Exception Handling in Python

Writing Robust Code that Handles Errors Gracefully

CS 101 - Fall 2025

What Are Exceptions?

Exceptions are errors that occur during program execution.

. . .

Without proper handling, they **crash** your program!

. . .

```
# This will crash!
number = int("hello")
```

. . .

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

Why Handle Exceptions?

- **Prevent crashes** - Keep your program running
- **User-friendly** - Show helpful messages instead of cryptic errors
- **Data protection** - Don't lose work when errors occur
- **Professional code** - Handle edge cases gracefully

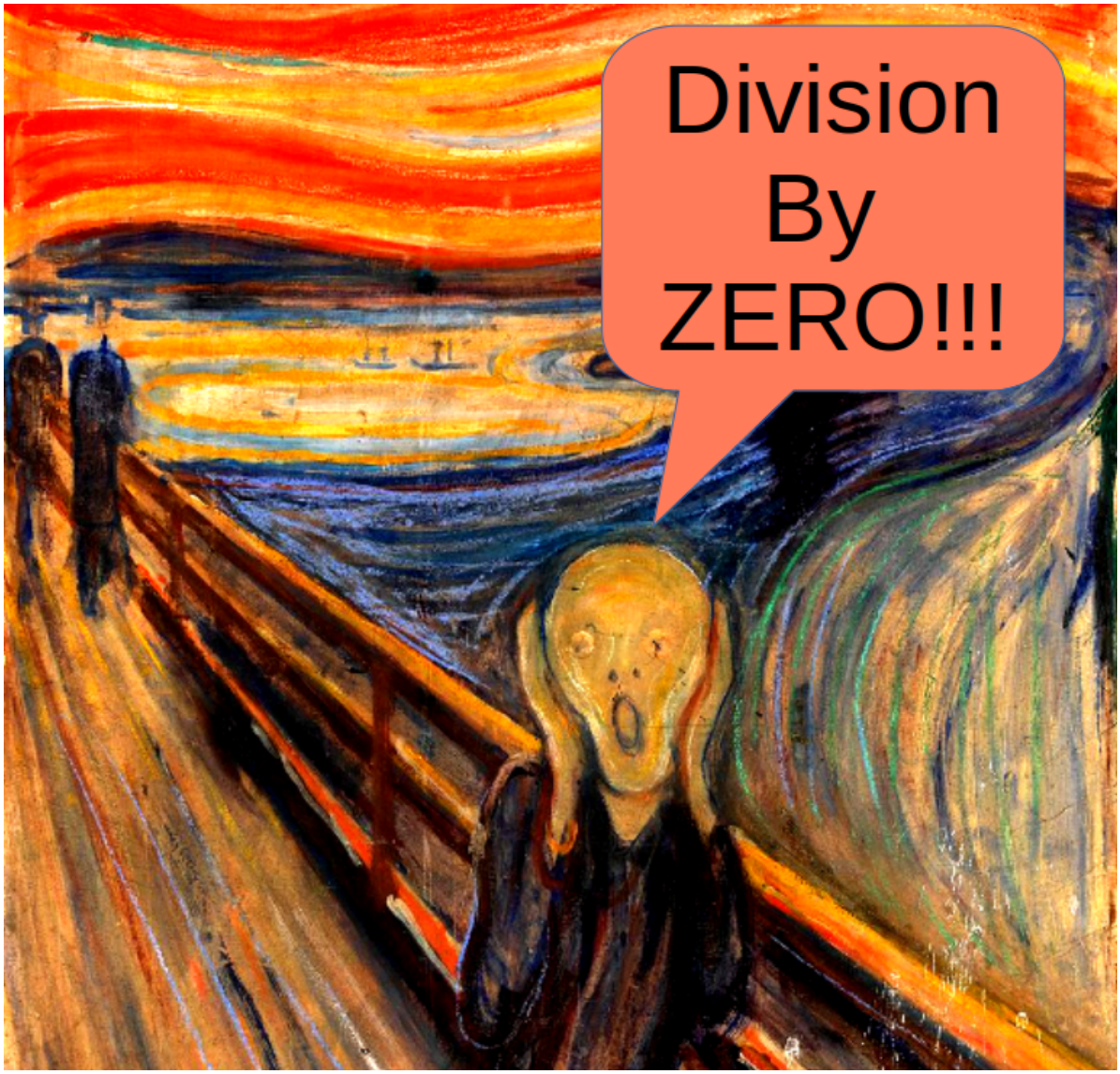
The Basic Pattern: try-except

```
try:
    # Code that might cause an error
    risky_operation()
except:
    # What to do if an error occurs
    print("Oops! Something went wrong")
```

. . .

Let's see a real example!

What Makes Software Crash?




```
1/0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[1], line 1
----> 1 1/0

ZeroDivisionError: division by zero
```

- `ZeroDivisionError` type errors result when a value is being divided by zero
- Typically the program will crash
- To avoid crashing, programmers will design a conditional statement (`if-else`) to check if a divisor is a zero to prevent a crash
- Or, they would introduce code for **exception handling**

Simple Example: Division

```
try:
    result = 10 / 0
    print(f"Result: {result}")
except:
    print("Cannot divide by zero!")
```

. . .

Output:

```
Cannot divide by zero!
```

. . .

The program continues running!

A Better Approach: Specific Exceptions

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
except TypeError:
    print("Invalid types for division!")
```

. . .

Why is this better?

- Handle different errors differently
- More precise error messages
- Easier to debug

Common Exception Types

Exception	When It Happens
`ValueError`	Invalid value (e.g., `int("hello")`)
`ZeroDivisionError`	Division by zero
`TypeError`	Wrong type (e.g., `"5" + 5`)
`FileNotFoundError`	File doesn't exist
`IndexError`	List index out of range
`KeyError`	Dictionary key doesn't exist

Challenge #1: Fix the Code

This code crashes. Add exception handling to make it safe:

```
user_input = input("Enter a number: ")
number = int(user_input)
result = 100 / number
print(f"100 divided by {number} is {result}")
```

. . .

Hints:

- What if the user enters text instead of a number?
- What if they enter 0?

Solution #1

```python
user_input = input("Enter a number: ")

try:
    number = int(user_input)
    result = 100 / number
    print(f"100 divided by {number} is {result}")
except ValueError:
    print("Please enter a valid number!")
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

Getting Error Details: The `as` Keyword

```python
try:
    age = int(input("Enter your age: "))
except ValueError as error:
    print(f"Invalid input: {error}")
```

. . .

Example output:

```
Enter your age: twenty
Invalid input: invalid literal for int() with base 10: 'twenty'
```

The `else` Clause

Runs **only if no exception occurs**:

```python
try:
    number = int(input("Enter a number: "))
except ValueError:
    print("That's not a valid number!")
else:
    print(f"You entered: {number}")
    print("Great job!  ")
```

The `finally` Clause

Runs **no matter what** - even if there's an exception:

```
try:
    file = open("data.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File not found!")
finally:
    print("Cleanup: Closing file handle")
    # This always runs!
```

. . .

Perfect for **cleanup operations**!

Complete Exception Structure

```
try:
    # Try to do something risky
    risky_code()
except SpecificError:
    # Handle specific error
    handle_error()
except AnotherError as e:
    # Handle another error with details
    print(f"Error: {e}")
else:
    # Success! No exceptions occurred
    success_code()
finally:
    # Always runs - cleanup code
    cleanup()
```

Challenge #2: File Reader

Create a function that safely reads a file:

```
def read_file_safely(filename):
    # Add exception handling here!
    file = open(filename, 'r')
    content = file.read()
    file.close()
    return content
```

. . .

Requirements: - Handle `FileNotFoundError` - Always close the file (use `finally`) - Return `None` if file doesn't exist

---

Solution #2

```
def read_file_safely(filename):
    try:
        file = open(filename, 'r')
        content = file.read()
        return content
    except FileNotFoundError:
        print(f"File '{filename}' not found!")
        return None
    finally:
        try:
            file.close()
        except:
            pass  # File was never opened
```

Multiple Exceptions in One Line

When the same action handles multiple errors:

```
try:
    value = int(input("Enter a number: "))
    result = 100 / value
except (ValueError, ZeroDivisionError):
    print("Invalid input or division by zero!")
```

. . .

```
# Or handle them separately:
except (ValueError, ZeroDivisionError) as error:
    print(f"Error occurred: {type(error).__name__}")
```

Real-World Example: Calculator

```
def safe_calculator():
    try:
        num1 = float(input("First number: "))
        operator = input("Operator (+, -, *, /): ")
        num2 = float(input("Second number: "))

        if operator == "+":
            result = num1 + num2
        elif operator == "/":
            result = num1 / num2
        # ... more operators

        print(f"Result: {result}")
    except ValueError:
        print("Please enter valid numbers!")
    except ZeroDivisionError:
        print("Cannot divide by zero!")
```

Challenge #3: List Access

Fix this code to handle invalid indices:

```
def get_item(my_list, index):
    item = my_list[index]
    return item

# Test it:
fruits = ["apple", "banana", "cherry"]
print(get_item(fruits, 5))
```

. . .

Bonus: Also handle if `index` is not a number!

Solution #3

```
def get_item(my_list, index):
    try:
        item = my_list[index]
        return item
    except IndexError:
        print(f"Index {index} is out of range!")
        return None
    except TypeError:
        print("Index must be an integer!")
        return None

# Test it:
fruits = ["apple", "banana", "cherry"]
print(get_item(fruits, 5))    # Index error
print(get_item(fruits, "2"))  # Type error
```

Raising Your Own Exceptions

So far we've been **catching** exceptions. But you can also **raise** them!

. . .

Why raise exceptions?

- Signal that something is wrong in your code
- Enforce rules and constraints
- Communicate errors to calling code
- Make your functions fail early and clearly

The `raise` Statement

Basic syntax:

```
raise ExceptionType("Error message")
```

. . .

Common built-in exceptions to raise:

- `ValueError` - Invalid value
- `TypeError` - Wrong type
- `RuntimeError` - General runtime error
- `NotImplementedError` - Feature not implemented yet

Example: Validating Input

```
def set_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative!")
    if age > 150:
        raise ValueError("Age seems unrealistic!")
    return age
```

. . .

Using it:

```
try:
    user_age = set_age(-5)
except ValueError as error:
    print(f"Error: {error}")
```

. . .

Output: `Error: Age cannot be negative!`

When to Raise Exceptions

1. **Invalid input** - User provides bad data
2. **Violated preconditions** - Function requirements not met
3. **Impossible operations** - Can't complete the requested action
4. **Resource problems** - File missing, network down, etc.

. . .

Principle: Raise exceptions when you **can't** fix the problem, but the **caller** might be able to!

Practical Example: Division Function

```
def safe_divide(numerator, denominator):
    """Divide two numbers with validation."""

    if not isinstance(numerator, (int, float)):
        raise TypeError("Numerator must be a number")

    if not isinstance(denominator, (int, float)):
        raise TypeError("Denominator must be a number")

    if denominator == 0:
```

```
        raise ValueError("Cannot divide by zero")

    return numerator / denominator
```

Using the Division Function

```
try:
    result = safe_divide(10, 2)
    print(f"Result: {result}")  # Works fine

    result = safe_divide(10, 0)  # This raises ValueError

except TypeError as e:
    print(f"Type error: {e}")
except ValueError as e:
    print(f"Value error: {e}")
```

. . .

Output:

```
Result: 5.0
Value error: Cannot divide by zero
```

Custom Exception Classes

Create your own exception types:

```
class NegativeNumberError(Exception):
    """Raised when a negative number is not allowed"""
    pass

def calculate_square_root(num):
    if num < 0:
        raise NegativeNumberError(
```

```
            "Cannot calculate square root of negative number"
        )
    return num ** 0.5
```

Challenge #4: Validate Email

Create a function that validates an email address:

```
def validate_email(email):
    # Your code here
    # Raise ValueError if:
    # - No @ symbol
    # - No dot after @
    # - Email is empty
    pass
```

. . .

Test cases:

```
validate_email("student@college.edu")  # Should work
validate_email("invalid-email")         # Should raise error
```

Solution #4

```
def validate_email(email):
    if not email:
        raise ValueError("Email cannot be empty!")

    if "@" not in email:
        raise ValueError("Email must contain @")

    if "." not in email.split("@")[1]:
        raise ValueError("Email must have domain extension")
```

```
    return True

try:
    validate_email("test@example.com")
    print("  Valid email!")
except ValueError as error:
    print(f"  {error}")
```

Real-World: API Request Handler

```
import requests

def fetch_data(url):
    try:
        response = requests.get(url, timeout=5)
        response.raise_for_status()
        return response.json()
    except requests.Timeout:
        print("Request timed out!")
    except requests.ConnectionError:
        print("Network connection failed!")
    except requests.HTTPError as e:
        print(f"HTTP error: {e}")
    except Exception as e:
        print(f"Unexpected error: {e}")
    finally:
        print("Request completed")
```

Best Practices

1. **Be specific** - Catch specific exceptions, not all errors
2. **Don't hide errors** - Log or display useful information
3. **Clean up resources** - Use `finally` for files, connections
4. **Fail gracefully** - Provide helpful error messages
5. **Don't overuse** - Not everything needs try-except

6. **Document exceptions** - Tell users what errors to expect

---

Common Mistakes to Avoid

```
#  DON'T: Catch everything blindly
try:
    do_something()
except:
    pass  # Silent failure is bad!

#  DO: Be specific and informative
try:
    do_something()
except SpecificError as error:
    log_error(error)
    notify_user("Operation failed")
```

Big Challenge: Bank Account

Create a `BankAccount` class with exception handling:

```
class InsufficientFundsError(Exception):
    pass

class BankAccount:
    def __init__(self, balance=0):
        # Initialize account
        pass

    def deposit(self, amount):
        # Add exception handling for negative amounts
        pass

    def withdraw(self, amount):
        # Raise InsufficientFundsError if balance too low
```

```
        # Raise ValueError if amount is negative
        pass
```

Big Solution

```
class InsufficientFundsError(Exception):
    pass

class BankAccount:
    def __init__(self, balance=0):
        if balance < 0:
            raise ValueError("Initial balance cannot be negative")
        self.balance = balance

    def deposit(self, amount):
        if amount <= 0:
            raise ValueError("Deposit amount must be positive")
        self.balance += amount
        return self.balance

    def withdraw(self, amount):
        if amount <= 0:
            raise ValueError("Withdrawal amount must be positive")
        if amount > self.balance:
            raise InsufficientFundsError(
                f"Insufficient funds. Balance: ${self.balance}"
            )
        self.balance -= amount
        return self.balance
```

Testing the Bank Account

```
try:
    account = BankAccount(100)
    print(f"Balance: ${account.balance}")

    account.deposit(50)
    print(f"After deposit: ${account.balance}")

    account.withdraw(200)  # This will fail!

except ValueError as e:
    print(f"Invalid operation: {e}")
except InsufficientFundsError as e:
    print(f"Transaction failed: {e}")
```

Quick Reference Card

Clause	Purpose	When It Runs
`try:`	Code that might fail	Always attempted
`except:`	Error handling	Only if exception occurs
`else:`	Success code	Only if NO exception
`finally:`	Cleanup code	ALWAYS runs

Key Takeaways

- Exceptions help create **robust, user-friendly** programs
- Use `try-except` to handle errors **gracefully**
- Be **specific** with exception types
- **Always clean up** resources with `finally`
- **Raise exceptions** to signal problems in your code
- Good error handling = **professional code**!
