Computer Science By SAMPAT LILER

These complete notes have been made for class 12th board computer science exam.

Exception Handling in Python

In Python, errors can occur due to various reasons:

- Syntax errors (violating Python's rules)
- Runtime errors (occurring during execution)
- Logical errors (wrong logic but no error message)

Among these, **exceptions** are runtime errors that disrupt normal program execution. Python allows handling these errors using exception handling mechanisms, which prevent abrupt program termination.

Syntax Errors

A **syntax error** occurs when the Python interpreter detects incorrect syntax in the code. These errors must be fixed before running the program.

Example:

```
print("Hello"  # Missing closing parenthesis
```

Error Output:

```
SyntaxError: unexpected EOF while parsing
```

The interpreter detects a missing closing parenthesis and reports a SyntaxError.

Exceptions

Even if the syntax is correct, an error may occur during execution. These are called **exceptions** and include errors like:

- Division by zero
- Accessing an undefined variable
- Opening a non-existent file

Example:

```
num = 10 / 0  # Division by zero
```

Error Output:

```
ZeroDivisionError: division by zero
```

Since dividing by zero is undefined, Python raises a ZeroDivisionError.

Built-in Exceptions

Python provides several built-in exceptions to handle common errors. Some examples:

Exception Name	Description
SyntaxError	Raised for incorrect syntax
ZeroDivisionError	Raised when dividing by zero
ValueError	Raised when an operation receives an invalid argument
NameError	Raised when a variable is not defined
IndexError	Raised when accessing an out-of-range list index
TypeError	Raised when an operation is performed on incompatible types

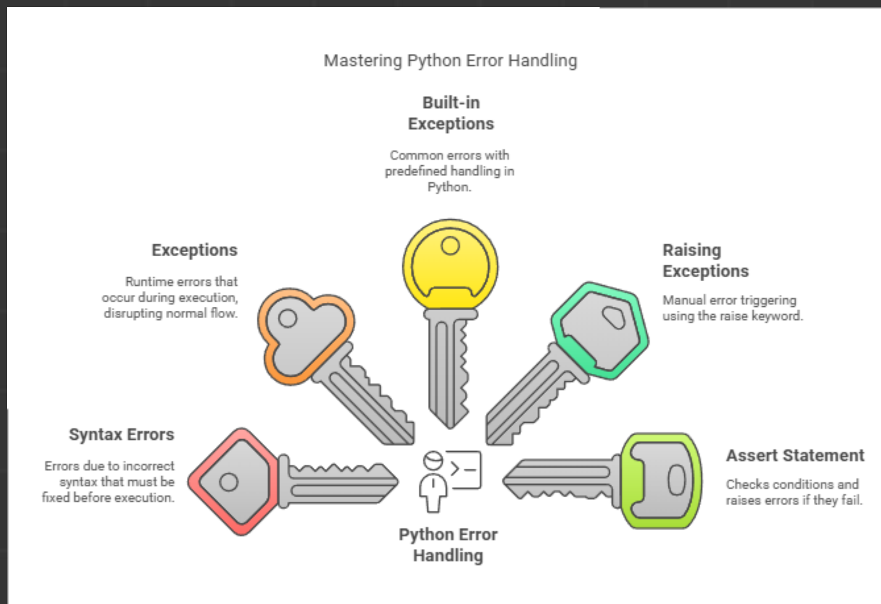
Example:

```
x = int("abc")  # Invalid integer conversion
```

Error Output:

ValueError: invalid literal for int() with base 10: 'abc'

COMPLETE NOTES AND LECTURES BY SAMPAT LILER SIR



Raising Exceptions

Python allows raising exceptions manually using the raise keyword.

Example 1: Raising an Exception

```
x = -5
if x < 0:
    raise ValueError("Negative numbers are not allowed")
```

Output:

ValueError: Negative numbers are not allowed

The raise Statement

Syntax:

```
raise ExceptionType("Custom error message")
```

Raising IndexError

```
list1 = [1, 2, 3]
index = 5
if index >= len(list1):
    raise IndexError("Index out of range")
```

Output:

IndexError: Index out of range

The assert Statement

The assert statement checks a condition and raises an AssertionError if it fails.

```
x = -10
```

```
assert x >= 0, "Negative number detected"
```
Output:

```
AssertionError: Negative number detected
```

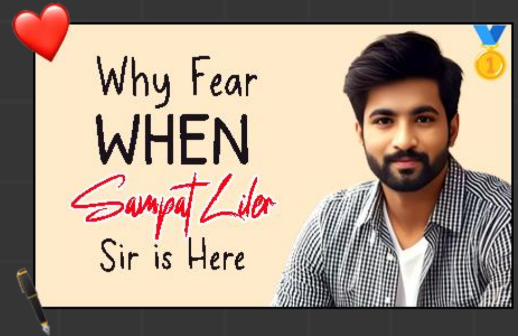
Handling Exceptions

Exception handling prevents program crashes by allowing us to catch and handle errors.

Need for Exception Handling

Without handling, an error stops the program execution. Exception handling:

- Prevents abrupt termination
- Allows alternative solutions
- Improves program reliability



Process of Exception Handling

When an exception occurs, Python:

1. Creates an **exception object**
2. Searches for an **exception handler**
3. If found, executes the handler; otherwise, the program stops.

Catching Exceptions

Python uses try...except to catch exceptions.

Syntax:

```
try:
    # Code that may raise an exception

except ExceptionType:
    # Code to handle the exception
```

Example: Handling ZeroDivisionError

```
try:
    x = int(input("Enter a number: "))
    result = 10 / x
    print("Result:", result)
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

Output 1 (User enters 5):

```
Result: 2.0
```

Output 2 (User enters 0):

```
Cannot divide by zero!
```

Handling Multiple Exceptions

We can handle different errors using multiple except blocks.

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
```

```
except ZeroDivisionError:
    print("Cannot divide by zero!")
except ValueError:
    print("Invalid input! Please enter a number.")
```

Output 1 (User enters 0):

Cannot divide by zero!

Output 2 (User enters abc):

Invalid input! Please enter a number.

Catching All Exceptions

If we don't know the type of exception, we can use except: without specifying an error.

```
try:
    x = int(input("Enter a number: "))
    result = 10 / x
except:
    print("An error occurred!")
```

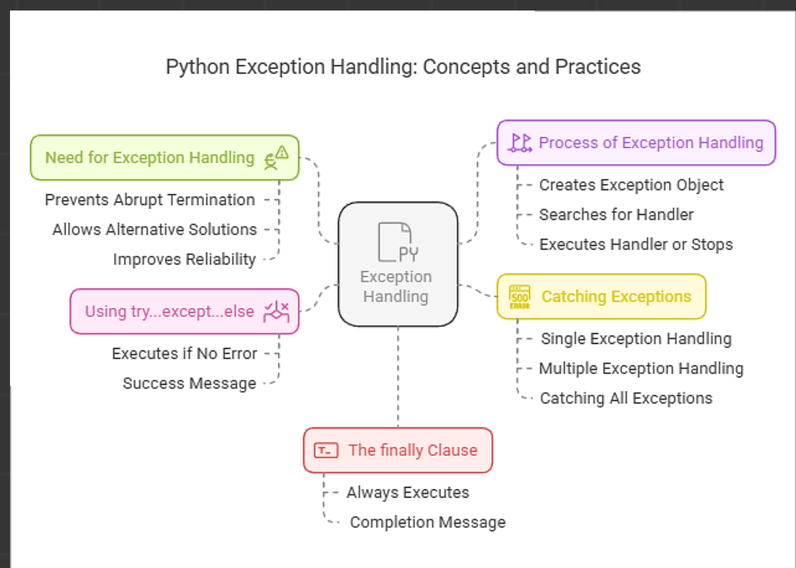
Using try...except...else

The else block runs only if no exception occurs.

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

Output 1 (User enters 5):

Division successful! Result: 2.0

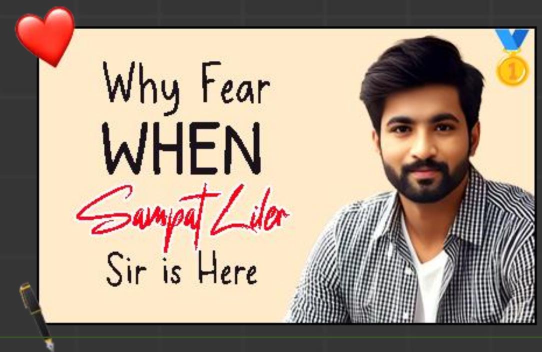


The finally Clause

The finally block **always executes**, whether an exception occurs or not.

Example:

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



Output 1 (User enters 5):
Division successful! Result: 2.0
Execution complete!
Output 2 (User enters 0):
Cannot divide by zero!
Execution complete!

Summary

- **Syntax errors** occur due to incorrect syntax and must be fixed before execution.
- **Exceptions** occur during runtime and need to be handled.
- **Built-in exceptions** include ZeroDivisionError, ValueError, IndexError, etc.
- The **raise statement** manually raises exceptions.
- The **assert statement** checks conditions and raises AssertionError if false.
- Exception handling uses **try...except** to catch errors and prevent program crashes.
- The **else block** runs if no exception occurs.
- The **finally block** always executes, ensuring resource cleanup.

---

2. File Handling in Python

---

1. Introduction to Files

A **file** is a named location on a computer's storage device where data is stored permanently. When we run a Python program, data exists only during execution. If we want to **store data permanently** (like employee records, sales data, etc.), we need to use files.
In Python, files help in:

- Storing data permanently.
- Avoiding repetitive data entry.
- Managing large volumes of data efficiently.

Types of Files
There are two main types of files:

1. **Text Files** – Human-readable files containing characters (e.g., .txt, .csv, .py).
2. **Binary Files** – Machine-readable files containing 0s and 1s (e.g., images, videos, .exe files).

---

2. Opening and Closing a Text File

To work with files in Python, we use the open() function.
Opening a File

```
file_object = open("example.txt", "r")  # Opens the file in read mode
```

Here, "r" is the **mode** which specifies how the file will be accessed.
Modes of Opening Files

Mode	Description
"r"	Read mode (default). File must exist.
"w"	Write mode. Creates a new file or overwrites if exists.
"a"	Append mode. Adds data at the end of the file.
"r+"	Read & Write mode. File must exist.
"w+"	Read & Write mode. Overwrites file if exists.
"a+"	Append & Read mode. Creates file if not exists.

Closing a File

Once done, always **close** the file to free system resources.

```
file_object.close()
```

Using with Statement (Auto-close)

The with statement **automatically closes** the file after execution.

```
with open("example.txt", "r") as file:
    data = file.read()  # Read file content
```

COMPLETE NOTES AND LECTURES BY SAMPAT LILER SIR

3. Writing to a Text File

To write data into a file, open it in **write ("w") or append ("a") mode**.

Using write() Method

Writes a string to the file.

```
file = open("example.txt", "w")
file.write("Hello, this is a test file!\n")
file.close()
```

Using writelines() Method

Writes multiple lines at once.

```
file = open("example.txt", "w")
lines = ["Hello!\n", "Python file handling is easy.\n"]
file.writelines(lines)
file.close()
```

⚠ If opened in "w" mode, it **overwrites existing content**.

---

4. Reading from a Text File

To read data, open the file in **read ("r") mode**.

Using read() Method

Reads the entire file content.

```
file = open("example.txt", "r")
data = file.read()
print(data)
file.close()
```

Using readline() Method

Reads **one line** at a time.

```
file = open("example.txt", "r")
print(file.readline())  # Reads the first line
file.close()
```

Using readlines() Method

Reads **all lines** and returns a list.

```
file = open("example.txt", "r")
lines = file.readlines()
print(lines)
file.close()
```

---

5. Setting Offsets in a File (seek() and tell())

Sometimes, we may need to **move** within a file while reading or writing.

tell() Method

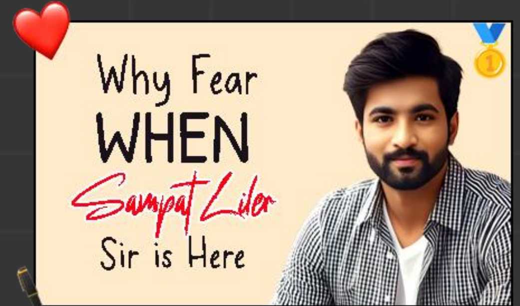
Returns the **current position** of the file pointer.

```
file = open("example.txt", "r")
print(file.tell())  # Shows position
```

seek() Method

Moves the file pointer to a specific byte position.

```
file = open("example.txt", "r")
file.seek(5)  # Moves to the 5th byte
print(file.read())  # Reads from position 5
file.close()
```



---

6. Creating and Traversing a Text File

Creating a File and Writing Data

```
file = open("practice.txt", "w")
file.write("Python file handling example.\n")
file.write("Learning file operations.\n")
file.close()
```

Reading a File Line by Line

```
file = open("practice.txt", "r")
for line in file:
    print(line.strip())  # strip() removes newline characters
file.close()
```

---

7. The Pickle Module (Binary File Handling)

Python provides the pickle module for **serializing (pickling) and deserializing (unpickling)** Python objects into binary files.

Pickling (Saving Python Object to File)

```
import pickle

data = {"Name": "John", "Age": 25, "City": "New York"}
file = open("data.pkl", "wb")  # Open in binary write mode
pickle.dump(data, file)  # Dump data
file.close()
```

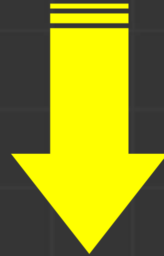
Unpickling (Loading Object from File)

```
file = open("data.pkl", "rb")  # Open in binary read mode
loaded_data = pickle.load(file)  # Load data
file.close()

print(loaded_data)  # {'Name': 'John', 'Age': 25, 'City': 'New York'}
```

Summary

- **Files** allow permanent data storage.
- **Text files** store readable characters; **binary files** store data in bytes.
- Use open("filename", mode) to **open a file**.
- Use close() or with statement to **close a file**.
- **Write data** using write() and writelines().
- **Read data** using read(), readline(), and readlines().
- **Move within a file** using seek() and tell().
- **Pickle module** is used for storing and retrieving Python objects.



👉 Subscribe Youtube Channel - Anvira Education - YouTube

👉 Join Course - Https://Anviraeducation.Com/

👉 Follow Us On Facebook - Https://Www.Facebook.Com/Anviraedu

👉 Follow Us On Instagram - https://www.instagram.com/anvira_edu/

👉 Sampat Sir Instagram - https://www.instagram.com/writersampat/

👉 Join Our Telegram Channel - https://t.me/Anviraeducation20