Python

Full stack Skills Bootcamp

Introducing Python Error Handling and Exceptions

What is Error Handling?

- Error handling is a critical aspect of programming that allows developers to manage unexpected events gracefully. Exceptions are special conditions that arise during the execution of a program, often indicating an error or an unusual circumstance.

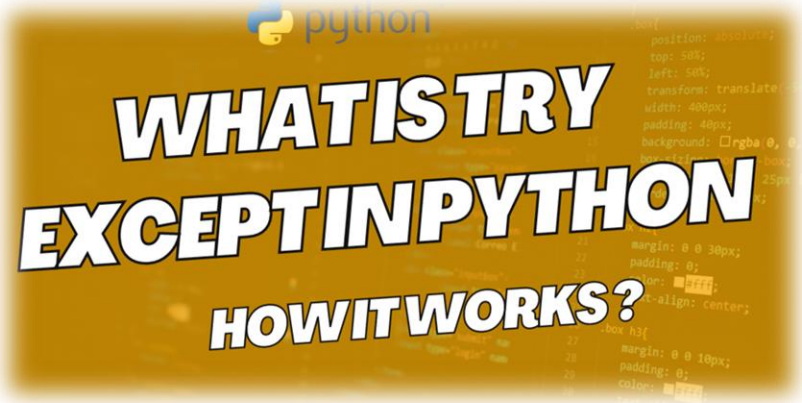
- Importance:

Proper error handling ensures that programs do not crash unexpectedly and can provide informative feedback to users and developers. Understanding how to manage exceptions allows for robust and resilient code.

Basic Try-Except Example

```python
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Error: You cannot divide by zero!")
```

```
Error: You cannot divide by zero!
```



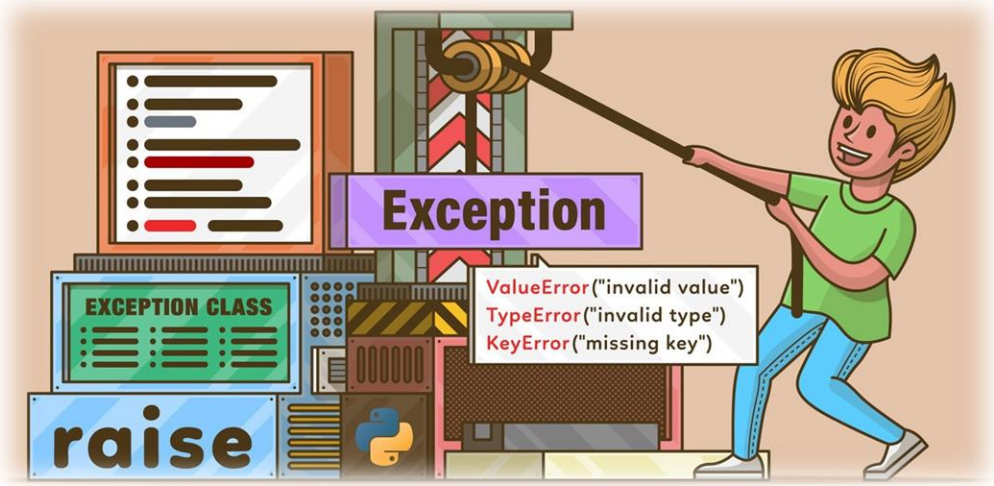
- In this example, attempting to divide by zero raises a ZeroDivisionError. Instead of crashing the program, the error is caught in the except block, allowing you to respond with a user-friendly message.
- This structure helps isolate problematic code and makes it clear where exceptions may occur.

Handling Multiple Exceptions

```python
try:
    number = int("Step8up")
except (ValueError, TypeError) as e:
    print(f"Error: {e}")
```

```
Error: invalid literal for int() with base 10: 'Step8up'
```

- Python allows you to handle multiple exceptions in a single except clause by enclosing them in parentheses.
- Here, converting a string that cannot be interpreted as an integer raises a ValueError. The except block catches both ValueError and TypeError, allowing for concise error handling.
- This feature is useful when you anticipate multiple types of errors that could occur from a single operation.



Using Finally to Execute Code Regardless of an Exception

```
try:
    file = open("test.txt", "r")
except FileNotFoundError:
    print("Error: File not found.")
finally:
    print("This will always execute, whether an exception occurs or not.")
```

- The finally block will execute after the try block, regardless of whether an exception was raised or not. This is especially useful for cleanup operations, such as closing files or releasing resources.
- In this example, if the specified file does not exist, a FileNotFoundError is caught, and a message is printed. Regardless of the error, the finally block ensures that a closing message is always displayed.

Using Else Block

```python
try:
    number = 5
    result = number / 2
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
else:
    print(f"The result is {result}.")
```

```
The result is 2.5
```

- This slide illustrates the use of the else block for handling successful operations. If the division succeeds, the result is printed.
- This structure allows for clearer logic in your code, distinguishing between error cases and regular flow without nesting.

Raising Custom Exceptions

```python
age = -1
if age < 0:
    raise ValueError("Age cannot be negative!")
```

```
Traceback (most recent call last):
  ...
ValueError: Age cannot be negative!
```

- You can raise exceptions intentionally using the raise statement. This is particularly useful for enforcing rules and constraints within your code.
- This example checks the validity of the age variable. If it's negative, a ValueError is raised, halting execution and providing a clear error message.

Exception Example with Detailed Logging

```python
try:
    my_list = [1, 2, 3]
    print(my_list[5])  # This will raise an IndexError
except IndexError:
    print("Error: Index out of range.")
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
```

```
Error: Index out of range.
```

- Detailed error logging helps developers understand what went wrong and where, improving debugging efforts and application maintenance.
- This example highlights how catching specific exceptions, like IndexError, can lead to clearer, more informative messages. The logging of the error gives insights into what caused the issue, aiding in troubleshooting.
- By catching different types of exceptions separately, you gain better control over how to respond to each situation.

Conclusion

Key Points

- Always use specific exceptions to catch and handle errors effectively. This prevents unintended behaviours and simplifies debugging.
- Avoid using broad except clauses, as they can mask real problems in the code.
- Consider implementing logging to capture exception details, which aids in understanding issues when they arise.
- Always clean up resources (like files and network connections) in the finally block to maintain system integrity.