**Exception Handling in Python**

An **exception** is an **error** that occurs during the execution of a program. When an exception is raised, it disrupts the normal flow of the program, and the program halts unless the exception is properly handled. Exceptions are used to signal that something unusual or unexpected has happened during the execution of the program (e.g., trying to divide by zero, accessing a file that doesn't exist).

In Python, exceptions are objects that represent specific types of errors. They provide detailed information about the problem that occurred, such as the error message, the type of error, and where in the code it occurred.

**Types of Exceptions in Python**

Python has a wide variety of built-in exceptions that represent different types of errors. Some common exceptions include:

* **ZeroDivisionError**: Raised when dividing by zero.
* **FileNotFoundError**: Raised when trying to access a file that does not exist.
* **ValueError**: Raised when a function receives an argument of the correct type but inappropriate value (e.g., passing a negative number to a function expecting only positive numbers).
* **IndexError**: Raised when trying to access an index that is out of range in a list or tuple.
* **TypeError**: Raised when an operation or function is applied to an object of inappropriate type (e.g., adding a string to an integer).
* **KeyError**: Raised when a dictionary key is not found.

**Exception Handling in Python**

Python provides a mechanism for handling exceptions using try, except, else, and finally blocks. This helps prevent the program from crashing when an error occurs and allows the programmer to deal with the error gracefully.

**Basic Syntax:**

try:

result = 10 / 0

except ZeroDivisionError as e:

print(f"Error: {e}")

**Explanation:**

1. **try block**: The code that might raise an exception is placed inside the try block.
2. **except block**: If an exception is raised in the try block, Python will look for an except block that matches the type of the exception. If a match is found, the corresponding code in the except block is executed.
3. **as e**: This syntax allows you to access the exception object itself and get additional information, such as the error message.

**Example of catching an exception:**

try:

num = int(input("Enter a number: "))

print(f"The square of the number is {num \*\* 2}")

except ValueError:

print("That's not a valid number!")

If the user enters something that is not a number (e.g., a letter), a ValueError will be raised, and the program will print "That's not a valid number!"

**Using Multiple Except Blocks**

You can handle different types of exceptions in different blocks:

try:

number = int(input("Enter a number: "))

result = 10 / number

except ZeroDivisionError:

print("Error: You can't divide by zero!")

except ValueError:

print("Error: Invalid input, please enter a valid number!")

Here, two exceptions are handled:

* If the user enters 0, a ZeroDivisionError is caught.
* If the user enters an invalid value (like a letter), a ValueError is caught.

**Else Block**

The else block is executed only if no exceptions were raised in the try block.

try:

num = int(input("Enter a number: "))

result = 10 / num

except ZeroDivisionError:

print("Error: Division by zero!")

except ValueError:

print("Error: Invalid input!")

else:

print(f"Result is: {result}")

If no exception occurs in the try block, the else block will execute, printing the result of the division.

**Finally Block**

The finally block is always executed, regardless of whether an exception occurs or not. It's typically used for cleaning up resources, such as closing files or releasing connections.

try:

file = open("file.txt", "r")

content = file.read()

except FileNotFoundError:

print("File not found!")

finally:

file.close() # Ensures that the file is closed even if an exception occurs

In this case, even if an error occurs while reading the file, the finally block ensures that the file is properly closed.

**Custom Exceptions**

You can also create your own exceptions by defining a new class that inherits from the built-in Exception class. This allows you to raise and handle your own specific errors.

class NegativeValueError(Exception):

pass

def check\_value(value):

if value < 0:

raise NegativeValueError("Value cannot be negative!")

try:

check\_value(-10)

except NegativeValueError as e:

print(e)

In this example, we define a custom exception called NegativeValueError and raise it when the value passed to check\_value is negative.

**Raising Exceptions**

You can raise exceptions manually using the raise keyword. This can be useful if you want to signal that something unexpected has happened in the program.

def check\_age(age):

if age < 0:

raise ValueError("Age cannot be negative.")

print(f"Your age is {age}")

try:

check\_age(-5)

except ValueError as e:

print(e)

In this example, the ValueError is raised manually when the age is negative.

**Summary**

* **Exceptions** are errors that disrupt the normal flow of a program.
* Python provides mechanisms to **handle exceptions** gracefully using try, except, else, and finally.
* You can define **custom exceptions** and raise exceptions using the raise keyword.
* Handling exceptions helps prevent programs from crashing and allows them to respond appropriately to error conditions.