Errors and Exceptions

Errors and Their Types in Python

Errors in Python occur when the interpreter encounters something it cannot execute. Errors can be broadly classified into **syntax errors** and **exceptions**.

---

1. Syntax Errors (Compile-Time Errors)

A **SyntaxError** occurs when the Python interpreter finds an incorrect syntax (wrong grammar of Python).

Example:

```
if True
    print("Hello")
```

Error Output:

```
SyntaxError: expected ':'
```

✓ **Fix:** Add a colon : after if True.

---

2. Exceptions (Runtime Errors)

Exceptions occur **at runtime** when a valid syntax is executed but results in an error.

Types of Exceptions in Python

Here are some common types of exceptions:

(i) NameError

Occurs when trying to use a variable that is not defined.

```
print(x)  # x is not defined
```

Error Output:

```
NameError: name 'x' is not defined
```

✓ **Fix:** Define x before using it.

(ii) TypeError

Occurs when an operation is performed on an incompatible type.

```
print(5 + "hello")  # Integer + String
```

Error Output:

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

✓ **Fix:** Convert data types before operations:

```
print(str(5) + "hello")  # Output: 5hello
```

(iii) ValueError

Occurs when a function receives an argument of the correct type but an inappropriate value.

```
num = int("abc")  # Cannot convert "abc" to an integer
```

Error Output:

```
ValueError: invalid literal for int() with base 10: 'abc'
```

✓ **Fix:** Ensure valid input before conversion.

(iv) IndexError

Occurs when trying to access an index that is out of range.

```
lst = [1, 2, 3]
print(lst[5])  # Index out of
range
```

Error Output:

```
IndexError: list index out of range
```

✓ **Fix:** Check list length before accessing indexes.

(v) KeyError

Occurs when trying to access a dictionary key that does not exist.

```
my_dict = {"a": 1, "b": 2}
print(my_dict["c"])  # Key 'c' does not exist
```

Error Output:

```
KeyError: 'c'
```

✓ **Fix:** Use .get() to handle missing keys:

```
print(my_dict.get("c", "Key not found"))  # Output: Key not found
```

(vi) AttributeError

Occurs when trying to access an attribute that does not exist.

```
x = 10
x.append(5)  # Integers do not have an append() method
```

Error Output:

```
AttributeError: 'int' object has no attribute 'append'
```

✓ **Fix:** Use correct data types.

(vii) ZeroDivisionError

Occurs when trying to divide a number by zero.

```
print(10 / 0)
```

Error Output:

```
ZeroDivisionError: division by zero
```

✓ **Fix:** Ensure the denominator is not zero before dividing.

(viii) FileNotFoundError

Occurs when trying to open a file that does not exist.

```
with open("nonexistent_file.txt", "r") as f:
    content = f.read()
```

Error Output:

```
FileNotFoundError: [Errno 2] No such file or directory: 'nonexistent_file.txt'
```

✓ **Fix:** Check if the file exists before opening.

(ix) ImportError / ModuleNotFoundError

Occurs when trying to import a module that does not exist.

```
import non_existent_module
```

Error Output:

```
ModuleNotFoundError: No module named 'non_existent_module'
```

✓ **Fix:** Install or check the module name.

Exception Handling in Python

Exception handling in Python allows you to gracefully handle runtime errors, preventing program crashes. This is done using try, except, else, and finally blocks.

1. Basic Exception Handling using try-except

A try block is used to enclose code that may raise an exception. If an error occurs, the except block handles it.

Example: Handling Division by Zero

```
try:
   result = 10 / 0  # Raises ZeroDivisionError
except ZeroDivisionError:
   print("Error: Cannot divide by zero!")
```

✓ **Output:**

```
Error: Cannot divide by zero!
```

2. Handling Multiple Exceptions

You can handle different types of exceptions separately.

Example: Handling ZeroDivisionError and ValueError

```
try:
   num = int(input("Enter a number: "))
   result = 10 / num
except ZeroDivisionError:
   print("Error: Cannot divide by zero!")
except ValueError:
   print("Error: Invalid input! Please enter a number.")
```

✓ **Input:**

```
Enter a number: abc
```

✓ **Output:**

```
Error: Invalid input! Please enter a number.
```

3. Catching Multiple Exceptions in One except Block

Instead of writing multiple except blocks, you can use a tuple to catch multiple exceptions in a single block.

Example:

```
try:
   num = int(input("Enter a number: "))
   result = 10 / num
except (ZeroDivisionError, ValueError) as e:
   print(f"Error: {e}")
```

4. Using else with try-except

The else block runs **only if no exceptions occur**.

Example:

```
try:
   num = int(input("Enter a number: "))
   result = 10 / num
except ZeroDivisionError:
   print("Error: Cannot divide by zero!")
except ValueError:
   print("Error: Invalid input! Please enter a number.")
else:
   print("Success! The result is:", result)
```

✓ **Input:** 5

✓ **Output:**

```
Success! The result is: 2.0
```

5. Using finally Block

The finally block **always executes**, whether an exception occurs or not.

Example:

```
try:
    f = open("file.txt", "r")
    content = f.read()
except FileNotFoundError:
    print("Error: File not found!")
finally:
    print("Execution completed.")
```

✓ **Output:**

```
Error: File not found!
Execution completed.
```

6. Raising Custom Exceptions Using raise

You can manually raise exceptions using raise.

Example: Raising ValueError

```
age = int(input("Enter your age: "))
if age < 0:
    raise ValueError("Age cannot be negative!")
```

✓ **Input:** -5

✓ **Output:**

```
ValueError: Age cannot be negative!
```

7. Creating Custom Exceptions

You can define your own exception classes by inheriting from Exception.

Example: Custom Exception for Negative Numbers

```
class NegativeNumberError(Exception):
    pass

num = int(input("Enter a positive number: "))
if num < 0:
    raise NegativeNumberError("Negative numbers are not allowed!")
```

✓ **Input:** -3

✓ **Output:**

```
NegativeNumberError: Negative numbers are not allowed!
```

8. Handling All Exceptions (Exception)

Using Exception in except can catch **all** types of errors.

Example:

```
try:
    x = int(input("Enter a number: "))
    result = 10 / x
```

```
except Exception as e:
    print(f"An error occurred: {e}")
```

✓ **Input:** "abc"

✓ **Output:**

```
An error occurred: invalid literal for int() with base 10: 'abc'
```

Conclusion

- **try-except**: Handles exceptions.
- **else**: Runs when no exceptions occur.
- **finally**: Runs **always**, even if an exception occurs.
- **raise**: Manually raises an exception.
- **Custom Exceptions**: Create user-defined exception classes.

Using exception handling properly ensures your program is **robust, user-friendly, and error-free!**

finally Block in Python

The finally block is used in Python to execute code **regardless of whether an exception occurs or not**. It is commonly used for **cleanup operations** like closing files, releasing resources, or disconnecting from databases.

---

1. Syntax of finally Block

```
try:
    # Code that may raise an exception
except ExceptionType:
    # Handling exception
finally:
    # Code that always executes
```

---

2. Example: finally Executes Always

```
try:
    print("Try block executing...")
    result = 10 / 2  # No exception
except ZeroDivisionError:
    print("Cannot divide by zero!")
finally:
    print("Finally block always executes!")
```

✓ **Output:**

```
Try block executing...
Finally block always executes!
```

---

3. finally Block When an Exception Occurs

Even if an exception occurs, the finally block still executes.

```
try:
    print("Trying to divide by zero...")
    result = 10 / 0  # Causes ZeroDivisionError
except ZeroDivisionError:
    print("Caught ZeroDivisionError!")
```

```
finally:
    print("Finally block executed!")
```

✓ **Output:**

```
Trying to divide by zero...
Caught ZeroDivisionError!
Finally block executed!
```

---

4. Using finally for Resource Cleanup

A common use case of finally is ensuring that a file or database connection is properly closed.

Example: Closing a File

```
try:
    f = open("example.txt", "r")
    content = f.read()
except FileNotFoundError:
    print("File not found!")
finally:
    print("Closing the file...")
    f.close()  # This ensures the file is always closed
```

✓ **Output (if file is missing):**

```
File not found!
Closing the file...
```

Even though an exception occurs (FileNotFoundError), the finally block executes, ensuring the file is closed.

---

5. finally with return Statement

Even if a function has a return statement inside try or except, the finally block **still executes before returning**.

```
def test_finally():
    try:
        return "Try block executed"
    finally:
        print("Finally block executed!")

print(test_finally())
```

✓ **Output:**

```
Finally block executed!
Try block executed
```

Even though return is inside try, finally executes **before returning**.

---

6. finally with raise

If an exception is raised inside try and not caught in except, the finally block **still runs before the program crashes**.

```
try:
    print("Before exception")
    raise ValueError("Something went wrong!")  # Raising an exception
finally:
    print("Finally executed before crashing!")
```

✓ **Output:**

```
Before exception
Finally executed before crashing!
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
```

```
ValueError: Something went wrong!
```

The finally block **executes before the exception terminates the program**.

---

7. finally in Nested Try Blocks

A finally block inside a nested try-except also executes.

```
try:
    try:
        print("Inner try block")
        raise ZeroDivisionError
    finally:
        print("Inner finally block")
except ZeroDivisionError:
    print("Exception handled in outer block")
finally:
    print("Outer finally block")
```

✓ Output:

```
Inner try block
Inner finally block
Exception handled in outer block
Outer finally block
```

Both finally blocks **execute regardless of the exception**.

Conclusion

- finally **always executes**, even if an exception occurs.
- Used for **cleanup tasks** like closing files, releasing memory, or disconnecting databases.
- Executes **before returning** if return is present.
- **Executes even if an exception is raised** and not caught.

The finally block ensures **reliable cleanup** and helps prevent resource leaks in programs! 🚀