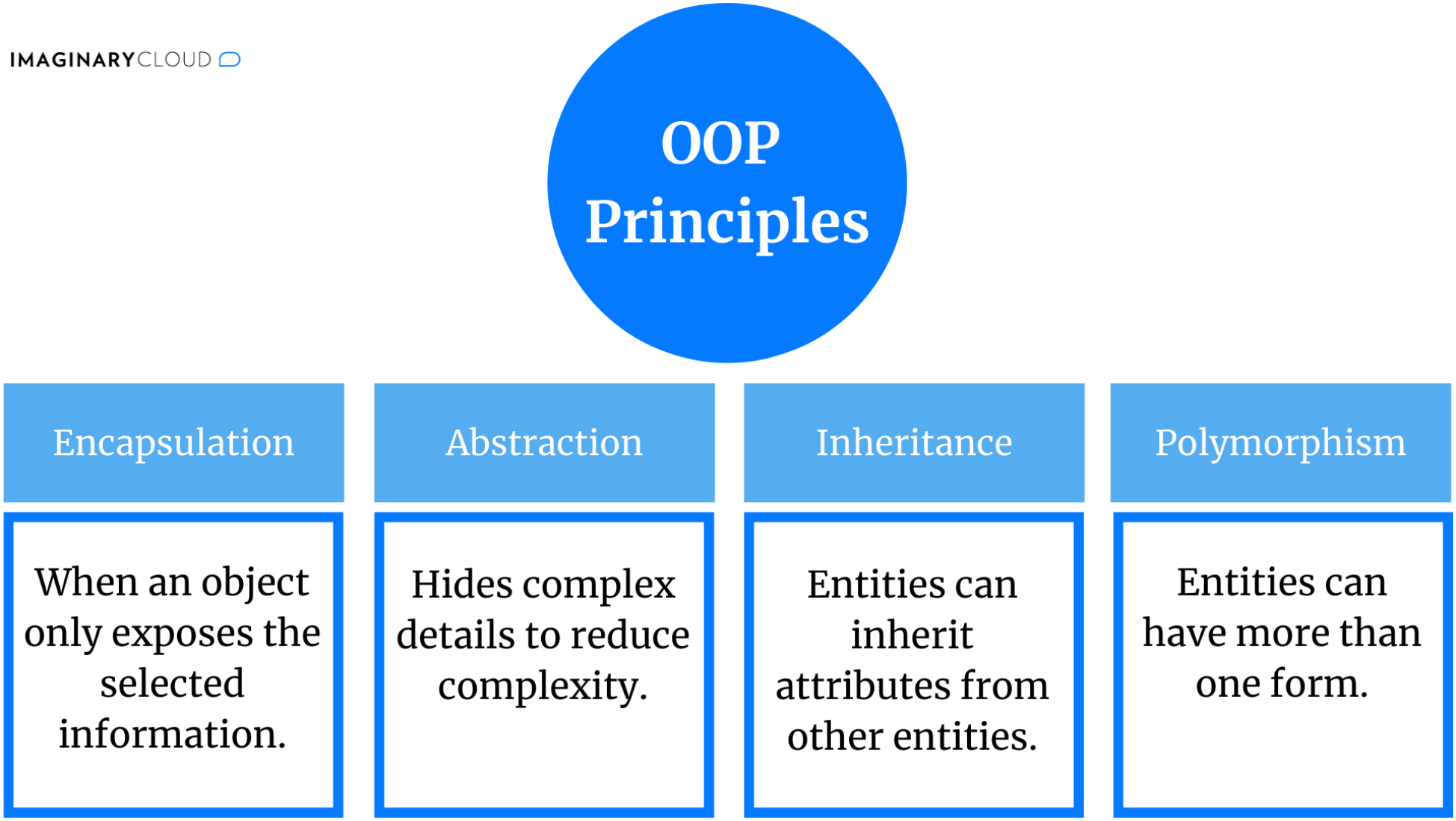

Figure 6: The Four Pillars of OOP

Source: Natalia Terlecka, Mariana Berga

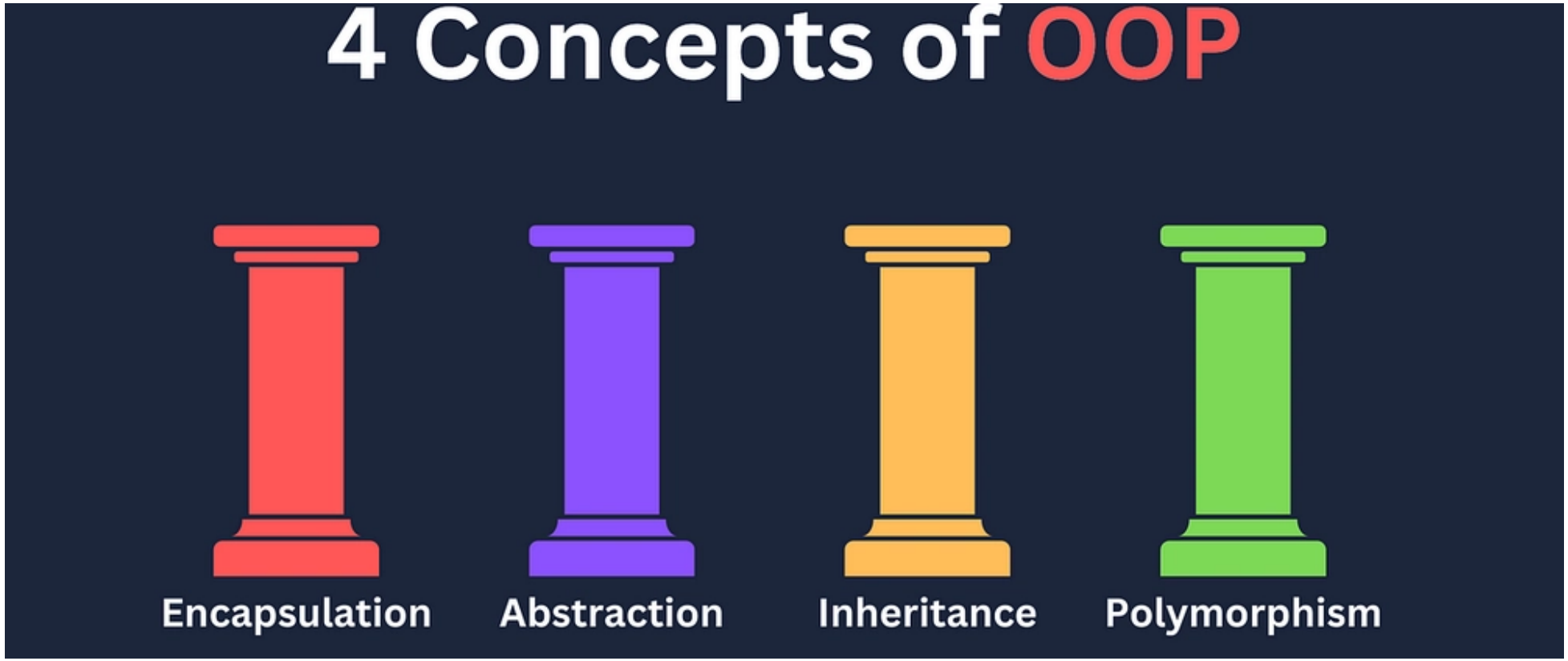

Figure 7: The Four Pillars of OOP

Source: _Khojiakbar_

3.4.1 1. Encapsulation

Encapsulation is the bundling of data (attributes) and the methods that operate on that data into a single, self-contained unit (the object).

A crucial part of encapsulation is **data hiding**—restricting direct access to an object's internal data from outside the object. Instead, access is provided through public methods (a "public interface").

Note: A **public** member can be accessed from anywhere in the program, while **private** and **protected** members restrict access to certain scopes. Private members are accessible only within the class itself, whereas protected members are accessible within the class and its subclasses.

- **Benefits**: This prevents accidental or improper modification of an object's data, making the code more robust and secure. It allows the internal implementation of an object to change without affecting the code that uses it.
- **Example**: Think of a car's dashboard. You, the driver, interact with the speedometer to read the speed and with the ignition to start the car. You cannot reach in and manually spin the engine's crankshaft. The car's internal mechanics (its data) are encapsulated, and you interact with it through a safe, public interface (the dashboard).

In Python, data hiding is achieved by a naming convention: prefixing an attribute with a single underscore (`_`) indicates it is **protected** (intended for internal or subclass use), while prefixing with a double underscore (`__`) makes it **private** (not easily accessible from outside the class).

3.4.2 Encapsulation Example: Public vs Private Attributes

```
class Motorcycle:
    def __init__(self, wheels, frame, seat):
        self.wheels = wheels      # Public attribute
        self.__frame = frame      # Private attribute
        self.seat = seat          # Public attribute

# Create a Motorcycle object
bike = Motorcycle(2, "Steel", "Leather")

# Accessing public attributes (works fine)
print("Wheels:", bike.wheels)
print("Seat:", bike.seat)

# Attempting to access private attribute directly (raises AttributeError)
print("Frame:", bike.__frame)
```

Output:

```
Wheels: 2
Seat: Leather
Traceback (most recent call last):
  File "<stdin>", line X, in <module>
AttributeError: 'Motorcycle' object has no attribute '__frame'
```

3.4.3 Encapsulation Example: The BankAccount Class

```
class BankAccount:
    def __init__(self, owner, balance=0.0):
        self.owner = owner  # Public attribute
        self.__balance = balance  # Private attribute

    # Public method to get the balance (a "getter")
    def get_balance(self):
        return self.__balance

    # Public method to deposit money
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited ${amount:.2f}. New balance: ${self.__balance:.2f}")
        else:
            print("Deposit amount must be positive.")

    # Public method to withdraw money
    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew ${amount:.2f}. New balance: ${self.__balance:.2f}")
        else:
            print("Invalid withdrawal amount or insufficient funds.")
```

```
25
26 account = BankAccount("John Doe", 100.0)
27
28 # We can't access the private attribute directly.
29 # print(account.__balance)  # This would raise an AttributeError.
30
31 # We must use the public interface.
32 print(f"Initial balance: ${account.get_balance():.2f}")
33 account.deposit(50.0)
34 account.withdraw(30.0)
35 account.withdraw(500.0) # This will fail safely.
36 print(f"Final balance: ${account.get_balance():.2f}")
```

Output:

```
1 Initial balance: $100.00
2 Deposited $50.00. New balance: $150.00
3 Withdrew $30.00. New balance: $120.00
4 Invalid withdrawal amount or insufficient funds.
5 Final balance: $120.00
```

In this example, the `balance` attribute is made private (using a double underscore prefix, `__balance`) to prevent direct access or modification from outside the class. This ensures that the balance cannot be seen or altered.

This design enforces **encapsulation** and **abstraction**, two key principles of Object-Oriented Programming.

All interactions with the balance—such as checking the current amount, depositing, or withdrawing money—must go through the specific public methods provided by the class (`get_balance()`, `deposit()`, `withdraw()`).

Note:

- **Encapsulation**: In the way that it provides **data protection**, data and methods are bundled together to safeguard the internal state of an object and control access through a public interface.
- **Abstraction**: In the way that it **reduces complexity**, only the essential functionalities of an object are exposed to the outside world, hiding complex implementation details.

3.4.4 2. Abstraction

Abstraction is the principle of hiding the complex implementation details of an object and exposing only the essential features or functionalities to the user. It simplifies a complex system by providing a high-level view.

Note: Abstraction focuses on *what* an object does (**functional**), while **encapsulation** focuses on *how* an object does it (**security**).

- **Benefits**: Abstraction reduces complexity and allows developers to use a component without understanding its internal workings. This makes code easier to read and maintain.
- **Example**: Consider a television remote control. You interact with simple buttons like "Power," "Volume Up," and "Channel Down." You don't need to know about the infrared signals, frequency modulation, or the circuit board's logic. The complexity is abstracted away behind a simple user interface.

3.4.5 Abstraction Example: The `Circle` Class (Abstraction)

```python
import math

# This class abstracts the details of circle calculations.
# It hides the implementation (private methods __area and __circumference)
# but exposes a simple, public interface to get the results.
class Circle:
    def __init__(self, diameter):
        self.__diameter = diameter      # Private attribute
        self.__radius = diameter / 2    # Private attribute

    # Private method to calculate area (implementation detail)
    def __area(self):
        return math.pi * self.__radius ** 2

    # Private method to calculate circumference (implementation detail)
    def __circumference(self):
        return 2 * math.pi * self.__radius

    # --- Public Interface (The Abstraction) ----

    # Public "getter" for the area
    def get_area(self):
        return self.__area()

```

```
25      # Public "getter" for the circumference
26      def get_circumference(self):
27          return self.__circumference()
28
29      # Public method to get all details for printing
30      def get_details(self):
31          print(f"Diameter: {self.__diameter}")
32          print(f"Radius: {self.__radius}")
33          # Use the public getters (good practice)
34          print(f"Area: {self.get_area():.2f}")
35          print(f"Circumference: {self.get_circumference():.2f}")
36
37  # Create Circle objects
38  circle1 = Circle(10)
39  circle2 = Circle(7)
40
41  # Access details through the simple print method
42  print("--- Details for Circle 1 ---")
43  circle1.get_details()
44  print() # blank line
45  print("--- Details for Circle 2 ---")
46  circle2.get_details()
47
48  total_area = circle1.get_area() + circle2.get_area() # We can now adding the area.
49  print(f"\nTotal area of both circles: {total_area:.2f}")
```

Output:

```
--- Details for Circle 1 ---
Diameter: 10
Radius: 5.0
Area: 78.54
Circumference: 31.42

--- Details for Circle 2 ---
Diameter: 7
Radius: 3.5
Area: 38.48
Circumference: 21.99

Total area of both circles: 117.02
```

3.4.6 Abstraction Example: The `CoffeeMachine` Class (Abstraction)

```
# This class abstracts away the details of making different coffee drinks.
class CoffeeMachine:
    def __init__(self):
        self.__water_level = 100
        self.__beans_level = 100

    # These are private implementation details.
    def __grind_beans(self):
        print("Grinding coffee beans...")
        self.__beans_level -= 10

    def __heat_water(self):
        print("Heating water...")
        self.__water_level -= 20

    def __pour_espresso(self):
        print("Pouring espresso shot...")

    # This is the public, abstract interface.
    def make_espresso(self):
        print("--- Making Espresso ---")
        if self.__water_level >= 20 and self.__beans_level >= 10:
            self.__heat_water()
            self.__grind_beans()
```

```
            self.__pour_espresso()
            print("Your espresso is ready!")
        else:
            print("Error: Not enough water or beans.")

# The user has a very simple interaction.
machine = CoffeeMachine()
machine.make_espresso()
```

Output:

```
--- Making Espresso ---
Heating water...
Grinding coffee beans...
Pouring espresso shot...
Your espresso is ready!
```

3.4.7 3. Inheritance

Inheritance is a mechanism that allows a new class (the **subclass** or **child class**) to inherit attributes and methods from an existing class (the **superclass** or **parent class**). This creates an "is-a" relationship (e.g., a Dog "is-a" Animal).

- **Benefits**: Inheritance promotes code reusability (DRY - Don't Repeat Yourself) and establishes a logical hierarchy between classes.
- **Example**: In biology, a Dog or a Lion "is-a" Mammal, and a Mammal "is-a" Animal. A lion, as well as a dog, inherits general animal traits (like the ability to breathe) and more specific mammal traits (like having fur), but it also has its own unique behavior (roaring[4], or barking in dogs).

[4]**Roar** (verb/noun): the loud, deep sound made by a lion or similar large animal; used here to illustrate a class-specific behavior

Inheritance Example UML Diagram:

This diagram shows how Lion and Dog inherit from Mammal, which in turn inherits from Animal.

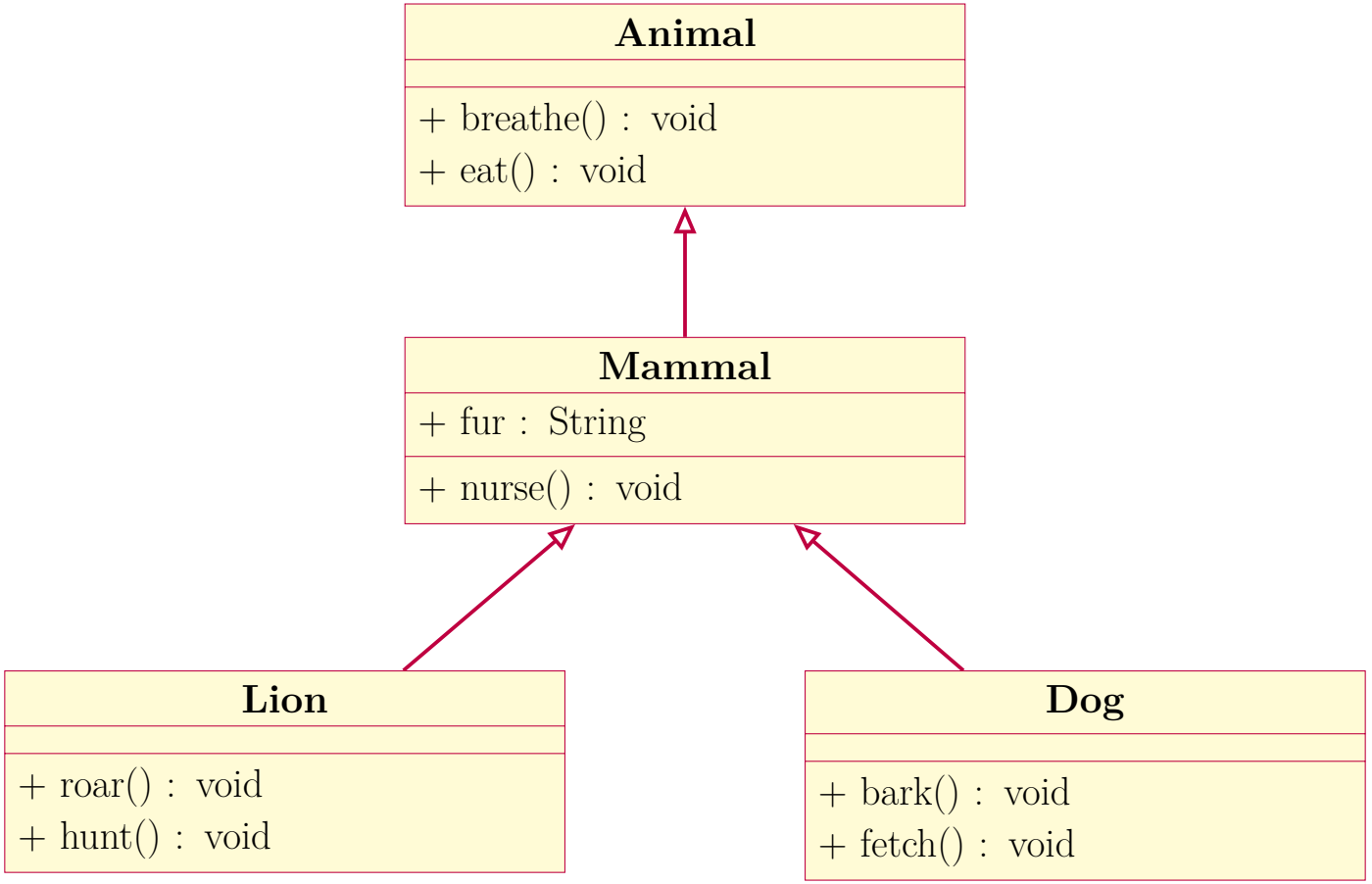

Figure 8: UML Class Diagram showing inheritance: Lion and Dog "is-a" Mammal, Mammal "is-a" Animal.

3.4.8 Inheritance Example: Animal, Dog, and Lion

This example illustrates **inheritance** in Python. The 'Dog' and 'Lion' classes inherit from the parent class 'Animal'. Each child class has its own method to exhibit specific behavior. We also demonstrate handling attempts to call a method that a child class does not have.

```
# Parent class
class Animal:
    def breathe(self):
        print("Generic breathing sound")

# Child class - Dog
class Dog(Animal):
    def bark(self):
        print("Woof!")

# Child class - Lion
class Lion(Animal):
    def roar(self):
        print("Roar!")

# Create objects
dog1 = Dog()
lion1 = Lion()


```

```
21
22 # Access methods
23 dog1.breathe()  # Calls Animal's breathe() method
24 dog1.bark()     # Calls Dog's bark() method
25 lion1.breathe() # Calls Animal's breathe() method
26 lion1.roar()    # Calls Lion's roar() method
27 print()
28
29 # Attempt to call a method that doesn't exist in Lion
30 try:
31     lion1.bark()  # Lion does not have bark()
32 except AttributeError as e:
33     print(f"Error: {e}")
```

Output:

```
1 Generic breathing sound
2 Woof!
3 Generic breathing sound
4 Roar!
5
6 Error: 'Lion' object has no attribute 'bark'
```

3.4.9 Inheritance Example: Shapes with Stored Area

This example demonstrates inheritance with geometric shapes.

- The '`Shape`' class stores the area, and child classes '`Circle`' and '`Rectangle`' calculate it via '`calculate_area()`'.
- The '`super()`' function is used in the child classes '`Circle`' and '`Rectangle`' to call the constructor of the parent class '`Shape`'. This ensures that the parent class initializes the '`color`' attribute correctly. Using '`super()`' allows the child class to extend the parent class functionality without rewriting its initialization code, promoting code reuse and maintaining the inheritance hierarchy.

```
1  # Parent class
2  class Shape:
3      def __init__(self, color):
4          self.color = color
5          self._area = None  # Protected attribute to store area
6
7      def get_color(self):
8          return self.color
9
10     def get_area(self):
11         return self._area
12
```

```
13
14
15
16 # Child class - Circle
17 class Circle(Shape):
18     def __init__(self, color, radius):
19         super().__init__(color)
20         self.radius = radius
21         self.calculate_area()
22
23     def calculate_area(self):
24         self._area = 3.14 * self.radius ** 2
25
26 # Child class - Rectangle
27 class Rectangle(Shape):
28     def __init__(self, color, width, height):
29         super().__init__(color)
30         self.width = width
31         self.height = height
32         self.calculate_area()
33
34     def calculate_area(self):
35         self._area = self.width * self.height
36
37
38
```

```
39
40 # Create objects
41 circle1 = Circle("Red", 5)
42 rect1 = Rectangle("Blue", 4, 6)
43
44 print(f"Circle color: {circle1.get_color()}, area: {circle1.get_area()}")
45 print(f"Rectangle color: {rect1.get_color()}, area: {rect1.get_area()}")
```

Output:

```
1 Circle color: Red, area: 78.5
2 Rectangle color: Blue, area: 24
```

Note:

- We make the '`_area`' attribute **protected** so that it cannot be directly modified from outside the class, providing **security** and maintaining encapsulation.
- Access to the area is only through the '`get_area()`' method.

3.4.10 Inheritance Example: Vehicle, Car, and Motorcycle Classes (with Brand Info)

This example demonstrates how child classes (‘`Car`’ and ‘`Motorcycle`’) inherit attributes and methods from a parent class (‘`Vehicle`’) and add their own specific behavior.

The ‘`super()`’ function ensures the parent constructor is called. The ‘`accelerate()`’ and ‘`brake()`’ methods now show the brand and the action performed.

```
# Superclass (Parent)
class Vehicle:
    def __init__(self, brand, year):
        self.brand = brand
        self.year = year
        self.speed = 0

    def accelerate(self, amount):
        self.speed += amount
        print(f"{self.brand} accelerates by {amount} km/h.")
        print(f"Speed is now {self.speed} km/h.")

    def brake(self, amount):
        self.speed -= amount
        print(f"{self.brand} brakes by {amount} km/h.")
        print(f"Speed is now {self.speed} km/h.")

```

```
18
19 # Subclass - Car
20 class Car(Vehicle):
21     def __init__(self, brand, year, num_doors):
22         super().__init__(brand, year)
23         self.num_doors = num_doors
24
25     def turn_on_radio(self):
26         print(f"{self.brand}'s radio is on!")
27
28 # Subclass - Motorcycle
29 class Motorcycle(Vehicle):
30     def __init__(self, brand, year, has_sidecar):
31         super().__init__(brand, year)
32         self.has_sidecar = has_sidecar
33
34     def make_engine_sound(self):
35         print(f"{self.brand} engine is making a sound!")
36
37
38 # Create objects
39 car1 = Car("Ford", 2021, 4)
40 car2 = Car("Toyota", 2020, 2)
41 bike1 = Motorcycle("Harley-Davidson", 2019, False)
42 bike2 = Motorcycle("Ducati", 2022, False)
43
```

```
# Perform actions
car1.accelerate(50)
car2.accelerate(30)
bike1.accelerate(80)
bike2.accelerate(60)
print()

car1.brake(20)
bike1.brake(40)
print()

car1.turn_on_radio()
bike1.make_engine_sound()
bike2.make_engine_sound()
```

Output:

```
Ford accelerates by 50 km/h.
Speed is now 50 km/h.
Toyota accelerates by 30 km/h.
Speed is now 30 km/h.
Harley-Davidson accelerates by 80 km/h.
Speed is now 80 km/h.
Ducati accelerates by 60 km/h.
Speed is now 60 km/h.

```

```
Ford brakes by 20 km/h.
Speed is now 30 km/h.
Harley-Davidson brakes by 40 km/h.
Speed is now 40 km/h.

Ford's radio is on!
Harley-Davidson engine is making a sound!
Ducati engine is making a sound!
```

3.4.11 4. Polymorphism

Polymorphism (from Greek, meaning "many forms") is the ability of an object to take on many forms. In practice, it means that a single interface (like a method name) can be used for objects of different classes, and each object will respond in its own specific way.

- **Benefits**: Polymorphism allows for flexible and decoupled code. You can write functions that operate on a general type without needing to know the specific subtype they are dealing with.

3.4.11.1 Polymorphism via Inheritance (Method Overriding)

- When a subclass provides its own implementation of a method that is inherited[5] from or defined in its superclass, it is called **method overriding**. This is the most common form of polymorphism in Object-Oriented Programming.
- **Method overriding** is a **programming technique** that allows a subclass to replace or layer on top of an existing function from the parent class, providing specialized behavior while retaining the interface of the original method.

[5]Inheritance:

– **Dictionary meaning**: the act of receiving something from a predecessor or parent.

– **In programming**: the mechanism by which a class (child/subclass) can acquire attributes and methods from another class (parent/superclass).

3.4.12 Polymorphism Example: Animal, Dog, and Lion

This demonstrates **polymorphism**. All classes have a `breathe()` method, but each works differently (polymorphism).

> **Note:** In the earlier example 3.4.8, we demonstrated inheritance, where the methods—or functions—were inherited under different names.

The `Dog` and `Lion` classes inherit from `Animal`. In this example, the child classes override the same method name (`breathe()`), demonstrating **method overriding**. Each child class provides its own implementation of the `breathe()` method to exhibit specialized behavior.

```
1  # Parent class
2  class Animal:
3      def breathe(self):
4          print("Generic breathing")
5
6  # Child class - Dog
7  class Dog(Animal):
8      def breathe(self):
9          print("Dog breathing through nose")
10
11
12
```

```
13 # Child class - Lion
14 class Lion(Animal):
15     def breathe(self):
16         print("Lion breathing heavily")
17 
18 dog1 = Dog()
19 lion1 = Lion()
20 
21 dog1.breathe()
22 lion1.breathe()
```

Output:

```
1 Dog breathing through nose
2 Lion breathing heavily
```

3.4.13 Polymorphism Example: Vehicle, Car, and Motorcycle

```
# Parent class
class Vehicle:
    def move(self):
        print("Vehicle is moving")

# Child class - Car
class Car(Vehicle):
    def move(self):
        print("Car is driving on the road")

# Child class - Motorcycle
class Motorcycle(Vehicle):
    def move(self):
        print("Motorcycle is riding on the road")

car1 = Car()
bike1 = Motorcycle()

car1.move()
bike1.move()
```

Output:

```
Car is driving on the road
Motorcycle is riding on the road
```

This demonstrates **polymorphism** using a parent class '`Vehicle`' and child classes '`Car`' and '`Motorcycle`'. Each child class overrides the '`move()`' method of the parent class to provide its own specific behavior.

In this example, both '`Car`' and '`Motorcycle`' share the same method name '`move()`' from the parent class '`Vehicle`', but each provides a different implementation. This is a classic example of **polymorphism** in object-oriented programming.

3.4.14 Polymorphism Example: Polymorphism via Duck Typing

In Python, polymorphism is often achieved through a concept called **duck typing**. The idea is: "If it walks like a duck and it quacks like a duck, then it must be a duck." Note: Python is a dynamically typed language, so it is less strict about object types.

In other words, Python does not require objects to share a common parent class; it only cares whether the object has the required methods. This allows functions to work with any object that implements the expected behavior.

Note: In the following examples, the class is named '`Duck`', but this is just a class name. A class does **not** need to be named 'Duck' to demonstrate the concept of duck typing.

3.4.14.1 Polymorphism via Classical Inheritance

This example demonstrates **classical inheritance polymorphism**, where child classes inherit from a common parent class and override its methods. The function works with any object of the parent class or its subclasses.

> **Note:** This example uses inheritance. All classes share a common parent '`Animal`'. The function works because of method overriding, **not duck typing**.

```
1  class Animal:
2      def speak(self):
3          print("Some generic animal sound")
4
5  class Dog(Animal):
6      def speak(self):
7          print("Woof!")
8
9  class Cat(Animal):
10     def speak(self):
11         print("Meow!")
12
13 class Duck(Animal):
14     def speak(self):
15         print("Quack!")
16
```

```
17 def make_animal_speak(animal_object):
18     animal_object.speak()
19 
20 dog = Dog()
21 cat = Cat()
22 duck = Duck()
23 
24 make_animal_speak(dog)   # Output: Woof!
25 make_animal_speak(cat)   # Output: Meow!
26 make_animal_speak(duck)  # Output: Quack!
```

Output:

```
1 Woof!
2 Meow!
3 Quack!
```

3.4.14.2 Polymorphism via Duck Typing

In Python, **duck typing** allows polymorphism without requiring a common parent class. As long as an object has the required method, it can be used by a function.

> **Note:** None of the classes share a common parent. The function works with all objects because they implement the required ‘speak()‘ method. This is true **duck typing**.

```
1  class Dog:
2      def speak(self):
3          print("Woof!")
4
5  class Cat:
6      def speak(self):
7          print("Meow!")
8
9  class Robot:
10     def speak(self):
11         print("Beep Boop!")
12
13 def make_animal_speak(animal_object):
14     animal_object.speak()
15
16
17
```

```
dog = Dog()
cat = Cat()
robot = Robot()

make_animal_speak(dog)    # Output: Woof!
make_animal_speak(cat)    # Output: Meow!
make_animal_speak(robot)  # Output: Beep Boop!
```

Output:

```
Woof!
Meow!
Beep Boop!
```

3.5 Summary: Introduction to Object-Oriented Programming

This topic introduces **Object-Oriented Programming (OOP)**, a programming paradigm centered around "objects" rather than separate functions and data. An object bundles its own data (attributes) and behaviors (methods) into a single unit.

The two core concepts are:

- **Class:** The blueprint or template for creating objects (e.g., `Student`, `Animal`, `Lion`).
- **Object:** A specific instance of a class (e.g., a `Student` object named `student1`).

A class is defined using the **`__init__()` constructor** to initialize an object's attributes (like `self.name`) and **methods** (like `def speak(self)`) to define its behaviors.

3.5.1 Four Pillars of OOP:

1. **Encapsulation:** The practice of bundling data and methods together and restricting access to an object's internal data. This is done by making attributes **private** (e.g., `self.__balance`). This protects the data from being changed incorrectly, forcing the use of public methods (like `deposit()`) instead.
2. **Abstraction:** Hiding complex implementation details and exposing only a simple interface. The `CoffeeMachine` example illustrates this: the user only needs to call `make_espresso()`, and all the complex private steps (`__grind_beans`, `__heat_water`) are hidden.

3. **Inheritance:** A mechanism where a new class (child class) can inherit attributes and methods from an existing class (parent class). This creates an *"is-a"* relationship (e.g., a `Dog` *is-a* `Animal`) and promotes code reuse.

4. **Polymorphism:** The ability of different objects to respond to the same method call in their own specific way. For example, a `Dog` object and a `Cat` object can both use a `.speak()` method, but one outputs "Woof!" and the other "Meow!". In Python, this is often achieved via "Duck Typing."

Topic 4:

Working with Objects and Structuring Programs

Topic 4: Working with Objects and Structuring Programs

4.1 Modeling Objects with Classes: Attributes and Behaviors

Classes enable programmers to model real-world objects by encapsulating their **attributes** (data) and **behaviors** (operations).

These are represented in code through **data members** and **member functions**—or, in Python terminology, **attributes** and **methods**.

Once a class has been defined, its name can be used to **create** objects (instances) of that class type.

- **Data Members (Attributes):** Variables within a class that represent an object's state or properties. Data members store information about the object and are typically accessed or modified through member functions (methods). They describe what the object *has*.
- **Member Functions (Methods):** Functions defined within a class that specify the behavior or operations of the object. Methods enable interaction with and manipulation of data members, performing actions or computations. They describe what the object *does*.

In general OOP terminology (e.g., C++ or Java), we refer to *data members* and *member functions*. In Python, however, the common terms are *attributes* and *methods*. Both refer to the same underlying concepts: attributes hold data, and methods define behavior.

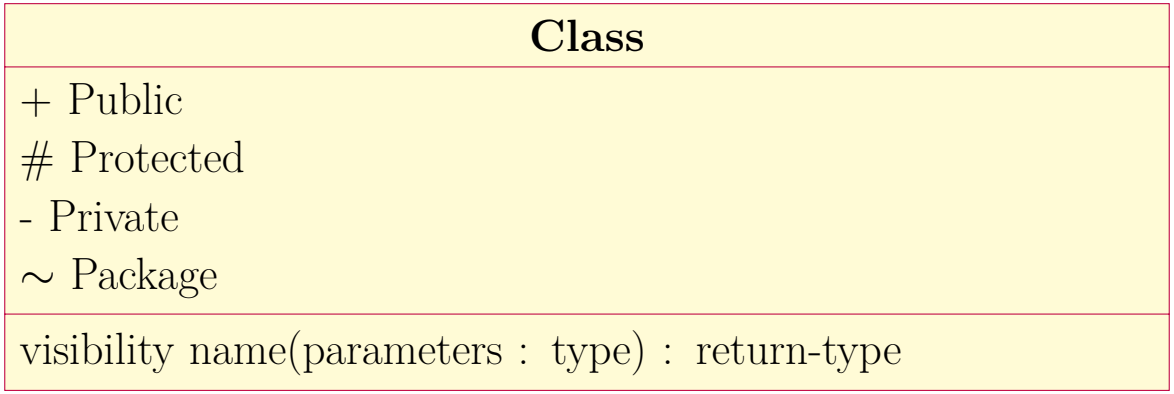

Figure 9: Class Diagram with Attributes
(Public, Protected, Private, Package)

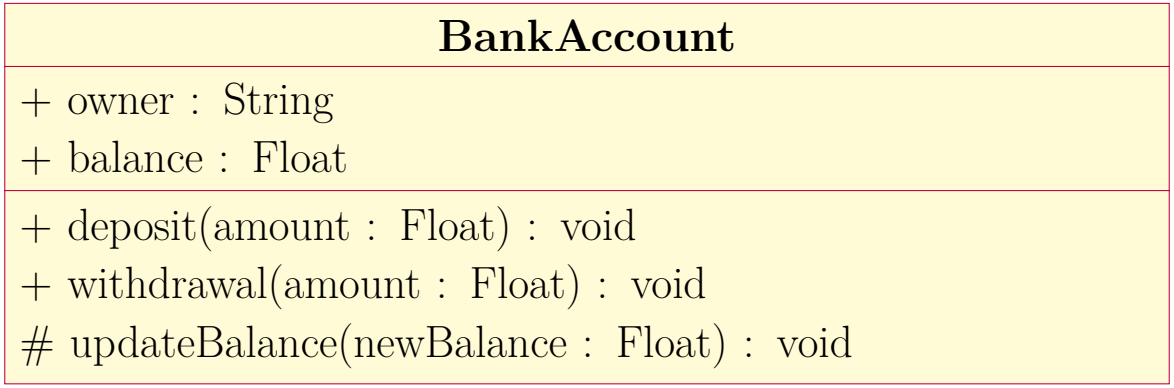

Figure 10: BankAccount Class with Attributes and Operations

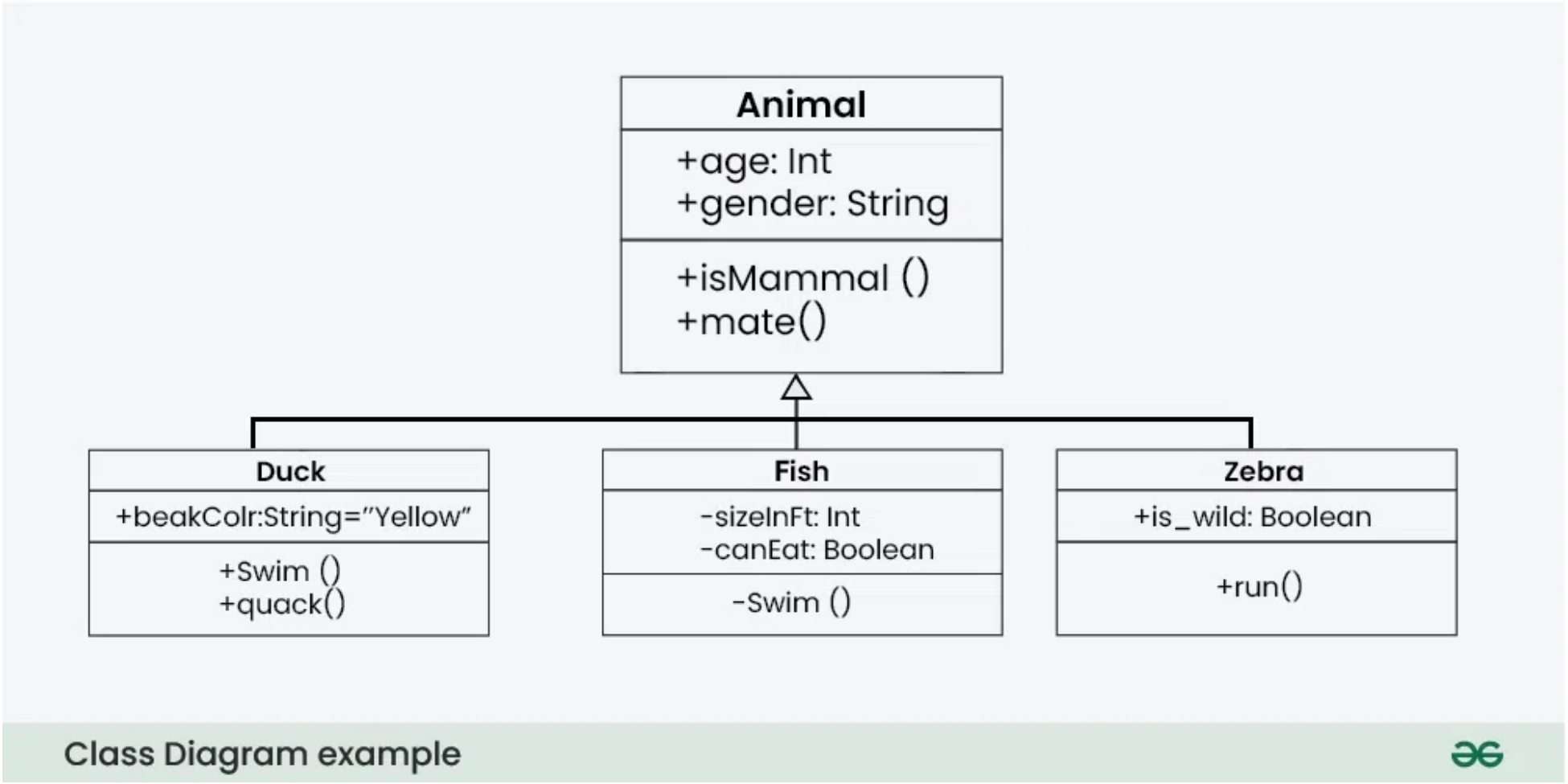

Figure 11: UML Class Diagram

Source: geeksforgeeks.org

4.2 Designing with Class Diagrams (UML)

Before building a house, you need a blueprint. In OOP, our blueprint is often a **UML Class Diagram**.

UML (Unified Modeling Language) is a standard visual language used to design and document software systems. A class diagram shows the structure of our classes, their attributes, their methods, and their relationships.

However, UML is not limited to class diagrams. It provides a variety of diagram types to represent different aspects of a system. For example, **Use Case Diagrams** describe system functionality from a user's perspective, **Sequence Diagrams** show interactions between objects over time, and **Activity Diagrams** model workflows or processes. Together, these diagrams offer a comprehensive way to visualize and communicate software design.

A basic class diagram is a box with three sections:

- **Section 1 (Top):** The Class Name (e.g., `Student`)
- **Section 2 (Middle):** Attributes (the data, e.g., `name`)
- **Section 3 (Bottom):** Methods (the behaviors, e.g., `get_info()`)

We also use **visibility notation** to show which members are public or private:

- `+` (Public): Can be accessed from anywhere. (e.g., `+name`)
- `-` (Private): Can only be accessed from within the class. In Python, this corresponds to **private** attributes (e.g., `__balance`).
- `#` (Protected): Can be accessed within the class and its subclasses. In Python, this corresponds to **protected** attributes (e.g., `_area`).

In UML class diagrams, we usually do *not* include underscores (such as `_` or `__`) in attribute names. The visibility is indicated entirely by the notation (`+`, `-`, `#`) rather than by the naming convention. For example, an attribute might simply appear as `- balance` or `# area`, without underscores.

4.2.1 UML Class Diagram Example: `Student` Class

This diagram shows the blueprint for the **`Student`** class we saw in Topic 3.

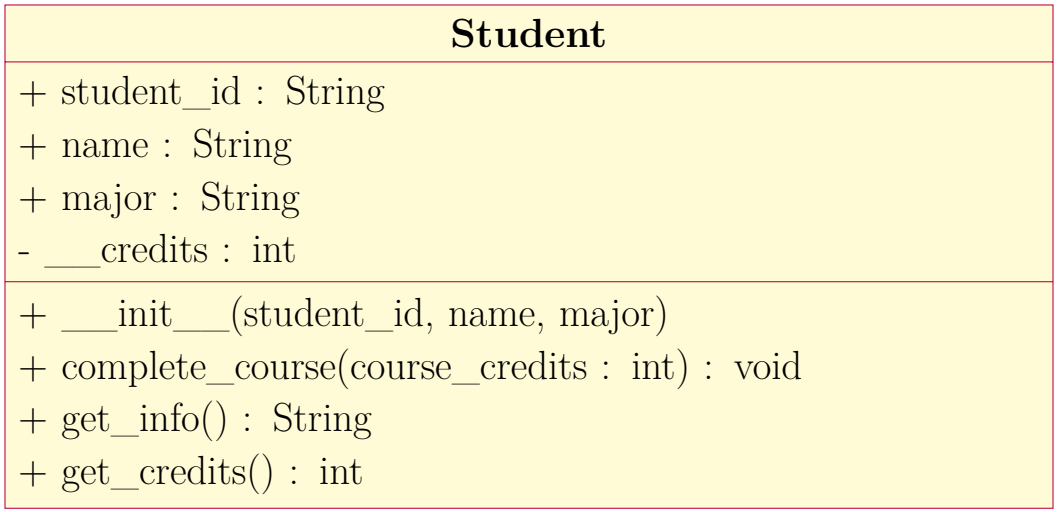

Figure 12: UML Class Diagram for a `Student` class. Note the `+` and `-` visibility markers.

This diagram clearly communicates that **`student_id`**, **`name`**, and **`major`** are public, while **`__credits`** is private. It also shows the public methods available to use.

4.2.2 Code Implementation for the Student UML

Here is the complete Python code that implements the `Student` class blueprint from the UML diagram. Notice how the `+` and `-` symbols map to **public** (e.g., `self.name`) and **private** (e.g., `self.__credits`) members. Type hints are also included to match the UML specification strictly.

```
# This class definition matches the UML diagram, including types.
class Student:
    # + __init__(student_id: str, name: str, major: str)
    def __init__(self, student_id: str, name: str, major: str) -> None:
        # + student_id : String
        self.student_id: str = student_id
        # + name : String
        self.name: str = name
        # + major : String
        self.major: str = major
        # - __credits : int
        self.__credits: int = 0  # Private attribute, initialized to 0

    # + complete_course(course_credits: int) : None
    def complete_course(self, course_credits: int) -> None:
        """Adds credits for a completed course."""
        if course_credits > 0:
            self.__credits += course_credits
            print(f"{self.name} completed a course for {course_credits} credits.")
        else:
```

```
            print("Invalid credit amount.")

    # + get_info() : str
    def get_info(self) -> str:
        """Returns formatted student information."""
        return f"ID: {self.student_id}, Name: {self.name}, Major: {self.major}"

    # + get_credits() : int
    def get_credits(self) -> int:
        """Public getter for private attribute __credits."""
        return self.__credits

# --- Main part of the script ---
# We use this block to test the class
if __name__ == "__main__":
    # Create two objects (instances) from the Student class
    student1: Student = Student("S001", "Alice", "Computer Science")
    student2: Student = Student("S002", "Bob", "Physics")

    # Interact with the objects using their public methods
    student1.complete_course(3)
    student1.complete_course(4)
    student2.complete_course(4)

    print("\n--- Student Details ---")
    print(student1.get_info())
```

```
47      print(f"Total Credits: {student1.get_credits()}")
48
49      print(student2.get_info())
50      print(f"Total Credits: {student2.get_credits()}")
51
52      # Attempting to access private attribute (will fail)
53      try:
54          print("\nAttempting to access private credits...")
55          print(student1.__credits)
56      except AttributeError as e:
57          print(f"Error: {e}")
```

Output:

```
1  Alice completed a course for 3 credits.
2  Alice completed a course for 4 credits.
3  Bob completed a course for 4 credits.
4
5  --- Student Details ---
6  ID: S001, Name: Alice, Major: Computer Science
7  Total Credits: 7
8  ID: S002, Name: Bob, Major: Physics
9  Total Credits: 4
10
11 Attempting to access private credits...
12 Error: 'Student' object has no attribute '__credits'
```

4.2.3 UML Class Diagram Example: Inheritance (`Person` and `Student`)

This example demonstrates **inheritance**, one of the core principles of Object-Oriented Programming (OOP). The `Student` class inherits from the `Person` class, meaning it reuses and extends the attributes and behaviors of its parent.

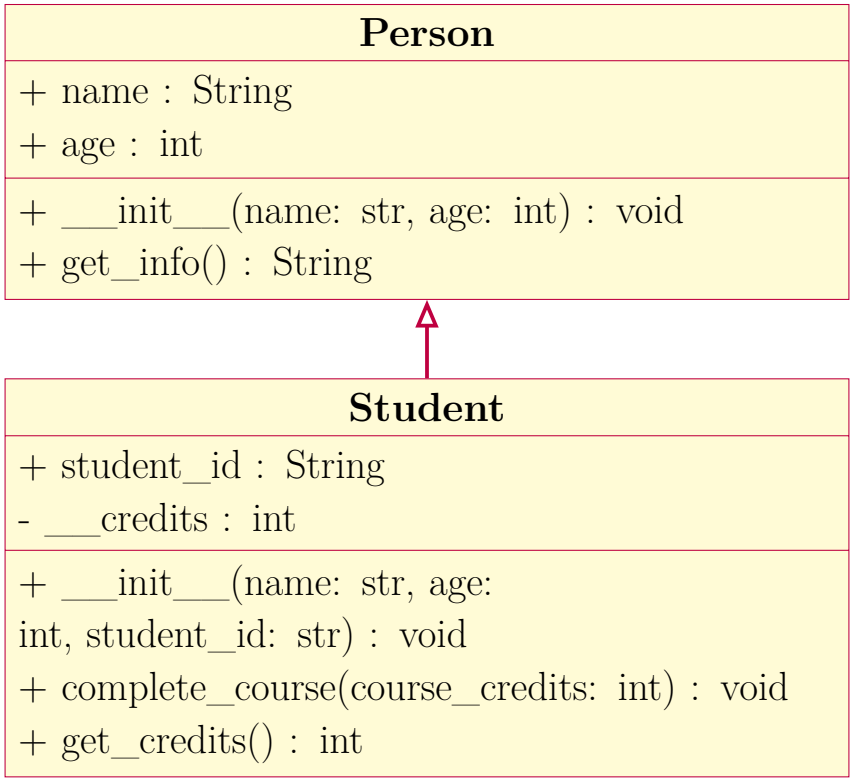

Figure 13: UML Class Diagram showing inheritance between `Person` (parent) and `Student` (child).

The arrow connecting the child class to the parent represents an **"is-a"** relationship, which indicates inheritance between a subclass and its parent class. Other types of lines in UML diagrams represent different kinds of relationships — not all lines imply an "is-a" (inheritance) relationship.

The diagram shows that `Student` inherits all public attributes and methods from `Person`, while adding its own data and behavior.

4.2.4 Code Implementation for the `Person`–`Student` UML

The following Python code implements the UML diagram above. Notice how the `Student` class uses the `super()` function to call the parent class's constructor.

```python
# Base class (Parent)
class Person:
    # + __init__(name: str, age: int)
    def __init__(self, name: str, age: int) -> None:
        # + name : String
        self.name: str = name
        # + age : int
        self.age: int = age

    # + get_info() : str
    def get_info(self) -> str:
        """Return basic information about the person."""
        return f"Name: {self.name}, Age: {self.age}"

# Derived class (Child)
class Student(Person):
    # + __init__(name: str, age: int, student_id: str)
    def __init__(self, name: str, age: int, student_id: str) -> None:
        # Call the parent class constructor
        super().__init__(name, age)
        # + student_id : String
```

```
22          self.student_id: str = student_id
23          # - __credits : int
24          self.__credits: int = 0
25
26      # + complete_course(course_credits: int) : None
27      def complete_course(self, course_credits: int) -> None:
28          """Adds credits for a completed course."""
29          if course_credits > 0:
30              self.__credits += course_credits
31              print(f"{self.name} earned {course_credits} credits.")
32          else:
33              print("Invalid credit amount.")
34
35      # + get_credits() : int
36      def get_credits(self) -> int:
37          """Public getter for private attribute __credits."""
38          return self.__credits
39
40  # --- Testing the classes ---
41  if __name__ == "__main__":
42      student1: Student = Student("Alice", 20, "S001")
43      student2: Student = Student("Bob", 22, "S002")
44
45      # Call inherited and new methods
46      print(student1.get_info())  # Inherited from Person
47      student1.complete_course(3)
```

```
48      print(f"Total Credits: {student1.get_credits()}")
49      print(f"Type of student1: {type(student1)}")  # Print type
50
51      print(student2.get_info())
52      student2.complete_course(4)
53      print(f"Total Credits: {student2.get_credits()}")
54      print(f"Type of student2: {type(student2)}")  # Print type
```

Output:

```
1  Name: Alice, Age: 20
2  Alice earned 3 credits.
3  Total Credits: 3
4  Type of student1: <class '__main__.Student'>
5  Name: Bob, Age: 22
6  Bob earned 4 credits.
7  Total Credits: 4
8  Type of student2: <class '__main__.Student'>
```

This example illustrates how inheritance allows code reuse. The `Student` class automatically gains the properties and behaviors of `Person`, while still being able to define its own additional attributes and methods.

Note: The objects `student1` and `student2` are instances of the same type, `Student`, which means they share the same structure and behavior defined in the class.

4.3 Objects and Instances

In Object-Oriented Programming, it is important to distinguish between a **class** and the concrete entities it creates.

Object: An **object** is a concrete, individual entity created based on a class. It has a specific state (the values of its attributes) and behavior (the methods it can perform). Essentially, an object is a real-world representation of the abstract concept defined by a class.

Instance: An **instance** is another way to describe an object. When you create an object from a class, you are said to "instantiate" the class. Every object is therefore an **instance** of a particular class.

Example in Python:

```
1 student1 = Student("Alice", 20, "S001")  # student1 is an object
```

Key Points:

- **Class:** A blueprint or template.
- **Object/Instance:** A concrete entity created from a class.
  - **Object**: Emphasizes the entity itself that exists in memory.
  - **Instance**: Emphasizes that the entity is created from a specific class.
- Multiple objects can be created from the same class, each with its own state.

4.4 Organizing Code: Objects in Separate Files (Modules)

As programs grow larger, putting all your classes and main program logic into a single `.py` file quickly becomes messy and hard to maintain.

This is considered bad practice. The solution is **Separation of Concerns (SoC)**[6]: each class can be placed in its own file (a **module**).

- **Why?** It makes your code more readable, maintainable, and reusable. For example, you can import your `Student` class into multiple programs without duplicating code.
- **How?** Use the `import` statement to connect files. Example:

```
1 from school_entities import Student
2 student1 = Student("Alice", 20, "S001")
```

- **Case Sensitivity:** Both the file name and class name are case-sensitive.
  When using `import`, the names must match exactly. For example, `School_Entities.py` is different from `school_entities.py`, and `Student` is different from `student`.
- **Multiple Classes per File:** While a single file can contain multiple classes, such as `Classroom` and `Student` in `school_entities.py`, it is generally avoided for clarity. Keeping one class per file improves code readability and maintenance.

[6]Separation of Concerns (SoC) is a software design principle that recommends organizing code so that each part addresses a distinct concern, improving modularity, readability, and maintainability.

4.4.1 Example: Separating the Student Class into Modules

Let's split our program into two files: `school_entities.py` and `main.py`.

File 1: `school_entities.py`

```python
# school_entities.py - defines only the Student class; can be imported as a module
class Student:
    def __init__(self, student_id, name, major):
        self.student_id = student_id
        self.name = name
        self.major = major
        self.__credits = 0  # Private attribute

    def complete_course(self, course_credits):
        if course_credits > 0:
            self.__credits += course_credits
            print(f"{self.name} completed a course.")
        else:
            print("Invalid credit amount.")

    def get_info(self):
        return f"ID: {self.student_id}, Name: {self.name}, Major: {self.major}"

    def get_credits(self):
        return self.__credits
```

File 2: main.py

```python
# main.py - main program - Import the Student class from school_entities.py
from school_entities import Student

# Create instances (objects) of the Student class
student1 = Student("S001", "Alice", "Computer Science")
student2 = Student("S002", "Bob", "Physics")

# Interact with the objects using their methods
student1.complete_course(3)
student2.complete_course(4)

print(student1.get_info())
print(f"Credits: {student1.get_credits()}")

print(student2.get_info())
print(f"Credits: {student2.get_credits()}")
```

Output:

```
Alice completed a course.
Bob completed a course.
ID: S001, Name: Alice, Major: Computer Science
Credits: 3
ID: S002, Name: Bob, Major: Physics
Credits: 4
```

4.4.2 The `if __name__ == "__main__":` Block

Before splitting code into separate modules, all classes and test code are often written in a single file. For example:

```python
# student.py - everything in one file

class Student:
    def __init__(self, student_id, name, major):
        self.student_id = student_id
        self.name = name
        self.major = major
        self.__credits = 0

    def complete_course(self, course_credits):
        self.__credits += course_credits

    def get_info(self):
        return f"ID: {self.student_id}, Name: {self.name}, Major: {self.major}"

# Test code
student1 = Student("S001", "Alice", "Computer Science")
print(student1.get_info())
```

In this case, if you try to import the `Student` class from this file into another program, the test code runs immediately, which is usually undesirable.

To prevent this, Python provides the `if __name__ == "__main__":` block. Wrapping code inside this block ensures that it **only runs when the file is executed directly**, not when it is imported as a module.

```python
# school_entities.py - class definitions only

class Student:
    ...

if __name__ == "__main__":
    # Code here runs only when this file is executed directly
    student1 = Student("S001", "Alice", "Computer Science")
    print(student1.get_info())
```

Benefits:

- When another file imports `school_entities.py`, the code inside the `if __name__ == "__main__":` block in `school_entities.py` does **not run**.
- When executing `python school_entities.py`, the test code inside the `if` block **will run**, which is perfect for testing classes in isolation.

4.5 Managing Multiple Objects (Lists of Objects)

In a real application, you don't just have one or two objects—you have hundreds or thousands. The most common way to manage them is by putting them in a **list**.

This is where OOP becomes truly powerful. You can manage complex data simply by managing a list of objects.

4.5.1 Creating a List of Objects

We can easily create many objects using a loop and append them to a list.

```
from school_entities import Student

# A list to hold all of our student objects
student_roster = []

# Data we might get from a file or database
student_data = [
    ("S001", "Alice", "Computer Science"),
    ("S002", "Bob", "Physics"),
    ("S003", "Charlie", "Mathematics")
]

# Loop through the data and create an object for each student
for s_id, s_name, s_major in student_data:
```

```
    # Create the object
    new_student = Student(s_id, s_name, s_major)
    # Add the new object to our list
    student_roster.append(new_student)

print(f"Created {len(student_roster)} student objects.")
```

Output:

```
Created 3 student objects.
```

4.5.2 Removing Objects Safely from a List

Modifying a list while iterating over it directly can lead to **skipped elements** or **unexpected behavior**. We can avoid this by iterating over a **copy** of the list.

```
# Create a roster with consecutive IDs for demonstration
student_roster = [
    Student("S001", "Alice", "CS"),
    Student("S002", "Bob", "Physics"),
    Student("S002", "Brian", "Math"),
    Student("S003", "Charlie", "CS")
]

# Unsafe removal: directly modifying the list while iterating
for student in student_roster:
    if student.student_id == "S002":
        student_roster.remove(student)

print("After unsafe removal:", [s.student_id for s in student_roster])

# Recreate the roster
student_roster = [
    Student("S001", "Alice", "CS"),
    Student("S002", "Bob", "Physics"),
    Student("S002", "Brian", "Math"),
    Student("S003", "Charlie", "CS")
]
```

```
23
24 # Safe removal: iterate over a copy
25 for student in student_roster[:]:  # shallow copy
26     if student.student_id == "S002":
27         student_roster.remove(student)
28
29 print("After safe removal:", [s.student_id for s in student_roster])
```

Output:

```
1 After unsafe removal: ['S001', 'S002', 'S003']
2 After safe removal: ['S001', 'S003']
```

Explanation:

- **Unsafe removal:** When modifying the list while iterating, the second student with ID "S002" was skipped. This demonstrates why direct removal is risky.
- **Safe removal:** Iterating over a **shallow copy** (`student_roster[:]`) is sufficient for removing objects from the outer list safely. All matching students are removed correctly.
- For nested modifications, a deep copy would be required, but for filtering or removing top-level objects, a shallow copy is enough.

4.6 Comparing Objects in Python

4.6.1 Mixed Types Comparison Example

Before comparing custom objects, it helps to understand how Python handles object comparison in general.

Simple Example with Mixed Types:

```
a = [1, "hello", [3, 4]]
b = [1, "hello", [3, 4]]
c = a

print(a == b)  # True: same contents (value comparison)
print(a is b)  # False: different outer list objects
print(a is c)  # True: same object in memory

# Comparing individual elements
print(a[0] is b[0])      # True: small integers are immutable and may share memory
print(a[1] is b[1])      # True: short strings may be interned and share memory
print(a[2] is b[2])      # False: inner lists are mutable, different objects
print(a[0] is a[-1])     # False: first element vs last element (different types)
```

Output (example):

```
True
False
True
True
True
False
False
```

Explanation:

- `==` compares the **values stored inside** the objects recursively (list contents and nested elements).
- `is` checks if two variables point to the **same object** in memory.
- Immutable objects like small integers or short strings may share memory, so `a[0] is b[0]` or `a[1] is b[1]` can be `True`.
- Mutable objects like lists always create a new object, so `a[2] is b[2]` is `False`.
- Mixed types in a list illustrate how Python handles identity and value comparison differently depending on mutability.

4.6.2 String and Tuple Comparison Example

Strings and tuples are **immutable** objects in Python. Python may reuse some immutable objects in memory (string interning, tuple caching), which affects `is` comparisons.

```
# String comparison
s1 = "hello"
s2 = "hello"
s3 = "".join(["he", "llo"])  # creates a new string object dynamically

print("String comparisons:")
print(s1 == s2)  # True: same sequence of characters (value comparison)
print(s1 is s2)  # True: may be same object due to interning
print(s1 == s3)  # True: same text value
print(s1 is s3)  # False: different object, created at runtime

# Tuple comparison
t1 = (1, 2, 3)
t2 = (1, 2, 3)
t3 = t1

print("\nTuple comparisons:")
print(t1 == t2)  # True: same contents
print(t1 is t2)  # False: different tuple objects
print(t1 is t3)  # True: same object reference


```

```
# Number comparison
n1 = 256
n2 = 256
n3 = 257
n4 = 257

print("\nNumber comparisons:")
print(n1 is n2)  # True: small integers are cached
print(n3 is n4)  # False: larger integers are different objects
```

Output:

```
String comparisons:
True
True
True
False

Tuple comparisons:
True
False
True

Number comparisons:
True
False
```

Explanation:

- `==` checks if the contents of the object are the same (value comparison), works for strings, tuples, and numbers.
- `is` checks whether two variables reference the same object in memory (identity comparison).
- Strings and tuples are immutable; Python may reuse some small immutable objects in memory.
- For numbers, Python caches small integers (typically -5 to 256), so `n1 is n2` is True, but larger numbers are distinct objects (`n3 is n4` is False).
- This contrasts with strings: short string literals are interned, so `s1 is s2` can be True, while dynamically created strings are different objects (`s1 is s3` is False).

4.6.3 Advanced Example: Mutable vs. Immutable Objects

```
# Immutable example (tuples)
t1 = (1, 2, 3)
t2 = (1, 2, 3)
print(t1 == t2)  # True: same contents
print(t1 is t2)  # False: different tuple objects

# Mutable example (lists)
l0 = [3, 4]           # shared inner list
l1 = [1, 2, l0]       # list containing l0
l2 = [1, 2, l0]       # another list containing the same l0 object
l3 = l1               # l3 references the same object as l1
print(l1 == l2)       # True: same structure and values
print(l1 is l2)       # False: l1 and l2 are different outer lists
print(l1 is l3)       # True: l3 is the same object as l1

l0.append(5)          # Mutating the shared inner list affects both l1 and l2
print(l1)             # [1, 2, [3, 4, 5]]
print(l2)             # [1, 2, [3, 4, 5]]
print(l3)             # [1, 2, [3, 4, 5]]

# Compare again after modification
print(l1 == l2)       # Still True: same nested structure and values
print(l1 is l2)       # Still False: different outer list objects
print(l1[2] is l2[2]) # True: both share the same inner list object (l0)
```

Output:

```
True
False
True
False
True
[1, 2, [3, 4, 5]]
[1, 2, [3, 4, 5]]
[1, 2, [3, 4, 5]]
True
False
True
```

Explanation:

- Both `l1` and `l2` contain the same inner list (`l0`), so modifying `l0` updates both structures.
- After modification, `l1 == l2` remains `True` because their contents are still equivalent.
- `l1 is l2` remains `False`, since they are different outer list objects in memory.
- `l1[2] is l2[2]` is `True`, showing that both lists share the same inner list object reference.
- This demonstrates how mutable shared objects can cause side effects when modified.

Understanding these reference relationships is essential when comparing complex data structures or designing functions that modify objects in place.

4.6.4 Recursive and Cross-Reference Example

Python allows data structures to reference themselves or each other. This can lead to complex, cyclic relationships where equality and identity comparisons behave differently.

```
# Self-referential (recursive) list
a = [1, 2]
a.append(a)  # list contains itself

print(a)       # [1, 2, [...]]
print(a is a[2])  # True: the third element refers to 'a' itself
print(a == a)     # True: same object

# Cross-referential lists
x = [1, 2]
y = [3, 4]
x.append(y)
y.append(x)

print(x)  # [1, 2, [3, 4, [...]]]
print(y)  # [3, 4, [1, 2, [...]]]

# Comparison and identity checks
print(x == y)       # False: different structure and order
print(x[2] is y)    # True: x contains a reference to y
print(y[2] is x)    # True: y contains a reference to x
```

Output:

```
[1, 2, [...]]
True
True
[1, 2, [3, 4, [...]]]
[3, 4, [1, 2, [...]]]
False
True
True
```

Explanation:

- In the first case, list `a` contains a reference to itself — printing it shows `[...]` to avoid infinite recursion.
- `a is a[2]` confirms that the self-reference points to the same object.
- In the cross-reference case, `x` and `y` each contain a reference to the other. This creates a cyclic data structure where equality (`==`) stops recursion once cycles are detected.
- Python's comparison mechanisms are designed to safely handle these cycles without causing infinite loops.

Recursive and cross-referential structures appear in advanced use cases such as graph representations, linked data structures, or circular dependencies. They also demonstrate why understanding identity and reference behavior is essential when designing complex, mutable objects.

Note: Operator Behavior May Vary

Although the general rules for `is` and `==` apply to most objects, their behavior is not universal. Python allows classes and data structures to **override** or **reinterpret** comparison and membership operations.

- The equality operator `==` can behave differently if a class defines a custom method such as `__eq__()`. This allows developers to specify what "equality" means for their objects.
- The identity operator **`is` cannot be overridden**; it always checks memory identity. However, some types (e.g., small integers or short strings) may appear identical because Python internally reuses immutable objects for efficiency.
- Membership tests using `in` also depend on the data type. For example:
  - `x in list` checks for value equality (`==`) within the list.
  - `x in dict` checks whether `x` is a **key** in the dictionary, not a value.
- Similarly, comparison operators like `<`, `>`, `<=`, and `>=` can be customized using special methods such as `__lt__()`, `__gt__()`, etc.

In short, while these operators have standard meanings, their actual behavior can differ depending on how an object's class is implemented.

4.6.5 Comparing Custom Objects

In Python, how two objects are compared depends on the operator used:

- **Identity comparison (`is`):** Checks whether two variables refer to the **exact same object** in memory.
- **Value (attribute) comparison:** Compares the **data stored inside the objects**—for example, the strings, numbers, or lists that make up their attributes—rather than the objects themselves.

> **Note:** The equality operators `==` and `!=` compare the values held by objects, not their identities. Relational operators such as `<`, `>`, `<=`, and `>=` can also be used when the underlying data types support ordering (e.g., numbers or strings).

Key Points:

- By default, `==` behaves like `is` unless the class defines a custom method such as `__eq__()`.
- The `is` operator **always** checks if two references point to the same object in memory.
- Value comparison focuses on the actual contents (the data) of objects, not their identity.
- Returning the full object from a search function is recommended—it keeps all its attributes and methods accessible.

Example:

```
student1 = Student("S001", "Alice", "CS")
student2 = Student("S001", "Alice", "CS")  # same data, different object
student3 = student1                        # same object as student1

print(student1 == student2)  # False: different objects, unless __eq__ is defined
print(student1 is student2)  # False: different objects
print(student1 == student3)  # True: same object
print(student1 is student3)  # True: same object

# Compare values stored in attributes
if student1.student_id == student2.student_id:
    print("Same student ID")

# Compare values (int) stored in attributes
# if student1.age > student2.age:
#     print("student1 is older")  # Executes only if condition is True
```

Output:

```
False
False
True
True
Same student ID
```

4.6.6 Interacting with a List of Objects

Once objects are in a list, we can loop through them and call their methods.

```
# (Assuming student_roster list from previous example)

# Let's give all students 3 credits for a new course
for student in student_roster:
    student.complete_course(3)

print("\n--- Student Roster ---")
# Now let's print info for all students
for student in student_roster:
    print(student.get_info())
```

Output:

```
Alice completed a course.
Bob completed a course.
Charlie completed a course.

--- Student Roster ---
ID: S001, Name: Alice, Major: Computer Science
ID: S002, Name: Bob, Major: Physics
ID: S003, Name: Charlie, Major: Mathematics
```

4.6.7 Finding an Object in a List

You can write a simple function to search the list for a specific object.

```
# (Assuming student_roster list from previous example)

def find_student_by_id(student_list, target_id):
    for student in student_list:
        # We can access public attributes for comparison
        if student.student_id == target_id:
            return student  # Return the whole object
    return None # Not found

# --- Search for a student ---
target_id = "S002"
found_student = find_student_by_id(student_roster, target_id)

if found_student:
    print(f"\nFound student: {found_student.get_info()}")
else:
    print(f"\nStudent {target_id} not found.")
```

Output:

```
Found student: ID: S002, Name: Bob, Major: Physics
```

4.6.8 Filtering a List of Objects

We can also loop through our list to create a *new* list based on a condition.

```
# (Assuming student_roster list from previous example)
# Let's find all students in Computer Science

# Create a new, empty list for the results
cs_students = []

for student in student_roster:
    if student.major == "Computer Science":
        cs_students.append(student)

# Print the filtered list
print("\n--- Computer Science Students ---")
for student in cs_students:
    print(student.get_info())
```

Output:

```
--- Computer Science Students ---
ID: S001, Name: Alice, Major: Computer Science
```

4.7 Summary: Working with Objects and Structuring Programs

This topic provided a practical guide to working with objects, moving from simple definitions to structured program design. Here are the core concepts covered:

- **Designing with UML:** Before coding, we use **UML Class Diagrams** as blueprints. These diagrams visualize a class's **Name**, **Attributes** (data), and **Methods** (behavior). Visibility markers (`+`, `-`, `#`) are used to define public, private, and protected members.
- **Code Organization (Modules):** To avoid a single, massive file, we practice **Separation of Concerns (SoC)**. This means placing classes into their own `.py` files, known as **modules**. We use the `import` statement (e.g., `from school_entities import Student`) to use these classes in other files.
- **The `__name__` Block:** To allow a file to be used as **both** a runnable script (for testing) and an importable module, we place test code inside an `if __name__ == "__main__":` block. This code only runs when the file is executed directly, not when it is imported.
- **Managing Multiple Objects:** Real applications use many objects. The most common way to manage them is by storing them in a **list**. This allows us to:
  - **Iterate:** Loop through the list to call methods on each object (e.g., `for student in roster: student.complete_course(3)`).
  - **Find:** Search the list for a specific object based on an attribute (e.g., find a student by `target_id`).

  - **Filter:** Create a new list containing a subset of objects that meet a condition (e.g., all students in a specific `major`).

- **Comparing Objects (`is` vs. `==`):** This is a fundamental concept in Python.
  - **Identity (`is`):** Checks if two variables point to the *exact same object in memory.* (`student1 is student3` was `True`.
  - **Equality (`==`):** Checks if the objects have the same *value.* By default, this behaves like `is` for custom objects.
  - **Attribute Comparison:** To check if two different objects (like `student1` and `student2`) have the same data, we must compare their attributes directly (e.g., `student1.student_id == student2.student_id`).