

## Modules, Error and Exceptions 2

### ✓ New section

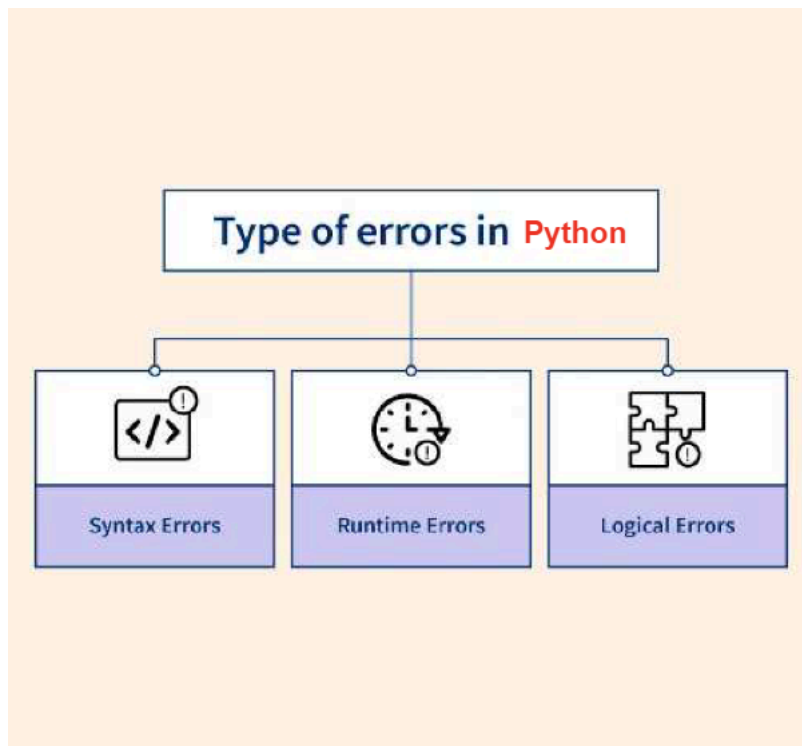
Start coding or [generate](#) with AI.

#### Agenda

1. Errors
  - Types of Error in Python
2. Exception Handling
  - try and except
  - Custom Exceptions
  - Handling Multiple Exceptions
  - else Clause
  - finally Clause
  - Raising Exceptions

### ✓ Errors

- Errors are problems in a program that causes the program to stop its execution.
- Errors in Python can be broadly categorized into syntax errors, runtime errors, and logical errors. Understanding these errors helps you debug your code more effectively. Here's a breakdown of different types of errors you might encounter in Python:
  1. **Syntax Errors:-** Syntax errors occur when the code you write does not follow Python's grammar rules. These errors are detected by the Python interpreter during the parsing stage before the code is executed.
  2. **Runtime Errors:-** Runtime errors occur during the execution of the program. These errors cause the program to stop abruptly and are usually due to issues like invalid operations or unexpected conditions.
  3. **Logical Errors:-** Logical errors are mistakes in the logic of your code that produce incorrect results but do not necessarily cause the program to crash. These errors are harder to detect because they do not raise explicit error messages.



```
def greet(name)
    print("Hello, " + name) # SyntaxError: invalid syntax
```

```
File "<ipython-input-27-6d6d7ffa43d8>", line 1
    def greet(name)
        ^
SyntaxError: expected ':'
```

```
if x = 10:
    print(x) # syntax Error
```

```
File "<ipython-input-29-428eea2c88b5>", line 1
    if x = 10:
        ^
SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of '='?
```

```
def func():
    print("Hello") # Indentation Error
```

```
File "<ipython-input-30-6d2a56eeb962>", line 2
    print("Hello") # Indentation Error
    ^
IndentationError: expected an indented block after function definition on line 1
```

```
numbers = [1, 2, 3]
print(numbers[5]) # IndexError: list index out of range
```

```
-----
IndexError                                Traceback (most recent call last)
<ipython-input-31-08d335cale95> in <cell line: 2>()
      1 numbers = [1, 2, 3]
----> 2 print(numbers[5]) # IndexError: list index out of range

IndexError: list index out of range
```

```
with open('non_existent_file.txt') as f:
    content = f.read() # FileNotFoundError: [Errno 2] No such file or directory
```

```
-----
FileNotFoundError                        Traceback (most recent call last)
<ipython-input-32-e948057df39e> in <cell line: 1>()
----> 1 with open('non_existent_file.txt') as f:
      2     content = f.read() # FileNotFoundError: [Errno 2] No such file or directory

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

```
numbers = [1, 2, 3, 4, 5]
for i in range(len(numbers)): # Should be range(len(numbers) - 1) if index is to be within bounds
    print(numbers[i + 1])
```

```
-----
IndexError                                Traceback (most recent call last)
<ipython-input-34-5ce67a8060d0> in <cell line: 2>()
      1 numbers = [1, 2, 3, 4, 5]
      2 for i in range(len(numbers)): # Should be range(len(numbers) - 1) if index is to be within bounds
----> 3     print(numbers[i + 1])

IndexError: list index out of range
```

## ✓ Exception Handling

### ✓ What is Exception?

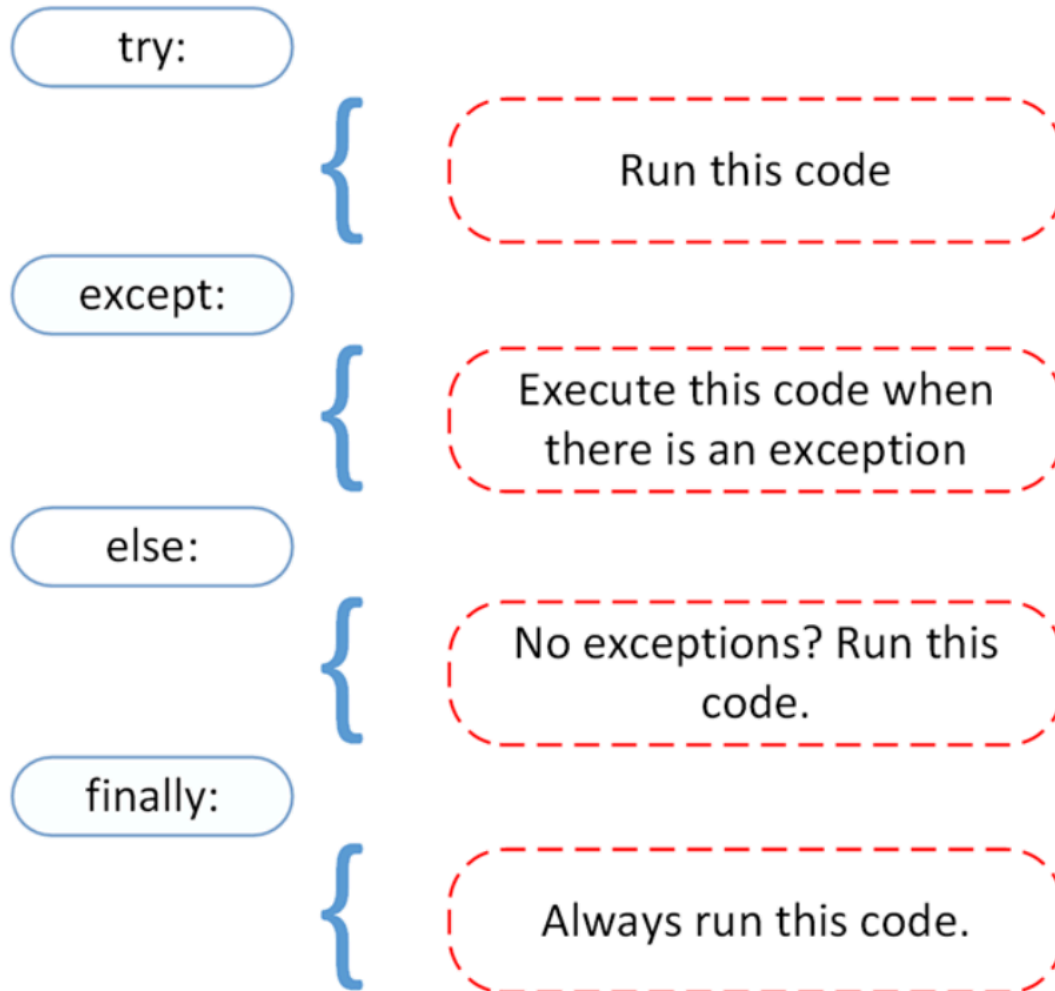
- An exception in Python is an incident that happens while executing a program that causes the regular course of the program's commands to be disrupted. When a Python code comes across a condition it can't handle, it raises an exception. An object in Python that describes an error is called an exception.

### What is Exception Handling?

- Exception handling in Python allows you to manage and respond to runtime errors gracefully. By catching and handling exceptions, you can prevent your program from crashing and provide meaningful error messages or recovery options.
- Exceptions are raised when some internal events occur which change the normal flow of the program.

### Why Exception handling is important?

- Exception Handling is essential for managing errors gracefully and preventing program crashes.
- It improves the robustness, readability, and maintainability of code by separating error-handling logic from regular code.
- It facilitates resource management, custom error responses, and controlled flow, leading to better user experiences and more reliable applications.




- The try block contains code that might raise an exception, and the except block contains code that runs if an exception occurs.

### ▼ try and except

```
try:
    # Code that may raise an exception
    result = 10 / 0
except ZeroDivisionError:
    # Code that runs if the exception occurs
    print("You can't divide by zero!")
```

🔄 You can't divide by zero!

```
try:
    result = 10 / 0
except:
    print("You can't divided by zero")
```

 You can't divided by zero

## ✓ Custom Exception

- In Python, you can create custom exceptions to handle specific situations in your code that aren't covered by the built-in exceptions. Custom exceptions are created by subclassing the built-in Exception class (or any other exception class), allowing you to define your own error messages and attributes.

```
class CustomError(Exception):
    pass

def do_something(value):
    if value > 100:
        raise CustomError("Value can't be greater than 100!")

try:
    do_something(80)
except CustomError as e:
    print(e)
```


Start coding or [generate](#) with AI.

## ✓ Handling Multiple Exceptions

- You can handle multiple exceptions by specifying different except blocks for each type of exception.

```
try:
    # Code that may raise an exception
    number = int(input("Enter a number: "))
    result = 10 / number
except ValueError:
    print("That's not a valid number!")


except ZeroDivisionError:
    print("You can't divide by zero!")
```

 Enter a number: uiuyi  
That's not a valid number!

## ✓ else clause

- The else clause in a try-except block in Python is used to define a block of code that should be executed only if no exceptions were raised in the try block. The else block is optional and comes after the try and except blocks. If an exception is raised and caught in the try block, the else block is skipped.


```
try:
    number = int(input("Enter a number: "))
    result = 10 / number
except ZeroDivisionError:
    print("You can't divide by zero!")
except ValueError:
    print("That's not a valid number!")
else:
    print(f"The result is {result}")
```

 Enter a number: 10  
The result is 1.0

## ✓ finally clause

- The finally clause in Python is used in conjunction with try and except blocks to define a block of code that will always execute, regardless of whether an exception was raised or not.
- The finally block is typically used for cleanup actions, such as closing files, releasing resources, or other tasks that should be performed no matter what.


```
try:
    file = open("example.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("The file was not found.")
finally:
    print("This will run no matter what.")
    if 'file' in locals():
        file.close()
```

 The file was not found.  
This will run no matter what.

## ▼ Base Exception

- BaseException is the root class for all built-in exceptions. It is the most fundamental base class from which all exception classes are derived, including user-defined exceptions. Understanding BaseException is crucial for exception handling and custom error creation.

```
try:
    # Code that may raise an exception
    raise ValueError("An example error")
    #number = int(input("Enter a number: "))
except BaseException as e:
    print(f"An exception occurred: {e}")
```

 An exception occurred: An example error

```
BaseException
├── SystemExit (raised when `sys.exit()` is called)
├── KeyboardInterrupt (raised when the user interrupts with Ctrl+C)
├── GeneratorExit (raised when a generator or coroutine is closed)
└── Exception
    ├── ArithmeticError
    ├── AttributeError
    ├── ValueError
    └── (many other standard exceptions)
```

## ▼ Raising Exceptions

- In Python, the raise statement is used to intentionally trigger an exception in your code. Raising exceptions allows you to signal that an error or unexpected condition has occurred, even if the Python interpreter itself has not detected any issue.
- This is useful in situations where you want to enforce certain conditions or validate input explicitly.

```
def check_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative!")
    return age
```

```
try:
    age = check_age(-5)
except ValueError as e:
    print(e)
```

 Age cannot be negative!

## ▼ Summary

- try: Code that may raise an exception.
- except: Code that runs if an exception occurs.
- else: Code that runs if no exception occurs.

- finally: Code that runs no matter what, usually for cleanup.

Using exception handling effectively helps make your programs more robust and user-friendly by managing errors gracefully.

```
class MyClass:
    def __init__(self,value):
        self.value = value

    def __eq__(self,other):
        return self.value == other.value

class MyOtherClass(MyClass):

    def __eq__(self,other):
        print(self.value,other.value)
        return self.value == 2*other.value

a = MyClass(10)
b = MyClass(10)
c = MyOtherClass(5)
d = MyOtherClass(20)
```

```
print(a==b)
print(a==c)
print(c==a)
print(a==d)
print(d==a)
```

```
True
5 10
False
5 10
False
20 10
True
20 10
True
```

```
import numpy as np
```

```
arr,step=np.linspace(0,10,5,retstep=True)
```

```
arr
```

```
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

```
step
```

```
np.float64(2.5)
```

```
np.logspace(1,3,5,base=2)#Here logarithms of numbers are spaced evenly with base 2
```

```
array([2.        ,  2.82842712,  4.        ,  5.65685425,  8.        ])
```

```
x=np.array([[1],[2]])
y=np.array([[3],[4]])
np.hstack((x,y))
```

```
array([[1,  3],
       [2,  4]])
```

```
np.vstack((x,y))
```

```
array([[1],
       [2],
       [3],
       [4]])
```

```
a=np.array([[1],[2],[3]])  
b=np.array([10,20,30])
```

```
print(a+b)
```

```
↕ [[11 21 31]  
   [12 22 32]  
   [13 23 33]]
```

```
a.shape
```

```
↕ (3, 1)
```

```
b.shape
```

```
↕ (3,)
```

Start coding or [generate](#) with AI.

Double-click (or enter) to edit

Start coding or [generate](#) with AI.