Errors and Exception Handling in Python

This page equips you with the knowledge to identify, handle, and prevent errors in your Python programs. You’ll learn about common errors, exception handling mechanisms, and how to write robust code.

Concept Overview

Topics:

* Understanding Python Errors
* Mastering Exception Handling

Learning Objectives

* Differentiate between syntax errors and exceptions.
* Identify common Python exceptions and their causes.
* Utilize try, except,else, and finally blocks for exception handling.
* Raise exceptions with raise and create custom exceptions.
* Write code that anticipates and gracefully handles potential errors.

Introduction

Errors are inevitable in programming. Python errors can be broadly categorized into two main types:

1. **Syntax Errors**: These errors occur when the code violates Python’s grammar rules. They are typically detected during the code compilation phase and prevent the program from running at all.
2. **Exceptions**: Exceptions are runtime errors that arise during program execution. They indicate unexpected situations that the program encounters. Exceptions allow you to write code that can gracefully handle these situations and prevent program crashes.

Detailed Explanation

Understanding Python Errors

**Common Syntax Errors**

* Missing colons (:) after statements
* Incorrect indentation (Python relies on indentation for code blocks)
* Unmatched parentheses or brackets
* Typos in variable or function names

**Common Exceptions**

1. NameError: Occurs when a variable or function is used before it’s defined.
2. TypeError: Raised when an operation is attempted on incompatible data types.
3. IndexError: Thrown when trying to access an element outside the list or sequence’s index range.
4. ZeroDivisionError: Occurs when attempting to divide by zero.
5. ValueError: Indicates an inappropriate value passed to a function or operation.

Mastering Exception Handling

Python provides a powerful mechanism called exception handling to manage errors. The fundamental structure involves try, except, else, and finally blocks:

try:

# Code that might raise an exception

except ExceptionType:

# Code to handle the exception

else:

# Code that executes if no exception occurs

finally:

# Code that always executes, regardless of exceptions

* try **block**: Contains the code that might potentially raise an exception.
* except **block**: Catches specific exceptions based on the ExceptionType. You can have multiple except blocks to handle different exception types.
* else **block**: Executes code only if no exceptions occur within the try block.
* finally **block**: Executes code regardless of whether an exception occurs or not. It’s commonly used for cleaning up resources like closing files.

You can also use the raise statement to explicitly raise an exception when encountering an error condition within your code:

def divide(x, y):

if y == 0:

raise ZeroDivisionError("Division by zero is not allowed")

return x / y

Custom Exceptions

What are Custom Exceptions?

Custom exceptions are user-defined exception classes that you create to handle specific errors or exceptional situations in your code. By deriving custom exceptions from the base Exception class in Python, you can create more meaningful and specific error messages for your applications.

Why Use Custom Exceptions?

* Specific Error Handling: Custom exceptions help in identifying and handling specific errors or situations unique to your application.
* Clarity and Readability: They provide clear and meaningful error messages, making your code more understandable and maintainable.
* Modularity: By encapsulating related error handling logic within custom exceptions, you can improve code modularity and organization.

Creating Custom Exceptions

To create a custom exception:

* Define a new class that inherits from the base Exception class or its subclasses (e.g., ValueError, TypeError, etc.).
* Optionally, you can add custom attributes or methods to your exception class based on your requirements.

class OutOfStockError(Exception):

"""Custom exception for handling out-of-stock items."""

def \_\_init\_\_(self, item\_name):

self.item\_name = item\_name

def \_\_str\_\_(self):

return f"Sorry, the item '{self.item\_name}' is out of stock."

# Sample Product Inventory

product\_inventory = {

"apple": 10,

"banana": 5,

"orange": 0, # Out of stock

"grapes": 3

}

Let’s break down the provided code:

**Custom Exception OutOfStockError**:

* We define a custom exception class OutOfStockError that inherits from the base Exception class. This custom exception is designed to handle out-of-stock items.
* The \_\_init\_\_ method initializes the exception with the name of the out-of-stock item.
* The \_\_str\_\_ method specifies the error message to be displayed when the exception is raised.

**Sample Product Inventory**:

* We have a dictionary product\_inventory that represents the available quantity of various items in our inventory.

**Function purchase\_item(item, quantity)**:

* This function simulates purchasing items from the inventory.
* It checks if the requested item is available in the inventory and if it’s in stock. If the item is in stock, it reduces the quantity accordingly.
* If the item is out of stock, it raises the OutOfStockError custom exception.

Testing the Custom Exception

def purchase\_item(item, quantity):

try:

if product\_inventory[item] == 0:

raise OutOfStockError(item)

else:

print(f"Purchase successful: {quantity} {item}(s)")

product\_inventory[item] -= quantity

except KeyError:

print(f"Sorry, '{item}' is not available in our inventory.")

# Testing the Custom Exception

try:

purchase\_item("apple", 3) # Purchase successful

purchase\_item("orange", 1) # Raises OutOfStockError

purchase\_item("watermelon", 1) # Item not available

except OutOfStockError as e:

print(e) # Output:

* We use a try-except block to handle exceptions that may occur during the purchase process.
* We attempt to purchase items like “apple”, “orange”, and “watermelon” with different quantities.
* If an OutOfStockError is raised (due to an out-of-stock item), we catch it and print the custom error message using print(e).

Practice Exercises

**Exercise 1**: Handling ZeroDivisionError

Instructions:

* Write a program that takes two numbers as input from the user and divides the first number by the second number.
* Handle the ZeroDivisionError exception to inform the user if they attempt to divide by zero.

**Exercise 2**: File Handling with FileNotFoundError

Instructions:

* Write a program that attempts to open and read data from a file specified by the user.
* Handle the FileNotFoundError exception to provide a meaningful message if the file does not exist.

**Exercise 3**: Custom Exception

Instructions:

* Create a custom exception class called ValueTooHighError that inherits from the Exception class.
* Write a program that takes a number as input and raises a ValueTooHighError exception if the number is greater than 100.

Additional Resources

* [Python Exception: An Introduction](https://savanna.alxafrica.com/rltoken/91zfUFxLB1rIJFczLBEnxw)
* [Errors and Exceptions](https://savanna.alxafrica.com/rltoken/2-kCEKmD5sKni5LwfWT87w)
* [Python Tutorial for Beginners | Exception Handling - Video](https://savanna.alxafrica.com/rltoken/JwV9L_uASKJYrbofPc7wqw)
* [A brief guide to python exception](https://savanna.alxafrica.com/rltoken/sl6prMYJjedC-47ITqI4Fg)