Error handling in Python is primarily done using try, except, else, and finally blocks. This mechanism allows you to handle runtime errors gracefully, ensuring that your program can recover or fail in a controlled manner.

Basic Structure

```
try:
    # Code that might raise an exception
except SomeException as e:
    # Code that runs if the exception occurs
else:
    # Code that runs if no exception occurs
finally:
    # Code that always runs, regardless of whether an exception occurred
```

Key Components:

1. **`try` block: You place code that might throw an exception inside this block.**
2. **`except` block: When an exception occurs, the code in the `except` block executes. You can specify a particular exception type or catch all exceptions.**
3. **`else` block (optional): This block runs if no exceptions were raised in the `try` block.**
4. **`finally` block (optional): Code in this block will always execute, whether an exception occurs or not, useful for cleanup operations like closing files or network connections.**

Example: Handling a Specific Exception

```
try:
    num = int(input("Enter a number: "))
    print(10 / num)
except ValueError:
    print("Invalid input! Please enter an integer.")
except ZeroDivisionError:
    print("Division by zero is not allowed.")
else:
    print("The operation was successful.")
finally:
    print("End of operation.")
```

In this example:

- If a ValueError (invalid input) or ZeroDivisionError occurs, a specific message is printed.
- The else block runs if no exceptions occur.
- The finally block runs regardless of whether an exception occurs, which is useful for cleanup.

Common Exceptions in Python:

- ValueError: Raised when a function gets an argument of the right type but inappropriate value (e.g., converting "abc" to an integer).
- ZeroDivisionError: Raised when trying to divide by zero.
- FileNotFoundError: Raised when trying to access a file that doesn't exist.
- TypeError: Raised when an operation or function is applied to an object of an inappropriate type.

Catching Multiple Exceptions

You can catch multiple exceptions using a tuple:

```
try:
    # Code that might raise either an IOError or ValueError
except (IOError, ValueError) as e:
    print(f"An error occurred: {e}")
```

Raising Exceptions

You can also raise your own exceptions using the `raise` keyword.

```
def divide(a, b):
    if b == 0:
        raise ZeroDivisionError("You can't divide by zero!")
    return a / b
try:
    result = divide(10, 0)
except ZeroDivisionError as e:
    print(e)
```

Custom Exceptions

You can create custom exceptions by subclassing the `Exception` class:

```
class CustomError(Exception):
    pass

try:
    raise CustomError("This is a custom error.")
except CustomError as e:
    print(e)
```