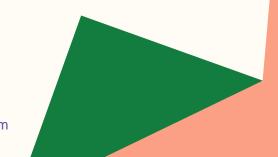


Day 14 - Python Basics

Shaida Muhammad

Thursday, March 20, 2025

ShaidaSherpao@gmail.com



Agenda

Python Online Free Ramzan Course 2025 Taught by: Shaida Muhammad

1	What are errors?	6	Raising exceptions with raise
2	Types of errors (syntax errors, runtime errors, logic errors)	7	Exception chaining
3	Handling errors with try, except, else, and finally	8	Custom exceptions
4	Handling multiple exceptions	9	Best practices for error handling
5	Nested try-except blocks	10	Hands-on practice

What are Errors?

- **Definition:** Errors are issues in a program that prevent it from running correctly.
- Types of Errors:
 - Syntax Errors: Mistakes in the code structure (e.g., missing colon).
 - Runtime Errors: Errors that occur during execution (e.g., division by zero).
 - Logic Errors: Errors in the program's logic (e.g., incorrect calculation).
- Example of a Runtime Error:

print(10 / 0) # ZeroDivisionError



Handling Errors with try and except

- Purpose: Prevent the program from crashing by handling errors gracefully.
- Syntax:

```
try:
    # Code that might cause an error
except ErrorType:
    # Code to handle the error
```

```
try:
    print(10 / 0)
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

- Key Points:
 - Use try to wrap code that might raise an error.
 - Use except to handle specific errors.



Handling Multiple Exceptions

- Purpose: Handle different types of errors in the same try block.
- Syntax:

```
try:

# Code that might cause an error
except ErrorType1:

# Handle ErrorType1
except ErrorType2:

# Handle ErrorType2
```

```
try:
    num = int(input("Enter a number: "))
    print(10 / num)
except ValueError as e:
    print("Invalid input! Please enter a number.", e)
except ZeroDivisionError as e:
    print("Cannot divide by zero!", e)
```

- Key Points:
 - Use multiple except blocks to handle different errors.
 - Order matters: Specific exceptions should come before general ones.



Using **else** and **finally**

else Block: Runs if no error occurs in the try block.

```
try:
    print(10 / 2) # Output: 5.0
except ZeroDivisionError:
    print("Cannot divide by zero!")
else:
    print("Division successful!") # Output: Division successful!
```

• finally Block: Runs regardless of whether an error occurs.

```
try:
    print(10 / 0)
except ZeroDivisionError:
    print("Cannot divide by zero!")
finally:
    print("Execution complete.")
```

- Key Points:
 - Use else for code that should run only if no errors occur.
 - Use finally for cleanup actions (e.g., closing files).

Nested Try-Except Blocks

Purpose: Handle errors in nested code blocks.

Syntax:

```
try:

# Outer try block

try:

# Inner try block

except ErrorType:

# Handle inner error

except ErrorType:

# Handle outer error
```

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

- Key Points:
 - Use nested try-except blocks for granular error handling.



Raising Exceptions with raise

- Purpose: Manually trigger an exception.
- Syntax:

```
raise Exception("Error message")
```

Example:

```
age = -1
```

if age < 0:

raise ValueError("Age cannot be negative!")

- Key Points:
 - Use raise to enforce specific conditions or rules.



Exception Chaining

- **Purpose:** Preserve the original exception when raising a new one.
- Syntax:

```
raise NewException from OriginalException
```

```
try:
    print(10 / 0)
except ZeroDivisionError as e:
    raise ValueError("Invalid operation") from e
```

- Key Points:
 - Use from to chain exceptions and preserve the traceback.

Custom Exceptions

- Purpose: Create your own exception types for specific use cases.
- Syntax:

```
class CustomError(Exception):
    pass
```

• Example:

```
class NegativeNumberError(Exception):
    def __init__(self, message="Number cannot be negative!"):
        self.message = message
        super().__init__(self.message)
```

```
num = -5
if num < 0:
    raise NegativeNumberError</pre>
```

- Key Points:
 - Custom exceptions inherit from the Exception class.
 - Use them to make error handling more specific and meaningful.

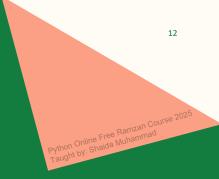
Best Practices for Error Handling

- 1. Be Specific: Catch specific exceptions instead of using a bare except.
- 2. Use finally for Cleanup: Ensure resources are released (e.g., closing files).
- **3. Avoid Silent Failures:** Always log or handle exceptions meaningfully.
- **4. Use Custom Exceptions:** For better readability and maintainability.
- **5. Test Error Cases:** Ensure your code handles errors as expected.



Hands-On Practice

```
Task 1: Handle multiple exceptions.
     try:
          num = int(input("Enter a number: "))
          print(10 / num)
     except ValueError:
          print("Invalid input! Please enter a
number.")
     except ZeroDivisionError:
          print("Cannot divide by zero!")
Task 2: Use else and finally blocks.
     try:
          print(10 / 2)
     except ZeroDivisionError:
          print("Cannot divide by zero!")
     else:
          print("Division successful!")
     finally:
          print("Execution complete.")
Task 3: Raise a ValueError for invalid input.
     age = -1
     if age < 0:
          raise ValueError("Age cannot be
negative!")
```



class NegativeNumberError(Exception):
 pass
num = -5
if num < 0:
 raise NegativeNumberError("Number</pre>

Task 4: Create and use a custom exception.

• Task 5: Use nested try-except blocks.

cannot be negative!")

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

Recap

- Errors can be syntax errors, runtime errors, or logic errors.
- Use try, except, else, and finally to handle errors gracefully.
- Handle multiple exceptions with multiple except blocks.
- Use nested try-except blocks for granular error handling.
- Use raise to manually trigger exceptions and from for exception chaining.
- Create custom exceptions for specific use cases.
- Follow best practices for effective error handling.



Homework

- 1. Write a program that handles a ValueError when converting user input to an integer.
- 2. Create a custom exception InvalidEmailError and raise it for invalid email formats.
- 3. Write a function that divides two numbers and handles all possible errors (e.g., ZeroDivisionError, TypeError).
- 4. Write a program with nested try-except blocks to handle errors when working with a list that has both numbers and non-numbers



Python Online Free Ramzan Course 2025
Python Online Free Ramzan Course 2025
Taught by: Shaida Muhammad

Q&A

- Do you have any questions?
- Share your thoughts.



Closing

Next class: File Handling