PYTHON EXCEPTION HANDLING

Error handling is a crucial part of any programming language — and Python provides a very clean and readable way to handle errors using **exceptions**. When an error occurs, rather than letting the program crash, Python allows us to **catch** and **respond** to it using a structured flow called *exception handling*. This ensures your programs remain robust, user-friendly, and easier to debug.

What Are Exceptions?

An **exception** is an unwanted or unexpected event that occurs during the execution of a program, disrupting its normal flow. For instance, if you try to divide a number by zero or access a file that doesn't exist, Python will raise an exception.

Without exception handling, your program will crash when it encounters such errors. With it, you can gracefully deal with errors, show meaningful messages to the user, and continue running other parts of your program.

The Try-Except Block:

Python's core mechanism for handling exceptions is the try-except block. The idea is simple: you "try" to execute a block of code, and if an error occurs, Python will jump to the except block to handle the issue.

Syntax:

```
try:
   # Code that might raise an error
   risky_operation()
except SomeException:
   # Code that runs if the error occurs
   handle_it()
```

Example:

```
try:
   x = 10 / 0
except ZeroDivisionError:
   print("You can't divide by zero.")
```

Here, if we didn't catch ZeroDivisionError, Python would crash the program. Instead, we show a friendly message and recover from the situation.

The else and finally Clauses:

Python allows two additional optional blocks with try-except:

- **else**: Runs if the try block runs successfully (no exception occurs).
- **finally**: Runs **always**, no matter what, usually for cleanup activities.

Example:

```
try:
    number = int("42")
except ValueError:
    print("Invalid number format.")
else:
    print("Conversion successful.")
finally:
    print("This always executes.")
```

This structure is very helpful when you want to take specific actions when the code succeeds or fails, or always close files or network connections.

Catching Multiple Exceptions:

Sometimes, a block of code might raise more than one type of error. You can catch multiple exceptions either separately or together.

Example (Single except for multiple errors):

```
try:
    result = 10 + "10"
except (TypeError, ValueError) as e:
    print(f"An error occurred: {e}")
```

Example (Multiple except blocks):

```
try:
    value = int("ten")
except ValueError:
    print("ValueError occurred.")
except TypeError:
    print("TypeError occurred.")
```

This approach gives you more control and allows handling each error differently.

Raising Exceptions Manually:

There are cases where you may want to **raise** an exception intentionally to indicate something went wrong.

Example:

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

set_age(-5)  # Will raise a ValueError
```

You can raise built-in exceptions or define your own.

Creating Custom Exceptions:

When you're building a large application or framework, defining **custom exception classes** can make your error handling much cleaner and semantically clear.

Example:

```
class InvalidInputError(Exception):
    """Custom exception for invalid user input"""
    pass

def process_input(data):
    if not isinstance(data, int):
        raise InvalidInputError("Input must be an integer.")

try:
    process_input("abc")
except InvalidInputError as e:
    print(f"Caught custom error: {e}")
```

You can even add more methods or attributes to your custom exceptions if needed.

Nested Try Blocks:

Nested try blocks can be used when you need finer control over error handling in specific sub-parts of a larger process.

```
try:
    print("Outer try block")
    try:
        x = int("NaN")
    except ValueError:
        print("Inner try caught a ValueError")
except Exception:
    print("Outer try caught an exception")
```

Nested structures can help isolate problems more precisely in complex logic flows.

The finally Block and Resource Management:

The finally clause is especially useful when you need to perform **cleanup operations**, such as closing files or releasing database connections, whether an exception was raised or not.

Example:

```
try:
    file = open("data.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File does not exist.")
finally:
    print("Closing file.")
    try:
        file.close()
    except:
        pass  # If file was never opened
```

In real-world applications, especially in file or network operations, finally ensures that your program doesn't leave open resources or memory leaks.

Best Practices in Python Exception Handling:

1. **Catch specific exceptions**: Avoid using a bare except: unless absolutely necessary.
2. **Log errors**: Especially in production code, always log errors (using logging module).
3. **Avoid hiding bugs**: Don’t catch exceptions and just pass unless there's a good reason.
4. **Use exceptions to handle real errors**: Don't use them for normal flow control.
5. **Group related errors**: You can handle similar errors together, but not unrelated ones.

Summary:

Python’s exception handling system is flexible, clean, and highly readable. Whether you're writing a simple script or a complex backend service, knowing how to use try, except, else, and finally will help you write more stable and error-resistant code. As your project grows, you can start using custom exceptions to make your error messages more meaningful and easier to debug.

Presented By:

Narmeen Asghar