**1. Try and Except Blocks**

In Python, exceptions are errors that occur during execution. To handle exceptions and prevent crashes, Python provides try and except blocks.

**Syntax:**

try:

# Code that may raise an exception

risky\_code()

except ExceptionType:

# Handle the exception

print("An error occurred!")

**Example:**

try:

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

result = 10 / num

except ZeroDivisionError:

print("You cannot divide by zero!")

except ValueError:

print("Invalid input! Please enter a number.")

* If the user enters 0, ZeroDivisionError is caught.
* If the user enters a non-numeric value, ValueError is caught.

**2. Defining Custom Exception Handlers**

Python allows you to define your own exception classes by inheriting from the built-in Exception class.

**Example:**

class NegativeNumberError(Exception):

"""Custom Exception for negative numbers."""

def \_\_init\_\_(self, value):

self.value = value

super().\_\_init\_\_(f"Negative numbers are not allowed: {value}")

def check\_positive(num):

if num < 0:

raise NegativeNumberError(num)

return num

try:

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

print(check\_positive(number))

except NegativeNumberError as e:

print(e)

* Here, we define NegativeNumberError that gets raised when a user enters a negative number.

**3. The Exception Hierarchy**

Python exceptions follow a hierarchy, where all exception classes inherit from the base class BaseException.

**Exception Hierarchy (Simplified)**

BaseException

├── SystemExit

├── KeyboardInterrupt

├── GeneratorExit

└── Exception

├── ArithmeticError

│ ├── ZeroDivisionError

│ ├── FloatingPointError

│ └── OverflowError

├── ValueError

├── IndexError

├── KeyError

├── AttributeError

├── TypeError

├── ImportError

├── IOError (OSError)

└── NameError

You can catch multiple exceptions using their place in the hierarchy.

**Example:**

try:

x = int("abc") # Raises ValueError

except ValueError as e:

print(f"Caught a ValueError: {e}")

except Exception as e:

print(f"Some other error occurred: {e}")

* If an exception is not caught at a lower level, it propagates up the hierarchy.

**4. Chaining Except Blocks**

When handling exceptions, you can have multiple except blocks to handle different types of errors.

**Example:**

try:

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

print(10 / num)

except ZeroDivisionError:

print("Cannot divide by zero.")

except ValueError:

print("Invalid input. Please enter a valid number.")

except Exception as e:

print(f"An unexpected error occurred: {e}")

* Python will check each except block in order and execute the first matching one.

**5. Using Python Shell**

The **Python Shell (REPL - Read-Eval-Print Loop)** allows you to execute Python commands interactively.

**How to open the Python shell:**

* Open the terminal and type:
* python
* In Windows, you can use py instead of python.

**Example Usage:**

>>> x = 5

>>> y = x + 10

>>> print(y)

15

The Python shell is useful for quick calculations, testing snippets, and debugging.

**6. Using Command-Line Arguments in a Script**

Python scripts can accept command-line arguments using sys.argv.

**Example:**

import sys

print("Script Name:", sys.argv[0])

if len(sys.argv) > 1:

print("Arguments:", sys.argv[1:])

else:

print("No arguments provided.")

**Run the script in the terminal:**

python script.py arg1 arg2 arg3

Output:

Script Name: script.py

Arguments: ['arg1', 'arg2', 'arg3']

* sys.argv[0] is the script name.
* sys.argv[1:] contains the passed arguments.

**7. Overview of argparse**

The argparse module provides a more powerful way to handle command-line arguments.

**Example using argparse:**

import argparse

parser = argparse.ArgumentParser(description="A simple calculator")

parser.add\_argument("num1", type=int, help="First number")

parser.add\_argument("num2", type=int, help="Second number")

parser.add\_argument("--operation", choices=["add", "sub", "mul", "div"], required=True, help="Operation to perform")

args = parser.parse\_args()

if args.operation == "add":

result = args.num1 + args.num2

elif args.operation == "sub":

result = args.num1 - args.num2

elif args.operation == "mul":

result = args.num1 \* args.num2

elif args.operation == "div":

result = args.num1 / args.num2

print("Result:", result)

**Run the script:**

python script.py 10 5 --operation add

Output:

Result: 15

**Why use argparse?**

* Provides argument validation.
* Automatically generates help (-h or --help).
* Handles optional and required arguments easily.

**Conclusion**

* try-except handles runtime errors.
* Custom exceptions improve error handling.
* Exception hierarchy determines how errors are caught.
* Multiple except blocks handle different errors.
* The Python shell is useful for testing.
* sys.argv allows passing arguments to scripts.
* argparse provides a structured way to handle command-line arguments.