Error handling

Joachim Vandekerckhove

Winter 2025

Introduction to Error Handling

Why is error handling important?

- Errors are inevitable in programming

Introduction to Error Handling

Why is error handling important?

- Errors are inevitable in programming
- Proper handling improves software robustness and usability

Introduction to Error Handling

Why is error handling important?

- Errors are inevitable in programming
- Proper handling improves software robustness and usability
- Helps with debugging and logging

Introduction to Error Handling

Why is error handling important?

- Errors are inevitable in programming
- Proper handling improves software robustness and usability
- Helps with debugging and logging
- Good error handling prevents crashes

Introduction to Error Handling

Why is error handling important?

- Errors are inevitable in programming
- Proper handling improves software robustness and usability
- Helps with debugging and logging
- Good error handling prevents crashes
- Most importantly, it prevents unexpected behavior

Categories of Errors in Python

- **Syntax Errors**

Categories of Errors in Python

- **Syntax Errors**
  - Caught before execution
  - Invalid Python syntax

Categories of Errors in Python

- **Syntax Errors**
  - Caught before execution
  - Invalid Python syntax
- **Runtime Errors (Exceptions)**

Categories of Errors in Python

- **Syntax Errors**
  - Caught before execution
  - Invalid Python syntax
- **Runtime Errors (Exceptions)**
  - Occur during program execution
  - Many types: ZeroDivisionError, TypeError, IndexError, etc.
  - Can be caught and handled

Categories of Errors in Python

- **Syntax Errors**
  - Caught before execution
  - Invalid Python syntax
- **Runtime Errors (Exceptions)**
  - Occur during program execution
  - Many types: ZeroDivisionError, TypeError, IndexError, etc.
  - Can be caught and handled
- **Logical Errors**

Categories of Errors in Python

- **Syntax Errors**
  - Caught before execution
  - Invalid Python syntax
- **Runtime Errors (Exceptions)**
  - Occur during program execution
  - Many types: ZeroDivisionError, TypeError, IndexError, etc.
  - Can be caught and handled
- **Logical Errors**
  - Program runs but produces wrong results
  - Hardest to detect – no error messages
  - Require careful testing and debugging

Syntax Errors

Example of a Syntax Error:

Syntax Errors

Example of a Syntax Error:

```
>>> print "Hello"
  File "<stdin>", line 1
    print "Hello"
          ^^^^^^^
SyntaxError: Missing parentheses in call to 'print'. Did you mean
    print(...)?
```

Logical Errors

Example of a Logical Error:

Logical Errors

Example of a Logical Error:

```
>>> # Task: Average the first five numbers
>>> total = 0
>>> for i in range(5):
...     total += i
>>> print(f"Average: {total/5}")
Average: 2.0
>>> # Bug: off by one
```

Runtime Errors

`ZeroDivisionError`

- Occurs when dividing by zero

Runtime Errors

`ZeroDivisionError`

- Occurs when dividing by zero
- Common in mathematical calculations

Runtime Errors

`ZeroDivisionError`

- Occurs when dividing by zero
- Common in mathematical calculations

Runtime Errors

`ZeroDivisionError`

- Occurs when dividing by zero
- Common in mathematical calculations

```
>>> 10 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Runtime Errors

`IndexError`

- Occurs when accessing a non-existent index

Runtime Errors

`IndexError`

- Occurs when accessing a non-existent index
- Common when working with sequences (lists, tuples, etc.)

Runtime Errors

`IndexError`

- Occurs when accessing a non-existent index
- Common when working with sequences (lists, tuples, etc.)

Runtime Errors

`IndexError`

- Occurs when accessing a non-existent index
- Common when working with sequences (lists, tuples, etc.)

```
>>> nums = [1, 2, 3]
>>> nums[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Runtime Errors

`TypeError`

- Occurs when an operation is performed on incompatible types

Runtime Errors

`TypeError`

- Occurs when an operation is performed on incompatible types
- Common when mixing different data types

Runtime Errors

`TypeError`

- Occurs when an operation is performed on incompatible types
- Common when mixing different data types

Runtime Errors

TypeError

- Occurs when an operation is performed on incompatible types
- Common when mixing different data types

```
>>> "hello" + 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str

>>> len(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
```

Common Python Exceptions

Exception	Description
SyntaxError	Invalid syntax
TypeError	Wrong type
ValueError	Invalid value
IndexError	Bad sequence index
KeyError	Key not found
NameError	Name not found
AttributeError	Missing attribute
ImportError	Import failed

Exception	Description
FileNotFoundError	File not found
IOError	I/O operation failed
ZeroDivisionError	Division by zero
AssertionError	Assert failed
RuntimeError	Generic error
OverflowError	Value too large
MemoryError	Out of memory
OSError	System error

I/O Related Exceptions

FileNotFoundError

I/O Related Exceptions

FileNotFoundError

- File path doesn't exist
- Common in: file operations, config loading, data processing
- Example: `open('missing.txt', 'r')`

I/O Related Exceptions

FileNotFoundError

- File path doesn't exist
- Common in: file operations, config loading, data processing
- Example: `open('missing.txt', 'r')`

I/O Related Exceptions

FileNotFoundError

- File path doesn't exist
- Common in: file operations, config loading, data processing
- Example: `open('missing.txt', 'r')`

I/O Related Exceptions

FileNotFoundError

- File path doesn't exist
- Common in: file operations, config loading, data processing
- Example: `open('missing.txt', 'r')`

IOError

I/O Related Exceptions

FileNotFoundError

- File path doesn't exist
- Common in: file operations, config loading, data processing
- Example: `open('missing.txt', 'r')`

IOError

- General input/output failures
- Disk full, pipe broken, network timeout
- Example: writing to a closed file

Data Structure Exceptions

IndexError

Data Structure Exceptions

IndexError

- Accessing sequence beyond bounds
- Example: `alphabet[27]` on a list of size 26

Data Structure Exceptions

IndexError

- Accessing sequence beyond bounds
- Example: `alphabet[27]` on a list of size 26

Data Structure Exceptions

IndexError

- Accessing sequence beyond bounds
- Example: `alphabet[27]` on a list of size 26

KeyError

Data Structure Exceptions

IndexError

- Accessing sequence beyond bounds
- Example: `alphabet[27]` on a list of size 26

KeyError

- Missing dictionary key
- Example: `dict['missing_key']`

Data Structure Exceptions

IndexError

- Accessing sequence beyond bounds
- Example: `alphabet[27]` on a list of size 26

KeyError

- Missing dictionary key
- Example: `dict['missing_key']`

Data Structure Exceptions

IndexError

- Accessing sequence beyond bounds
- Example: `alphabet[27]` on a list of size 26

KeyError

- Missing dictionary key
- Example: `dict['missing_key']`

AttributeError

Data Structure Exceptions

IndexError

- Accessing sequence beyond bounds
- Example: `alphabet[27]` on a list of size 26

KeyError

- Missing dictionary key
- Example: `dict['missing_key']`

AttributeError

- Accessing non-existent object attribute
- Example: `sdt.flaseAlarms()`

Type and Value Exceptions

`TypeError`

Type and Value Exceptions

`TypeError`

- Operation on incompatible types
- Example: `"U" + 2`

Type and Value Exceptions

`TypeError`

- Operation on incompatible types
- Example: `"U" + 2`

Type and Value Exceptions

`TypeError`

- Operation on incompatible types
- Example: `"U" + 2`

`ValueError`

Type and Value Exceptions

`TypeError`

- Operation on incompatible types
- Example: `"U" + 2`

`ValueError`

- Right type but invalid value
- Common in: parsing, conversion operations
- Example: `int("not a number")`

Type and Value Exceptions

`TypeError`

- Operation on incompatible types
- Example: `"U" + 2`

`ValueError`

- Right type but invalid value
- Common in: parsing, conversion operations
- Example: `int("not a number")`

Type and Value Exceptions

`TypeError`

- Operation on incompatible types
- Example: `"U" + 2`

`ValueError`

- Right type but invalid value
- Common in: parsing, conversion operations
- Example: `int("not a number")`

Type and Value Exceptions

`TypeError`

- Operation on incompatible types
- Example: `"U" + 2`

`ValueError`

- Right type but invalid value
- Common in: parsing, conversion operations
- Example: `int("not a number")`

`NameError`

Type and Value Exceptions

`TypeError`

- Operation on incompatible types
- Example: `"U" + 2`

`ValueError`

- Right type but invalid value
- Common in: parsing, conversion operations
- Example: `int("not a number")`

`NameError`

- Using undefined variable
- Example: typos, scope issues, using variable before assignment

Resource and System Exceptions

`MemoryError`

Resource and System Exceptions

`MemoryError`

- Out of memory
- Example: creating huge lists/arrays, infinite recursion

Resource and System Exceptions

`MemoryError`

- Out of memory
- Example: creating huge lists/arrays, infinite recursion

Resource and System Exceptions

`MemoryError`

- Out of memory
- Example: creating huge lists/arrays, infinite recursion

`OverflowError`

Resource and System Exceptions

`MemoryError`

- Out of memory
- Example: creating huge lists/arrays, infinite recursion

`OverflowError`

- Arithmetic operation too large
- Example: very large exponential operations

Resource and System Exceptions

`MemoryError`

- Out of memory
- Example: creating huge lists/arrays, infinite recursion

`OverflowError`

- Arithmetic operation too large
- Example: very large exponential operations

Resource and System Exceptions

`MemoryError`

- Out of memory
- Example: creating huge lists/arrays, infinite recursion

`OverflowError`

- Arithmetic operation too large
- Example: very large exponential operations

`ImportError`

Resource and System Exceptions

`MemoryError`

- Out of memory
- Example: creating huge lists/arrays, infinite recursion

`OverflowError`

- Arithmetic operation too large
- Example: very large exponential operations

`ImportError`

- Module import fails
- Example: importing non-installed package

Special Case Exceptions

`ZeroDivisionError`

Special Case Exceptions

`ZeroDivisionError`

- Division or modulo by zero
- Example: `x / 0` or `y % 0`

Special Case Exceptions

`ZeroDivisionError`

- Division or modulo by zero
- Example: `x / 0` or `y % 0`

Special Case Exceptions

`ZeroDivisionError`

- Division or modulo by zero
- Example: `x / 0` or `y % 0`

`AssertionError`

Special Case Exceptions

`ZeroDivisionError`

- Division or modulo by zero
- Example: `x / 0` or `y % 0`

`AssertionError`

- Assert statement fails

Special Case Exceptions

`ZeroDivisionError`

- Division or modulo by zero
- Example: `x / 0` or `y % 0`

`AssertionError`

- Assert statement fails

`RuntimeError`

Special Case Exceptions

`ZeroDivisionError`

- Division or modulo by zero
- Example: `x / 0` or `y % 0`

`AssertionError`

- Assert statement fails

`RuntimeError`

- Base class for many custom exceptions
- When no other error type will do

Handling Exceptions: Try-Except Blocks

Basic Syntax

Handling Exceptions: Try-Except Blocks

Basic Syntax

```
try:
    x = 1 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

Handling Exceptions: Try-Except Blocks

Basic Syntax

```
try:
    x = 1 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

Why is this important?

- Prevents crashes and allows graceful failure handling

Handling Exceptions: Try-Except Blocks

Basic Syntax

```
try:
    x = 1 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

Why is this important?

- Prevents crashes and allows graceful failure handling
- Allows debugging by catching and logging errors

Handling Exceptions: Try-Except Blocks

Basic Syntax

```
try:
    x = 1 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

Why is this important?

- Prevents crashes and allows graceful failure handling
- Allows debugging by catching and logging errors

Handling Exceptions: Try-Except Blocks

Basic Syntax

```
try:
    x = 1 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

Why is this important?

- Prevents crashes and allows graceful failure handling
- Allows debugging by catching and logging errors

Do not use try-except to continue execution unless you know exactly what is going on!

Else and Finally Blocks

Using Else and Finally

- `else`: Runs only if no exception occurs

Else and Finally Blocks

Using Else and Finally

- `else`: Runs only if no exception occurs
- `finally`: Runs regardless of whether an exception occurred or not

Else and Finally Blocks

Using Else and Finally

- `else`: Runs only if no exception occurs
- `finally`: Runs regardless of whether an exception occurred or not

Else and Finally Blocks

Using Else and Finally

- `else`: Runs only if no exception occurs
- `finally`: Runs regardless of whether an exception occurred or not

Example

Else and Finally Blocks

Using Else and Finally

- `else`: Runs only if no exception occurs
- `finally`: Runs regardless of whether an exception occurred or not

Example

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

Else and Finally Blocks

Using Else and Finally

- `else`: Runs only if no exception occurs
- `finally`: Runs regardless of whether an exception occurred or not

Example

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

(What else can age not be?)

Avoid defensive programming!

- A practice you should avoid is to use `else:` to continue execution notwithstanding the fact that an exception occurred

Avoid defensive programming!

- A practice you should avoid is to use `else:` to continue execution notwithstanding the fact that an exception occurred
- If the user supplied invalid input, you should error and tell them what they did wrong

Avoid defensive programming!

- A practice you should avoid is to use `else:` to continue execution notwithstanding the fact that an exception occurred
- If the user supplied invalid input, you should error and tell them what they did wrong

Avoid defensive programming!

Catch multiple possible exceptions

```
try:
    num = int(input("Enter a number: "))
    if num < 0:
        raise ValueError("Number cannot be negative")
    if num > 100:
        raise ValueError("Number must be 100 or less")
except ValueError as e:
    print(f"Invalid number: {e}")
    raise
except TypeError as e:
    print(f"Wrong type of input: {e}")
    raise
except Exception as e:  # Catch any unexpected errors
    print(f"Unexpected error: {e}")
    raise

print("You entered:", num)
return num  # Actually use the validated input
```

Raising Exceptions

Why Raise Exceptions?

- To signal an error when a function receives invalid input.
- Makes debugging easier by identifying issues early.

Example

```
def check_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative!")
```

Using Appropriate Exceptions: Examples

Example: Input Validation

Input validation is convenient in separate methods.

Using Appropriate Exceptions: Examples

Example: Input Validation

Input validation is convenient in separate methods.

```
def process_age(age_str):
    try:
        age = int(age_str)  # ValueError if not a number
        if age < 0:
            raise ValueError("Age cannot be negative")
        if age > 150:
            raise ValueError("Age seems unrealistic")
        return age # Everything went well
    except ValueError as e:
        print(f"Invalid age: {e}")
        raise
    except Exception as e:  # Catch any unexpected errors
        print(f"Unexpected error processing age: {e}")
        raise
```

Using Appropriate Exceptions: More Examples

Example: File Operations

Using Appropriate Exceptions: More Examples

Example: File Operations

```
def read_config(filename):
    try:
        with open(filename) as f:  # FileNotFoundError
            data = json.loads(f.read())  # JSONDecodeError
            return data['settings']  # KeyError
    except FileNotFoundError:
        print("Config file missing")
        raise  # Re-raise the current exception
    except json.JSONDecodeError:
        print("Invalid JSON format")
        raise
    except KeyError:
        print("Missing 'settings' in config")
        raise
    except Exception as e:
        print(f"Unexpected error: {e}")
        raise
```

When to Use Custom Exceptions

Create Custom Exceptions When:

- Built-in exceptions don't clearly convey the error
- You need domain-specific error handling
- You want to group related errors
- You need to add custom error attributes

Examples of Good Custom Exceptions:

- `DatabaseConnectionError`
- `InvalidConfigurationError`
- `UserAuthenticationError`
- `APIRateLimitExceeded`

Creating and Using Custom Exceptions

Example: Custom Exception Class

Creating and Using Custom Exceptions

Example: Custom Exception Class

```
class ConfigError(Exception):
    def __init__(self, message, missing_keys=None):
        self.missing_keys = missing_keys or []
        super().__init__(message)

def validate_config(config):
    required = ['api_key', 'host', 'port']
    missing = [k for k in required if k not in config]
    if missing:
        raise ConfigError("Missing required keys", missing)
```

Logging Errors Instead of Printing

Why Use Logging?

- Avoids cluttering the console.

Logging Errors Instead of Printing

Why Use Logging?

- Avoids cluttering the console.
- Keeps a permanent record of errors.

Logging Errors Instead of Printing

Why Use Logging?

- Avoids cluttering the console.
- Keeps a permanent record of errors.
- Helps with debugging and monitoring in production.

Logging Errors Instead of Printing

Why Use Logging?

- Avoids cluttering the console.
- Keeps a permanent record of errors.
- Helps with debugging and monitoring in production.

Logging Errors Instead of Printing

Why Use Logging?

- Avoids cluttering the console.
- Keeps a permanent record of errors.
- Helps with debugging and monitoring in production.

Example

Logging Errors Instead of Printing

Why Use Logging?

- Avoids cluttering the console.
- Keeps a permanent record of errors.
- Helps with debugging and monitoring in production.

Example

```
import logging
logging.basicConfig(filename="errors.log", level=logging.ERROR)
try:
    1 / 0
except ZeroDivisionError as e:
    logging.error(f"Error: {e}")
```

Testing Exception Handling

Using unittest

Testing Exception Handling

Using unittest

```
import unittest

def divide(a, b):
    if b == 0:
        raise ZeroDivisionError("Cannot divide by zero!")
    return a / b

class TestDivide(unittest.TestCase):
    def test_zero_division(self):
        with self.assertRaises(ZeroDivisionError):
            divide(1, 0)
```

Best Practices for Error Handling

Dos and Don’ts

- **Catch specific exceptions** instead of using a broad `Exception`

Best Practices for Error Handling

Dos and Don’ts

- **Catch specific exceptions** instead of using a broad `Exception`
- **Use logging** instead of `print()` to track errors

Best Practices for Error Handling

Dos and Don’ts

- **Catch specific exceptions** instead of using a broad `Exception`
- **Use logging** instead of `print()` to track errors
- **Use** `finally` to clean up resources like file handles or database connections

Best Practices for Error Handling

Dos and Don’ts

- **Catch specific exceptions** instead of using a broad `Exception`
- **Use logging** instead of `print()` to track errors
- **Use** `finally` to clean up resources like file handles or database connections
- **Don’t suppress exceptions** without handling them properly

Best Practices for Error Handling

Dos and Don’ts

- **Catch specific exceptions** instead of using a broad `Exception`
- **Use logging** instead of `print()` to track errors
- **Use** `finally` to clean up resources like file handles or database connections
- **Don’t suppress exceptions** without handling them properly
- **Use custom exceptions** for better error categorization

Best Practices for Error Handling

Dos and Don’ts

- **Catch specific exceptions** instead of using a broad `Exception`
- **Use logging** instead of `print()` to track errors
- **Use** `finally` to clean up resources like file handles or database connections
- **Don’t suppress exceptions** without handling them properly
- **Use custom exceptions** for better error categorization
- **Write tests** to verify that exception handling works correctly

Error handling

Joachim Vandekerckhove

Winter 2025