Chapter 8 - Exceptions

1. Introduction to Exceptions

An **exception** is an event that disrupts the normal flow of a program's execution. When a function detects a problem, it raises an exception, which can be caught and handled to prevent the program from crashing.

1.1 What is an Exception?

An exception occurs when the program encounters an unexpected situation, like a division by zero, file not found, or an invalid operation. Python provides a robust way to handle such issues gracefully using `try`, `except`, `else`, and `finally` blocks.

---

2. Raising and Catching Exceptions

2.1 Raising Exceptions

In Python, you can raise exceptions using the `raise` keyword. This is often done when a certain condition occurs that makes it impossible to continue the program's execution.

Syntax:

```
raise Exception("This is a custom error message")
```

2.2 Catching Exceptions

The `try` block allows you to test a block of code for errors. If an error occurs, Python will move to the `except` block to handle it.

Syntax:

```
try:
    # Code that may raise an exception
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"Error occurred: {e}")
```

In the above example, the `ZeroDivisionError` exception is caught and handled, preventing the program from crashing.

---

3. Types of Exceptions

Python has several built-in exceptions, and you can define custom exceptions as well. Some common built-in exceptions are:

- `ZeroDivisionError`: Raised when dividing by zero.
- `FileNotFoundError`: Raised when trying to open a file that does not exist.
- `ValueError`: Raised when a function receives an argument of the correct type but an inappropriate value.
- `IndexError`: Raised when trying to access an element of a list using an invalid index.
- `KeyError`: Raised when trying to access a dictionary with a key that does not exist.
- `TypeError`: Raised when an operation or function is applied to an object of inappropriate type.

---

4. The `try`, `except`, `else`, and `finally` Blocks

4.1 `try` Block

The `try` block is where you write the code that might raise an exception.

4.2 `except` Block

The `except` block is used to catch and handle exceptions that are raised in the `try` block.

4.3 `else` Block

The `else` block executes if no exception is raised in the `try` block.

4.4 `finally` Block

The `finally` block is always executed, no matter what, even if an exception is raised or not. It's used for cleanup activities, such as closing files or releasing resources.

Syntax:

```
try:
    # Code that may raise an exception
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
except ValueError:
    print("Error: Invalid input. Please enter a valid number.")
else:
    print(f"Result is: {result}")
finally:
    print("Execution completed.")
```

In this example:

- The `try` block attempts to get a number from the user and perform a division.
- If the user inputs an invalid number or tries to divide by zero, the `except` block handles the exception.
- The `else` block executes only if no exception occurred.
- The `finally` block always executes, ensuring that any final actions, like cleanup, are performed.

---

5. Custom Exceptions

Python allows you to create your own custom exceptions. This can be useful when you want to raise an exception specific to your application.

5.1 Defining Custom Exceptions

To create a custom exception, you define a new class that inherits from the `Exception` class.

Example:

```
class InvalidAgeError(Exception):
    def __init__(self, message="Age must be a positive number"):
        self.message = message
        super().__init__(self.message)

# Raising a custom exception
def check_age(age):
    if age < 0:
```

```
        raise InvalidAgeError("Age cannot be negative!")
    return f"Your age is {age}"

try:
    print(check_age(-5))
except InvalidAgeError as e:
    print(f"Error: {e}")
```

In this example:

- `InvalidAgeError` is a custom exception that inherits from Python's built-in `Exception` class.
- The `check_age` function raises this custom exception if the age is negative.

---

6. Using Exceptions in Django

In Django, exceptions are used to handle errors gracefully, such as missing data, incorrect user input, or database connection issues. Django provides built-in exceptions to handle common scenarios.

6.1 Django's Built-In Exceptions

Some common exceptions in Django include:

- `Http404`: Raised when a requested page is not found.
- `ValidationError`: Raised when form validation fails.
- `PermissionDenied`: Raised when the user does not have the required permissions.

Example: Handling Http404 in Views

```
from django.http import Http404

def get_article(request, article_id):
    try:
        article = Article.objects.get(id=article_id)
    except Article.DoesNotExist:
        raise Http404("Article not found")
    return render(request, "article_detail.html", {"article": article})
```

In this example, the `Http404` exception is raised if the requested article does not exist in the database.

6.2 Custom Django Exceptions

You can define custom exceptions in Django, just like in regular Python. Custom exceptions can be used to handle application-specific errors.

Example: Custom Exception in Django

```
class CustomValidationError(Exception):
    pass

def validate_data(data):
    if not data:
        raise CustomValidationError("Data cannot be empty")

# Using the custom exception
try:
    validate_data("")
except CustomValidationError as e:
    print(f"Validation failed: {e}")
```

---

7. Best Practices for Exception Handling

7.1 Avoiding Empty `except` Blocks

Catching all exceptions with a general `except` block can hide bugs. Always catch specific exceptions whenever possible.

7.2 Using Exceptions for Control Flow

While exceptions should be used for handling errors, avoid using them for regular control flow, as this can make your code harder to read and maintain.

7.3 Logging Exceptions

It's important to log exceptions for debugging and troubleshooting. In Python, the `logging` module can be used to log errors.

Example: Logging Exceptions

```
import logging
```

```python
# Setting up logging
logging.basicConfig(filename="app.log", level=logging.ERROR)

try:
    1 / 0
except ZeroDivisionError as e:
    logging.error(f"Error occurred: {e}")
```

---

8. Exercise

Exercise 1: Handling Multiple Exceptions

Write a Python function `divide_numbers(a, b)` that divides two numbers. The function should:

- Raise a `ZeroDivisionError` if `b` is 0.
- Raise a `ValueError` if either `a` or `b` is not a number. Handle the exceptions in the `try` and `except` blocks and print appropriate error messages.

```python
def divide_numbers(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        print("Error: Cannot divide by zero")
    except ValueError:
        print("Error: Both inputs must be numbers")
    else:
        print(f"Result: {result}")
```

Exercise 2: Custom Exception for Invalid Input

Create a custom exception called `InvalidInputError`. Write a function `get_age()` that prompts the user for an age. If the user enters a negative age, raise an `InvalidInputError` with the message "Age cannot be negative".

```python
class InvalidInputError(Exception):
    pass

def get_age():
    try:
        age = int(input("Enter your age: "))
        if age < 0:
```

```
            raise InvalidInputError("Age cannot be negative")
        print(f"Your age is {age}")
    except InvalidInputError as e:
        print(f"Error: {e}")
```

Exercise 3: Django Exception Handling

Create a Django view that retrieves a `Product` by its ID. If the product is not found, raise a `Http404` exception with a custom message "Product not found".

```
from django.http import Http404
from .models import Product

def get_product(request, product_id):
    try:
        product = Product.objects.get(id=product_id)
    except Product.DoesNotExist:
        raise Http404("Product not found")
    return render(request, "product_detail.html", {"product": product})
```