ERROR & EXCEPTION HANDLING IN PYTHON

Handling errors effectively for robust Python applications

MUKESH KUMAR

AGENDA

- Introduction to Errors & Exceptions
- Handling Exceptions using try and except
- Handling Multiple Exceptions
- Using else and finally Blocks
- Raising Exceptions Using raise
- Best Practices for Exception Handling
- Real-World Use Cases
- Summary

Introduction to Errors

What is an error?

- A programming error is a mistake in a program that prevents it from running correctly. Errors can occur during the development or execution of a program

Types of Errors

Syntax Errors – Occur when Python encounters incorrect syntax in the code (e.g., missing colons, incorrect indentation).

Runtime Errors (Exceptions) – Occur while the program is running, often due to invalid operations (e.g., division by zero, accessing an undefined variable).

Logical Errors – Occur when the program runs without crashing but produces incorrect results due to a flaw in the logic.

Syntax Errors

- A syntax error in Python occurs when the interpreter can't understand your code because it doesn't follow the language's rules.
- The interpreter detects these errors when it parses the code, before execution.
- Syntax errors prevent the code from running

Causes of Syntax Errors

- **Incorrect punctuation** Missing, misplaced, or mismatched punctuation marks like parentheses, brackets, braces, quotes, commas, and colons can cause syntax errors.
- **Misspelled keywords** Using the wrong spelling or case for Python keywords will result in an error.
- **Indentation errors** Python uses indentation to define code blocks, so incorrect indentation will cause syntax errors.

Causes of Syntax Errors

- **Invalid variable names** Using illegal characters in variable names leads to syntax errors.
- **Misusing operators** Incorrect use of the assignment operator (=) can cause syntax errors.

Syntax Err Examples

- Refer Notebook : Syntax_Err_Examples.ipynb

Logical Errors

- A logical error in Python is a type of programming mistake that occurs when the code executes without crashing but produces incorrect or unexpected results.
- Unlike syntax errors, which are detected by the interpreter, logical errors go unnoticed during execution, making them particularly challenging to identify and fix

Characteristics of Logical Errors

- **No Crash:** The program runs without any runtime or syntax errors.
- **Incorrect Output:** The results produced by the program do not align with the intended logic or expected outcomes.
- **Difficult to Detect:** Since there are no error messages, identifying logical errors often requires careful debugging and testing.

Common Causes

- **Incorrect Assumptions**: Misunderstanding the problem can lead to flawed logic in the code.
- **Invalid Logic**: Implementing an incorrect algorithm or formula can yield wrong results.

Logical Err Example

- Refer notebook : Logical_Err_Examples.ipynb

Runtime Errors

- A runtime error in Python occurs during the execution of a program, after it has already passed the interpreter's syntax checks.
- These errors are not detected when the script is parsed but arise when a specific line of code is executed, leading the program to halt unexpectedly.
- Runtime errors are also known as exceptions

Runtime Errors

- All exceptions are runtime errors because exceptions occur during the execution of a program.

- Not all runtime errors are exceptions, as some runtime errors (like infinite loops or incorrect logic) do not necessarily raise exceptions but still lead to issues.

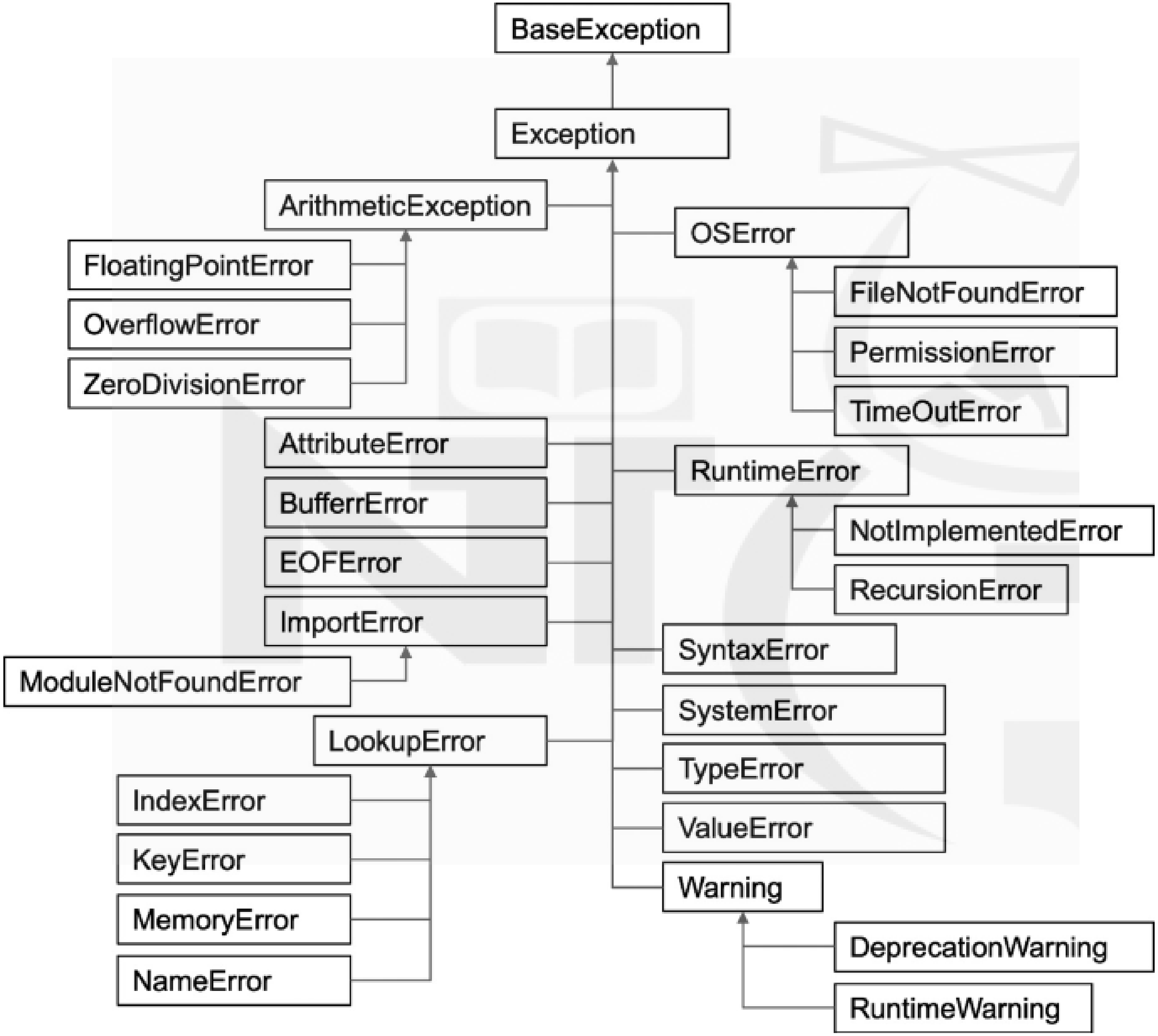
Introduction to Exceptions

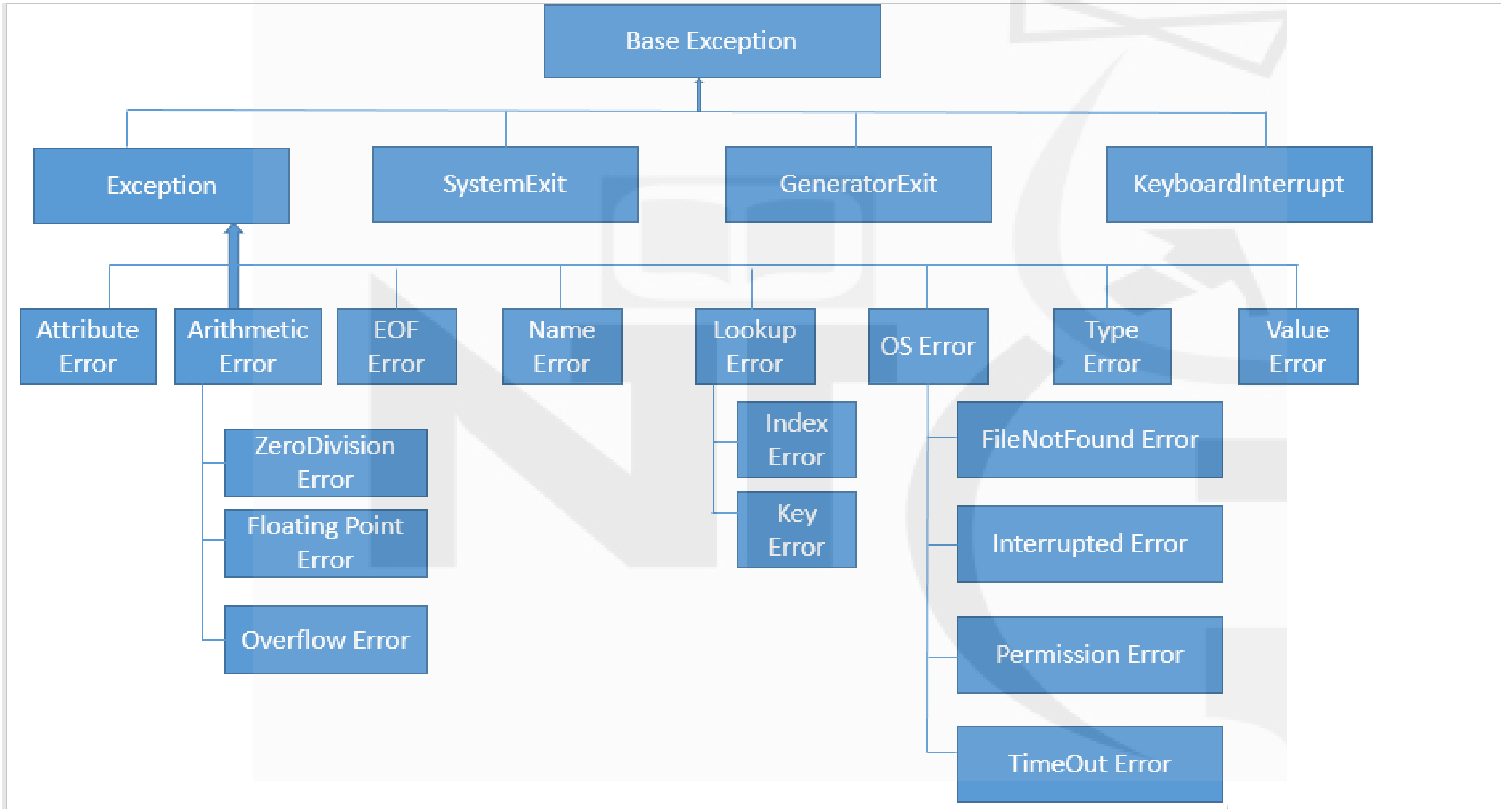
- What is an Exception?
  - Exceptions in Python are errors that occur during the execution of a program, disrupting the normal flow of the program
  - Common Python Exceptions:
    - - ZeroDivisionError
    - - IndexError
    - - KeyError
    - - TypeError

Built-in Exceptions

- Python's standard library includes a wide range of built-in exceptions that address common errors.

- When a built-in exception occurs, the corresponding exception handler code is executed, displaying the reason and the name of the exception.


BaseException
Exception
ArithmeticException
FloatingPointError
OverflowError
ZeroDivisionError
AttributeError
BufferrError
EOFError
ImportError
ModuleNotFoundError
LookupError
IndexError
KeyError
MemoryError
NameError
OSError
FileNotFoundError
PermissionError
TimeOutError
RuntimeError
NotImplementedError
RecursionError
SyntaxError
SystemError
TypeError
ValueError
Warning
DeprecationWarning
RuntimeWarning

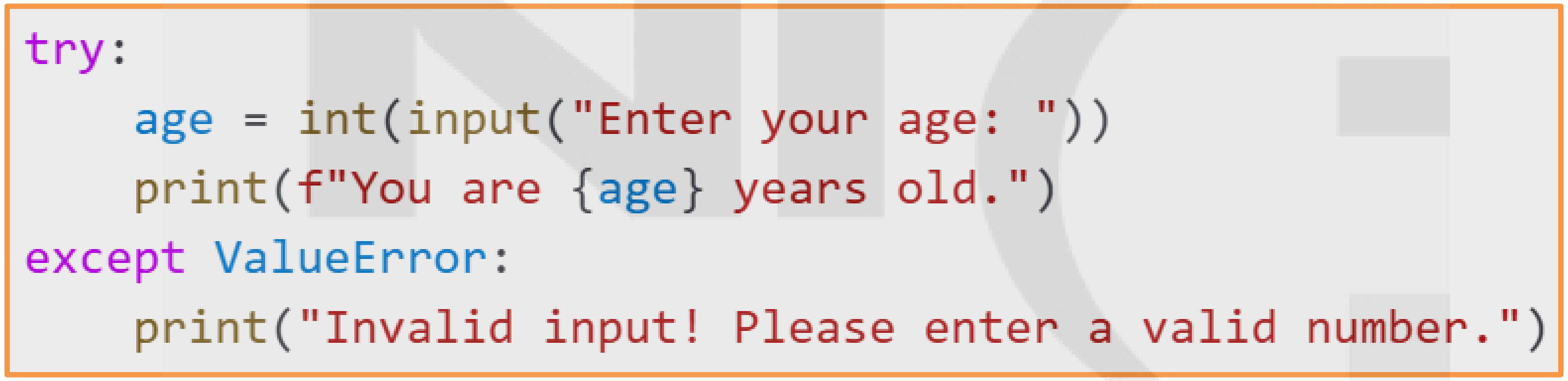

Base Exception
Exception
SystemExit
GeneratorExit
KeyboardInterrupt
Attribute Error
Arithmetic Error
EOF Error
Name Error
Lookup Error
OS Error
Type Error
Value Error
ZeroDivision Error
Floating Point Error
Overflow Error
Index Error
Key Error
FileNotFound Error
Interrupted Error
Permission Error
TimeOut Error

Exception Examples

- Refer Notebook: ExceptionExamples.ipynb

Exception Handling Using try and except

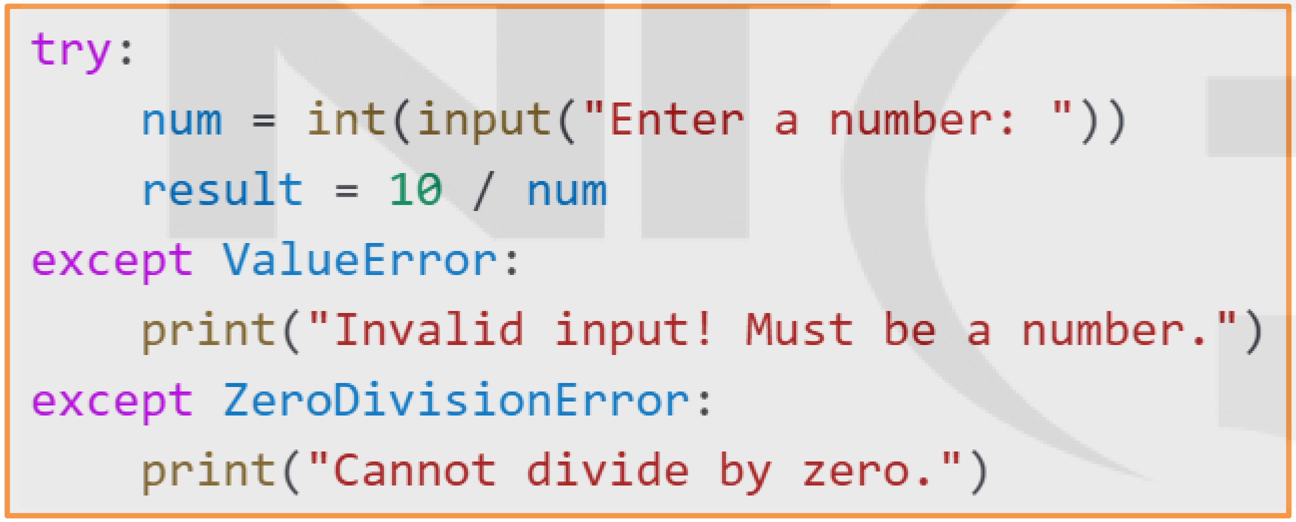
- Real-World Example: Handling User Input:



```python
try:
    age = int(input("Enter your age: "))
    print(f"You are {age} years old.")
except ValueError:
    print("Invalid input! Please enter a valid number.")
```

Handling Multiple Exceptions

- Using multiple except blocks:



```python
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ValueError:
    print("Invalid input! Must be a number.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

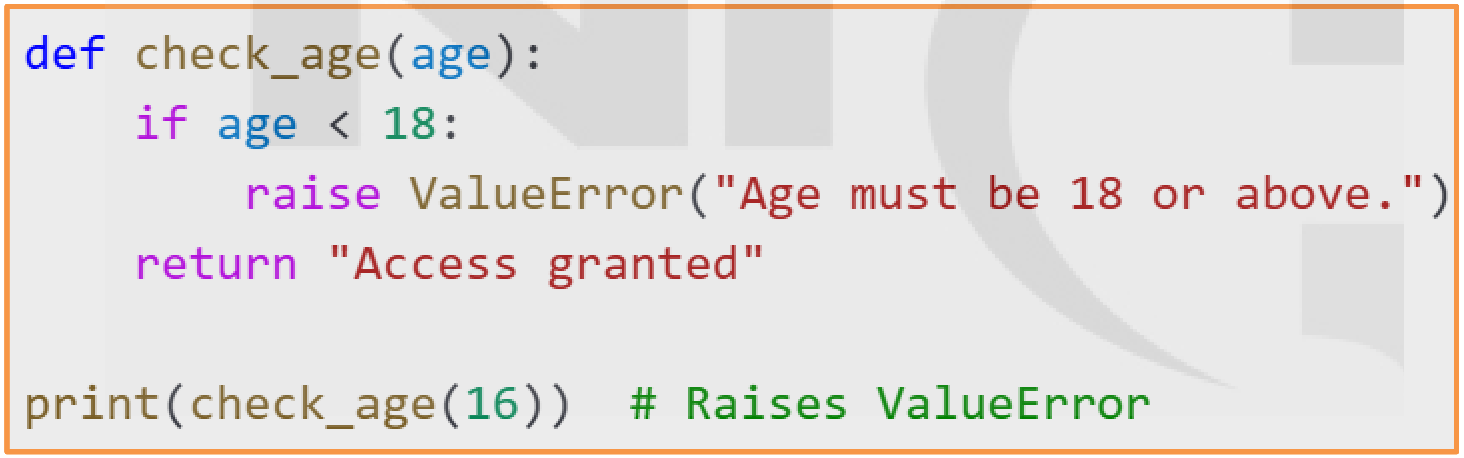
Using else and finally Blocks

- else executes if no exception occurs.
- finally executes always, regardless of exception occurrence.

```
try:
    num = int(input("Enter a number: "))
except ValueError:
    print("Invalid input!")
else:
    print("You entered:", num)
finally:
    print("Execution completed.")
```

Raising Exceptions Using raise

- Why use raise?
- To trigger custom error messages when conditions are met.



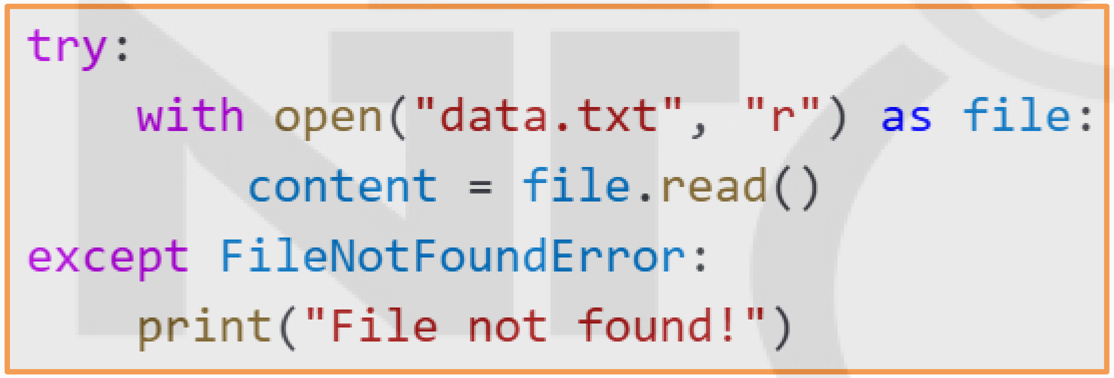
```python
def check_age(age):
    if age < 18:
        raise ValueError("Age must be 18 or above.")
    return "Access granted"

print(check_age(16))  # Raises ValueError
```

Best Practices for Exception Handling

- Be specific with exception types.
- Avoid using bare **except**: clauses.
- Use logging instead of **print().**
- Handle exceptions at the appropriate level.

Real-World Use Cases

- File Handling:



```python
try:
    with open("data.txt", "r") as file:
        content = file.read()
except FileNotFoundError:
    print("File not found!")
```

Summary

- What We Covered:
- - Types of Errors and Exceptions
- - Handling Exceptions with try-except
- - Using else, finally, and raise
- - Creating Custom Exceptions
- - Best Practices and Real-World Use Cases