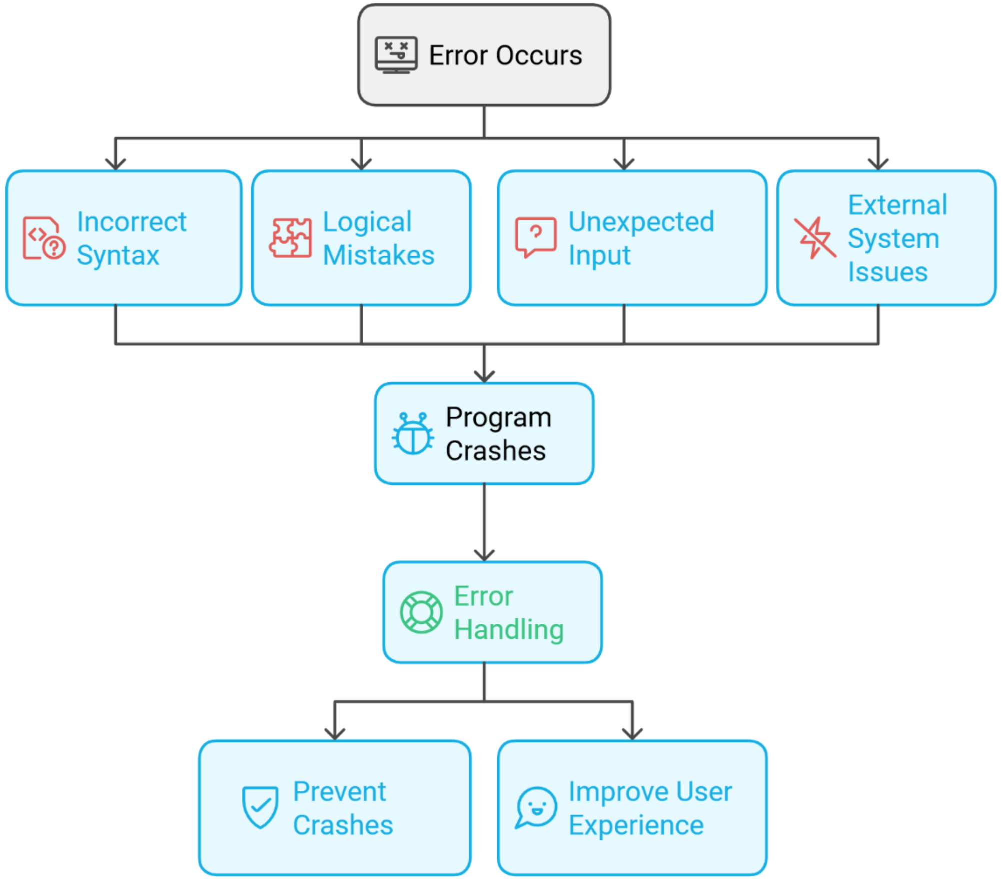
Error/Exception Handling

Dr. Muhammad Sajjad
RA. Wajahat Ullah

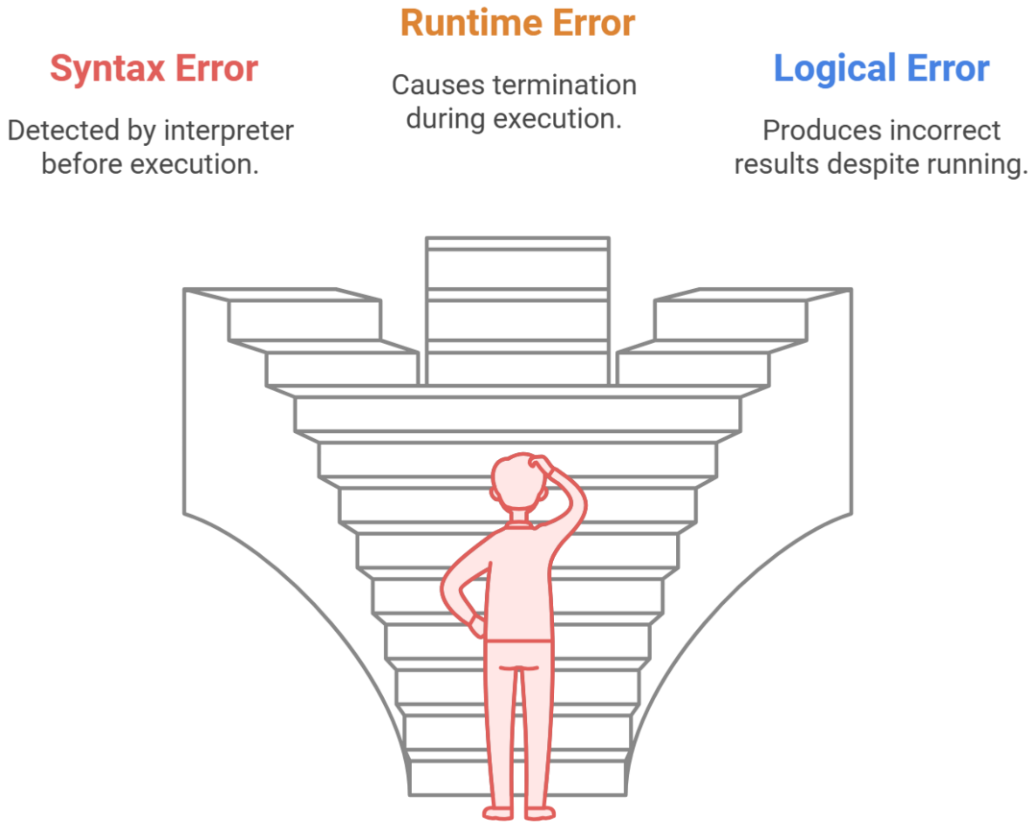
Errors in Programming

- **Errors** are unexpected or unwanted events that disrupt the normal flow of program execution.
- Errors can occur due to incorrect syntax, logical mistakes, unexpected input, or 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

- Various types of errors can possibly occur in a Python program, such as:
- **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.
- **Runtime error** occurs during the program execution and it causes the program to terminate.
- **Logical error** occurs when program runs but produces incorrect results due to incorrect logic of our program. This error is the hardest to detect.



Syntax Error

Detected by interpreter before execution.

Runtime Error

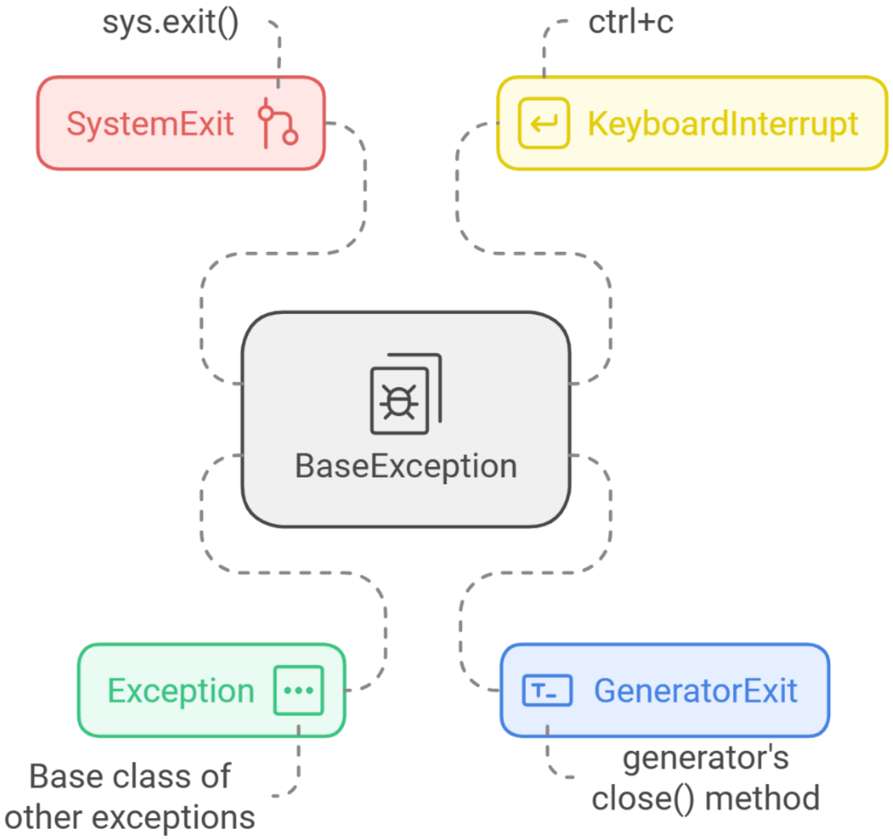
Causes termination during execution.

Logical Error

Produces incorrect results despite running.

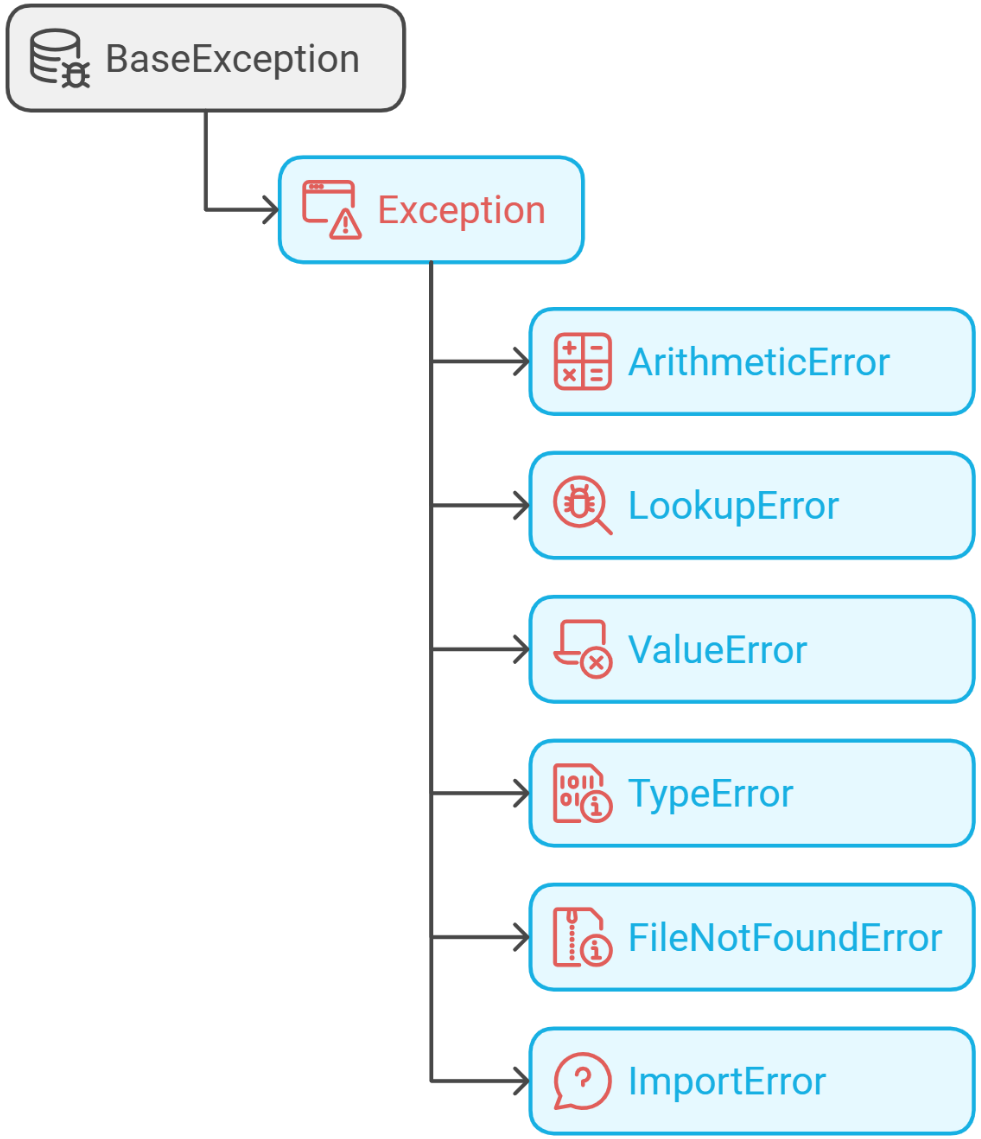
Python Error Hierarchy

- Errors/Exceptions in python are designed using a well-structured inheritance hierarchy.
- The base class of all type of exceptions is the `BaseException` class.
- Four main exception classes are derived from it
  - `SystemExit:` Used to terminate a python program e.g. `sys.exit()` raises this exception to exit the program.
  - `KeyboardInterrupt:` Raises when user interrupts program execution e.g. pressing `ctrl+c`
  - `GeneratorExit:` Raised when a generator's `close()` method is called
  - `Exception:` The main base class of all standard exceptions.



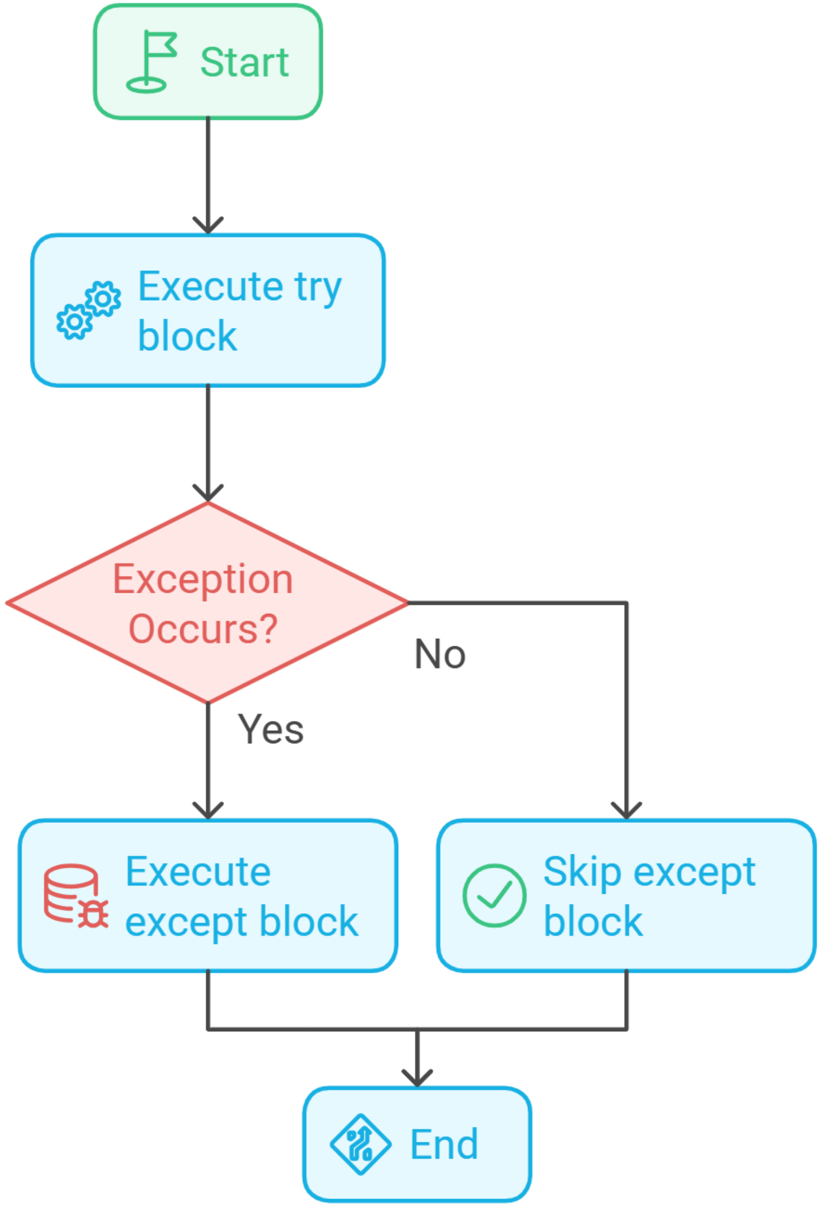
Python Error Hierarchy

- `Exception` is the direct subclass of `BaseException` which then acts as the superclass for all standard exceptions in Python.
- While handling errors, we usually catch instances of `Exception` or its subclasses.
- Following are the frequently encountered errors/exceptions in python:
  - `ArithmeticError:` during arithmetic operations e.g. `ZeroDivisionError`
  - `LookupError:` when accessing elements in collections e.g. `IndexError`, `KeyError`
  - `ValueError:` when a function receives an argument of the correct type but inappropriate value
  - `TypeError:` when an function is applied to an object of inappropriate type e.g. `len(2)`
  - `FileNotFoundError:` when trying to access a file that does not exist.
  - `ImportError:` when an import statement fails.



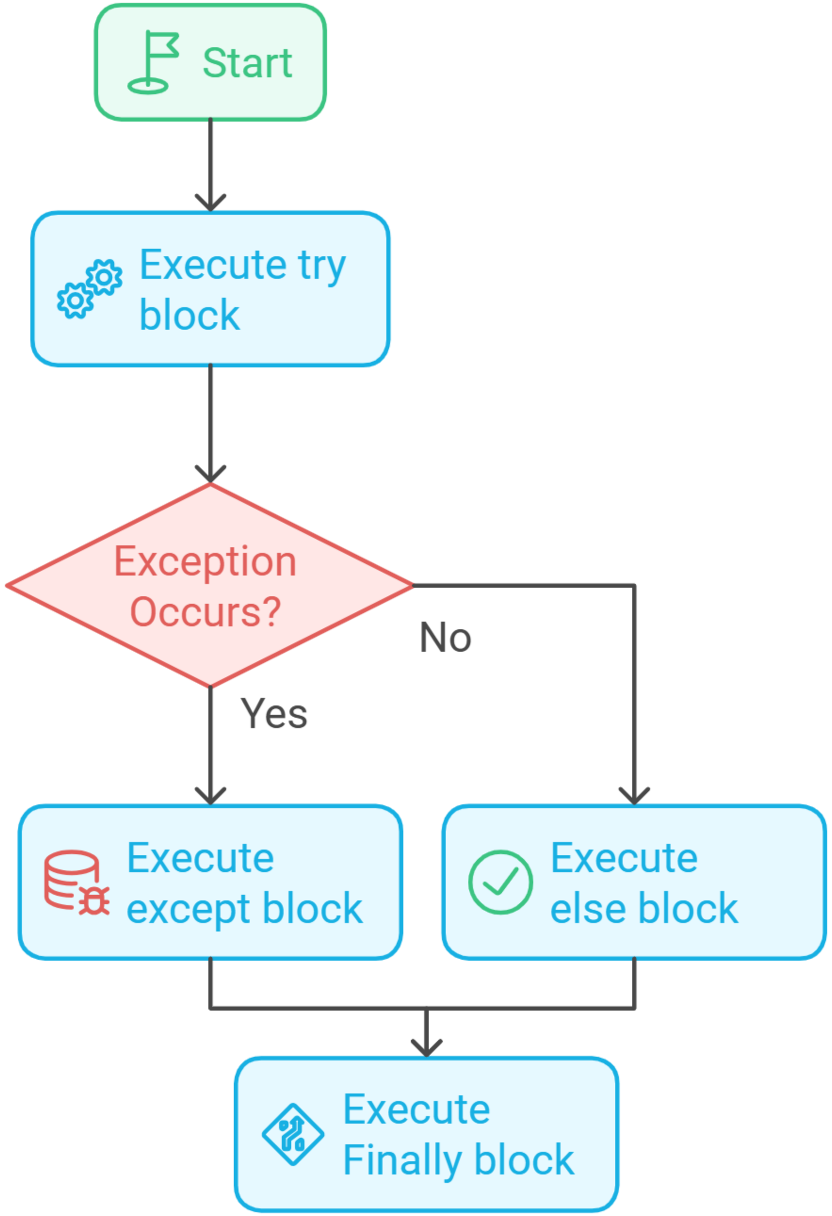
Exception Handling Syntax

- 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).



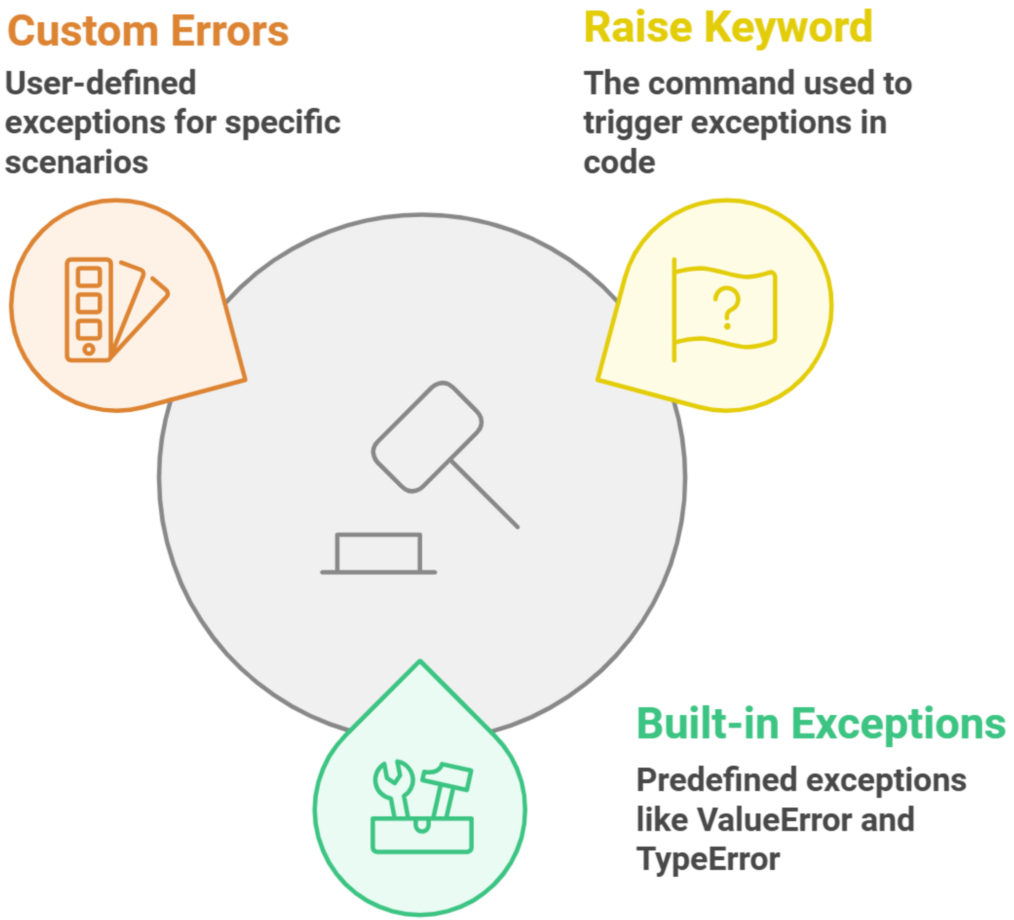
Exception Handling Syntax

- An optional `else` block can be added after all `except` blocks.
- This block is executed when no exceptions occur in the `try` block.
- Useful for code that runs only when there are no errors.
- Another optional `finally` block can be added at the end of all blocks.
- This block is always executed regardless of any errors are not.
- Commonly used for cleanup actions like closing files or releasing resources.



Custom Exceptions

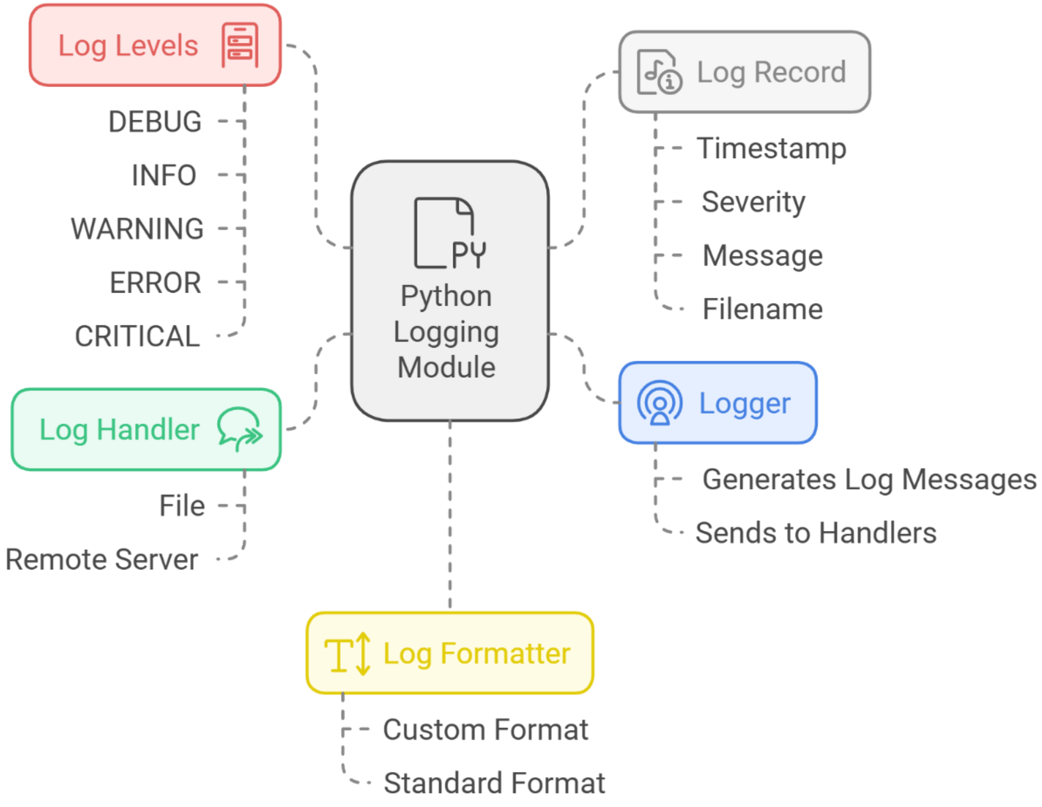
- We can manually throw an exception at any point in our program using the `raise` keyword.
- Useful for enforcing certain conditions and validating inputs.
- We can raise built-in exceptions like `ValueError`, `TypeError`, etc. or more general exception with a custom error message.
- We can also create custom errors specific to our problem by inheriting the `Exception` class.


Custom Errors
User-defined exceptions for specific scenarios
Raise Keyword
The command used to trigger exceptions in code
Built-in Exceptions
Predefined exceptions like ValueError and TypeError

Error Logging

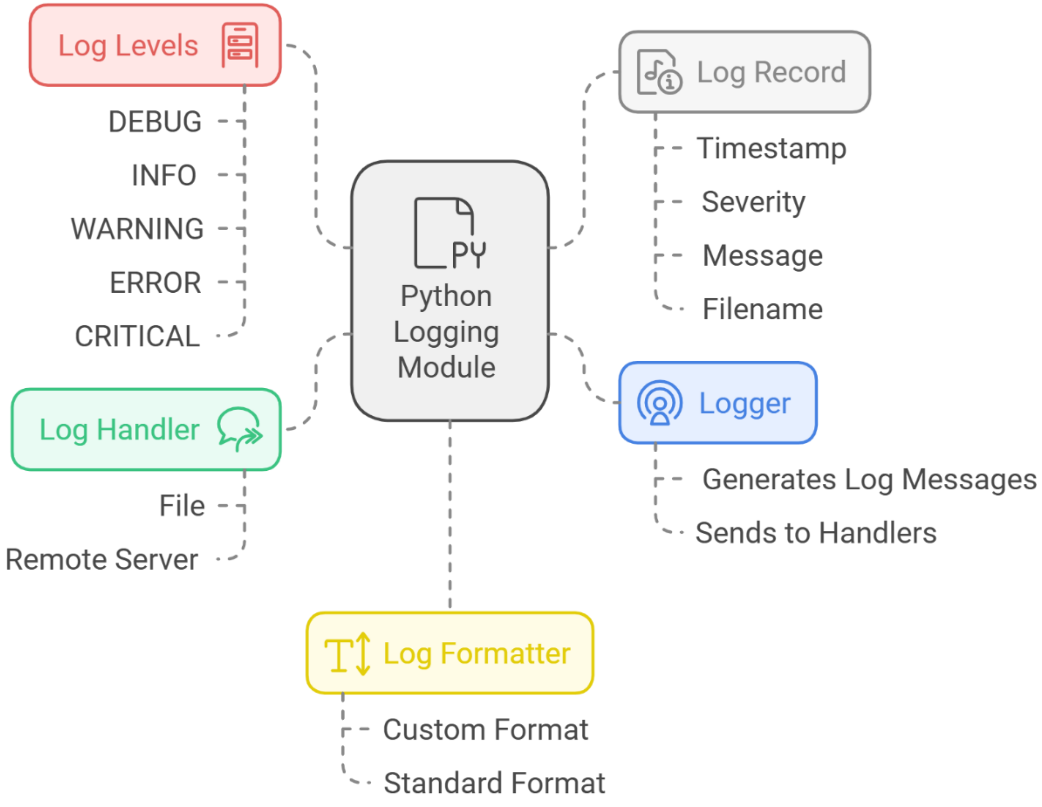
- Recording details about errors, exceptions, or other events in a program to a log file or other logging destination.
- Useful for tracking and debugging issues that occur during the execution of a program.
- Python’s `logging` module enables logging of errors, warnings, and other info to log files.
- We have 5 components of Logging:

1. Logger
2. Log Handler
3. Log Level
4. Log Formatter
5. Log Record



Components of Logging

- Python's `logging` module has 5 main components that work together to log information.
- **Logger** is the main interface for generating log messages. It captures log messages and sends them to handlers.
- **Log handler** sends log messages to destinations like a file, remote server, etc.
- **Log levels** classify the severity and importance of log messages (`DEBUG < INFO < WARNING < ERROR < CRITICAL`).
- **Log Formatter** Defines the format of the log message.
- **Log Record** contains metadata about the log message i.e. timestamp, severity, message, filename, etc.



Log Levels
DEBUG
INFO
WARNING
ERROR
CRITICAL
Log Handler
File
Remote Server
PY
Python Logging Module
Log Record
Timestamp
Severity
Message
Filename
Logger
Generates Log Messages
Sends to Handlers
Log Formatter
Custom Format
Standard Format