LEARN RISE SHINE
NAVTTC

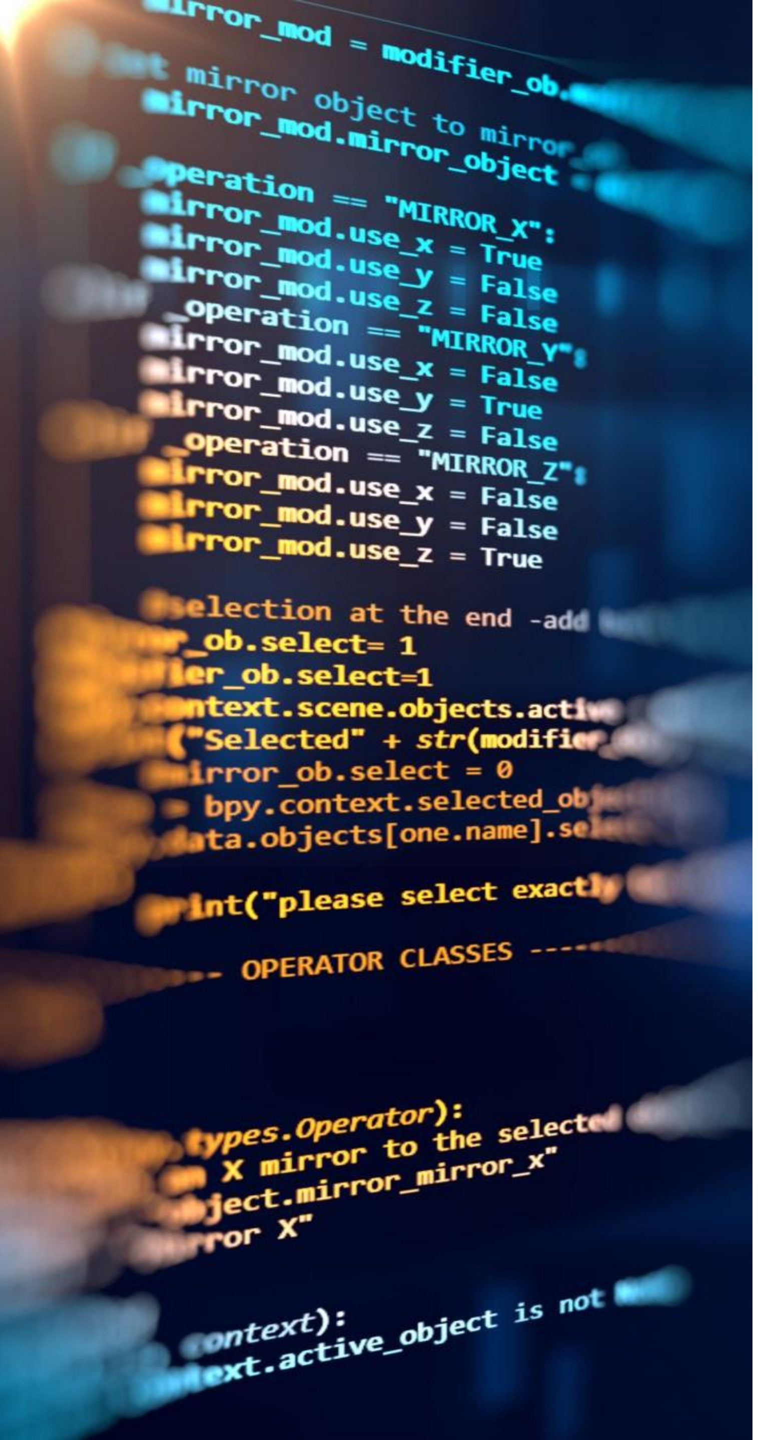
Summer of Code

Artificial Intelligence

(Machine Learning & Deep Learning)

Instructor
Wajahat Ullah
- *Research Assistant* (DIP Lab)

Duration
03 Months
(September – November)



Day 01 and 02 – Python Fundamentals (Functions)

Objectives:

- What are Functions?
- What is Scope?
- Lambda Expressions
- Error/Exception Handling
- Working with Files

Introduction to Functions

- Functions are named, reusable blocks of code designed to perform a specific task.

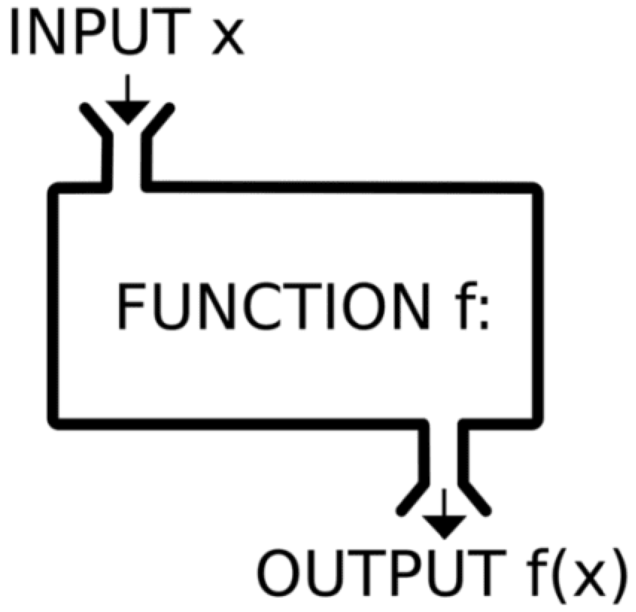
Types of Functions:

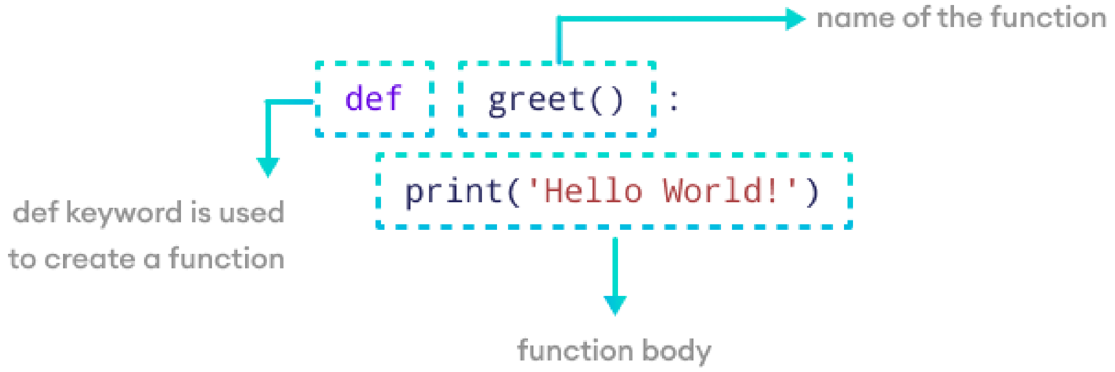
- Built-in functions (e.g., `print()`, `len()`)
- User-defined functions (created using `def`)

Example:

```
def greet(name):
    print(f"Hello, {name}!")
```

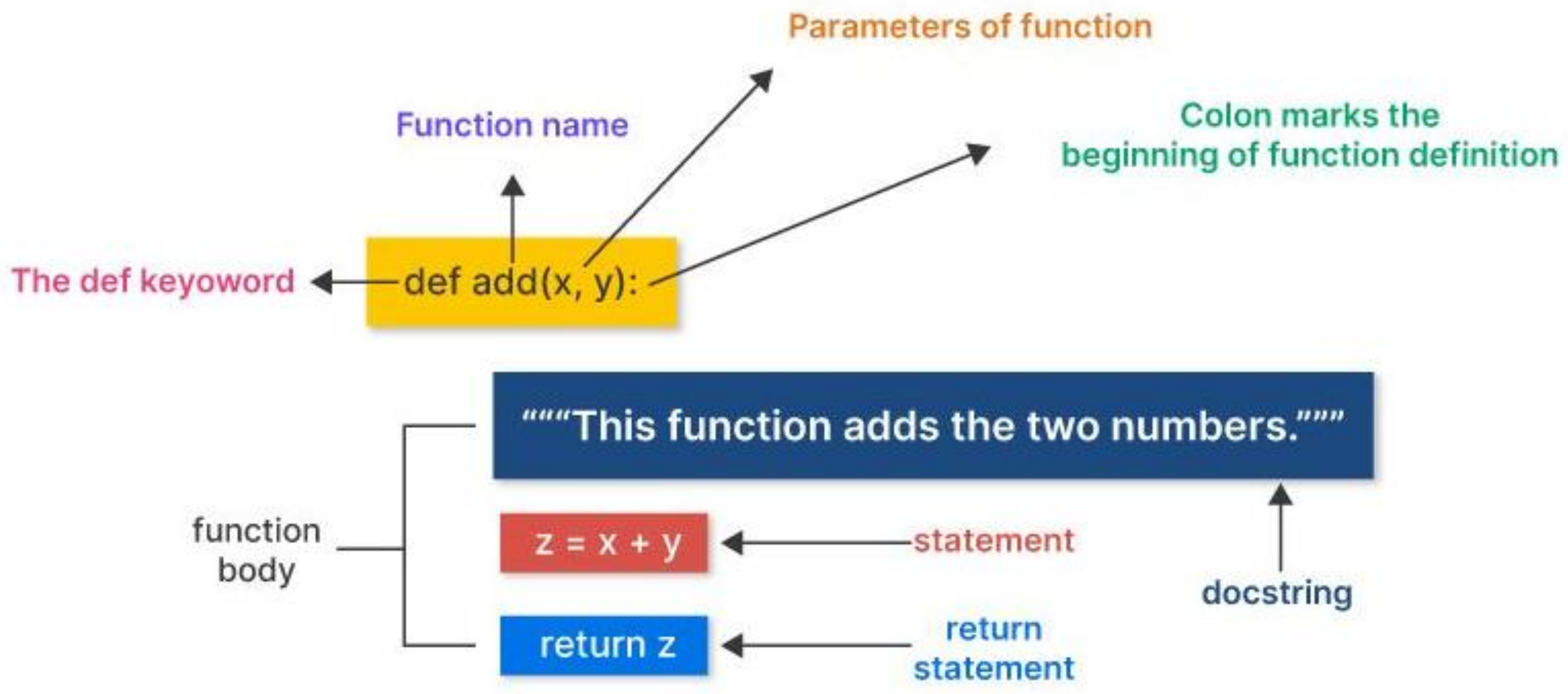
- In this example, `greet` is a function that takes one parameter, `name`.





```
def greet():
    print('Hello World!')
```

Functions in Python



Advantages of Using Functions

Why modular coding with functions boosts productivity and clarity

Reusability

Write once, call many times to reduce repetition.

Clarity

Encapsulate logic for easier understanding and maintenance.

Abstraction

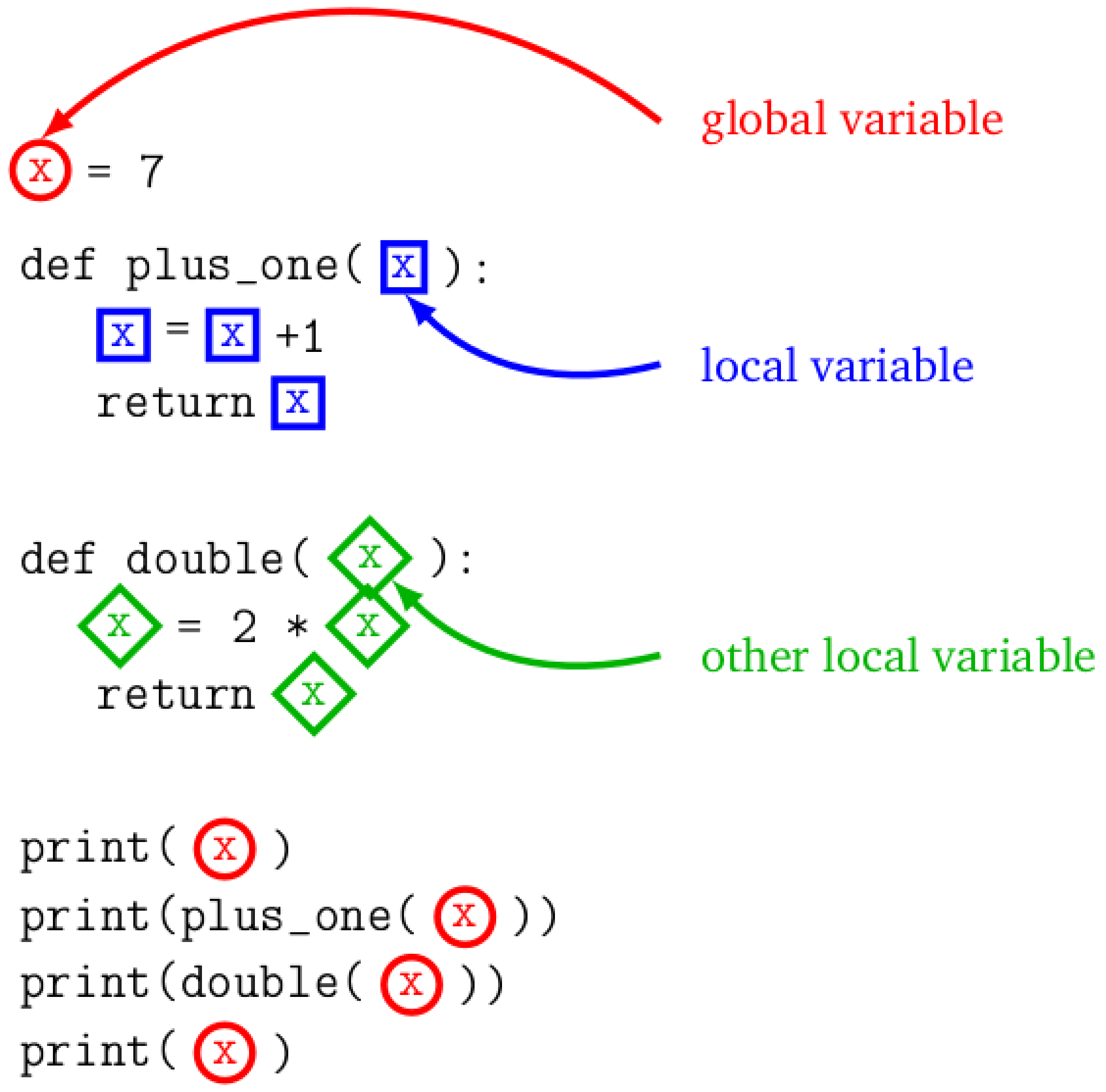
Hide complex details behind simple interfaces.

Modularity

Break programs into manageable, independent parts.

Variable Scope

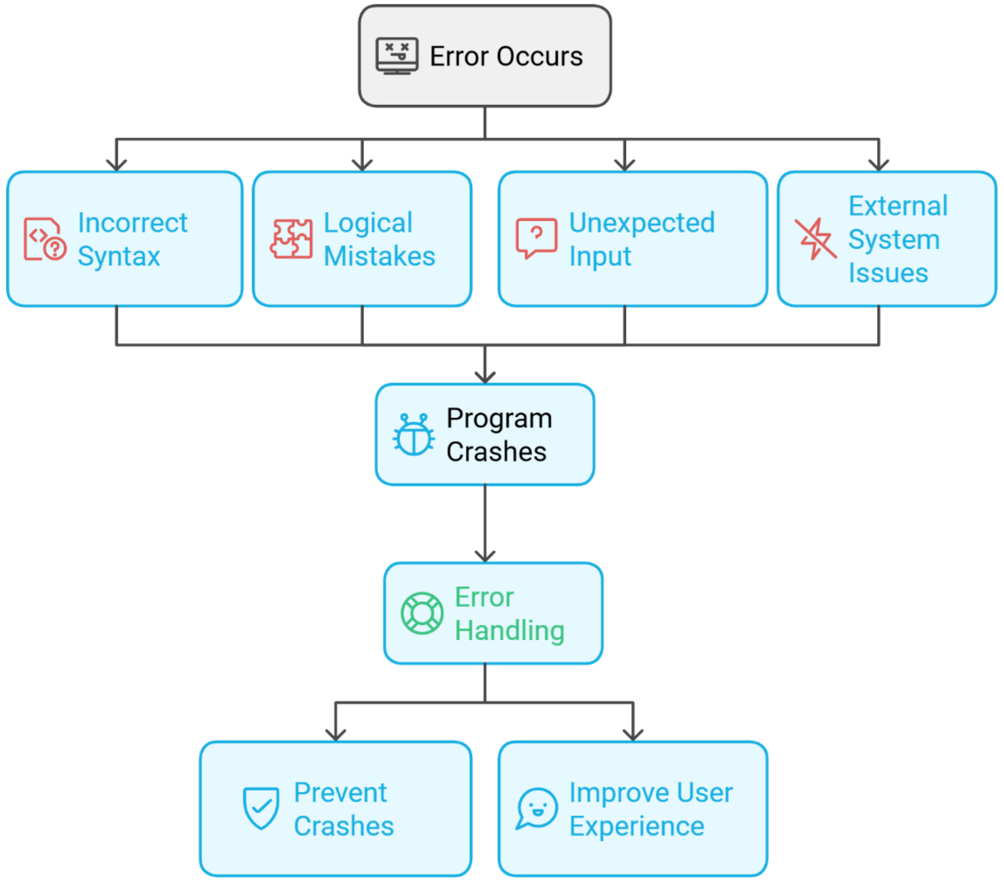
- **Scope** refers to the visibility and accessibility of variables in different parts of the code.
- **Local Scope:**
  Variables defined inside a function are local to that function and cannot be accessed outside.
- **Global Scope:**
  Variables defined outside a function are global and can be accessed inside the function.



```
x = 7

def plus_one( x ):
    x = x +1
    return x


def double( x ):
    x = 2 * x
    return x


print( x )
print(plus_one( x ))
print(double( x ))
print( x )
```

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.

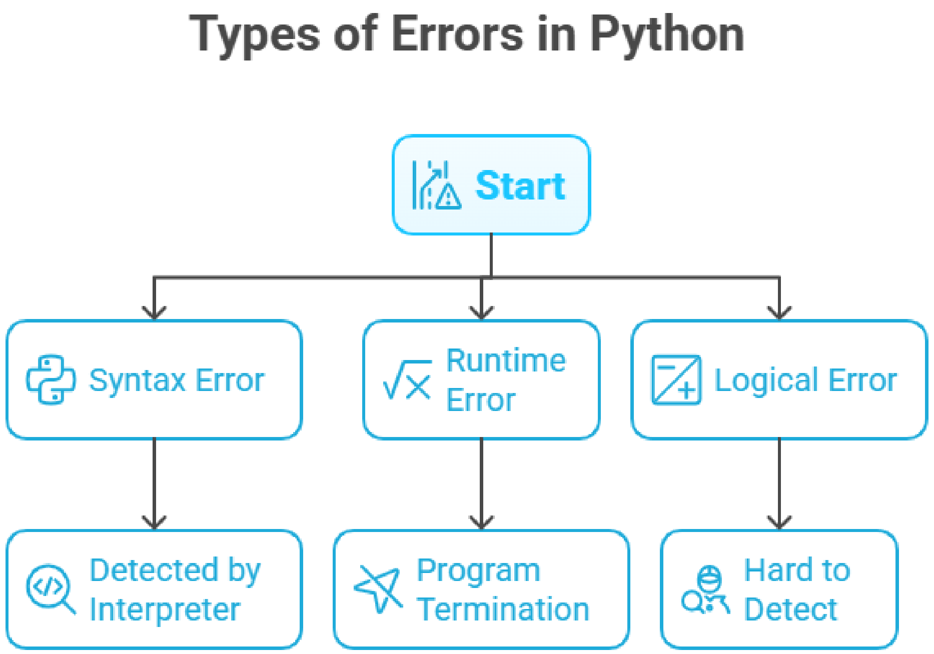
```
print(“Hello World”   # Missing parenthesis
```

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



Error vs Exception

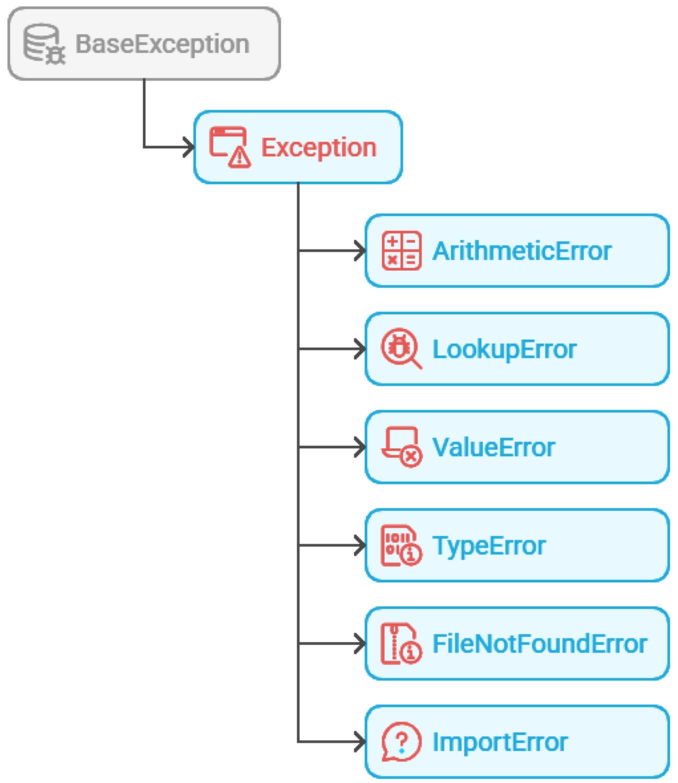
Error	Exception
Serious issues that usually terminate the program	Manageable issues that can be handled at runtime
Cannot typically be caught or handled	Can be caught using try-except blocks
MemoryError, RecursionError	*ZeroDivisionError, FileNotFoundError*

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

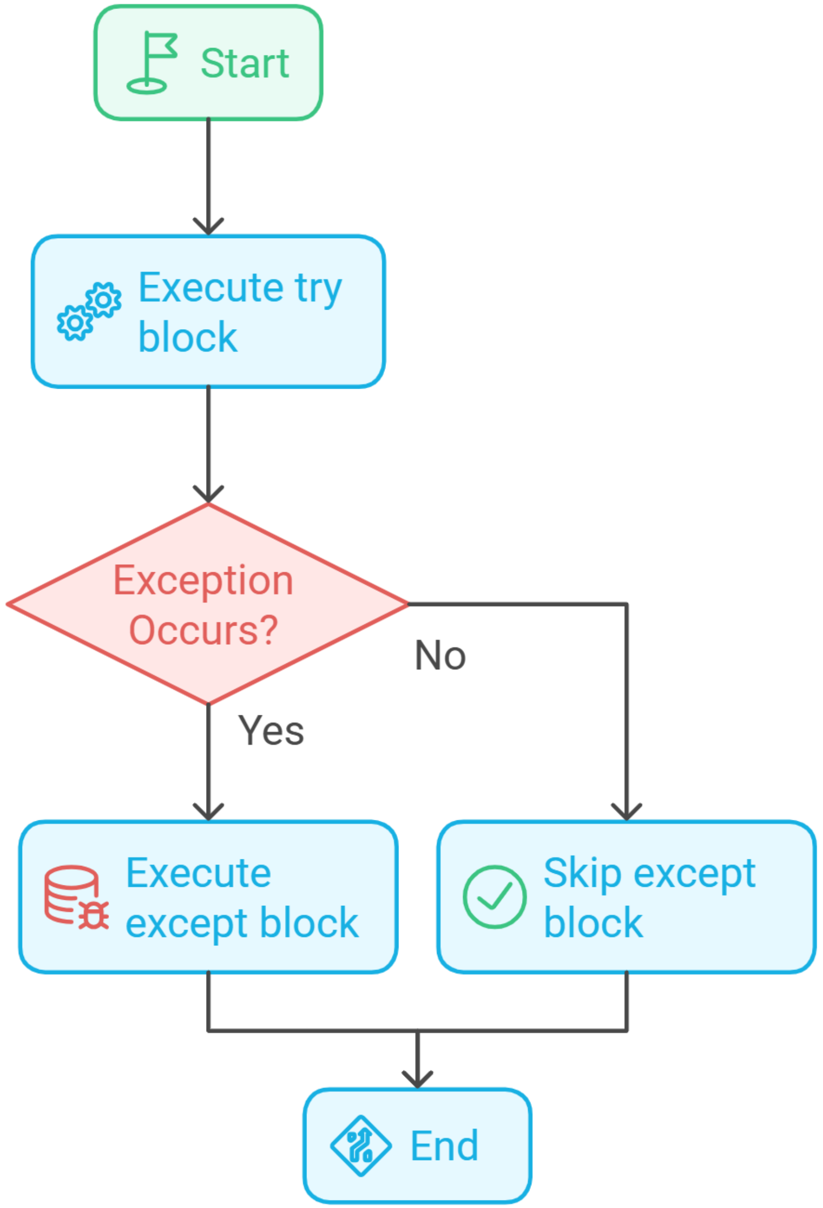


Exception Handling Syntax

- 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 Exception:
    # Code to handle the exception
```



Optional Blocks

`else` block:

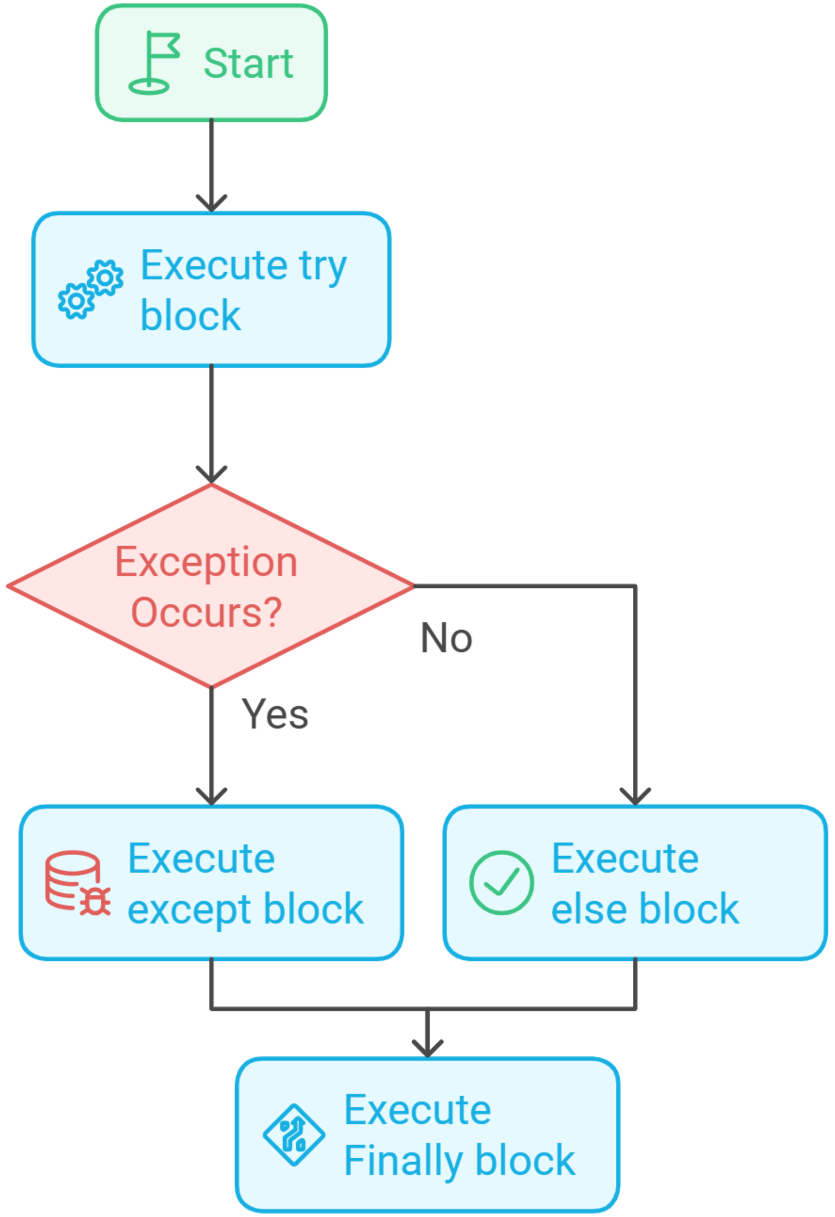
- Executes only when no exception occurs in *try*.

```
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”)
```

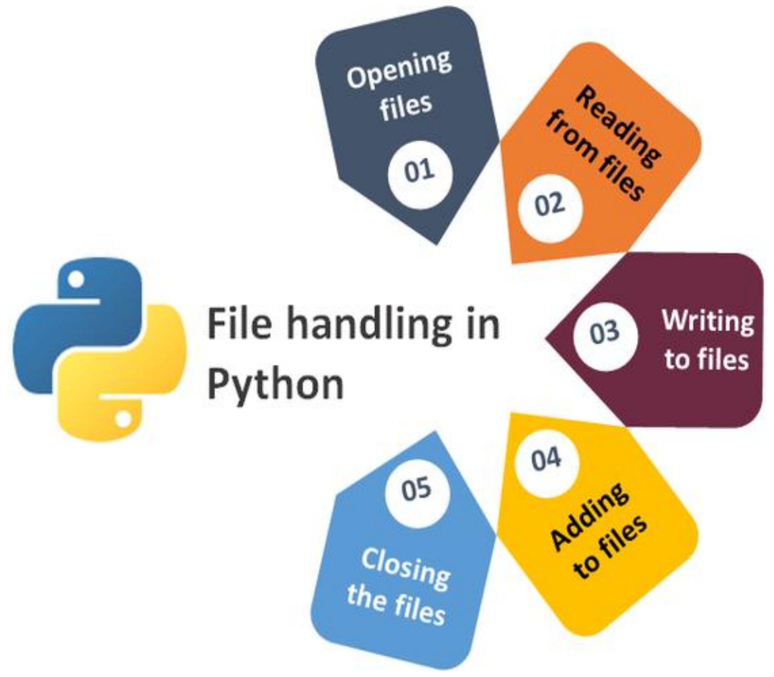


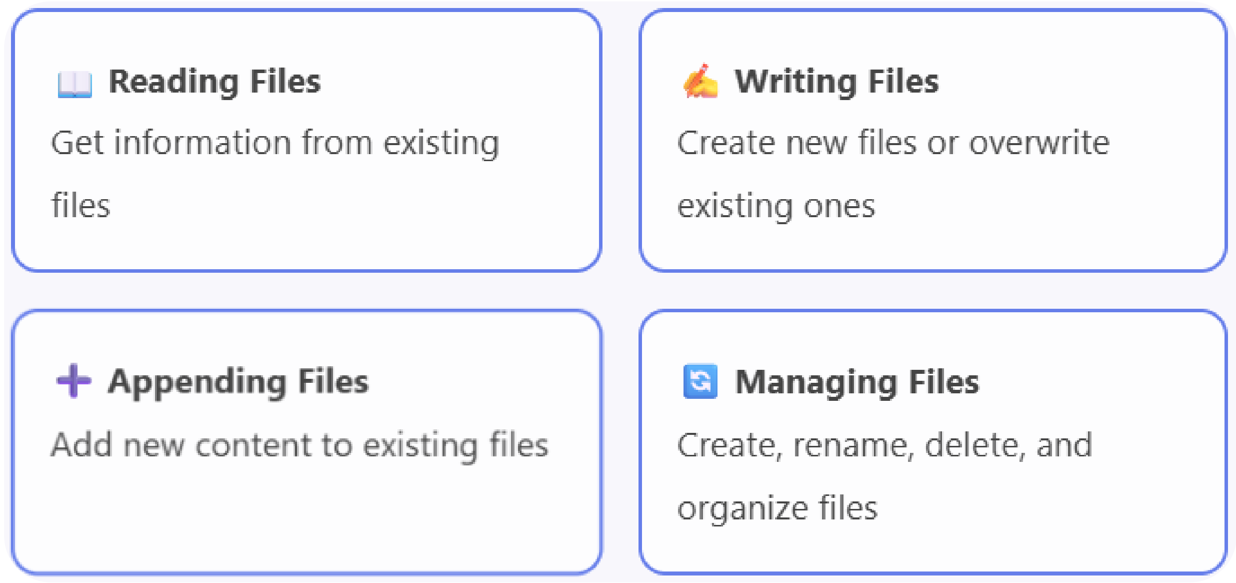
Introduction to File Handling

- 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

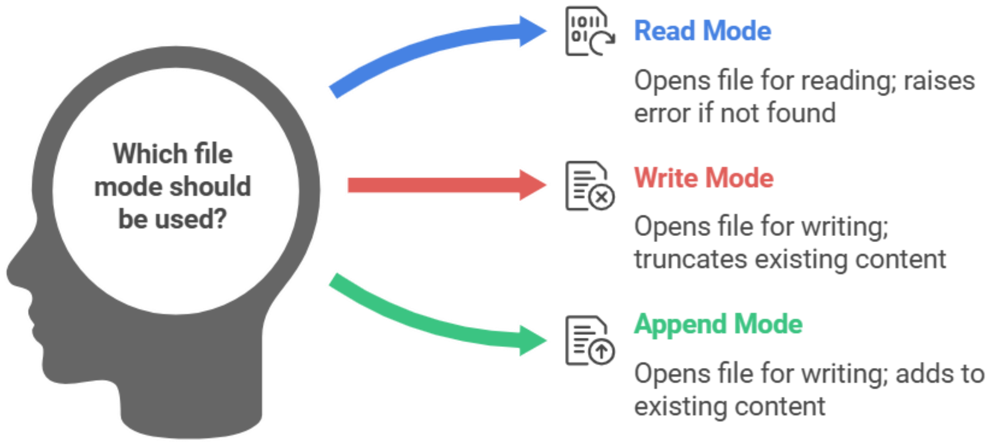
✍️ **Writing Files**
Create new files or overwrite existing ones

➕ **Appending Files**
Add new content to existing files

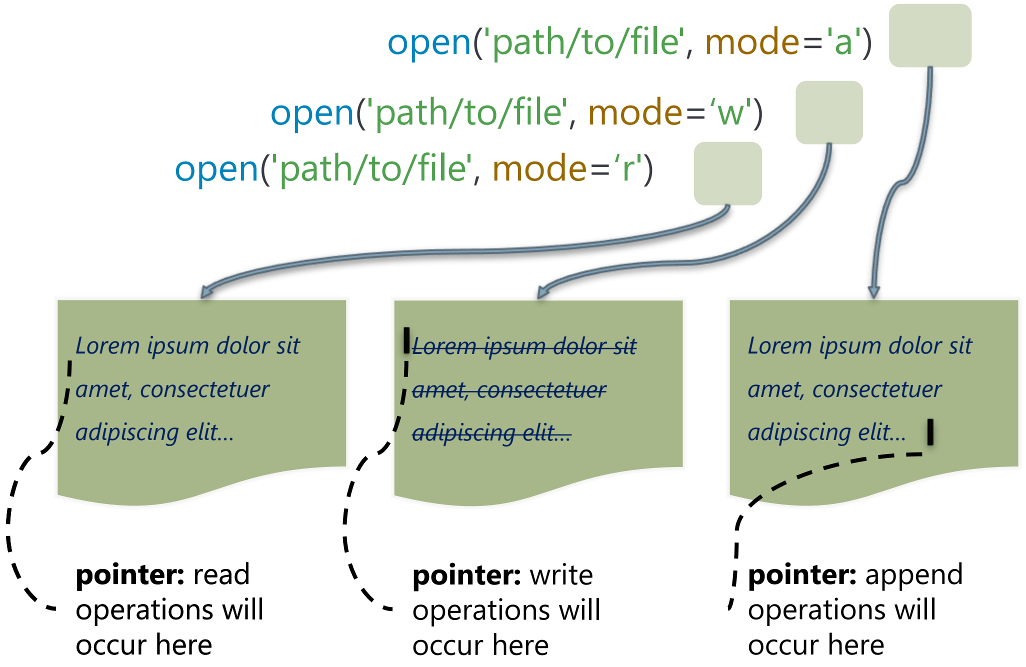
🔄 **Managing Files**
Create, rename, delete, and organize files

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



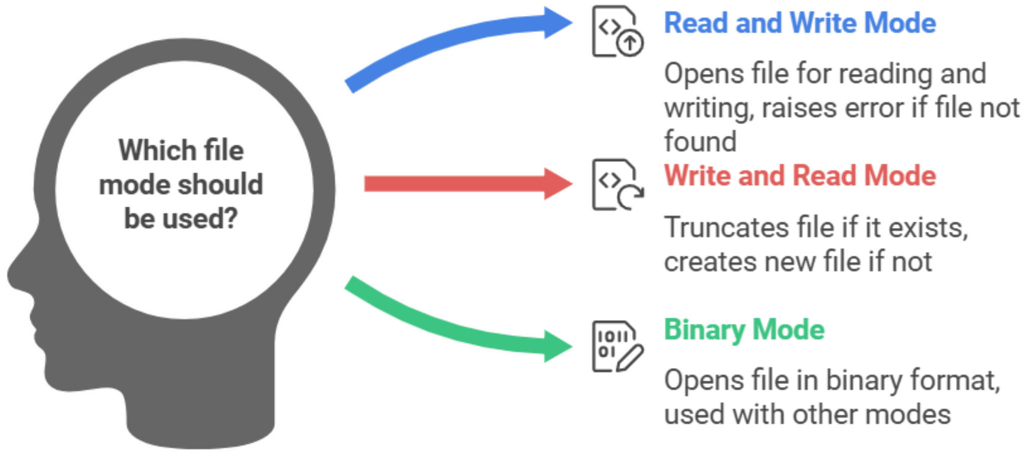
Which file mode should be used?
Read Mode
Opens file for reading; raises error if not found
Write Mode
Opens file for writing; truncates existing content
Append Mode
Opens file for writing; adds to existing content

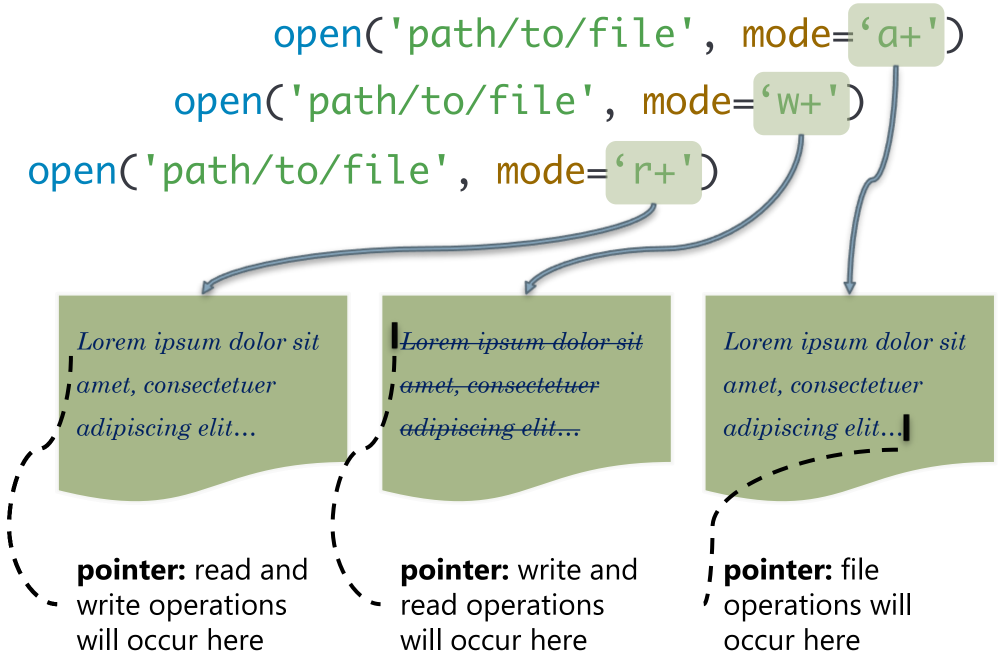


open('path/to/file', mode='a')
open('path/to/file', mode='w')
open('path/to/file', mode='r')
Lorem ipsum dolor sit amet, consectetuer adipiscing elit...
~~Lorem ipsum dolor sit amet, consectetuer adipiscing elit...~~
Lorem ipsum dolor sit amet, consectetuer adipiscing elit...
pointer: read operations will occur here
pointer: write operations will occur here
pointer: append operations will occur here

File Modes: Different Ways to Access Files

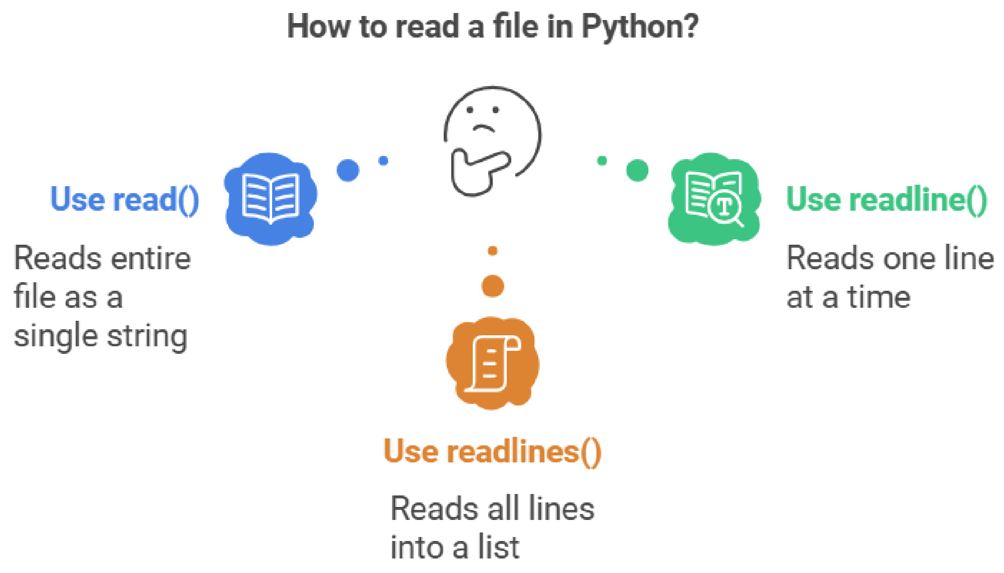
- A file mode determines the kind of operations that can be performed on the file.
- **Read and write mode – ‘`r+`’:**
  - Opens the file for both reading and writing
  - File pointer is placed at the beginning of the file
  - `FileNotFoundError` is raised, when the file does not exist
- **Write and 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.


Which file mode should be used?
Read and Write Mode
Opens file for reading and writing, raises error if file not found
Write and Read Mode
Truncates file if it exists, creates new file if not
Binary Mode
Opens file in binary format, used with other modes


```
open('path/to/file', mode=‘a+')
open('path/to/file', mode=‘w+')
open('path/to/file', mode=‘r+')
```
Lorem ipsum dolor sit amet, consectetuer adipiscing elit...
pointer: read and write operations will occur here
~~Lorem ipsum dolor sit amet, consectetuer adipiscing elit...~~
pointer: write and read operations will occur here
Lorem ipsum dolor sit amet, consectetuer adipiscing elit...
pointer: file operations will occur here

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.*


How to read a file in Python?
Use read()
Reads entire file as a single string
Use readline()
Reads one line at a time
Use readlines()
Reads all lines into a list

```
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.*

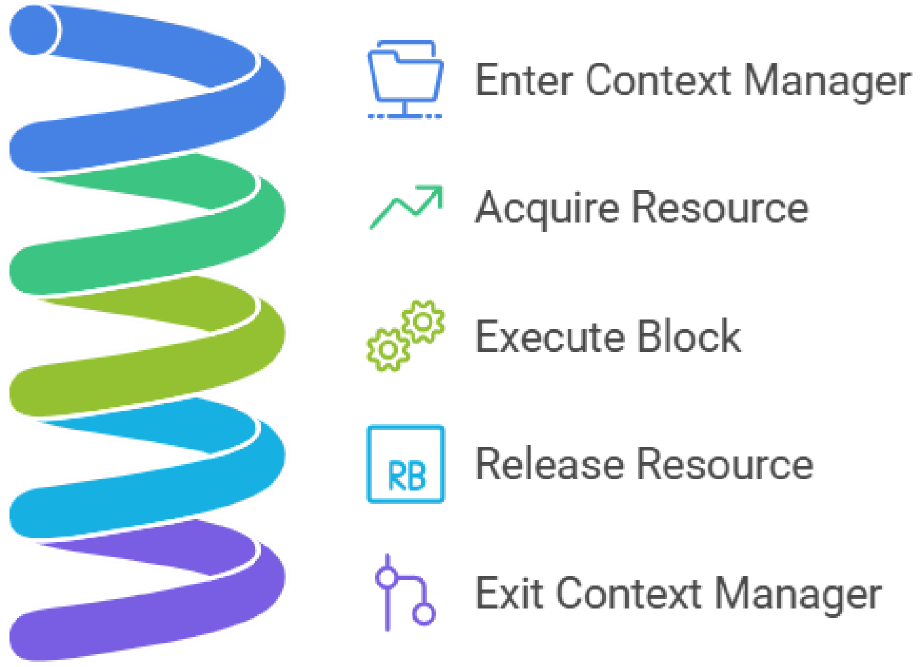
How to write a file to Python?
Use write()
Writes a string to a file without adding a newline
Use writelines()
Writes a list of strings to a file, each on new line

```
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

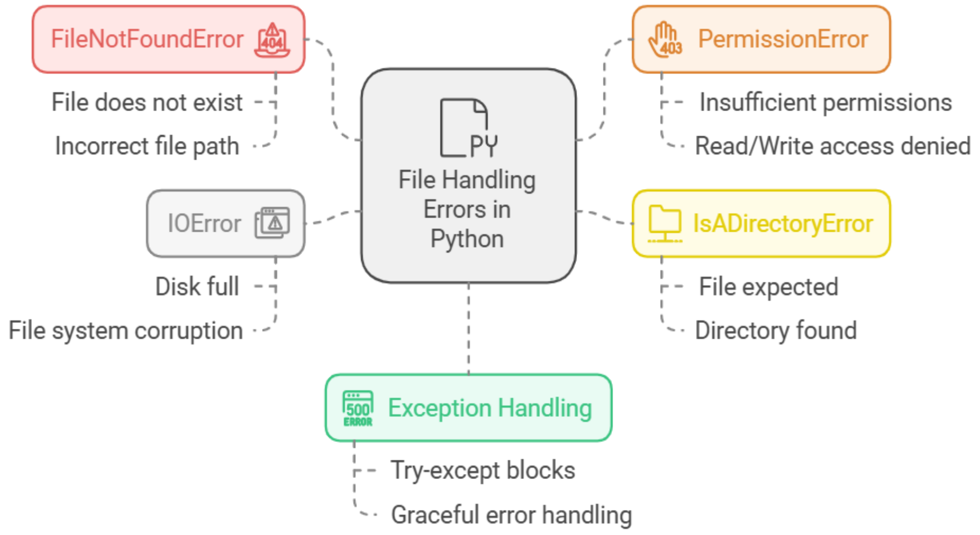
- File context manager is a way to handle files using the with statement.
- It ensures that the file is closed automatically after the block of code is executed.
- Even if an exception occurs within the block, the file will still be closed properly.
- It makes the code more readable and concise.
- File 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.")
```

Happy Coding