# 7
# Exceptions

Errors are an inevitable part of programming. Whether caused by incorrect logic, invalid user input, or unexpected system behavior, errors can disrupt the execution of a program and lead to unpredictable results. Proper error handling is crucial in ensuring that a program behaves robustly, gracefully recovers from unexpected situations, and provides meaningful feedback to users or developers.

Python provides a structured way to handle errors through exceptions. An **exception** is an event that interrupts the normal flow of a program when something goes wrong. Instead of allowing the program to crash unexpectedly, Python raises an exception, which can be caught and handled appropriately. This mechanism allows programmers to anticipate possible failures and respond accordingly, improving the program's stability and user experience.

There are two main types of errors in Python: **syntax errors** and **runtime errors**. Syntax errors occur when the Python interpreter encounters incorrect code structure, such as missing colons or unmatched parentheses. These errors prevent the program from running at all. On the other hand, runtime errors, or exceptions, occur during program execution. Examples include dividing by zero, accessing a non-existent list index, or trying to open a missing file. Unlike syntax errors, runtime errors may not appear until a specific part of the program is executed.

Python offers a powerful exception handling mechanism that allows developers to handle runtime errors gracefully. Using try and except blocks, programmers can catch exceptions and execute alternative code instead of letting the program crash. Additional constructs, such as else, finally, and raise, provide further control over error handling, ensuring that resources are cleaned up properly and errors are handled in a structured manner.

This chapters covers the following topics:

- Understanding Exceptions
- Common Built-in Exceptions
- Interpreting Traceback Messages
- Using try and except Blocks
- Raising Exceptions with raise
- Defining Custom Exceptions
- Using assert for Debugging
- Exception Propagation and Bubbling

This chapter explores Python's exception handling system in detail, covering the different types of exceptions, how to catch and manage them, and best practices for writing reliable, error-resilient code.

# Understanding Exceptions

In Python, errors fall into two main categories: **syntax errors** and **exceptions**. Syntax errors occur when the Python interpreter detects incorrect code structure, such as missing colons, unmatched parentheses, or invalid keywords, preventing the program from running at all. These errors must be fixed before execution. In contrast, exceptions occur during program execution when an operation cannot be completed successfully, such as dividing by zero, accessing an invalid list index, or working with incompatible types. While syntax errors are caught at compile time, exceptions are raised at runtime and can be handled using Python's exception handling mechanisms to prevent crashes and allow graceful recovery.

An **exception** is an event that occurs during the execution of a program and disrupts the normal flow of instructions. Unlike syntax errors, which prevent the program from running at all, exceptions occur when the program encounters an operation that it cannot complete successfully. If an exception is not handled, Python will terminate the program and display an error message, known as a **traceback**, to indicate where and why the error occurred.

**How Python Raises and Handles Exceptions**

When an error occurs in Python, the interpreter automatically **raises** an exception. This means Python creates an exception object that contains information about the error, including its type and an optional error message. If the program does not explicitly handle the exception, Python will **propagate** the error, eventually leading to a program crash.

For example, consider the following code:

```
result = 10 / 0  # This line will cause an exception
```

This code results in a ZeroDivisionError, as dividing by zero is mathematically undefined. Python raises an exception to indicate the error, and if no exception-handling mechanism is in place, the program will terminate.

## Common Built-in Exceptions

Python provides a wide range of built-in exceptions that cover various types of runtime errors. Some of the most commonly encountered exceptions include:

- **ZeroDivisionError** – Raised when attempting to divide a number by zero.

```
print(5 / 0)  # Raises ZeroDivisionError
```

- **TypeError** – Raised when an operation is performed on incompatible data types.

```
print("Hello" + 5)  # Raises TypeError
```

- **IndexError** – Raised when accessing an invalid index in a list or other sequence.

```
my_list = [1, 2, 3]

print(my_list[5])  # Raises IndexError
```

- **KeyError** – Raised when trying to access a non-existent key in a dictionary.

```
my_dict = {"name": "Alice"}

print(my_dict["age"])  # Raises KeyError
```

- **ValueError** – Raised when a function receives an argument of the right type but an invalid value.

```
int("hello")  # Raises ValueError
```

- **FileNotFoundError** – Raised when trying to open a non-existent file.

```
open("non_existent_file.txt")  # Raises FileNotFoundError
```

Python provides many other built-in exceptions, each designed to handle specific types of errors. Understanding these exceptions helps developers anticipate and manage potential failures effectively.

## Interpreting Exception Traceback Messages

When an exception occurs, Python generates a **traceback** message, which provides details about the error. A traceback includes:

1. The sequence of function calls (stack trace) that led to the exception.
2. The specific line number where the error occurred.
3. The type of exception raised.
4. A brief error message describing the issue.

For example, running the following code:

```
numbers = [1, 2, 3]

print(numbers[5])  # Accessing an out-of-range index
```

Produces an error message like:

Traceback (most recent call last):

```
  File "example.py", line 2, in <module>

    print(numbers[5])

IndexError: list index out of range
```

Breaking down the traceback:

- File "example.py", line 2: Indicates the file and line number where the error occurred.
- print(numbers[5]): Shows the specific code that caused the error.
- IndexError: list index out of range: Identifies the exception type and provides an error message.

Reading tracebacks effectively is an essential skill for debugging Python programs. By understanding where and why an error occurred, developers can quickly diagnose and fix issues in their code.

In the next section, we will explore how to handle these exceptions using Python's built-in error-handling mechanisms, ensuring that programs can recover gracefully instead of crashing.

# Using try and except Blocks

While exceptions indicate errors that occur during program execution, they don't necessarily have to cause a program to crash. Python provides built-in mechanisms to handle exceptions, allowing developers to respond to errors gracefully and continue execution.

The primary way to handle exceptions in Python is through the try and except statements. Code that may raise an exception is placed inside a try block, while the except block defines how to respond if an exception occurs.

For example:

```
try:
    result = 10 / 0  # This will raise ZeroDivisionError
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

In this case, when the ZeroDivisionError is raised inside the try block, Python immediately jumps to the corresponding except block, preventing the program from crashing.

## Handling Multiple Exceptions

Sometimes, a block of code may raise different types of exceptions. You can handle multiple exceptions separately:

```
try:
    num = int(input("Enter a number: "))  # Might raise ValueError
    result = 10 / num  # Might raise ZeroDivisionError
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
except ValueError:
    print("Error: Invalid input! Please enter a number.")
```

This approach allows different exceptions to be handled in specific ways.

## Catching Multiple Exceptions in One Block

If multiple exceptions should be handled in the same way, they can be grouped using parentheses:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except (ZeroDivisionError, ValueError):
    print("Invalid input or division by zero.")
```

Here, both ZeroDivisionError and ValueError are handled by the same except block.

### Using a Generic Exception Handler

To catch any exception (regardless of type), use except Exception:. This is useful when you don't know in advance what kind of errors might occur.

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except Exception as e:  # Catches any exception
    print(f"An error occurred: {e}")
```

However, using a generic exception handler should be done carefully, as it may hide unexpected bugs.

Using else and finally Blocks

else Block: Runs only if no exception occurs.

finally Block: Runs regardless of whether an exception occurs, often used for cleanup operations.

Example:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Cannot divide by zero.")
else:
    print(f"Result: {result}")  # Only runs if no exception occurs
finally:
    print("Execution completed.")  # Always runs
```

The finally block is useful for tasks such as closing files or releasing resources.

Handling exceptions properly ensures that Python programs can respond to errors in a controlled manner instead of crashing unexpectedly. In the next section, we will explore how to create custom exceptions for handling specific error cases.

# Raising Exceptions with raise

While Python automatically raises exceptions when an error occurs, there are cases where a programmer may want to trigger an exception manually. This is useful for enforcing constraints, validating input, or stopping execution when an unexpected situation arises.

## Manually Triggering Exceptions

Python provides the raise keyword to explicitly generate exceptions. It allows developers to signal that an error has occurred and should be handled.

For example:

```
raise ValueError("This is an error message.")
```

When this line runs, Python stops execution and raises a ValueError with the provided message.

## Raising Built-in Exceptions

Any built-in exception can be raised explicitly using raise, just as Python would do automatically under error conditions.

Example:

```
def divide(a, b):
    if b == 0:
        raise ZeroDivisionError("Cannot divide by zero.")
    return a / b
print(divide(10, 0))  # This will raise ZeroDivisionError
```

This prevents execution from continuing with an invalid operation.

## Creating Custom Error Messages

The raise statement can include a custom error message to provide more context about what went wrong.

Example:

```
def set_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative.")
    print(f"Age set to {age}")


set_age(-5)  # Raises ValueError with a custom message
```

Here, raise ValueError("Age cannot be negative.") ensures that negative values are not allowed and provides a clear error message.

**Re-raising Exceptions**

Sometimes, an exception is caught but needs to be re-raised for higher-level handling. This can be done using raise without specifying an exception:

```
try:
```

```
    x = int("abc")  # Raises ValueError
except ValueError as e:
    print("Caught an exception, re-raising...")
    raise  # Re-throws the same exception
```

Re-raising is useful when you want to log an error or take partial action before letting the exception propagate.

Raising exceptions with raise is a powerful way to enforce rules and ensure proper program behavior. In the next section, we will explore how to define and use custom exceptions for handling application-specific errors.

# Defining Custom Exceptions

While Python provides many built-in exceptions, there are situations where defining custom exceptions is necessary. Custom exceptions help make error handling more meaningful by explicitly representing domain-specific issues. They allow developers to differentiate different types of errors and handle them in a structured manner.

## Why and When to Create Custom Exceptions

Custom exceptions are useful when:

- The built-in exceptions do not clearly describe the error in your specific context.
- You want to provide more meaningful error messages and structured handling.
- You need to encapsulate domain-specific logic, making your code easier to debug and maintain.

For example, if an application deals with user accounts, an exception like InvalidUsernameError is more informative than a generic ValueError.

## Defining a Custom Exception Class

To create a custom exception, define a new class that inherits from Python's built-in Exception class. By convention, exception class names should end with Error.

Example:

```
class InvalidUsernameError(Exception):
    """Raised when a username does not meet the required format."""
    pass


def set_username(username):
    if " " in username:
```

```
        raise InvalidUsernameError("Username cannot contain
spaces.")
    print(f"Username set to: {username}")


set_username("invalid username")  # Raises InvalidUsernameError
```

Here, InvalidUsernameError is a subclass of Exception, and it behaves like any other exception but is more specific to the application's needs.

## Adding Custom Behavior to Exceptions

Custom exceptions can override the __init__ method to accept additional arguments, store extra data, or customize error messages.

Example:

```
class WithdrawalLimitError(Exception):
    """Exception raised when withdrawal exceeds the allowed
limit."""


    def __init__(self, amount, limit):
        self.amount = amount
        self.limit = limit
        super().__init__(f"Withdrawal of {amount} exceeds limit of
{limit}")


def withdraw(amount, limit=500):
    if amount > limit:
        raise WithdrawalLimitError(amount, limit)
    print(f"Withdrawal of {amount} successful.")


try:
    withdraw(600)
except WithdrawalLimitError as e:
    print(e)
```

In this example, WithdrawalLimitError includes details about the withdrawal attempt and limit, improving debugging and logging.

Custom exceptions make error handling more precise and improve the readability and maintainability of the code. In the next section, we will explore how to catch and handle exceptions using try-except blocks.

# Using assert for Debugging

Assertions are a built-in debugging feature in Python that help catch programming errors early by verifying assumptions about code. The assert statement acts as a checkpoint, ensuring that a condition is met during execution. If the condition evaluates to False, Python raises an AssertionError and halts execution.

## Understanding the assert Statement

The assert statement follows this syntax:

assert condition, "Error message (optional)"

- If condition evaluates to True, execution continues as normal.
- If condition evaluates to False, an AssertionError is raised, and the optional error message is displayed.

Example:

```
x = 10
assert x > 0, "x must be positive"
print("Execution continues since assertion passed.")
```

If x were negative, Python would raise:

AssertionError: x must be positive

## How Assertions Work and When They Should Be Used

Assertions are primarily used to detect bugs early by validating assumptions in the code. They are useful in:

- Debugging: Checking invariants during development, such as verifying input constraints in a function.
- Testing: Ensuring expected conditions hold before running further computations.
- Code documentation: Making implicit assumptions explicit to future developers.

Example:

def divide(a, b):

```
    assert b != 0, "Denominator must not be zero."
    return a / b


print(divide(10, 2))  # Works fine
print(divide(10, 0))  # Raises AssertionError
```

However, assertions should **not** be used for runtime error handling in production, as they can be disabled globally using Python's -O (optimize) flag. If optimizations are enabled, all assert statements are ignored, potentially leading to unexpected behavior.

**Differences Between assert and Explicit Exception Handling**

While assert statements can catch errors, they differ significantly from explicit exception handling using try-except:

- assert is best for catching *developer mistakes* and debugging, not user errors.
- Exceptions (raise) should be used for *recoverable runtime errors* that must be handled gracefully.
- Assertions are ignored in optimized mode (-O), whereas exception handling always executes.

Example of explicit exception handling:

```
def safe_divide(a, b):
    if b == 0:
        raise ValueError("Denominator must not be zero.")
    return a / b


print(safe_divide(10, 0))  # Raises ValueError
```

Unlike assert, explicit exception handling remains active even in optimized execution.

In summary, assert is a powerful tool for debugging and enforcing assumptions but should not replace proper exception handling. In the next section, we will explore how to use try-except blocks to handle exceptions gracefully during runtime.

# Bubbling Exceptions

Exception handling is not just about catching errors but also about catching them at the right level in the call stack. When an exception is raised in a deeply nested function, it propagates—or "bubbles up"—through the call stack until it is caught and handled. Understanding how to let exceptions bubble up and where to handle them ensures that errors are managed effectively while keeping code readable and maintainable.

## How Exception Bubbling Works

When an exception occurs inside a function, it immediately stops execution within that function. If the function does not handle the exception, it propagates (bubbles up) to the function that called it. This process continues up the call stack until the exception is either:

1. Caught and handled at an appropriate level, or
2. Left unhandled, causing the program to terminate with an error message.

**Example of Exception Bubbling**

Consider a scenario where multiple functions are involved, and an error occurs deep inside a nested function:

```
def inner_function():
    return 10 / 0  # ZeroDivisionError occurs here


def middle_function():
    return inner_function()  # Exception bubbles up


def outer_function():
    return middle_function()  # Exception continues to bubble up


try:
    outer_function()  # No handling in the functions, so the exception reaches here
except ZeroDivisionError:
    print("Error: Division by zero occurred!")
```

In this example:

- The error originates in inner_function().
- It is not handled there, so it bubbles up to middle_function(), then outer_function().
- Finally, the try-except block in the main program catches the exception.

This structure allows functions to focus on their intended purpose instead of handling errors they are not responsible for.

## Handling Exceptions at the Right Level

Catching exceptions too early (inside every function) can make debugging harder and introduce unnecessary complexity. Conversely, catching exceptions too late (or not at all) can cause program crashes. The key is to handle exceptions at a level where meaningful recovery or corrective action can be taken.

**Bad Practice: Handling Exceptions Too Early**

```
def inner_function():
    try:
        return 10 / 0  # Handling too early
    except ZeroDivisionError:
        print("Handled error, returning default value.")
        return 1
```

This approach prevents the calling function from knowing that something went wrong. Instead, it might be better to let the exception bubble up to a higher-level function that can decide how to handle it.

**Better Practice: Handling Exceptions Where Recovery Makes Sense**

```
def inner_function():
    return 10 / 0  # No handling here; let it bubble up


def process_data():
    try:
        result = inner_function()
        print(f"Processed result: {result}")
    except ZeroDivisionError:
        print("Error in processing: Division by zero. Please check
the input.")
```

In this improved version:

- inner_function() focuses on computation and does not suppress errors.
- process_data() handles the exception where it can take appropriate corrective action (such as notifying the user).

## Using raise to Explicitly Propagate Exceptions

If you catch an exception but still want it to bubble up, you can use raise to rethrow it:

```
def validate_input(value):
    if value < 0:
        raise ValueError("Negative values are not allowed!")


def process_input(value):
    try:
        validate_input(value)
    except ValueError as e:
        print(f"Warning: {e}")
        raise  # Re-raises the exception to bubble up further
```

This ensures that the exception is logged but still reaches higher levels where more significant decisions (such as terminating the program or requesting user input) can be made.

By understanding exception bubbling and handling errors at the appropriate level, you can write cleaner, more maintainable code. In the next section, we will explore the finally block, which ensures

that essential cleanup operations, like closing files or releasing resources, always occur, even when exceptions are raised.

# Best Practices for Exception Handling

Exception handling is a critical part of writing robust and maintainable Python programs. However, improper use of exception handling can introduce more problems than it solves. Following best practices ensures that exceptions are handled efficiently while maintaining code clarity and performance.

## Avoiding Bare except Clauses

A common mistake in exception handling is using a bare except: clause, which catches all exceptions, including system-exiting exceptions like KeyboardInterrupt and SystemExit. This can make debugging difficult by suppressing useful error messages.

**Bad Practice:**

```
try:
    result = 10 / 0
except:
    print("An error occurred.")  # Hides the actual exception
```

This catches all exceptions but doesn't provide any details about what went wrong.

**Better Approach:** Instead, always catch specific exceptions relevant to your code:

```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"Error: {e}")  # Prints: Error: division by zero
```

Catching specific exceptions makes it clear which errors are expected and allows for better debugging.

## Logging Exceptions for Debugging

Instead of merely printing error messages, logging exceptions ensures that errors are recorded for future analysis. The built-in logging module provides a better way to track exceptions.

**Using Logging:**

```
import logging

logging.basicConfig(level=logging.ERROR, filename="errors.log")

try:
```

```
    value = int("abc")  # Causes ValueError
except ValueError as e:
    logging.error("An error occurred: %s", e)
```

This logs the error in a file, making it easier to debug issues without exposing them to users.

## Using Exception Handling Only When Necessary

Exception handling should be used for handling unexpected errors, not for controlling normal program flow. Overusing try-except can lead to inefficiencies and obscure logic.

**Bad Practice:**

```
try:
    my_list = [1, 2, 3]
    item = my_list[3]  # IndexError
except IndexError:
    item = None  # Default value
```

Here, an exception is used where a simple conditional check would be more efficient.

**Better Approach:**

```
my_list = [1, 2, 3]
item = my_list[3] if len(my_list) > 3 else None  # Avoids
unnecessary exception
```

Using defensive programming (checking conditions before performing operations) can prevent unnecessary exception handling.

**Structuring Exception Handling for Maintainability**

Organizing exception handling improves readability and maintainability. Follow these principles:

1. **Use multiple specific except clauses** for different error types.
2. **Minimize the code inside try blocks** to reduce the scope of potential errors.
3. **Use finally for cleanup actions**, such as closing files or releasing resources.

## Well-structured exception handling:

```
try:
    with open("file.txt", "r") as file:
        content = file.read()
except FileNotFoundError:
    print("File not found. Please check the filename.")
except PermissionError:
```

```
    print("Permission denied. Cannot open the file.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
finally:
    print("Execution completed.")
```

This ensures:

- Errors are categorized and handled appropriately.
- Unexpected exceptions are still caught and reported.
- Cleanup tasks (like resource management) are performed in finally.

By following these best practices, exception handling becomes an asset rather than a liability. Thoughtfully handling errors improves code reliability, debugging, and performance. In the next section, we will explore the use of the finally block to ensure essential cleanup operations always occur.

# Summary

Exception handling is a crucial aspect of writing reliable and maintainable Python programs. Errors can arise from various sources, including invalid input, incorrect logic, or system-related issues. Without proper handling, these errors can cause programs to crash unexpectedly, leading to poor user experience and difficulty in debugging. Python provides a structured way to manage errors through exceptions, allowing developers to anticipate failures and ensure smooth program execution.

An exception is an event that disrupts normal execution when an operation cannot be completed successfully. Unlike syntax errors, which prevent a program from running, exceptions occur during execution and must be handled to prevent unexpected termination. Python automatically raises exceptions when it encounters an error, providing a traceback message that helps identify the source of the problem. Built-in exceptions such as ZeroDivisionError, TypeError, IndexError, and KeyError cover common failure scenarios, making it easier to detect and diagnose issues.

Handling exceptions in Python involves the use of try and except blocks, which catch and process errors without abruptly stopping execution. Multiple exceptions can be handled separately or together, ensuring that different error types receive appropriate responses. Additional constructs such as the else block allow code to execute only when no exceptions occur, while the finally block guarantees cleanup operations like closing files or releasing resources, regardless of whether an error was encountered. Raising exceptions manually with the raise statement provides a way to enforce rules and validate input, while defining custom exceptions allows for more specific error handling tailored to an application's needs.

Proper structuring of exception handling ensures that errors are caught at the appropriate level in the call stack. Allowing exceptions to bubble up to a meaningful location avoids unnecessary suppression of important issues while maintaining readability and clarity. Assertions provide a lightweight debugging tool for verifying assumptions during development but should not replace

robust error handling in production code. Logging exceptions instead of suppressing them aids in diagnosing issues and tracking unexpected behavior over time.

By following best practices such as avoiding overly broad except clauses, handling only expected exceptions, and ensuring minimal code inside try blocks, developers can prevent inefficient error management. Thoughtful exception handling contributes to more robust applications by preventing crashes, improving debugging, and ensuring that programs respond gracefully to unexpected situations.

# Exercises

1. **Handling a ZeroDivisionError**
   Write a program that prompts the user to enter two numbers and divides the first number by the second. Use a try-except block to catch a ZeroDivisionError and display a friendly error message if the user tries to divide by zero.

2. **Catching Multiple Exceptions**
   Create a program that asks the user for a number and attempts to convert it to an integer. Use a try-except block to catch both ValueError (if the input is not a number) and KeyboardInterrupt (if the user tries to interrupt the program with Ctrl+C). Display appropriate messages for each exception.

3. **Interpreting a Traceback Message**
   Run the following code and analyze the traceback message. Explain in comments what went wrong and which line caused the error.

```
my_list = [1, 2, 3]

print(my_list[5])  # This line will cause an error
```

4. **Using else and finally Blocks**
   Write a program that asks the user to input a filename. Try to open and read the file inside a try block. If the file does not exist, catch the FileNotFoundError and display an appropriate message. Use an else block to print the file's contents if no exception occurs. Ensure a finally block runs, printing "File operation complete." regardless of whether an exception occurred.

5. **Raising a Custom Exception**
   Define a function validate_age(age) that raises a ValueError with the message "Age cannot be negative!" if the provided age is negative. Call the function with a negative number and handle the exception using try-except.

6. **Creating a Custom Exception Class**
   Define a custom exception class InvalidPasswordError that inherits from Exception. Write a function check_password(password) that raises InvalidPasswordError if the password is less than 8 characters. Test the function and catch the exception.

7. **Using assert for Debugging**
   Write a function calculate_average(numbers) that takes a list of numbers and returns their average. Use an assert statement to ensure that the list is not empty before computing the average. Call the function with an empty list and observe the AssertionError.

8. **Understanding Exception Bubbling**
   Consider the following code:

```
def inner():
    return 5 / 0  # Error occurs here


def middle():
    return inner()


def outer():
    return middle()


try:
    outer()
except ZeroDivisionError:
    print("Caught a ZeroDivisionError!")
```

Explain in comments how the exception propagates from inner() to outer() and why the except block is able to catch it.

9. **Logging an Exception**
   Modify the program from Exercise 1 (division by zero) to log the exception using Python's logging module instead of just printing an error message. Ensure that errors are logged to a file named errors.log.

10. **Best Practices in Exception Handling**
    Rewrite the following poorly structured exception-handling code to follow best practices.

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except:
    print("Something went wrong!")  # Bad practice: Catches all exceptions blindly
```

- Ensure it catches specific exceptions (ValueError, ZeroDivisionError).
- Use an else block for successful execution.
- Use a finally block to print "Operation complete." regardless of success or failure.