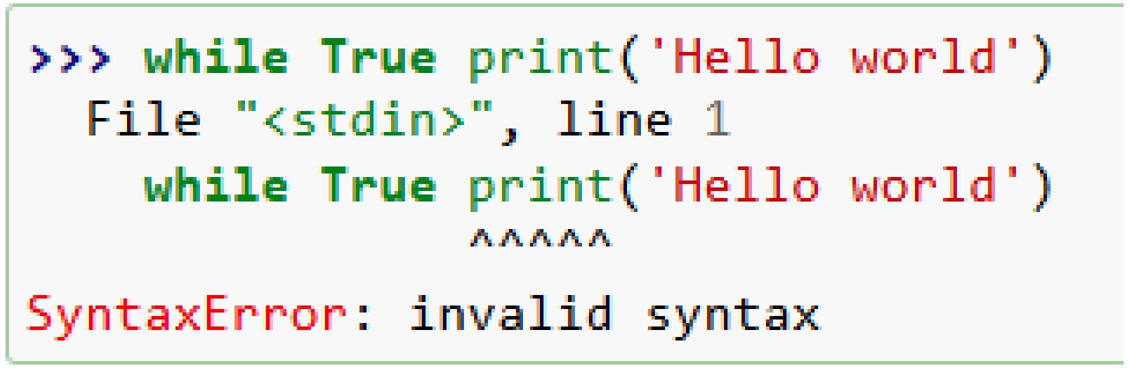
Errors and Exceptions

Syntax Errors

- Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:



```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                   ^^^^^
SyntaxError: invalid syntax
```

- The parser repeats the offending line and displays little arrows pointing at the place where the error was detected. Note that this is not always the place that needs to be fixed. In the example, the error is detected at the function print(), since a colon (':') is missing just before it.

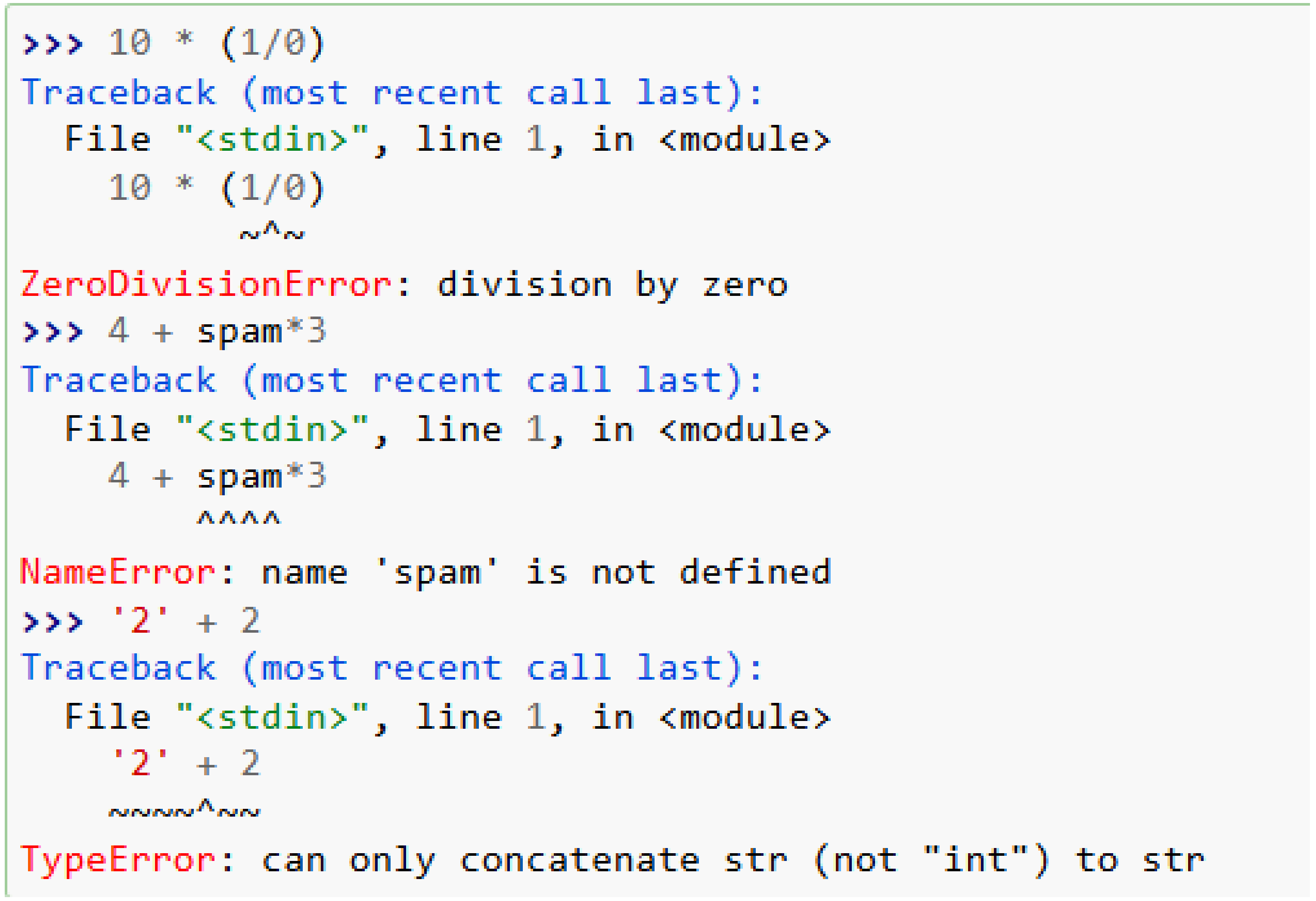
Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.

Errors detected during execution are called exceptions and are not unconditionally fatal.

Most exceptions are not handled by programs.

ERROR MESSAGES IN CASE OF EXCEPTIONS



```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    10 * (1/0)
          ~^~
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    4 + spam*3
        ^^^^
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    '2' + 2
    ~~~~^~~
TypeError: can only concatenate str (not "int") to str
```

Built-in Exceptions

Commonly occurring exceptions are usually defined in the compiler/interpreter.

These are called built-in exceptions.

Python’s standard library is an extensive collection of built-in exceptions that deals with the commonly occurring errors (exceptions) by providing the standardized solutions for such errors.

On the occurrence of any built-in exception, the appropriate exception handler code is executed which displays the reason along with the raised exception name.

The programmer then has to take appropriate action to handle it. Some of the commonly occurring built-in exceptions that can be raised in Python

Built-in exceptions in Python

SyntaxError:

- It is raised when there is an error in the syntax of the Python code.

ValueError

- It is raised when a built-in method or operation receives an argument that has the right data type but mismatched or inappropriate values.

IOError

- It is raised when the file specified in a program statement cannot be opened.

KeyboardInterrupt

- It is raised when the user accidentally hits the Delete or Esc key while executing a program due to which the normal flow of the program is interrupted.

Built-in exceptions in Python

ImportError

- It is raised when the requested module definition is not found.

EOFError

- It is raised when the end of file condition is reached without reading any data by input().

ZeroDivisionError

- It is raised when the denominator in a division operation is zero.

IndexError

- It is raised when the index or subscript in a sequence is out of range.

Built-in exceptions in Python

NameError

- It is raised when a local or global variable name is not defined.

IndentationError

- It is raised due to incorrect indentation in the program code.

TypeError

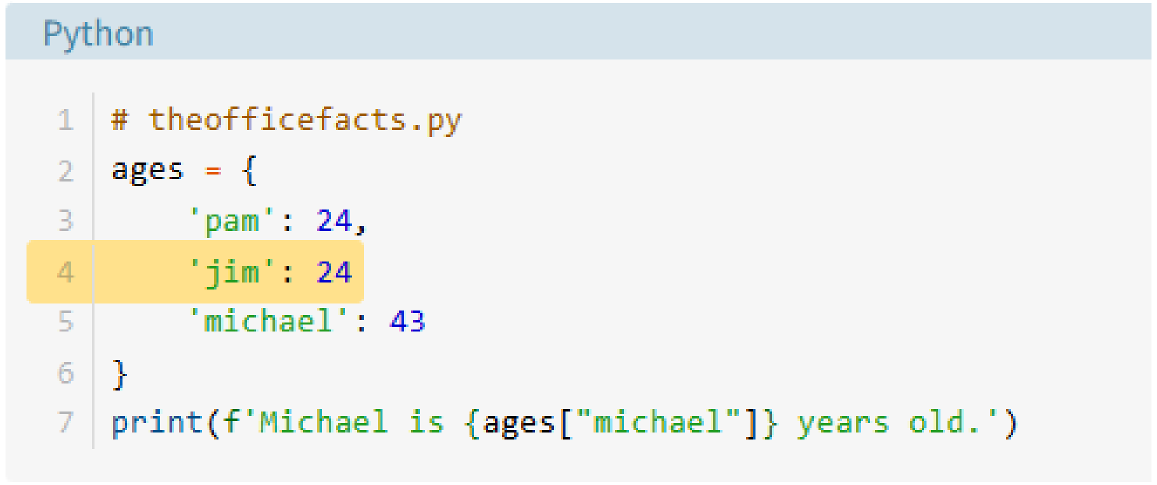
- It is raised when an operator is supplied with a value of incorrect data type.

OverFlowError

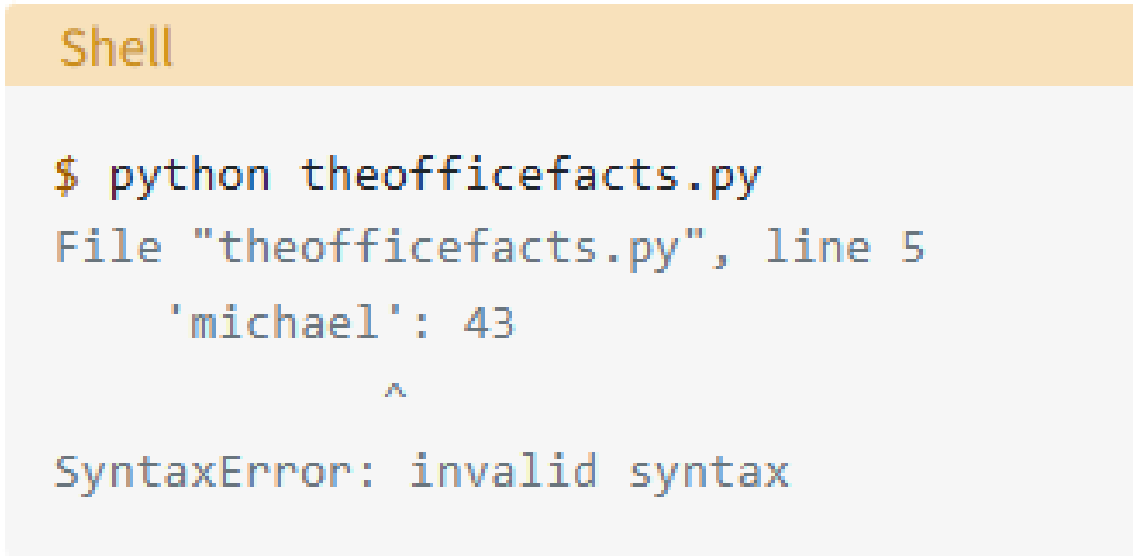
- It is raised when the result of a calculation exceeds the maximum limit for numeric data type.

SyntaxError

- You can see the invalid syntax in the dictionary literal on line 4. The second entry, 'jim', is missing a comma.



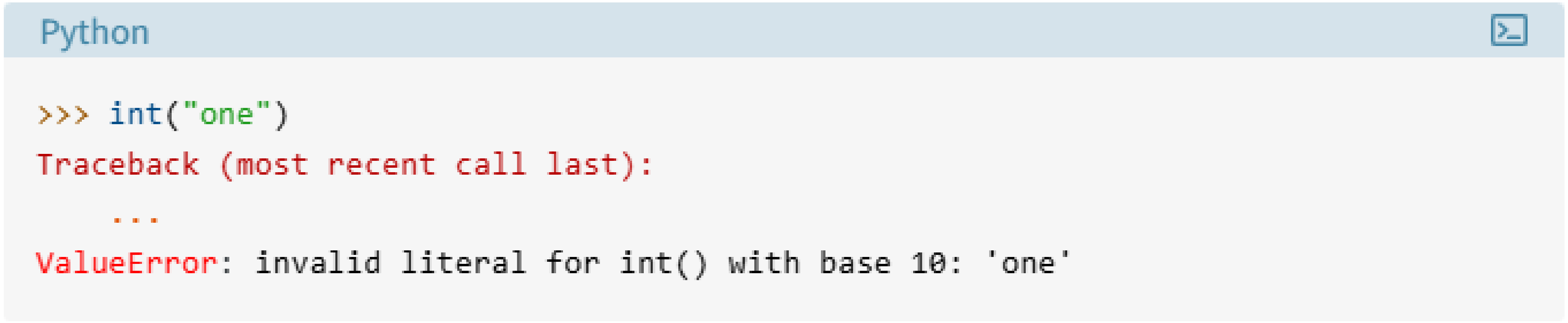
```python
1  # theofficefacts.py
2  ages = {
3      'pam': 24,
4      'jim': 24
5      'michael': 43
6  }
7  print(f'Michael is {ages["michael"]} years old.')
```



```
$ python theofficefacts.py
File "theofficefacts.py", line 5
    'michael': 43
            ^
SyntaxError: invalid syntax
```

ValueError

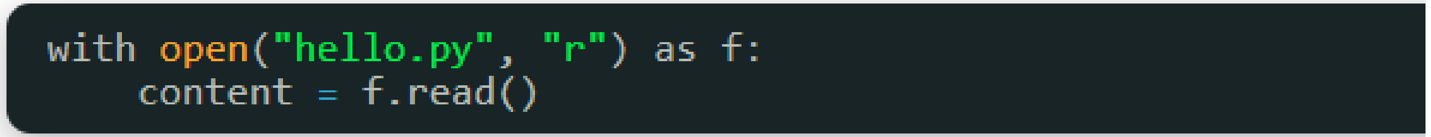
- ValueError is a built-in exception that gets raised when a function or operation receives an argument with the right type but an invalid value. It means the argument's type is okay, but its actual value isn't acceptable for the operation at hand.



```
>>> int("one")
Traceback (most recent call last):
    ...
ValueError: invalid literal for int() with base 10: 'one'
```

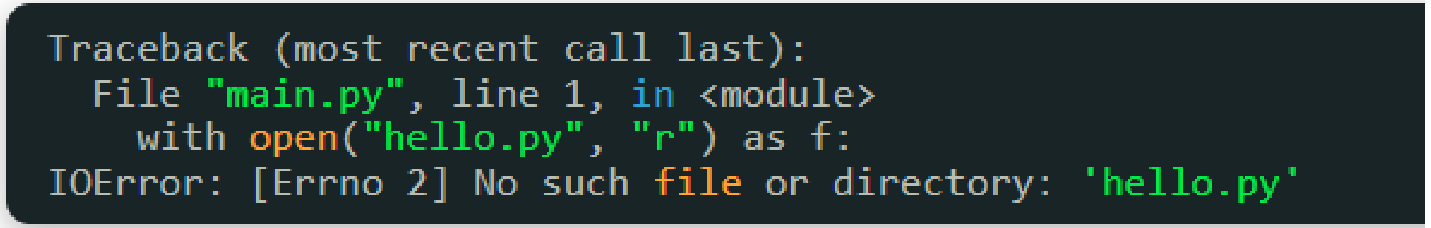
IOError

- IOError exception is raised when an input or output operation is taking place, there can be many reasons for this.
  - The file a user tried to access does not exist
  - The user does not have permission to access the file
  - File already in use
  - The device is running out of space to process



```
with open("hello.py", "r") as f:
    content = f.read()
```

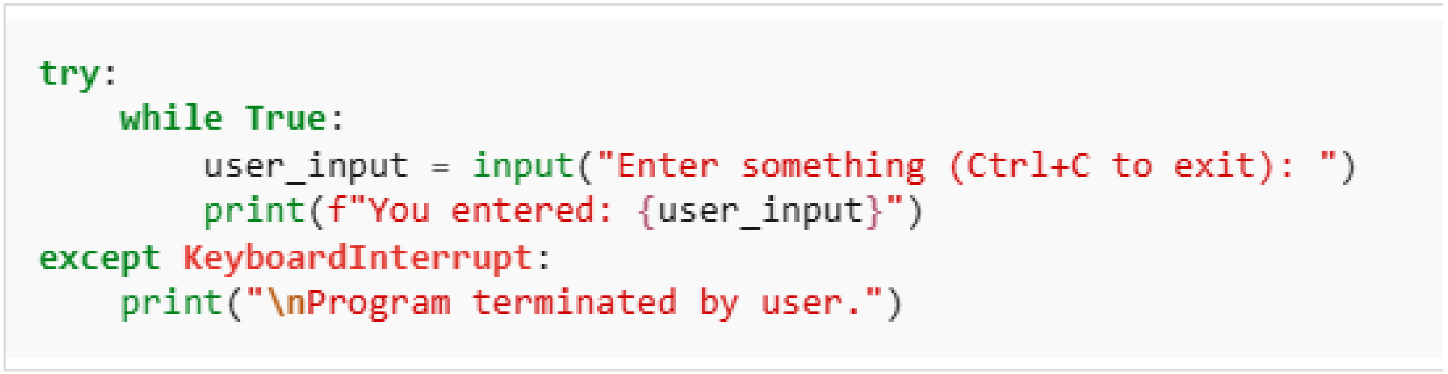
Output:



```
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    with open("hello.py", "r") as f:
IOError: [Errno 2] No such file or directory: 'hello.py'
```

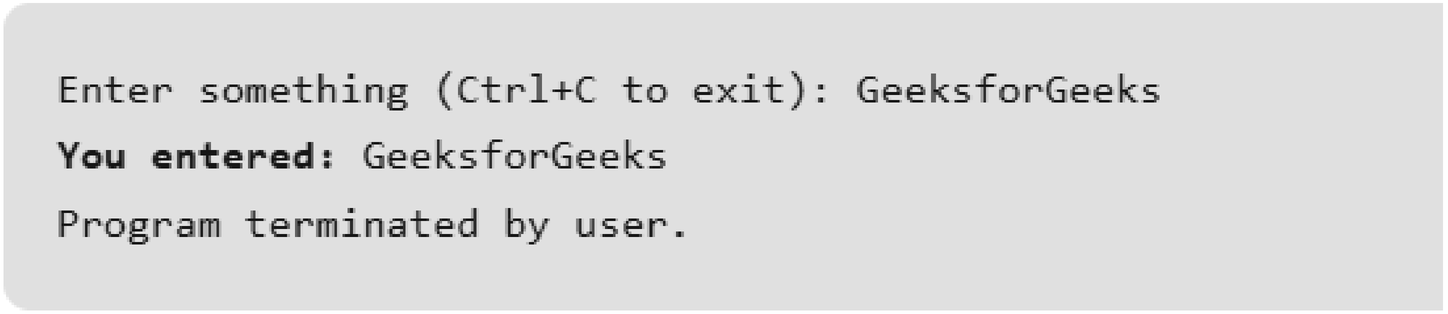
KeyboardInterrupt

- In Python, **KeyboardInterrupt** is a built-in exception that occurs when the user interrupts the execution of a program using a keyboard action, typically by pressing **Ctrl+C**.
- Handling **KeyboardInterrupt** is crucial, especially in scenarios where a program involves time-consuming operations or user interactions.



```
try:
    while True:
        user_input = input("Enter something (Ctrl+C to exit): ")
        print(f"You entered: {user_input}")
except KeyboardInterrupt:
    print("\nProgram terminated by user.")
```

Output

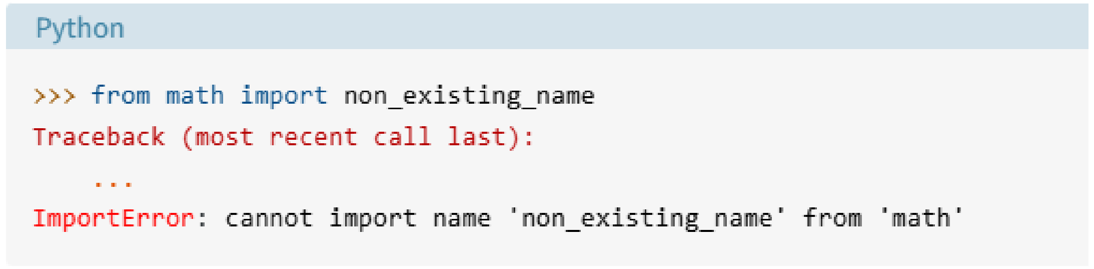


```
Enter something (Ctrl+C to exit): GeeksforGeeks
You entered: GeeksforGeeks
Program terminated by user.
```

ImportError

- ImportError is a built-in exception that occurs when an import statement doesn't successfully load a module or a name from a module.

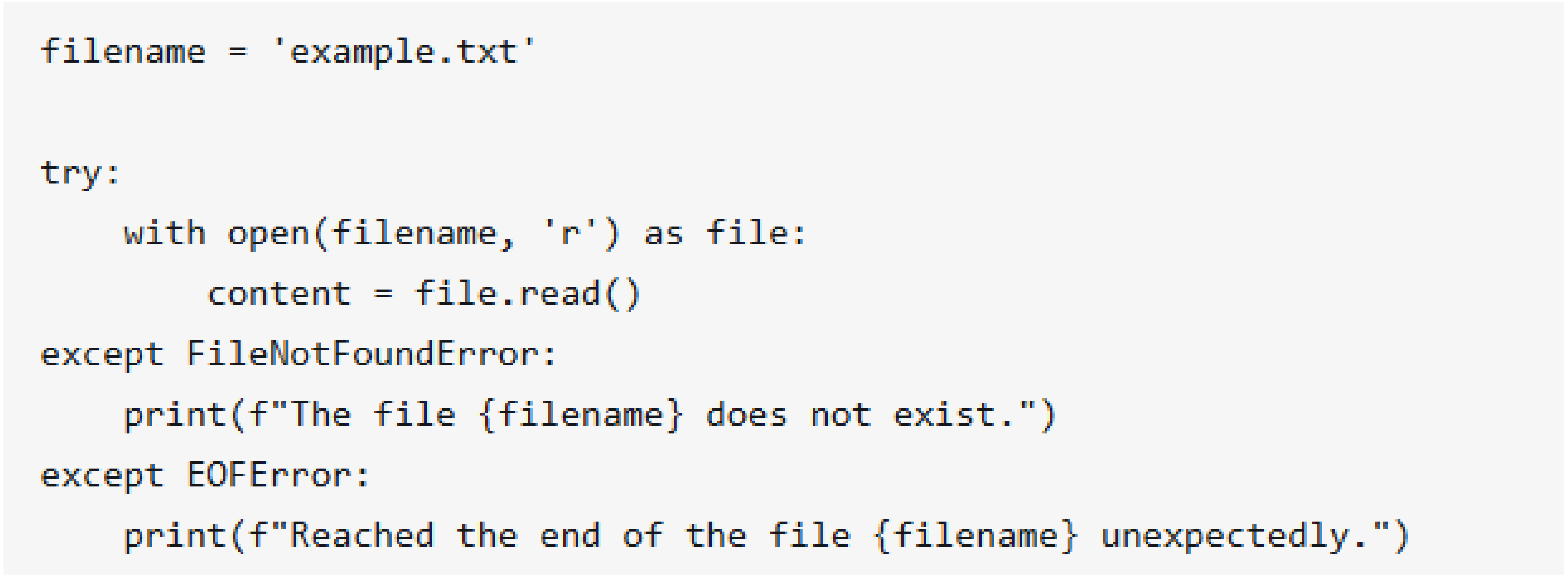
- This exception is part of Python's import system, which is responsible for loading and managing modules.
  - Importing a module that's not installed or not available in your current environment
  - Trying to import a specific name from a module when the name doesn't exist


Python

```
>>> from math import non_existing_name
Traceback (most recent call last):
    ...
ImportError: cannot import name 'non_existing_name' from 'math'
```

EOFERROR

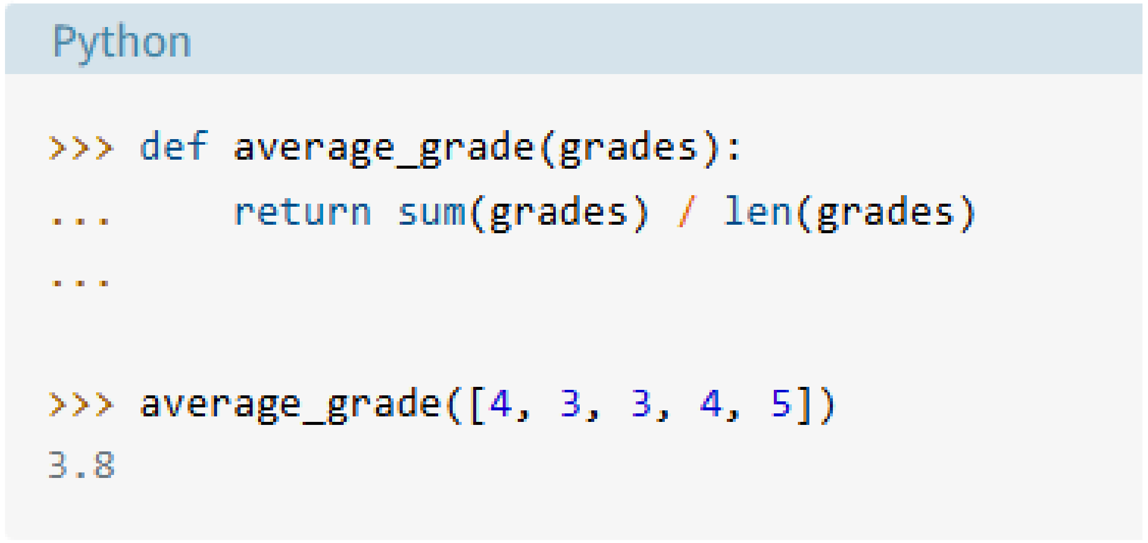
- EOF stands for End Of File. It's like the final period at the end of a book, signaling that there's nothing more to read. In the context of programming, EOF refers to a condition where a program reaches the end of an input stream (like a file or user input) and there is no more data to be read.



```
filename = 'example.txt'


try:
    with open(filename, 'r') as file:
        content = file.read()
except FileNotFoundError:
    print(f"The file {filename} does not exist.")
except EOFError:
    print(f"Reached the end of the file {filename} unexpectedly.")
```

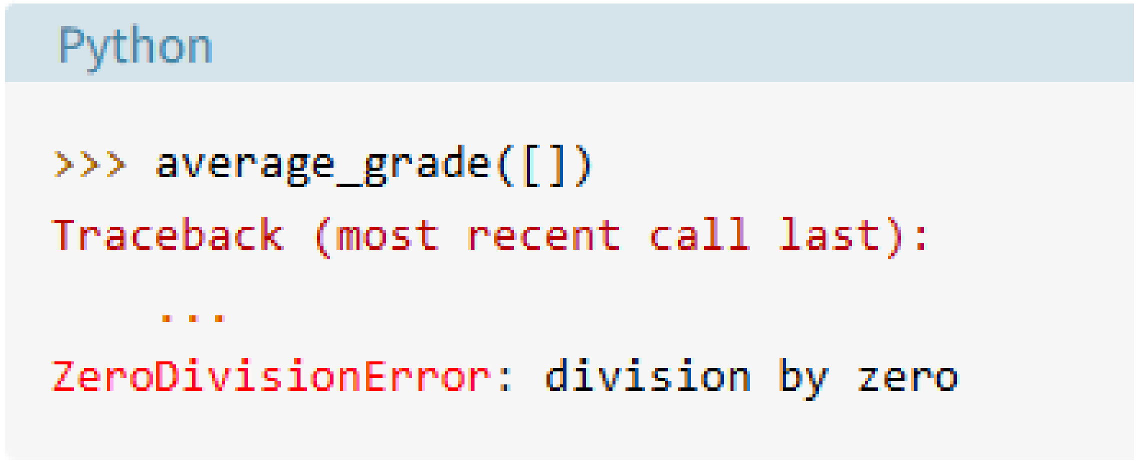
ZeroDivisionError

- Python raises the ZeroDivisionError exception when a division’s divisor or right operand is zero



Python

```
>>> def average_grade(grades):
...     return sum(grades) / len(grades)
...

>>> average_grade([4, 3, 3, 4, 5])
3.8
```

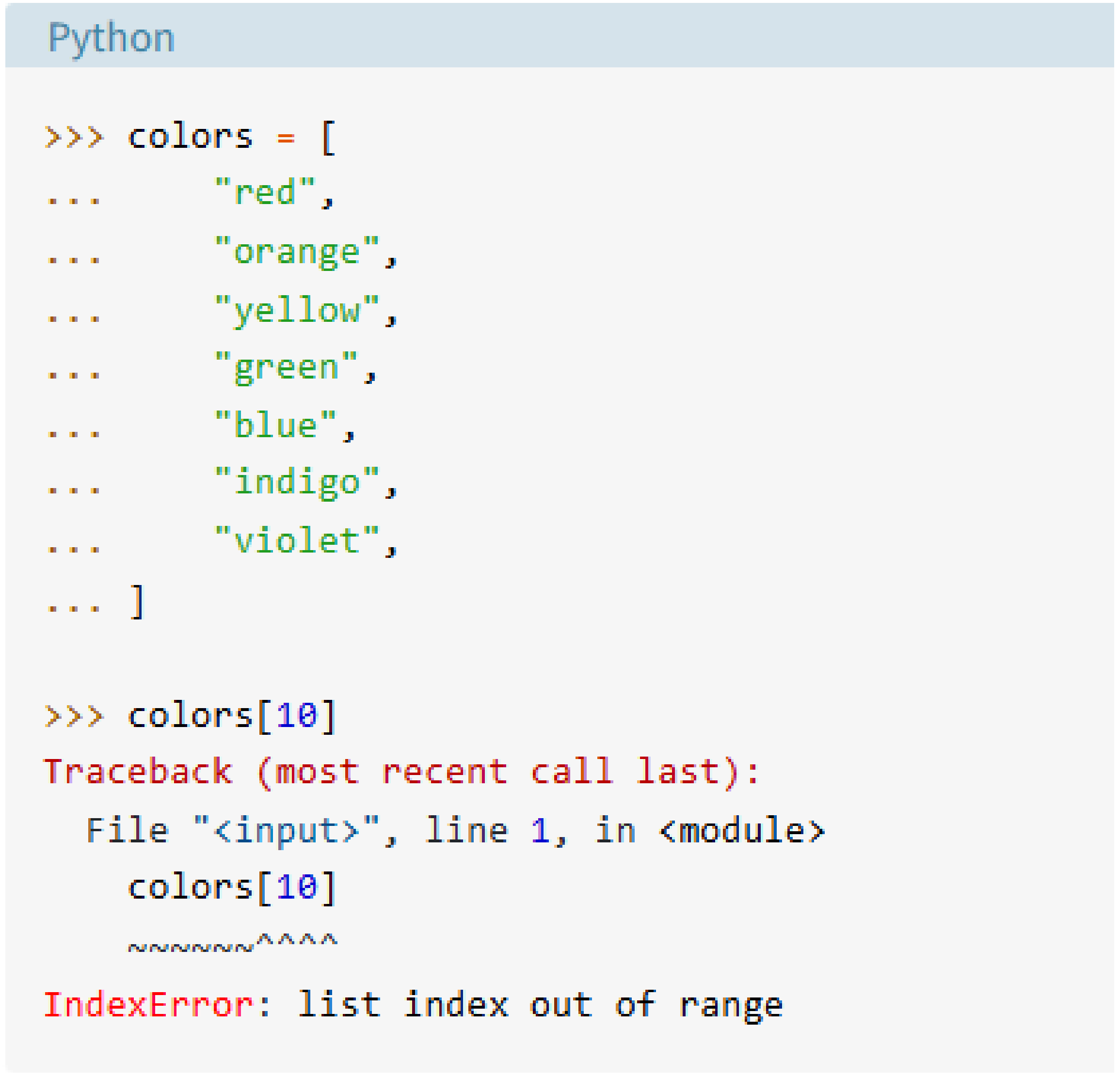


Python

```
>>> average_grade([])
Traceback (most recent call last):
    ...
ZeroDivisionError: division by zero
```

IndexError

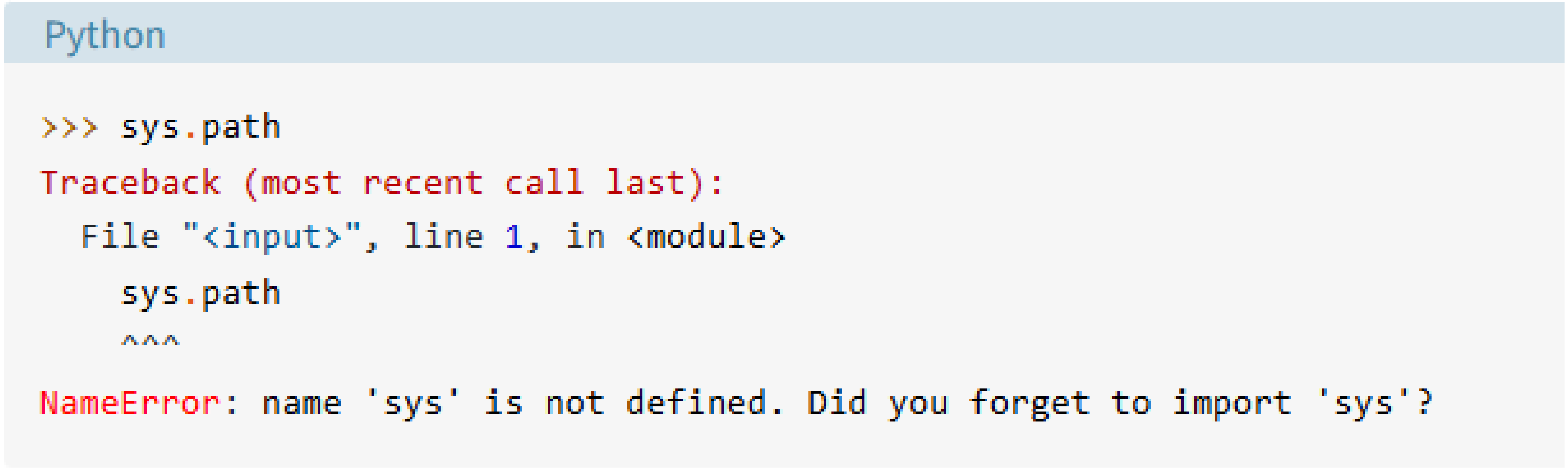
- The IndexError exception happens when you try to retrieve a value from a sequence using an out-of-range index:


Python

```
>>> colors = [
...     "red",
...     "orange",
...     "yellow",
...     "green",
...     "blue",
...     "indigo",
...     "violet",
... ]

>>> colors[10]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    colors[10]
    ~~~~~~^^^^
IndexError: list index out of range
```

NameError

- The NameError exception is also pretty common when you're starting with Python. This exception happens when you try to use a name that's not present in your current namespace. So, this exception is closely related to scopes and namespaces.
- For example, say that you're working in a REPL session. You've imported the sys module to use it in your current task. For some reason, you restart the interactive session and try to use sys without re-importing it:

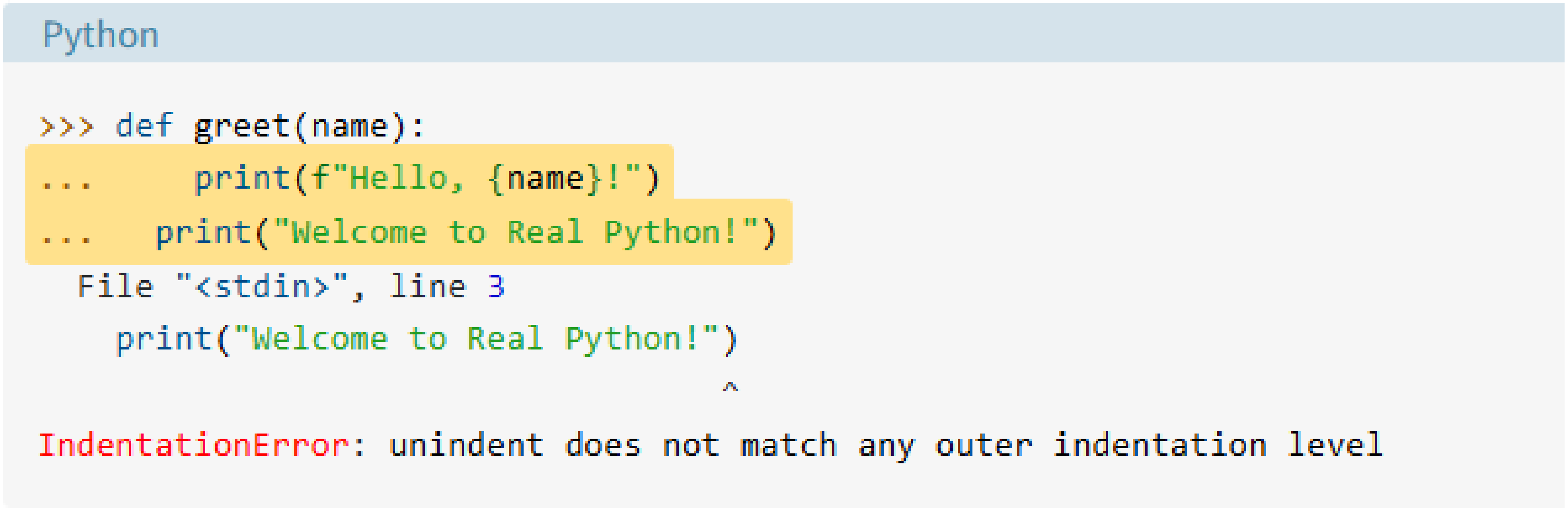


Python

```
>>> sys.path
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    sys.path
    ^^^
NameError: name 'sys' is not defined. Did you forget to import 'sys'?
```

IndentationError

- Unlike other programming languages, indentation is part of Python syntax. To delimit a code block in Python, you use indentation. Therefore, you may get an error if the indentation isn't correct. Python uses the IndentationError exception to denote this problem.

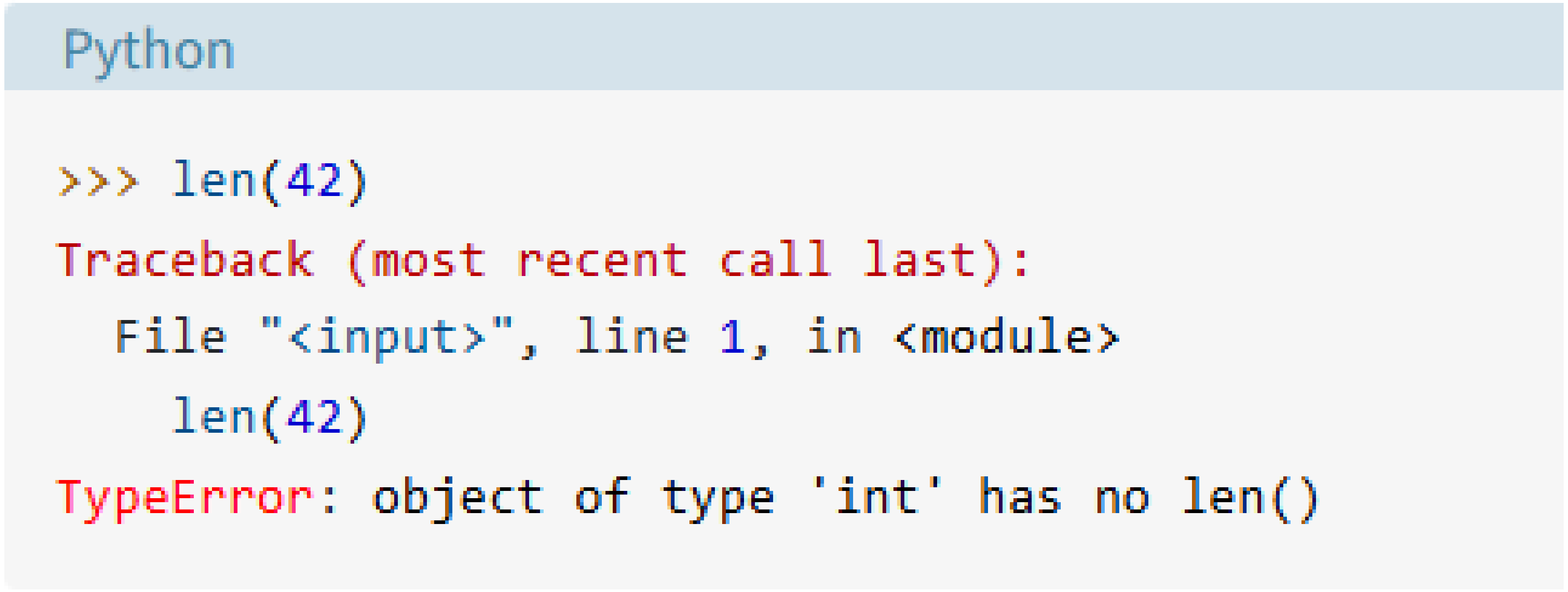


Python

```
>>> def greet(name):
...     print(f"Hello, {name}!")
...   print("Welcome to Real Python!")
  File "<stdin>", line 3
    print("Welcome to Real Python!")
                                   ^
IndentationError: unindent does not match any outer indentation level
```

TypeError

- Python raises a TypeError exception when you apply an operation or function to an object that doesn't support that operation. For example, consider calling the built-in len() function with a number as an argument:

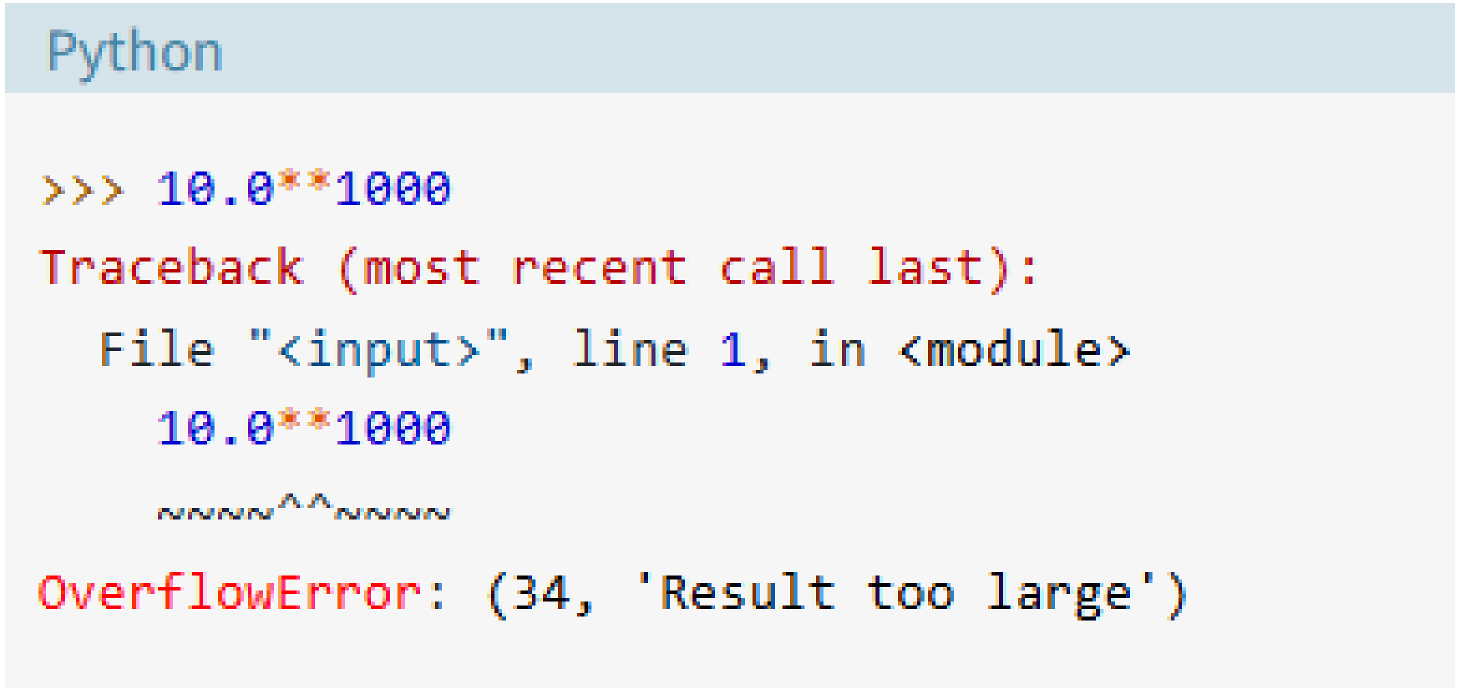


Python

```
>>> len(42)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    len(42)
TypeError: object of type 'int' has no len()
```

OverflowError

- The OverflowError isn’t that common. Python raises this exception when the result of an arithmetic operation is too large, and there’s no way to represent it:


Python

```
>>> 10.0**1000
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    10.0**1000
    ~~~~^^~~~~
OverflowError: (34, 'Result too large')
```

Raising Exceptions

- Each time an error is detected in a program, the Python interpreter raises (throws) an exception.
- Exception handlers are designed to execute when a specific exception is raised.
- Programmers can also forcefully raise exceptions in a program using the raise and assert statements.
- Once an exception is raised, no further statement in the current block of code is executed.
- So, raising an exception involves interrupting the normal flow execution of program and jumping to that part of the program (exception handler code) which is written to handle such exceptional situations.

The raise Statement

- Python has the raise statement as part of its syntax. You can use this statement to raise exceptions in your code as a response to exceptional situations.

- The raise statement can be used to throw an exception. The syntax of raise statement is:
- *raise exception-name[(optional argument)]*

The raise Statement

Python

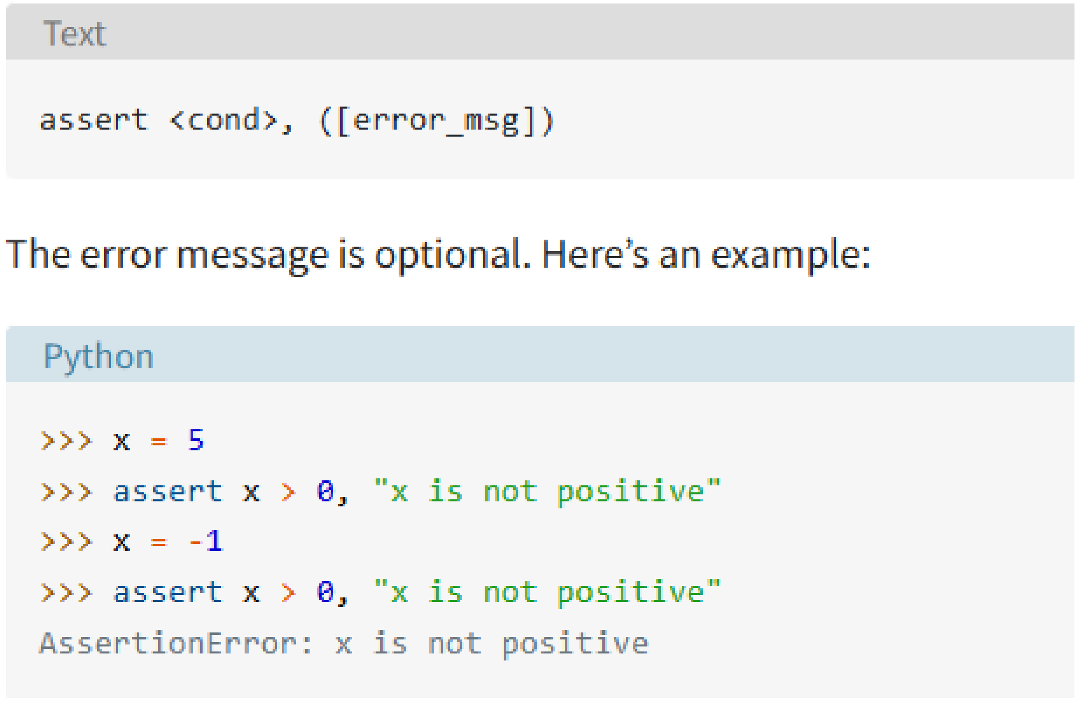
```
>>> def average_grade(grades):
...     return sum(grades) / len(grades)
...

>>> average_grade([5, 4, 5, 3])
4.25

>>> average_grade([])
Traceback (most recent call last):
    ...
ZeroDivisionError: division by zero
```

Python

```
>>> def average_grade(grades):
...     if not grades:
...         raise ValueError("empty grades not allowed")
...     return sum(grades) / len(grades)
...

>>> average_grade([5, 4, 5, 3])
4.25

>>> average_grade([])
Traceback (most recent call last):
    ...
ValueError: empty grades not allowed
```

The assert Statement

- An assert statement in Python is used to test an expression in the program code. If the result after testing comes false, then the exception is raised. This statement is generally used in the beginning of the function or after a function call to check for valid input. The syntax for assert statement is:
- *assert Expression[,arguments]*
- On encountering an assert statement, Python evaluates the expression given immediately after the assert keyword. If this expression is false, an AssertionError exception is raised which can be handled like any other exception.



Text

```
assert <cond>, ([error_msg])
```

The error message is optional. Here's an example:

Python

```
>>> x = 5
>>> assert x > 0, "x is not positive"
>>> x = -1
>>> assert x > 0, "x is not positive"
AssertionError: x is not positive
```

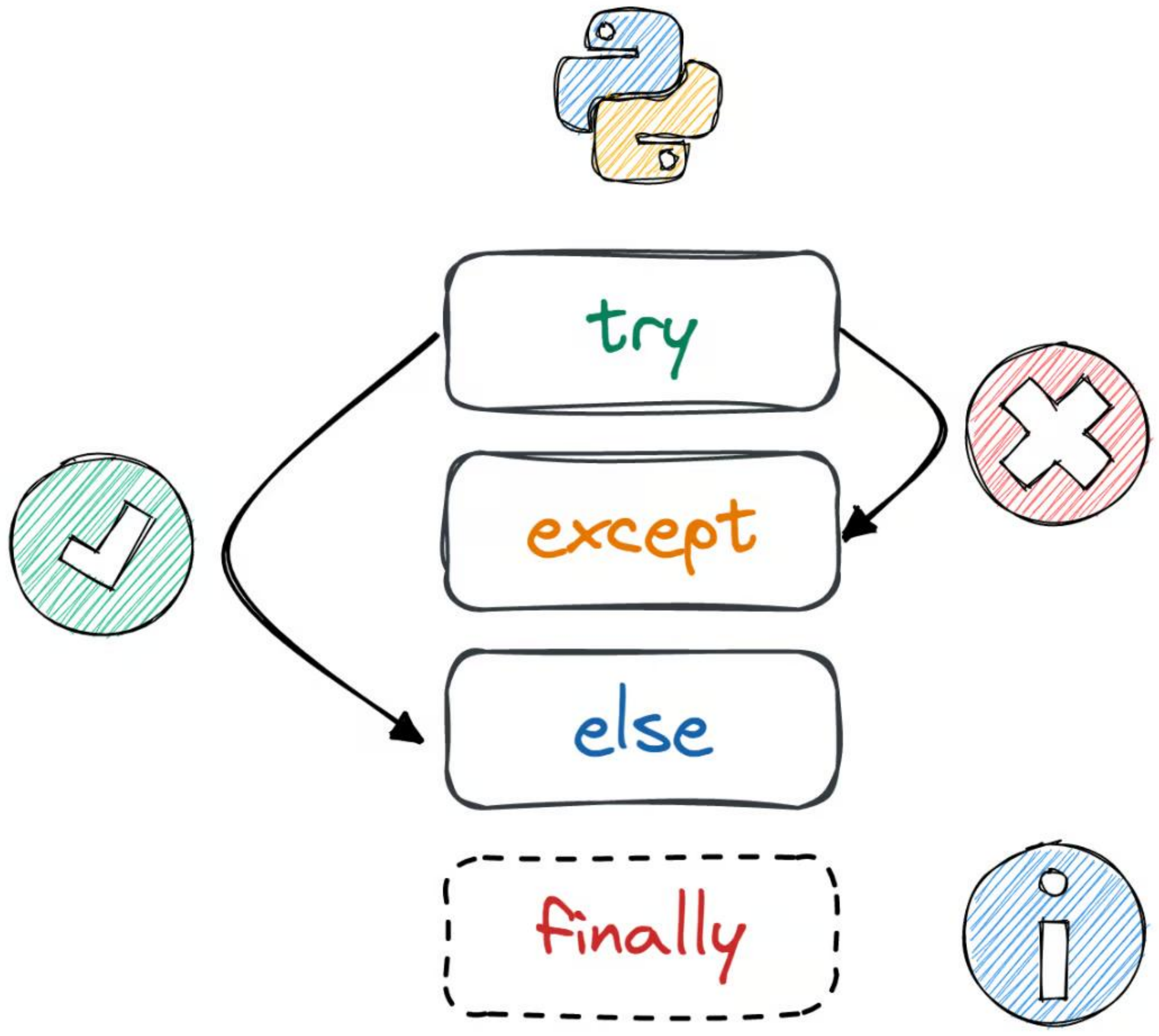
Exception Handling

Exception handling in Python is a mechanism to deal with errors that occur during the execution of a program.

These errors, also known as exceptions, can disrupt the normal flow of the program.

Exception handling allows you to gracefully manage these errors, prevent the program from crashing, and potentially take corrective actions.

EXCEPTION HANDLING



Exception Handling with try, except, else, and finally

After learning about errors and exceptions, we will learn to handle them by using

try,	except,	else, and	finally blocks.

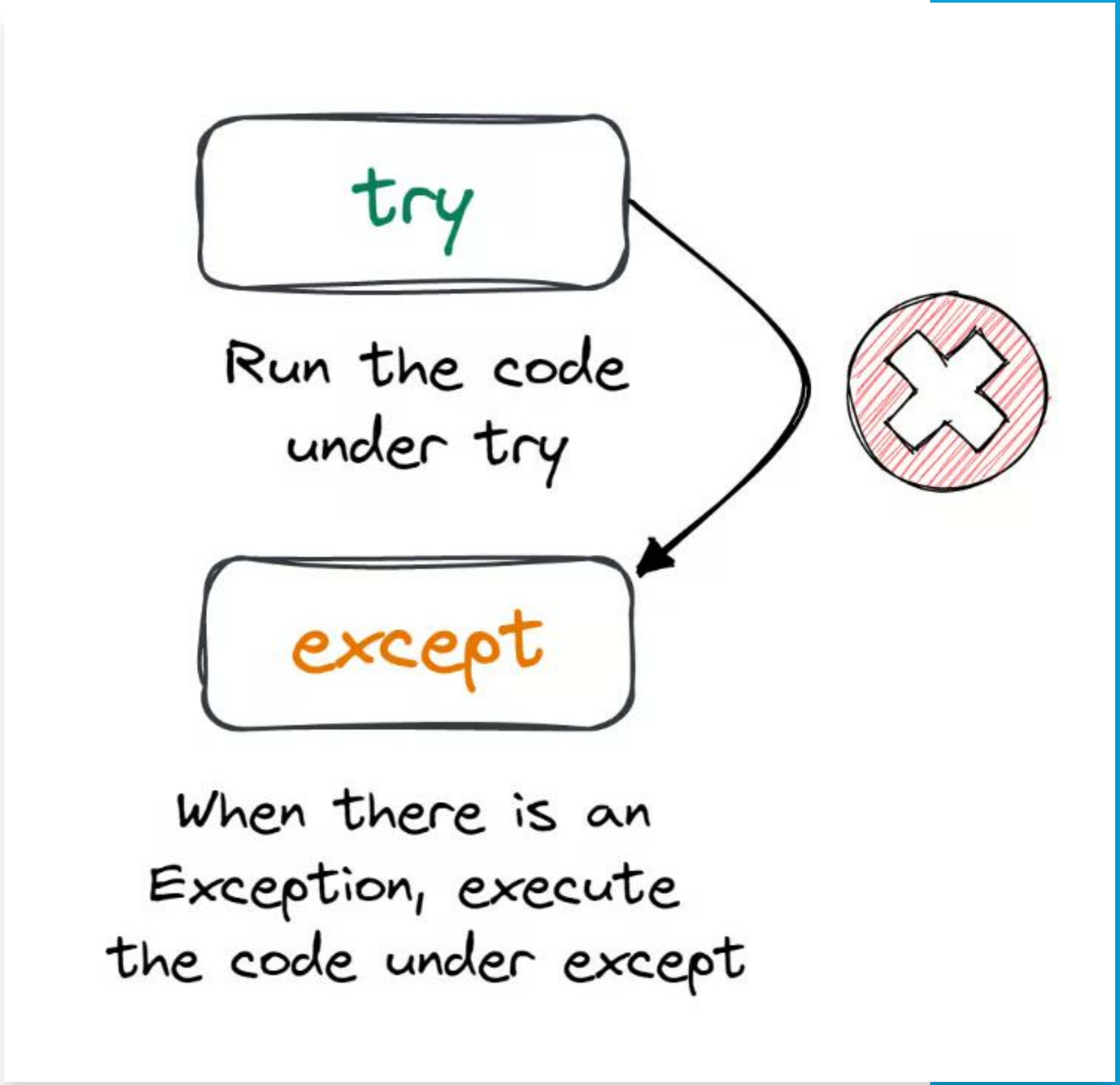
So, what do we mean by handling them?

In normal circumstances, these errors will stop the code execution and display the error message.

To create stable systems, we need to anticipate these errors and come up with alternative solutions or warning messages.

The try and except statement

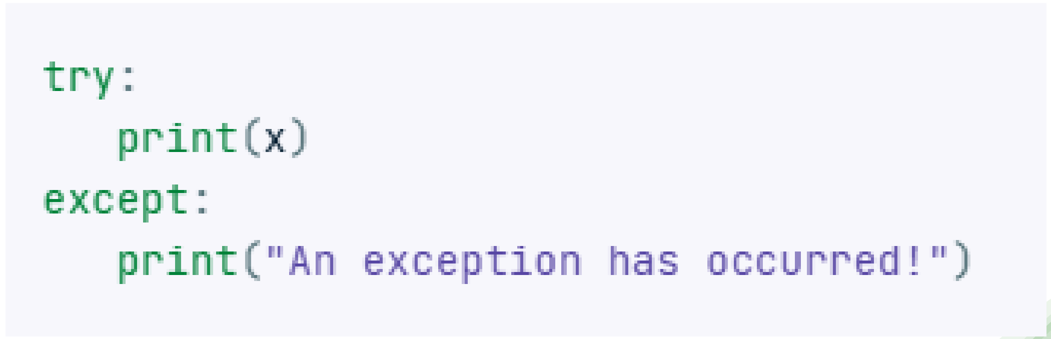
- The most simple way of handling exceptions in Python is by using the try and except block.
    - Run the code under the try statement.
    - When an exception is raised, execute the code under the except statement.



Simple example

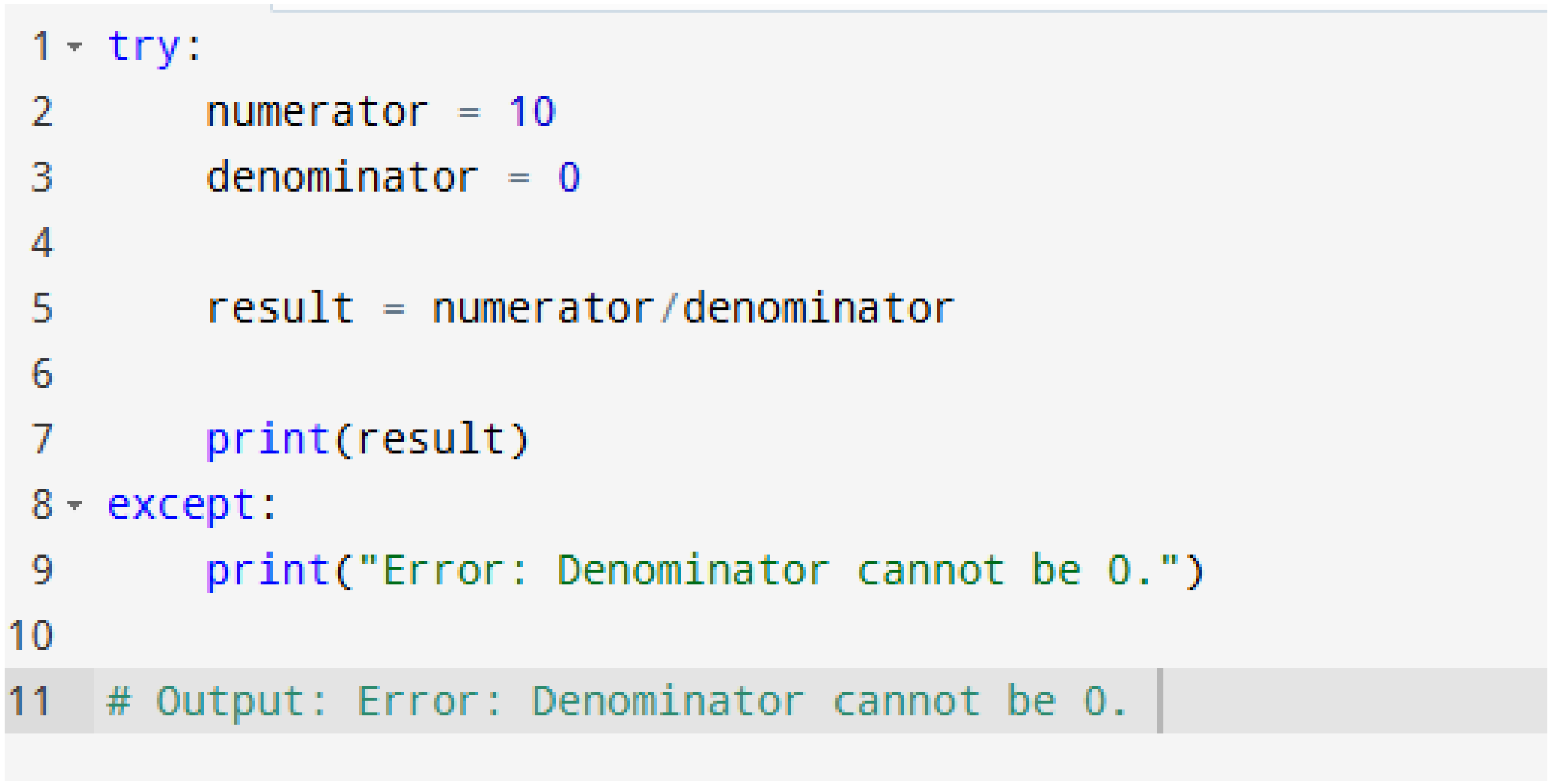
- we will try to print the undefined x variable. In normal circumstances, it should throw the error and stop the execution, but with the try and except block, we can change the flow behavior.

  - The program will run the code under the try statement.
  - As we know, that x is not defined, so it will run the except statement and print the warning.



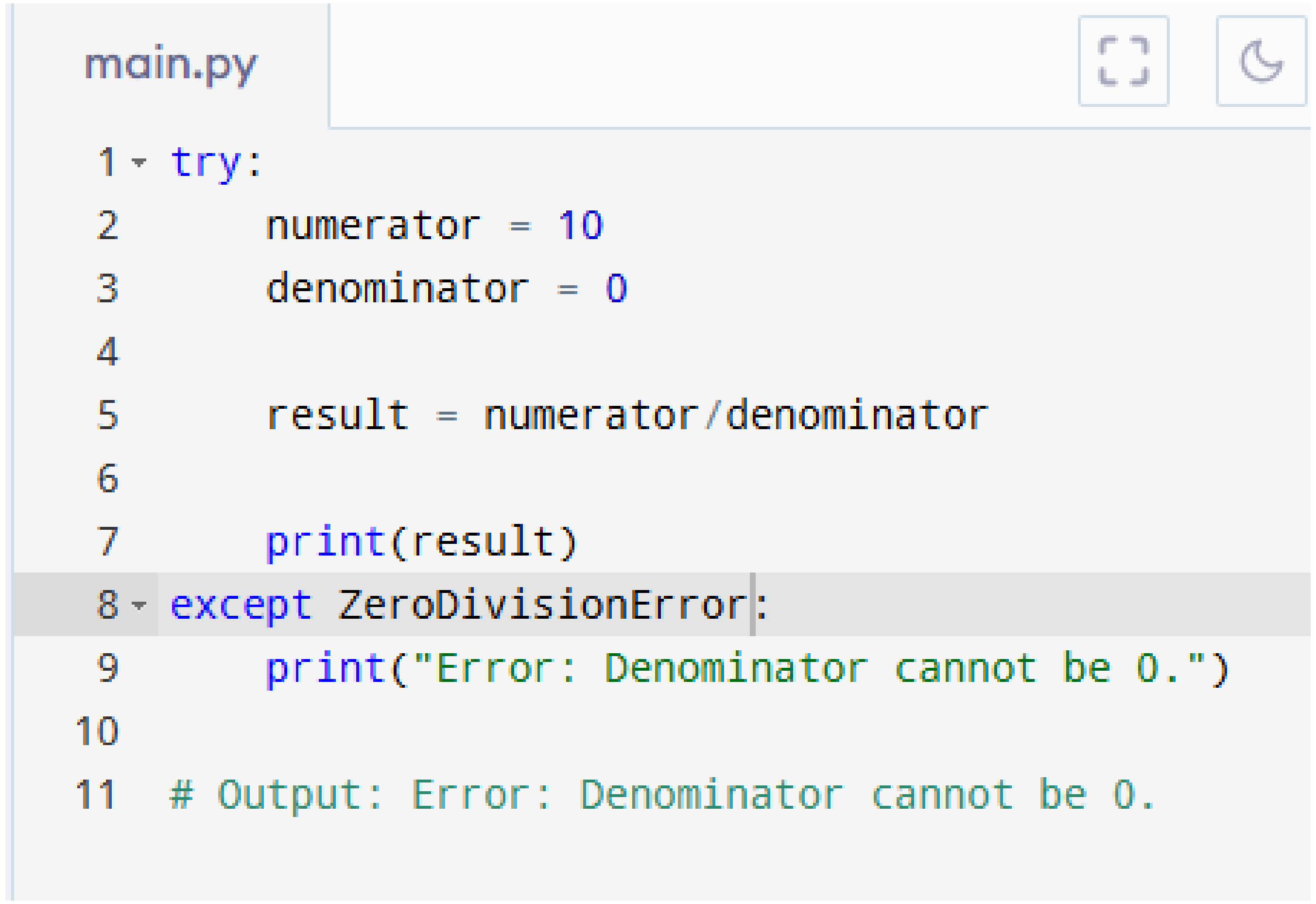
```python
try:
    print(x)
except:
    print("An exception has occurred!")
```

Try and except block



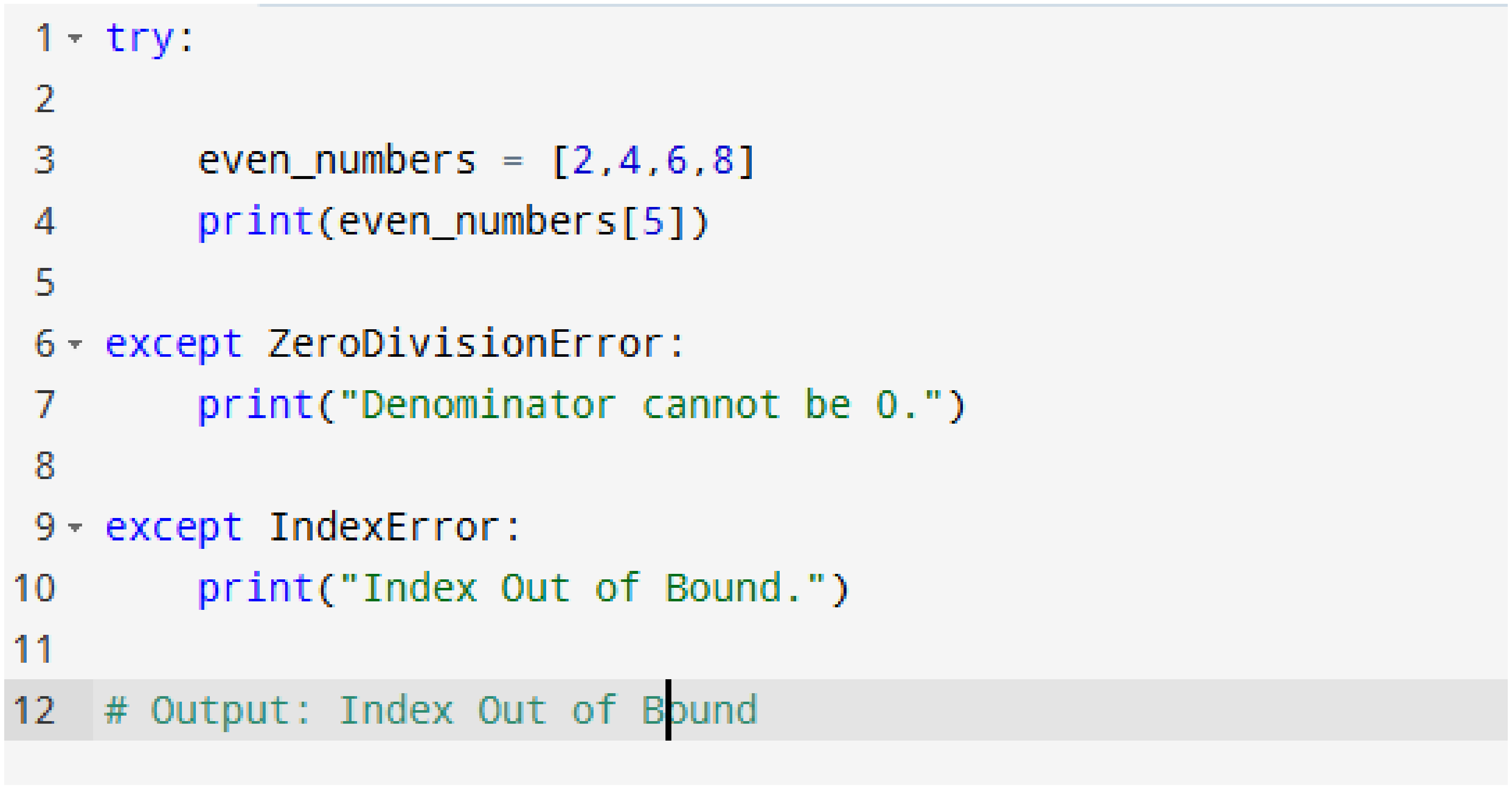
```python
try:
    numerator = 10
    denominator = 0

    result = numerator/denominator

    print(result)
except:
    print("Error: Denominator cannot be 0.")

# Output: Error: Denominator cannot be 0.
```

Catching Specific Exceptions in Python


main.py

```python
try:
    numerator = 10
    denominator = 0

    result = numerator/denominator

    print(result)
except ZeroDivisionError:
    print("Error: Denominator cannot be 0.")

# Output: Error: Denominator cannot be 0.
```

Try and multiple except block

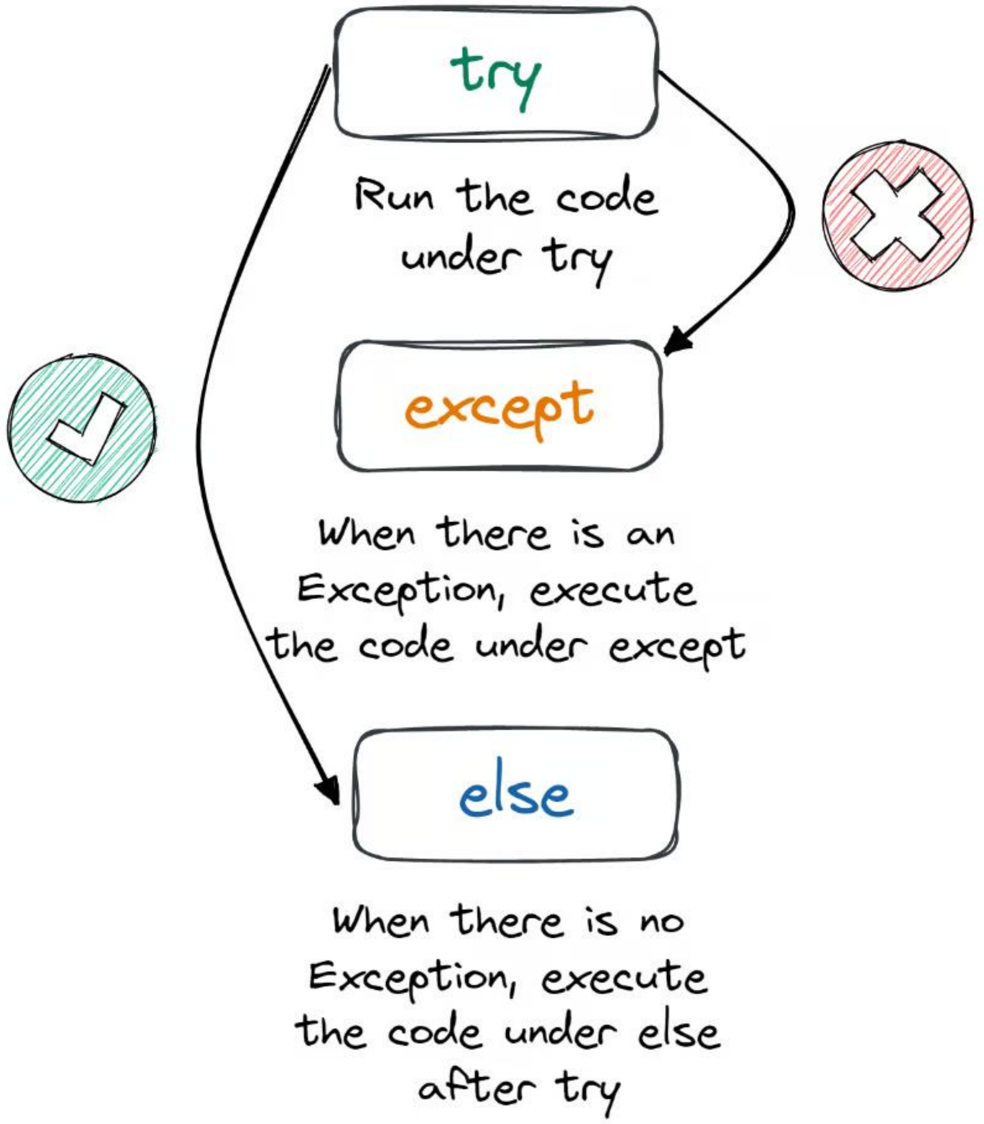


```
try:

    even_numbers = [2,4,6,8]
    print(even_numbers[5])

except ZeroDivisionError:
    print("Denominator cannot be 0.")

except IndexError:
    print("Index Out of Bound.")

# Output: Index Out of Bound
```

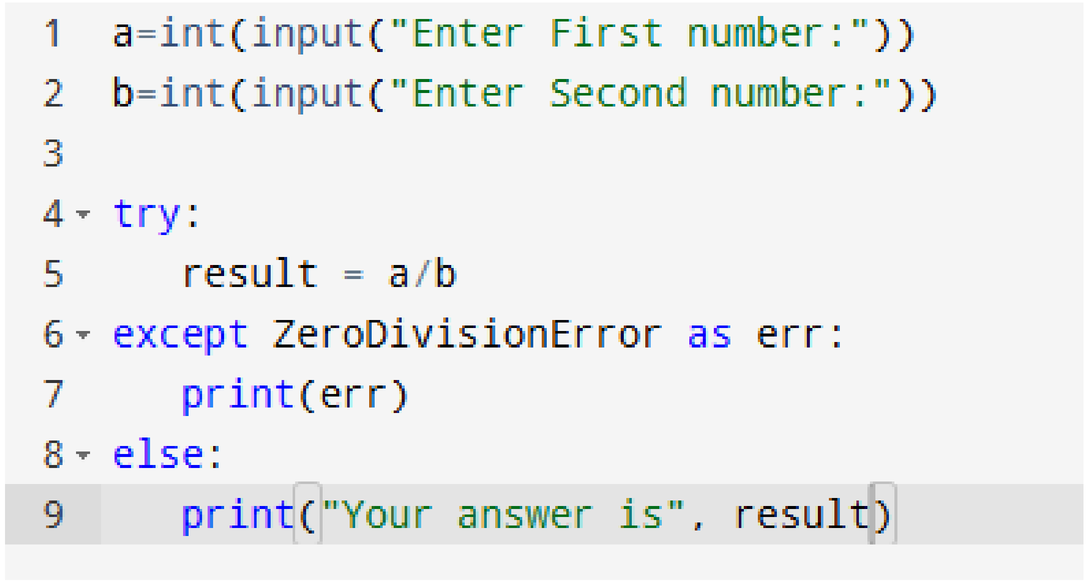
The try with else clause

- We have learned about try and except, and now we will be learning about the else statement.

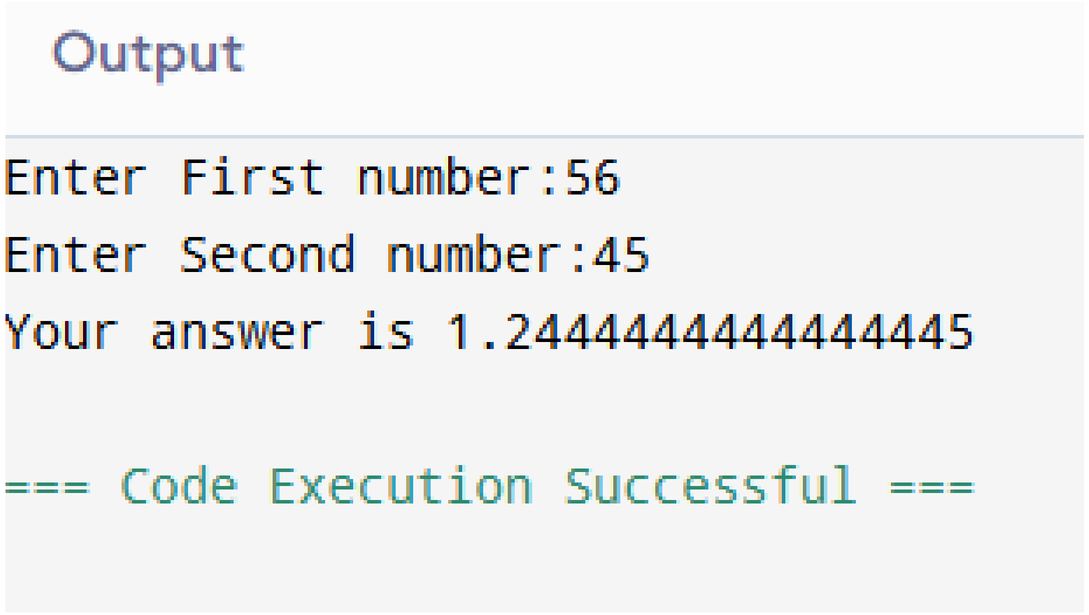
- When the try statement does not raise an exception, code enters into the else block. It is the remedy or a fallback option when you expect a part of your script will produce an exception. It is generally used in a brief setup or verification section where you don't want certain errors to hide.


try
Run the code under try
except
When there is an Exception, execute the code under except
else
When there is no Exception, execute the code under else after try

Example



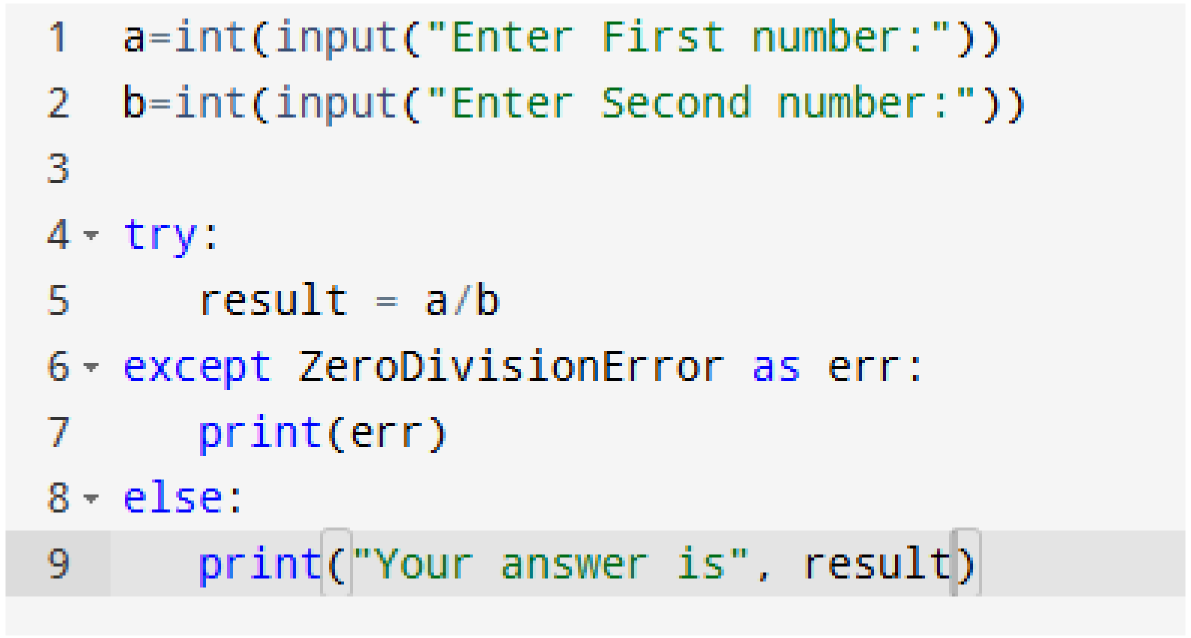
```python
a=int(input("Enter First number:"))
b=int(input("Enter Second number:"))

try:
    result = a/b
except ZeroDivisionError as err:
    print(err)
else:
    print("Your answer is", result)
```



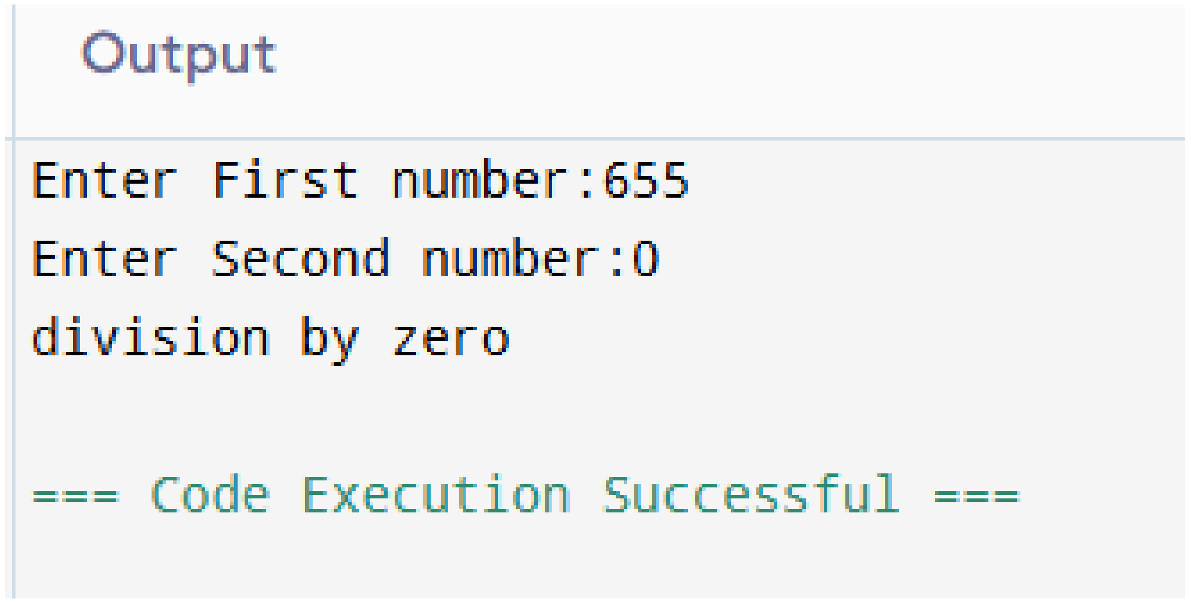
Output

```
Enter First number:56
Enter Second number:45
Your answer is 1.2444444444444445

=== Code Execution Successful ===
```

Example

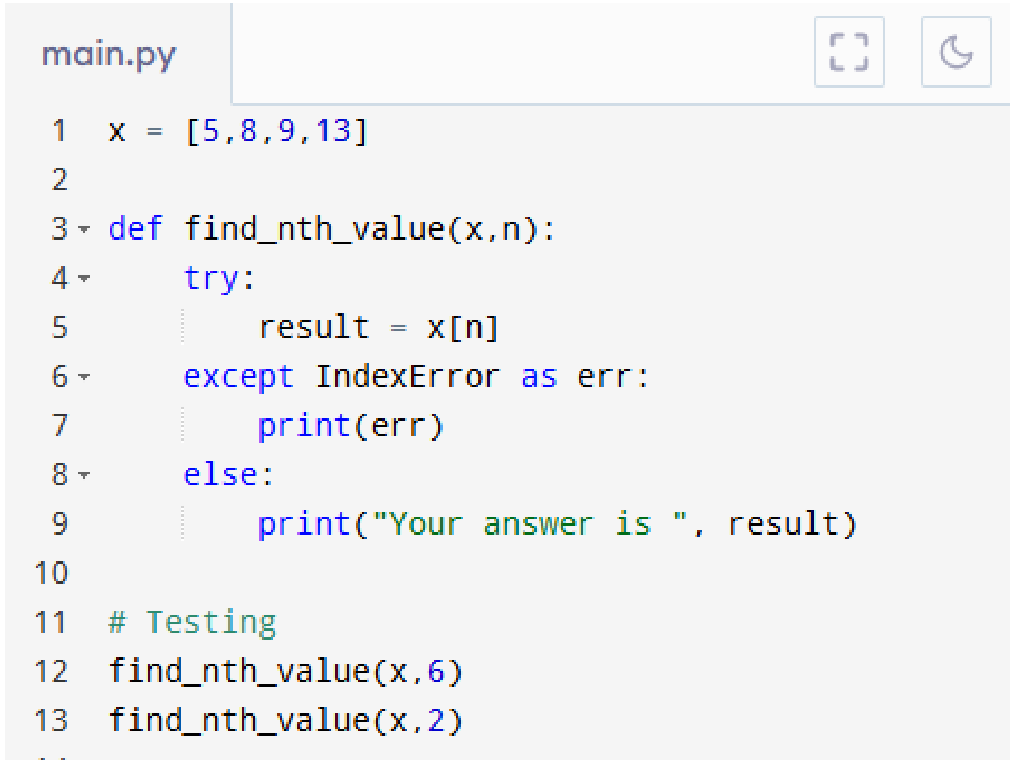


```python
a=int(input("Enter First number:"))
b=int(input("Enter Second number:"))

try:
    result = a/b
except ZeroDivisionError as err:
    print(err)
else:
    print("Your answer is", result)
```

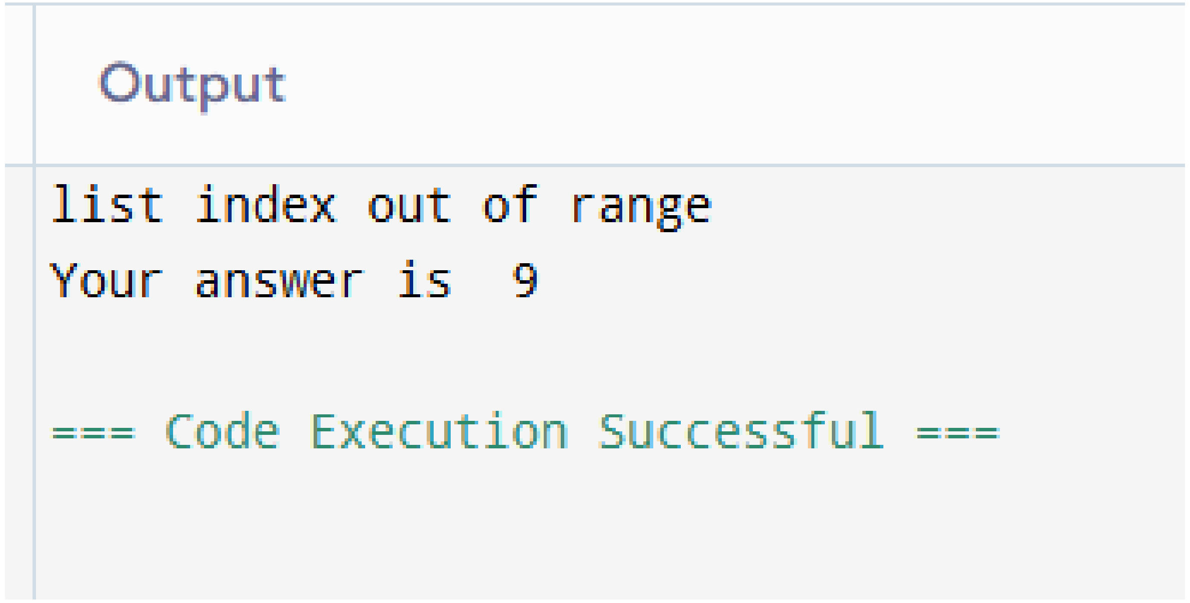


```
Enter First number:655
Enter Second number:0
division by zero


=== Code Execution Successful ===
```

Example


main.py

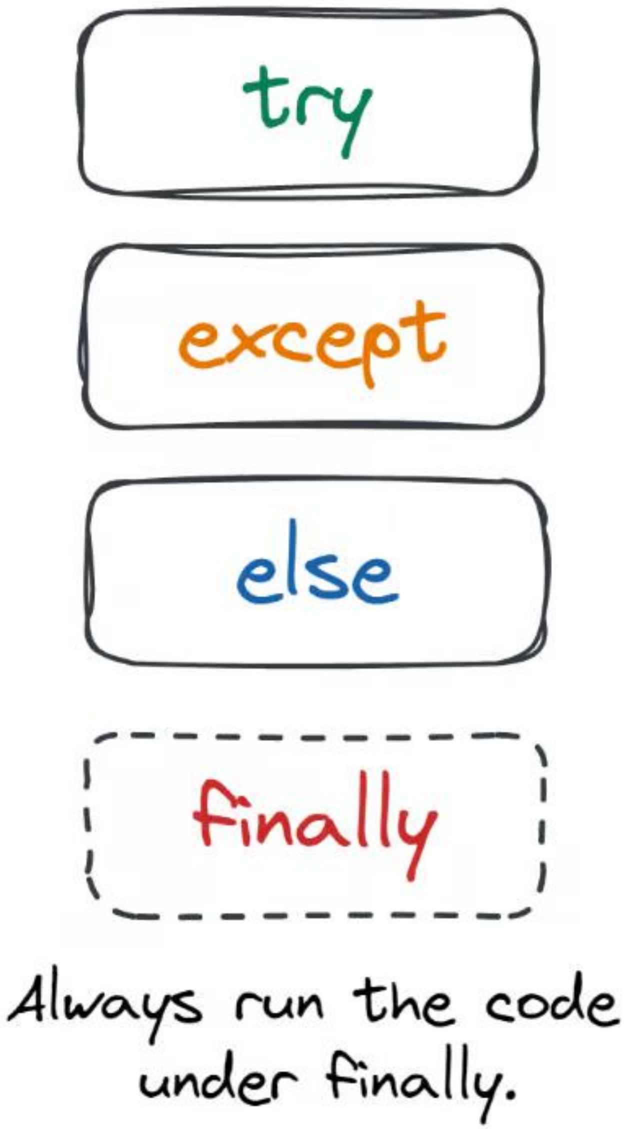
```python
x = [5,8,9,13]

def find_nth_value(x,n):
    try:
        result = x[n]
    except IndexError as err:
        print(err)
    else:
        print("Your answer is ", result)

# Testing
find_nth_value(x,6)
find_nth_value(x,2)
```


Output

```
list index out of range
Your answer is  9


=== Code Execution Successful ===
```

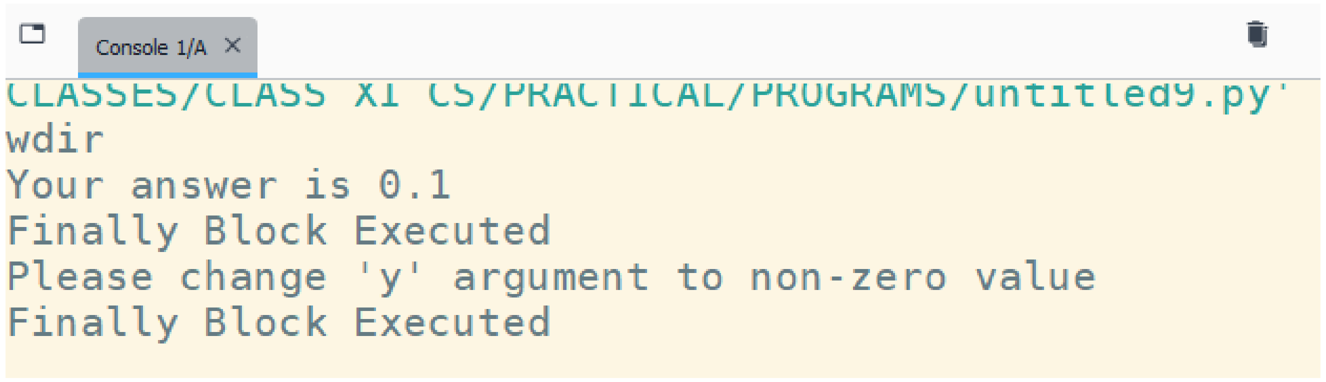
The finally keyword in Python

- The finally keyword in the try-except block is always executed, irrespective of whether there is an exception or not.
- In simple words, the finally block of code is run after the try, except and else block is final.
- It is quite useful in cleaning up resources and closing the object, especially closing the files.



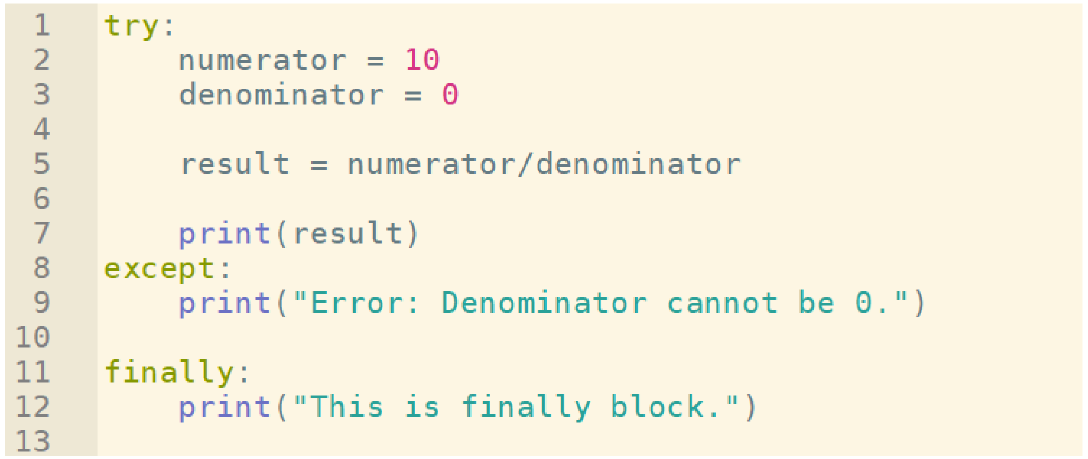
Example

```python
def divide(x,y):
    try:
        result = x/y
    except ZeroDivisionError:
        print("Please change 'y' argument to non-zero value")
    except:
        print("Something went wrong")
    else:
        print(f"Your answer is {result}")
    finally:
        print("Finally Block Executed")

divide(1,10)
divide(24,0)
```

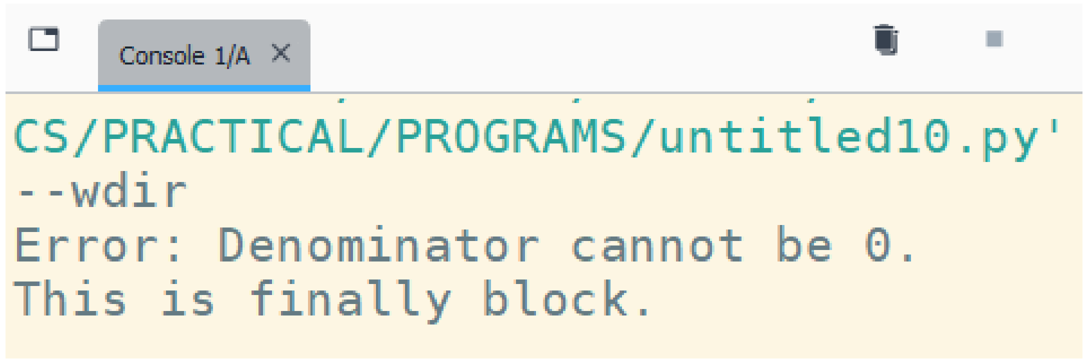


```
CLASSES/CLASS XI CS/PRACTICAL/PROGRAMS/untitled9.py'
wdir
Your answer is 0.1
Finally Block Executed
Please change 'y' argument to non-zero value
Finally Block Executed
```

Example



```python
try:
    numerator = 10
    denominator = 0

    result = numerator/denominator

    print(result)
except:
    print("Error: Denominator cannot be 0.")

finally:
    print("This is finally block.")

```



```
CS/PRACTICAL/PROGRAMS/untitled10.py'
--wdir
Error: Denominator cannot be 0.
This is finally block.
```