Introduction to Python Programming

Lecture 3 – HCCDA-AI

Dr. Muhammad Sajjad

R.A: Wajahat Ullah

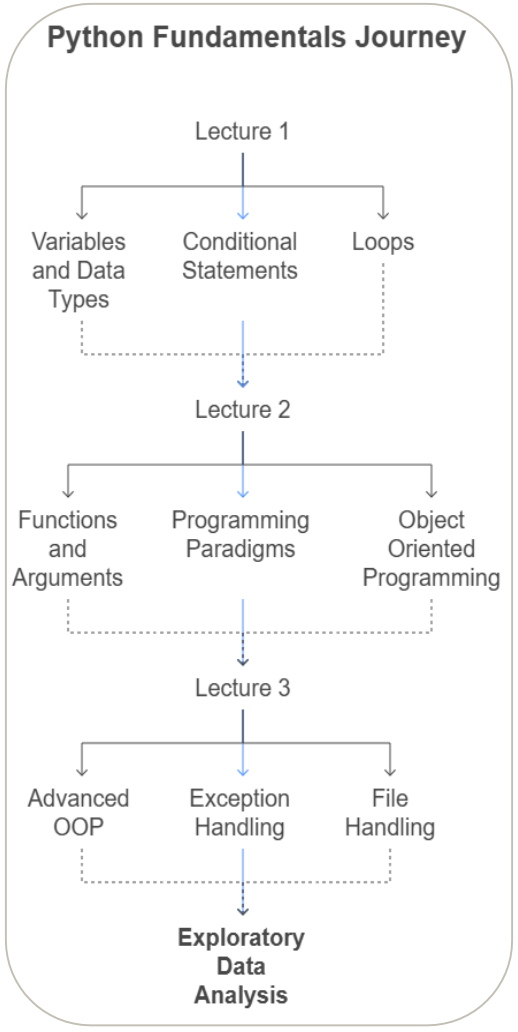
R.A: Imran Nawar

31 May 2025

Course Progress Overview

Python Fundamentals

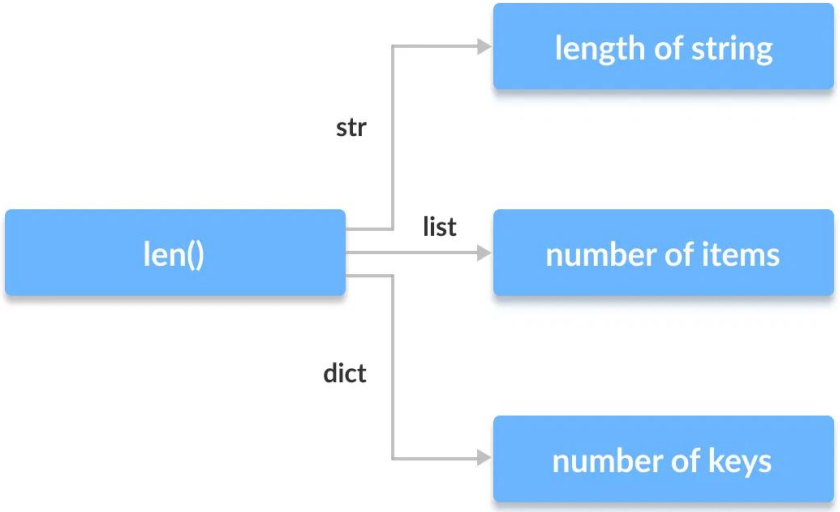
- **Lecture 1**:
  - Introduction to Programming, Installation and Setup
  - Variables, Data Types → *int, float, str, bool, list, tuple, dict*
  - Conditional Statements → *if, if-else, if-elif-else*
  - Loops → *while, for (range(), zip(), enumerate(), break, continue, pass)*
- **Lecture 2**:
  - Functions, Types of arguments → *positional , keyword, *args*
  - Programming Paradigms
  - Object Oriented Programming: Classes, Objects, Attributes, Methods
  - Constructor → __init__ Method
- **Lecture 3**:
  - Advanced OOP (Inheritance, Polymorphism, Abstraction, Encapsulation)
  - Exception Handling, File Handling
- **Coming Up Next**: **Exploratory Data Analysis (EDA)**



Polymorphism

Poly means “many” morph means “forms”

- **Polymorphism** allows the same method or operator to behave differently depending on the object’s type.
- Enable flexibility and reusability in code.
- **Key Concept:**
  - Same interface → Different behaviors depending on the object.
- **Real-world Analogy:**
  - A “Start” button works differently on different devices:
    - **Car:** Starts the engine
    - **Computer:** Boots the operating system
    - **Washing Machine:** Begins wash cycle



len() function *returns different things based on its arguments*

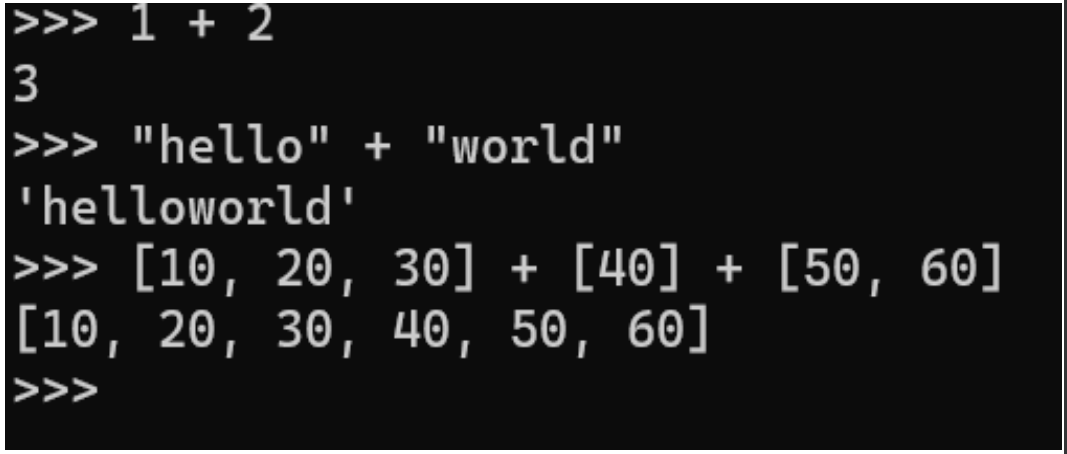
Types of Polymorphism in Python

1. Operator Overloading

- Same operator performs different operations based on operand types

Example: The + operator

- 5 + 3 → Addition (8)
- “Hello” + “World” → Concatenation (“HelloWorld”)
- [1,2] + [3,4] → List merging ([1, 2,3,4])



Types of Polymorphism

2. Method Overriding

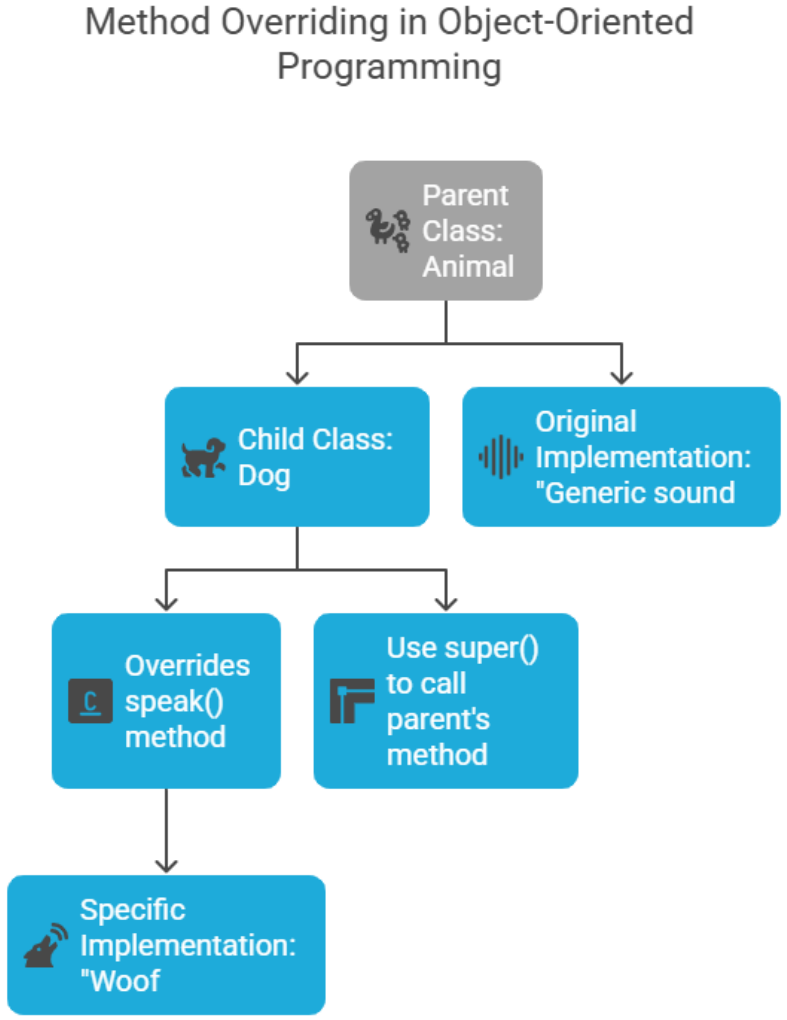
- A child class overrides a method from the parent class to provide a specific implementation.

Example:

```
class Animal:
    def speak(self):
        return "Generic sound"

class Dog(Animal):
    def speak(self):  # Overrides parent's method
        return "Woof"

animal = Animal()
dog = Dog()

print(animal.speak())  # Output: Generic sound
print(dog.speak())     # Output: Woof
```

- *Use super() if you need to call the parent's method.*

```
super().speak()
```



Types of Polymorphism

3. Duck Typing

- Objects are used based on their behavior, not their type.
- This is why Python is called “dynamically typed”, the type is determined at runtime based on the object’s behavior.

“If it walks like a duck and quacks like a duck, it’s a duck”

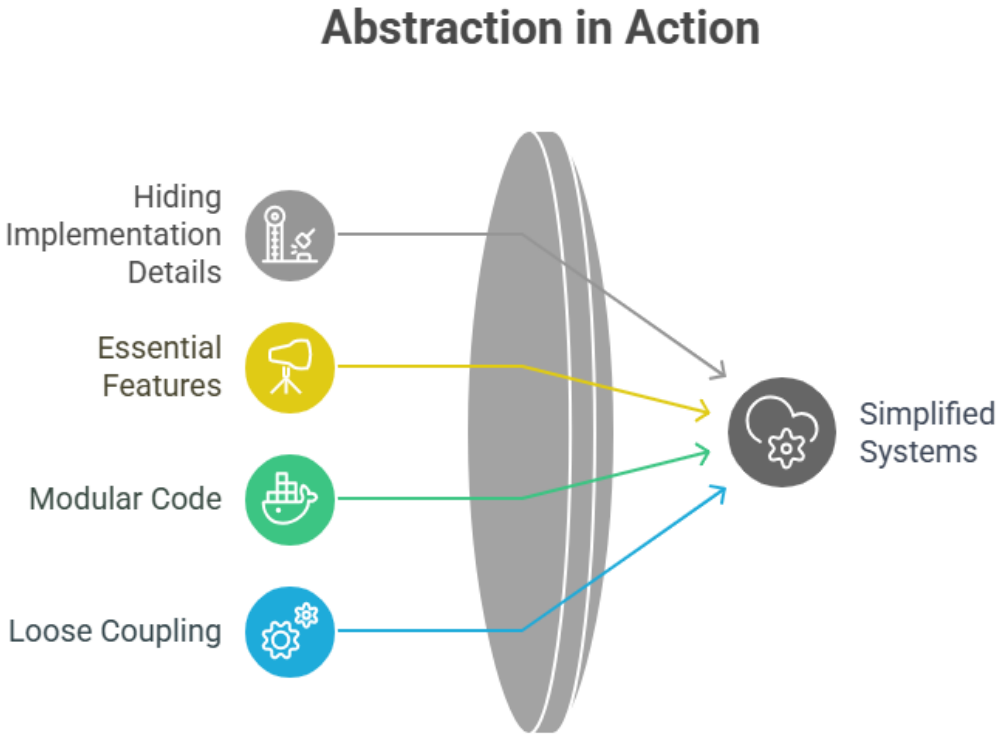
```
class Bird:
    def fly(self):
        print("Bird is flying")
class Plane:
    def fly(self):
        print("Plane is flying")
def take_off(flyable):
    flyable.fly()  # No need to know the object's class

take_off(Bird())
take_off(Plane())
```

Works because both objects implement the same method (*fly()*), even though they’re unrelated by inheritance.

Object-Oriented Programming (OOP)
Abstraction

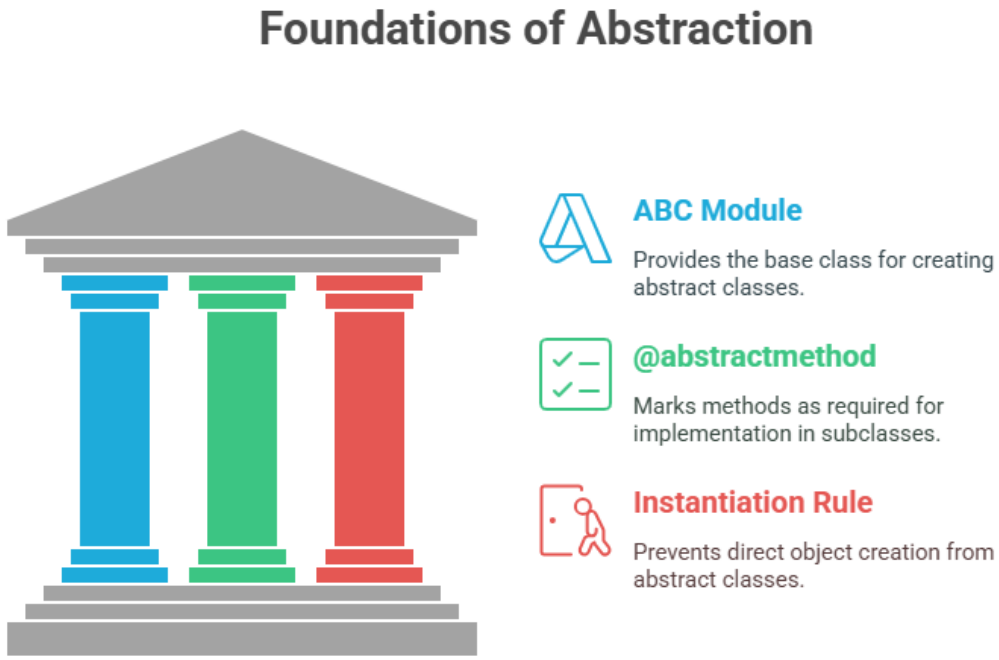
Abstraction

- Abstraction means hiding internal implementation details and showing only the essential features of an object.
  - Focus on what an object does, not how it does it.
- **Key Concepts:**
  - Simplifies concept systems by hiding the inner workings.
  - Helps in building cleaner, modular, and maintainable code.
  - Enables loose coupling between components or systems.
- **Real-world Analogy:**
  - You drive a car using the steering, pedals, and gear, but don't need to understand how the engine, fuel injection, or transmission work.



Abstraction in Python

- Python achieves abstraction via the *abc* module by creating blueprint classes that specify what methods subclasses must implement.
- **Purpose:**
  - To define a common interface that multiple classes must follow.
- **Key Components:**
  - **ABC;** Base class for creating abstract classes (from *abc* module).
  - **@abstractmethod:** Decorator that marks methods as "required" – any class inheriting from the abstract class must implement these methods.
  - **Important Rule:** Abstract classes cannot be instantiated directly (you cannot create objects from them).



Abstraction in Python

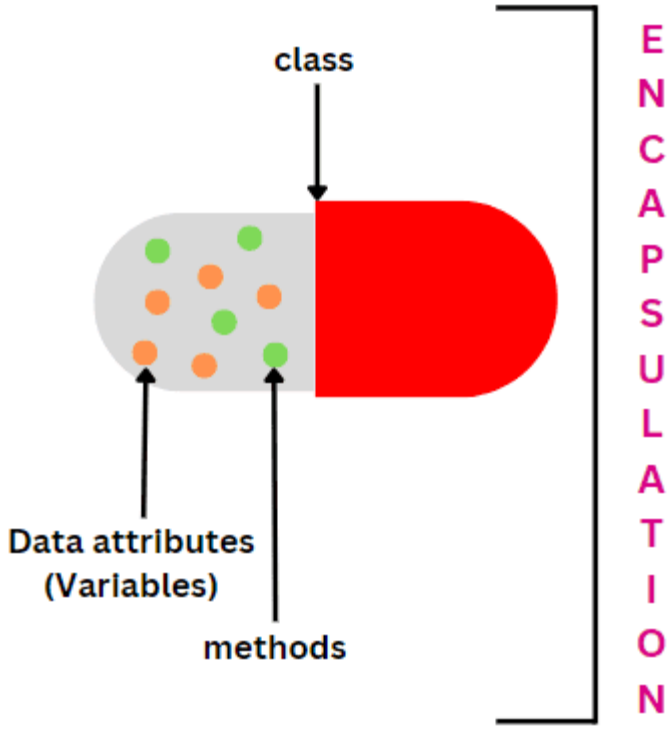
Example:

```
from abc import ABC, abstractmethod

class Animal(ABC):  # Abstract class
    @abstractmethod
    def sound(self):
        pass  # No implementation

class Dog(Animal):
    def sound(self):
        return "Woof"

class Cat(Animal):
    def sound(self):
        return "Meow"

# animal = Animal()  #  Error: Can't instantiate abstract class
dog = Dog()
print(dog.sound())  # Output: Woof
```

Object-Oriented Programming (OOP)
Encapsulation

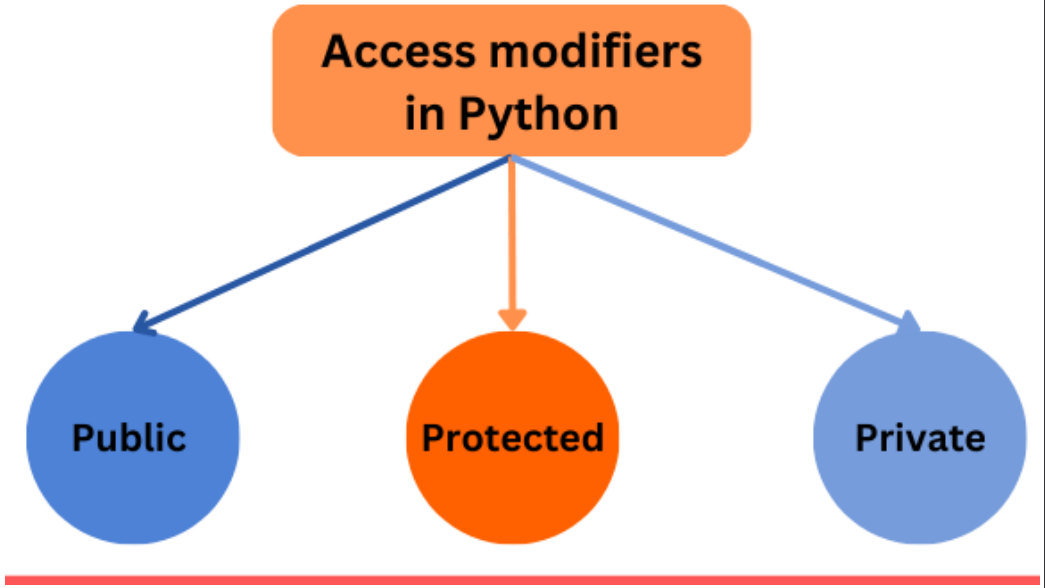
Encapsulation

- Encapsulation means putting related data and functions together in a class, like organizing items in a box, and controlling what others can access from outside.
- Encapsulation has two main ideas:
  - **Bundle together:** Keep related data and methods in one place (a class)
  - **Control access:** Decide what outsiders can see and change
- **Purpose:**
  - **Data Protection:** Prevent accidental or unauthorized modification of data.
  - **Controlled Access:** Interact with internal data only through well-defined interfaces (e.g., getter/setter methods).
  - **Implementation Hiding:** Hide internal logic from the outside world.
  - Improved Maintainability & Security.



Access Modifiers in Python

- Access modifiers control the level of access to class members.
- Unlike C++/Java, Python uses naming conventions (underscores) rather than strict keywords to achieve this.



Modifier	Syntax	Description
Public	name	Accessible from anywhere
Protected	_name	Intended for internal use; still accessible
Private	__name	Name mangling prevents direct external access

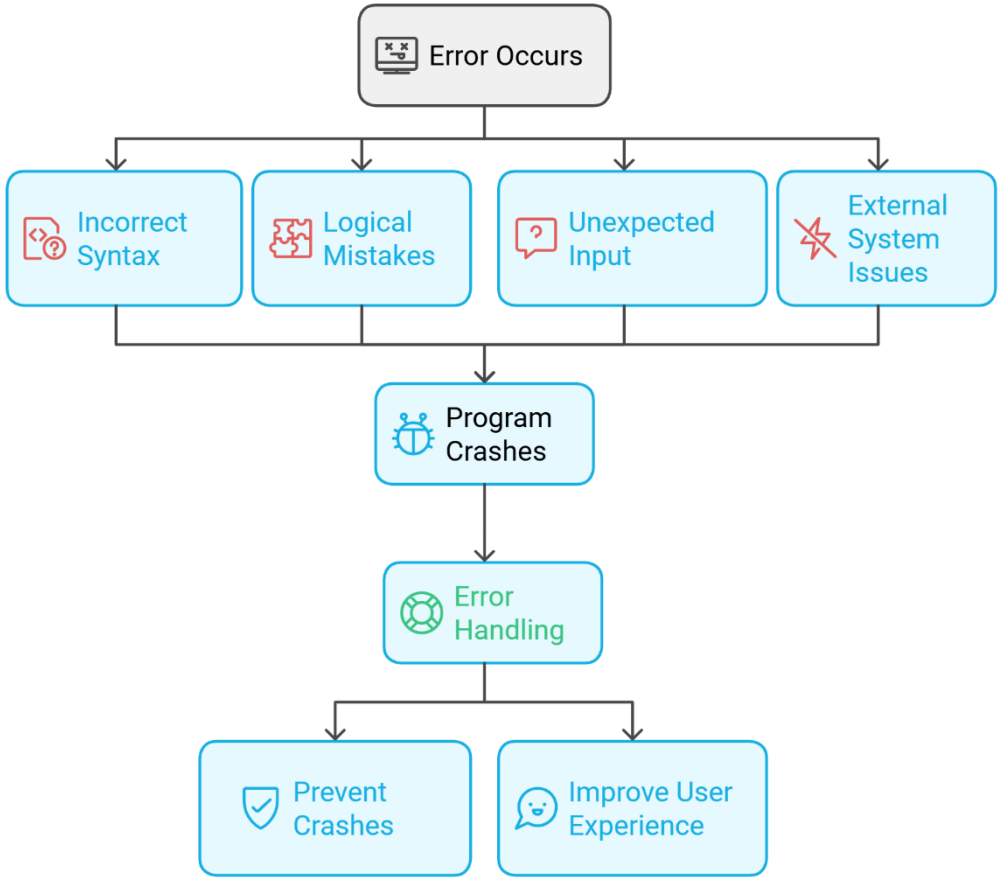
Example in Python

```
class Student:
    def __init__(self, name, age):
        self.name = name          # Public
        self.age = age            # Public
        self._grade = "A"         # Protected (by convention)
        self.__ssn = "123-45-6789" # Private (name mangled)

student = Student("John", 20)

print(student.name)      # Accessible (Public)
print(student._grade)    # Accessible (Protected, but should be avoided)
# print(student.__ssn)   # AttributeError (Private)
```

Error/Exception Handling

Errors in Programming

- **Errors** are unwanted events that disrupt the normal flow of program execution.
- They can rise from:
  - Incorrect syntax
  - Unexpected user input
  - Missing files
  - External system issues
- When an error occurs, a computer program crashes.
- **Error handling** is important to prevent program crashes and to improve user experience.



Types of Errors in Python

Three main types:

1. **Syntax error** occurs when the rules defined by the language are not followed while writing a program and is detected by python interpreter before execution.

   print("Hello World" # Missing closing parenthesis

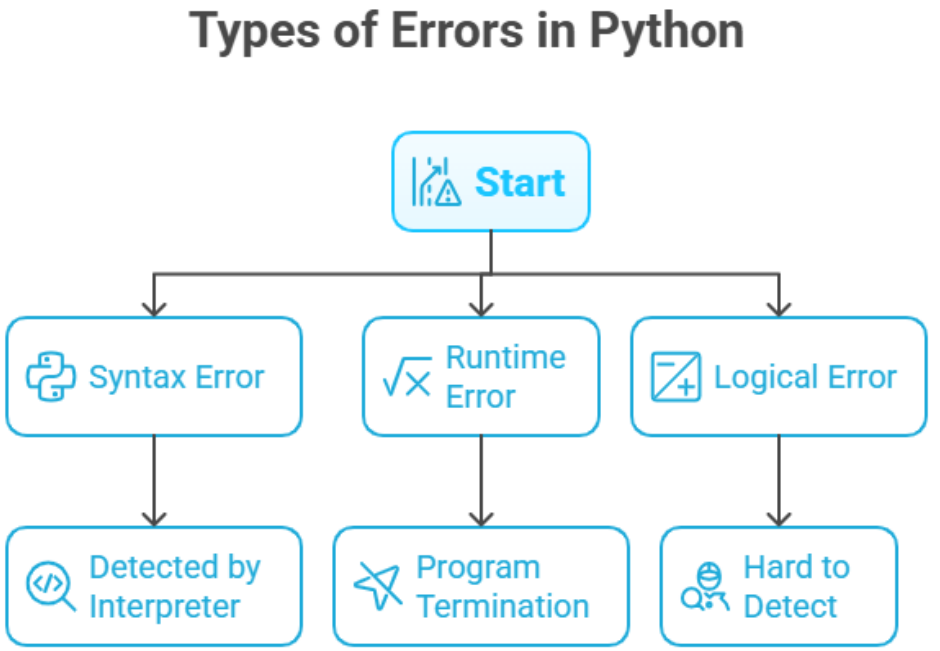
2. **Runtime error** occurs during the program execution and it causes the program to terminate.

   result = 10 / 0 # Division by zero

3. **Logical error** occurs when program runs but produces incorrect results due to incorrect logic of our program. This error is the hardest to detect.

   def calculate_average(numbers):

   return sum(numbers) / len(numbers) + 1 # Wrong formula!



Error vs Exception

Error	Exception
Serious issues that usually halt the program	Manageable issues that can be handled at runtime
Cannot typically be caught or handled	Can be caught using try-except blocks
MemoryError, RecursionError	*ZeroDivisionError, FileNotFoundError*

Exception Handling:

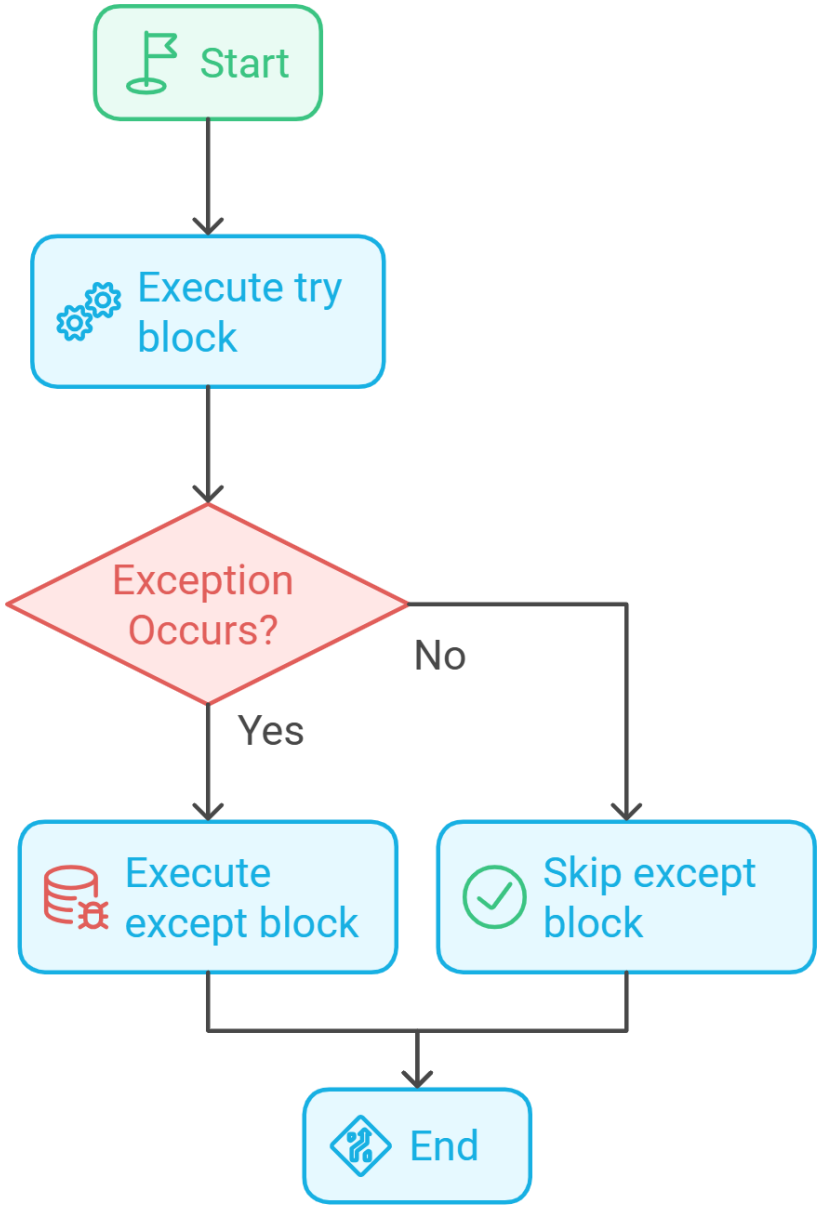
- The process of systematically responding to exception is called exception handling

Common Exceptions in Python

- **ZeroDivisionError** – Division by zero
- **IndexError** – Invalid index in a list or tuple
- **KeyError** – Accessing a non-existent dictionary key
- **TypeError** – Operation on incompatible data types
- **ValueError** – Incorrect value type
- **FileNotFoundError** – File or directory not found
- **ImportError** – Module not found or failed to load

Exception Handling Syntax

- Python provides the **try-except** block as a part of its error handling system.
- The **try** block is executed first and it contains code that might cause an exception.
- If no exception occurs in the **try** block, the **except** block is skipped. Otherwise, program execution jumps to the except block.
- We can specify the types of exceptions that we want to catch (possibly multiple).



Basic Syntax:

```
try:
    # Risky code
except SomeException:
    # Code to handle the exception
```

Optional Blocks

else block:

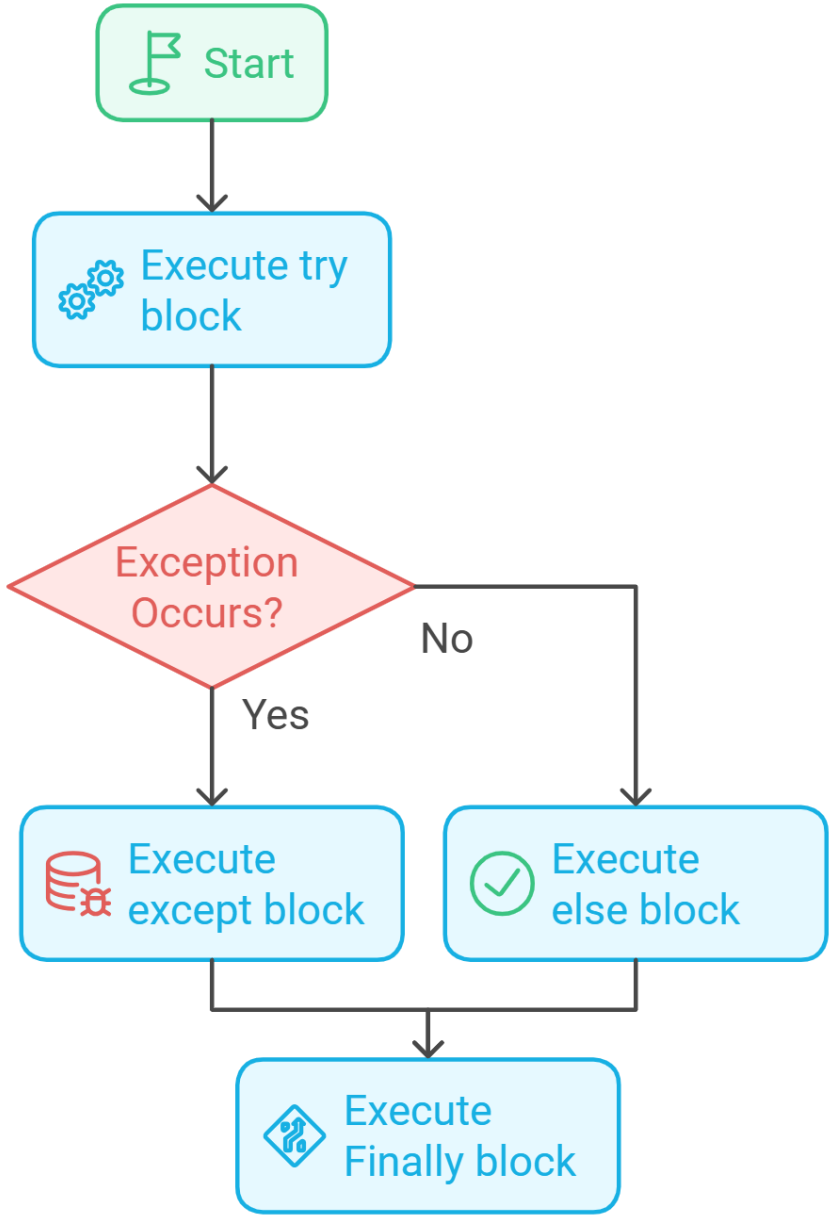
- Executes only when no exception occurs in *try*.

```
try:
    x = 10 / 2
except ZeroDivisionError:
    print("Error")
else:
    print("Success")  # Runs only if no exception
```

finally block:

- Always runs, regardless of whether exception occurs.

```
try:
    f = open("data.txt")
except FileNotFoundError:
    print("File missing")
finally:
    print("Cleanup or closing actions here")
```

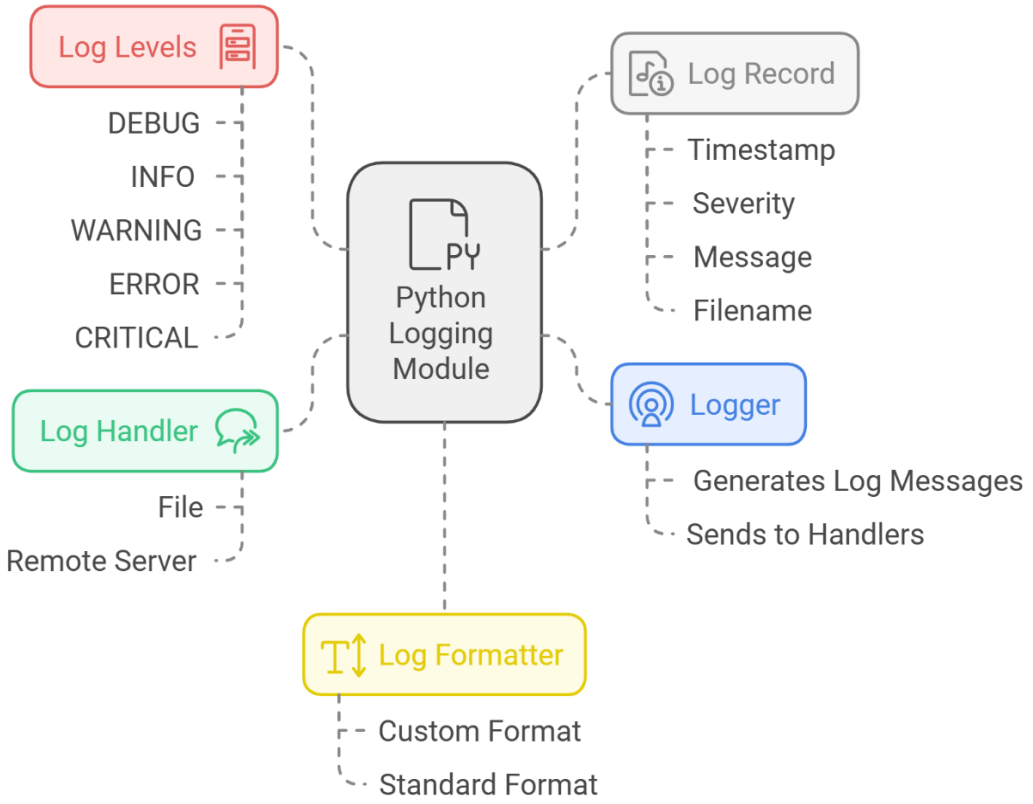


Error Logging in Python

- **What is Error Logging?**
  - Error logging means saving details about errors or issues that happen during your program's execution.
  - Instead of just printing messages, logs help you track and fix problems later.
- **Why is it useful?**
  - Helps in debugging and fixing bugs.
  - Stores error info even if the program crashes.
  - Useful in real-world apps and production systems.

application.log

```
1  [2025-05-31 06:41:42] Application started
2  [2025-05-31 06:41:42] An error occurred
3  [2025-05-31 06:41:42] Application stopped
```



Python's logging Module

Component	Purpose
Logger	Creates and manages log messages
Handler	Controls where logs go (file, console, etc.)
Formatter	Controls log message layout
Log Levels	DEBUG < INFO < WARNING < ERROR < CRITICAL

- A **log** is a record of events that happen while your program runs. Think of it like a diary for your program, it writes down what it's doing, what went wrong, and important milestones.

Logging Example:

```
import logging
# Setup logging configuration
logging.basicConfig(filename='errors.log' ,  level=logging.ERROR)


try:
    result = 10 / 0
except ZeroDivisionError:
    logging.error("Attempted division by zero", exc_info=True)
```

File Handling in Python

Introduction to File Handling

- File handling is the process of storing to and retrieving data from a persistent memory.

Why File Handling?

- **Persistence**: Data stored in variables is lost when a program ends. Files store data permanently until deleted.
- **Data Sharing**: Files help exchange data between different programs or systems.
- **Logging and Auditing**: Used to track application activity, debug issues, and maintain records.

📖 **Reading Files**

Get information from existing files

✍️ **Writing Files**

Create new files or overwrite existing ones

➕ **Appending Files**

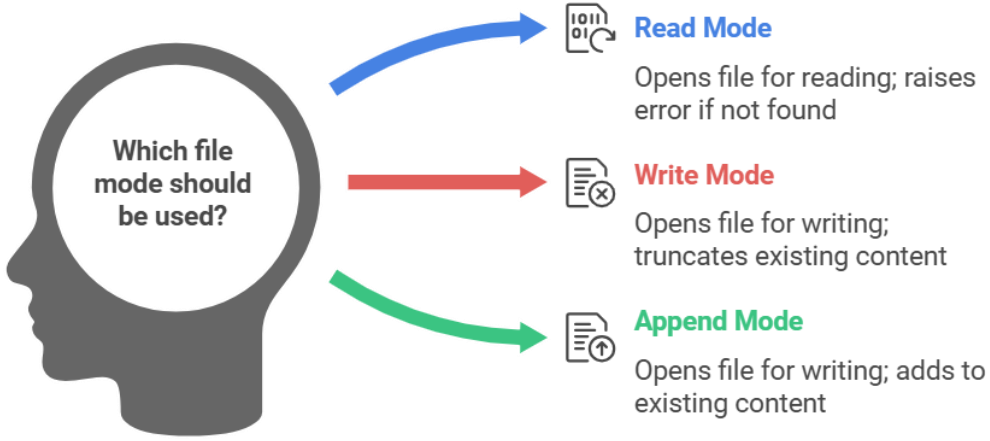
Add new content to existing files

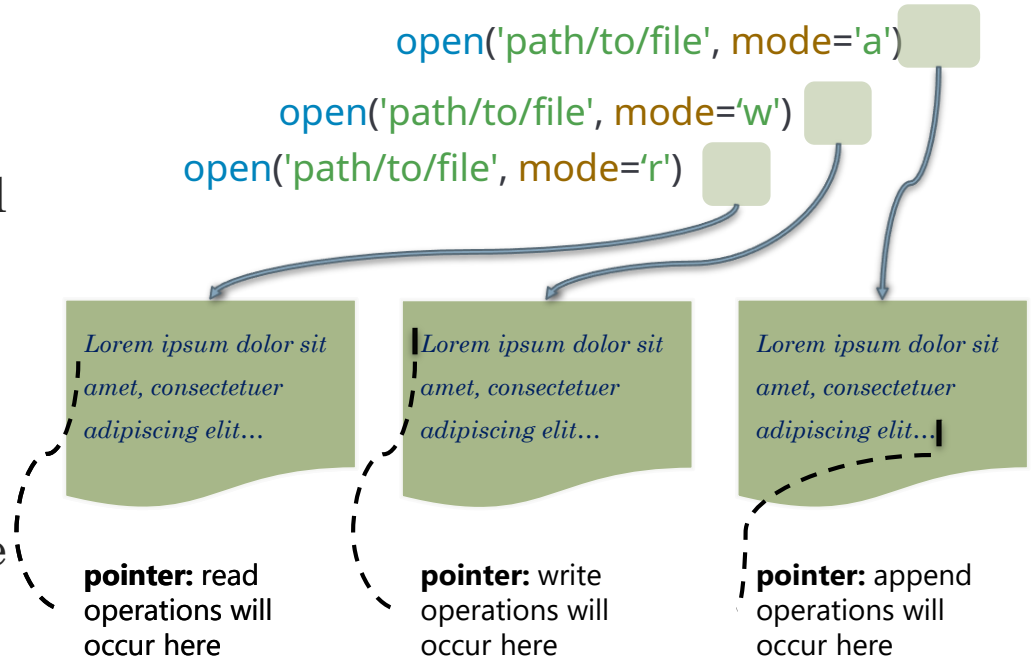
🔄 **Managing Files**

Create, rename, delete, and organize files

File Modes: Different Ways to Access Files

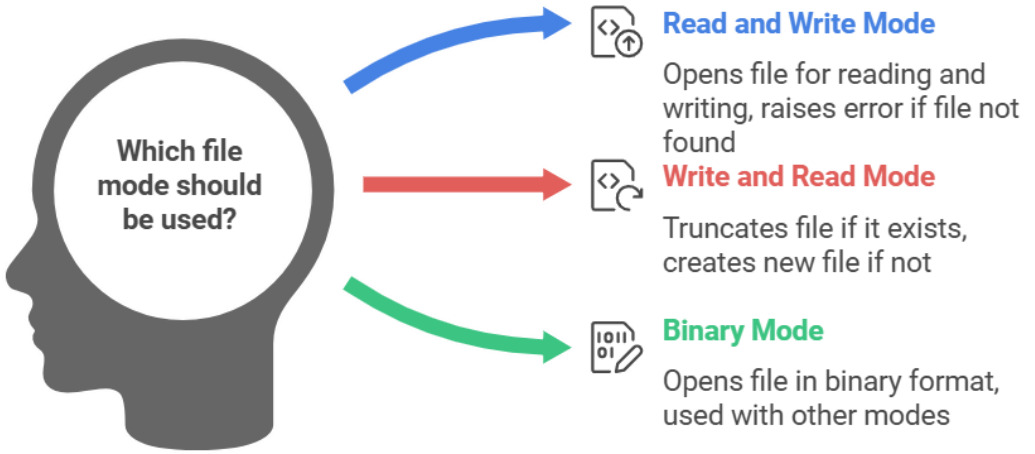
- A file mode determines the kind of operations that can be performed on the file.
- **Read mode – 'r':**
  - Opens the file for reading only
  - File pointer is placed at the beginning of the file
  - *FileNotFoundError* is raised, when the file does not exist
- **Write mode – 'w':**
  - Opens the file for writing only
  - If the file exists, its content is truncated (deleted)
  - A new file is created, when the file does not exist
- **Append mode – 'a':**
  - Opens the file for writing
  - File pointer is placed at the end of the file without modifying existing content
  - A new file is created, when the file does not exist

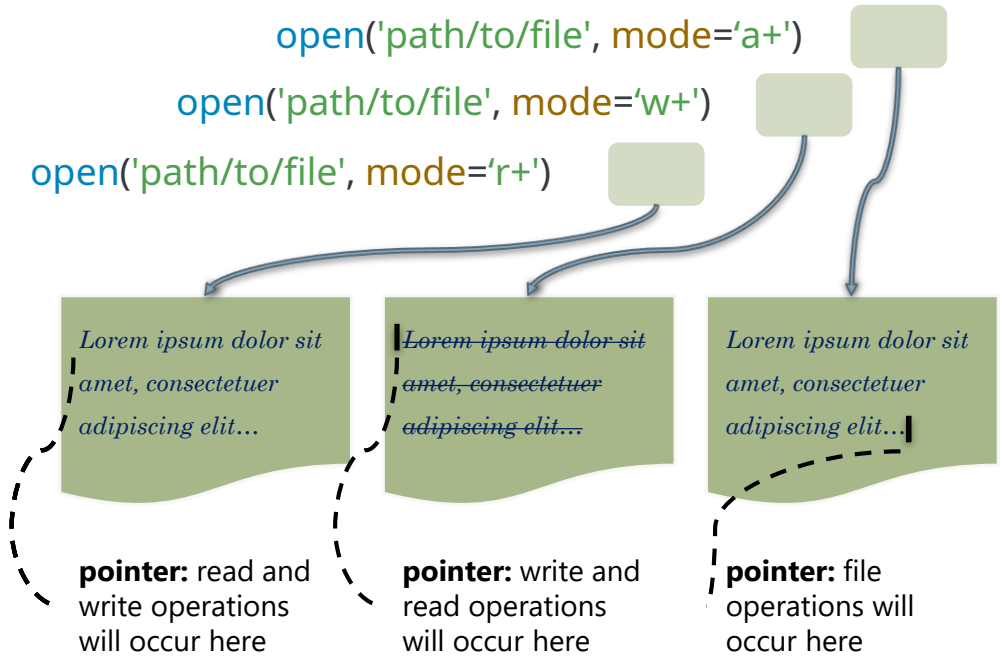




File Modes: Different Ways to Access Files

- A file mode determines the kind of operations that can be performed on the file.
- **Read and write mode – 'r+':**
  - Opens the file for both reading and writing
  - File pointer is placed at the beginning of the file
  - *FileNotFoundError* is raised, when the file does not exist
- **Write and read mode – 'w+':**
  - Opens the file for both writing and reading
  - If the file exists, its content is truncated (deleted)
  - A new file is created, when the file does not exist
- **Binary mode – 'b':**
  - Opens the file in binary mode.
  - Used in combination with other modes to work with the content of the file in binary format.





Reading Files

- Reading from files is fundamental operation in file handling.
- To read the contents of a file, it must be opened in ‘r’, ‘r+’, or ‘rb’ mode.
- The read() method reads the entire content of the file as a single string.
- The readline() method reads a single line from the file.
- The readlines() methods reads all lines in the file and returns them as a list of strings.
- ***Note:*** *In read mode,* ***file pointer*** *is placed at the beginning of the file initially.*


```
file = open('path/to/file', mode='r')
entire_file = file.read()
single_line = file.readline()
list_of_lines = file.readlines()
file.close()
```

Writing to Files

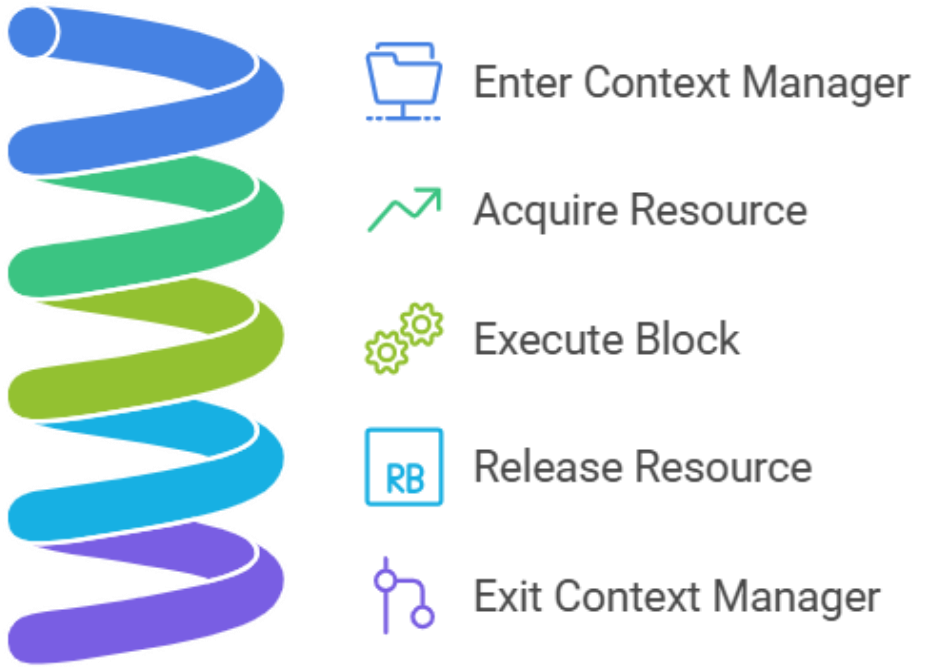
- Writing to files is necessary to store data in the files.
- To write some data to a file, it must be opened in ‘w’, ‘w+, ‘a’, ‘a+’, or ‘wb’ mode.
- The write() method writes a string to a file but doesn’t automatically add a newline at the end.
- The writelines() method writes a list of strings to the file, each on a new line.
- ***Note:*** *In write mode, initially the* ***file pointer*** *is placed at the beginning of the file while in append mode it is placed at the end.*


```
file = open('path/to/file', mode='w')
# write a line to the file
file.write('Hello, World!')
# write lines to the file
file.writelines(['I', 'Love', 'Python'])
file.close()
```

File Context Manager

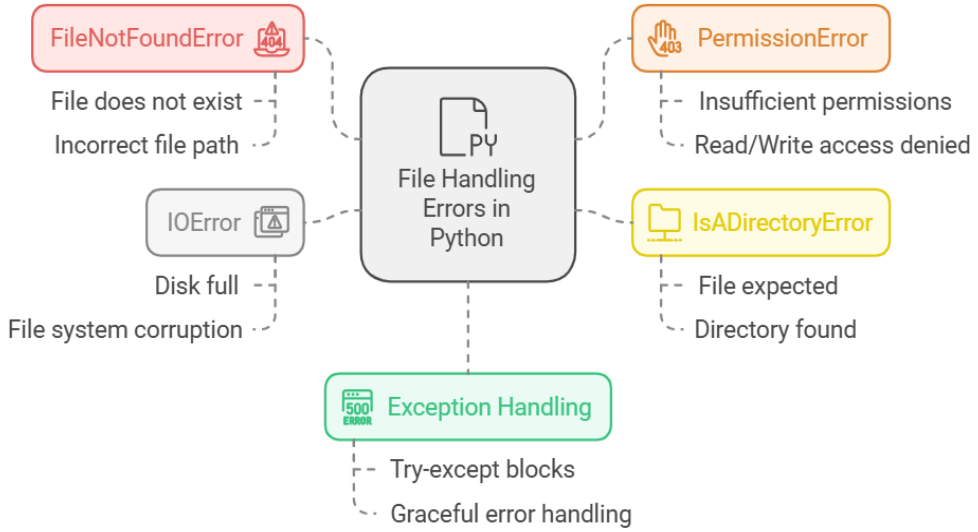
- File context manager is a way to handle files using the with statement.
- It ensures that the file is closed automatically after the block of code is executed.
- Even if an exception occurs within the block, the file will still be closed properly.
- It makes the code more readable and concise.
- File cannot be accessed outside of this context.

```
with open('path/to/file', mode='a') as file:
    # perform file operations
    ...
    # no need to close the file

# cannot perform operations with file here
```



File Exception Handling

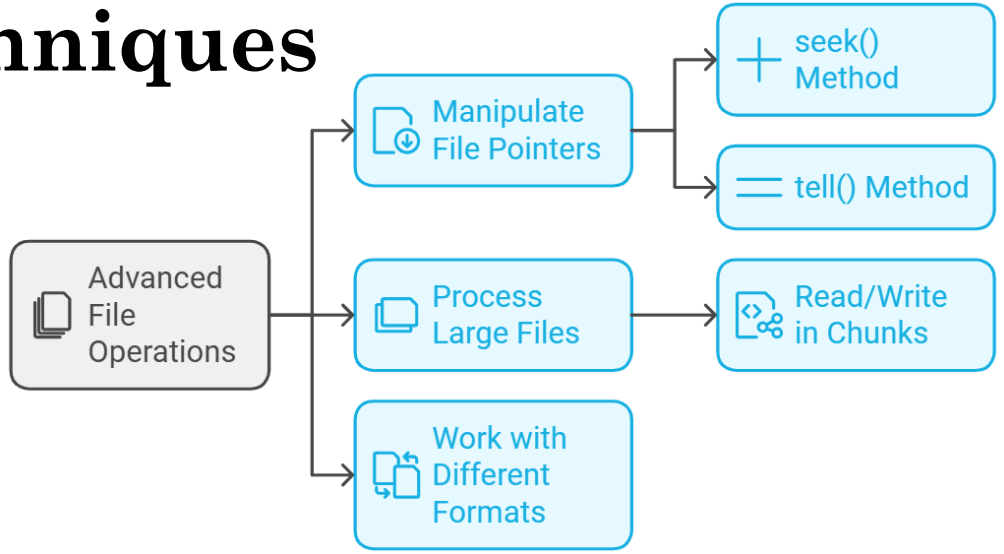
- Error handling is used when working with files to ensure that your program can gracefully handle unexpected situations.
- **FileNotFoundError:** Raised when trying to open a file that doesn't exist.
- **PermissionError:** Raised when the program does not have the necessary permissions to access the file.
- **IsADirectoryError:** Raised when a file is expected but a directory is found, or vice versa.
- **IOError:** Raised for various I/O related errors, such as disk full, file system corruption, etc.
- Python's exception handling mechanism is used to handle these situations using try-except blocks




```
try:
    file = open("random.txt", "r")
    content = file.read()
    file.close()
except FileNotFoundError:
    print("Error: 'example.txt' was not found.")
except PermissionError:
    print("Error: no permission to read 'example.txt'.")
except Exception as e:
    print(f"Error: Unexpected error occurred. {e}")
finally:
    file.close()
    print("File closed.")
```

Advanced File Handling Techniques

- Python provides advanced operations like manipulating file pointers, reading large files, and working with different formats.
- We can manipulate file pointer to read and write at specific positions:
  - The **seek()** method allows us to control the file pointer position.
  - The **tell()** method returns the current position of the file pointer.
- Reading or writing the entire file at once can be inefficient and may cause memory leaks.
- Therefore, large files are processed in chunks.

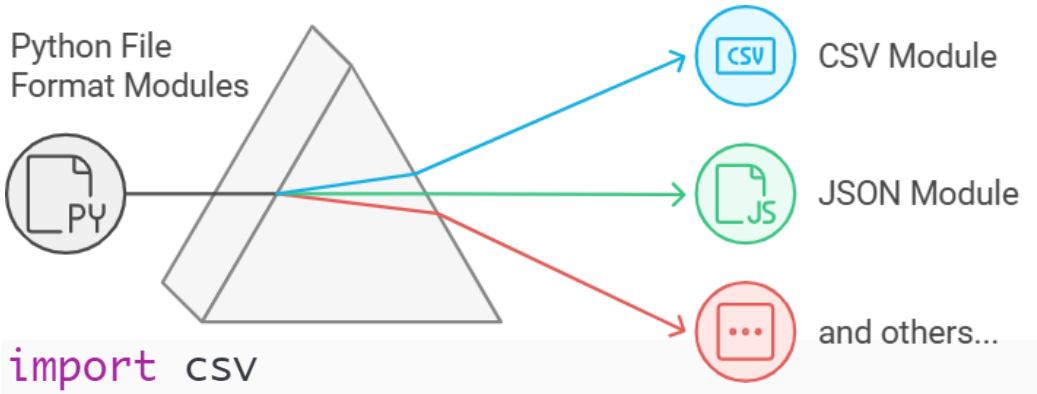



```python
with open('temp.txt', 'w+') as f:
    # gets the current file pointer position in the file
    f.tell()
    # sets the file pointer to new position in the file
    f.seek(0)
```

```python
with open(file_path, 'r') as file:
    while True:
        chunk = file.read(chunk_size)
        if not chunk:
            break
```

Advanced File Handling Techniques

- Python standard library has several modules to work with different file formats.
- The csv module allows working with CSV (Comma Separated Values) files which is a simple format for storing tabular data.
- Each line in the file represents a row and columns are typically separated by commas.
- The json module is used to work with JSON (JavaScript Object Notation) files, which is a lightweight data-interchange format.
- It is used to store structured data in a readable way.
- Its syntax is similar to python's dictionary.




```python
import csv

# Reading rows of a CSV file as
# lists using csv.reader
with open('example.csv', 'r') as csvfile:
    csvreader = csv.reader(csvfile)
    for row in csvreader:
        print(row)
```

```python
import json

# Reading from a JSON file
with open('example.json', 'r') as json_file:
    # json.load() reads from a file
    data = json.load(json_file)
    print("Content of example.json:\n", data)
```

Thank You