Exception Handling

What is an Exception?

- The term exception is shorthand for the phrase "exceptional event."
- An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

Errors

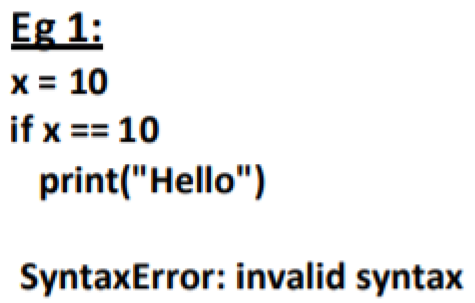
- When executing Python code, different errors may occur.
  - Coding errors made by the programmer(Syntax Errors), errors due to wrong input, or other unforeseeable things (Runtime Errors) like a user entering an invalid input to the program, or an error occurring while trying to establish connection between local machine and remote machine in distributed application.

- When an error occurs, Python’s default exception-handling behaviour kicks in: it stops the program and prints an error message. In other words, the program terminates abruptly.

- If you don’t want this default behaviour, you need to handle these exceptions.

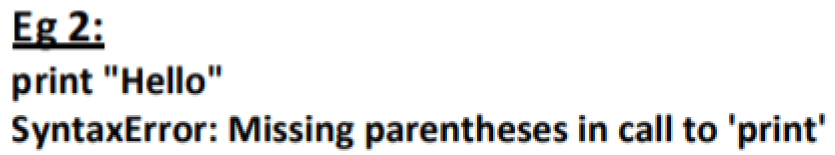
Syntax Errors:

- The errors which occur because of invalid syntax are called syntax errors.



Eg 1:

```
x = 10
if x == 10
  print("Hello")

 SyntaxError: invalid syntax
```



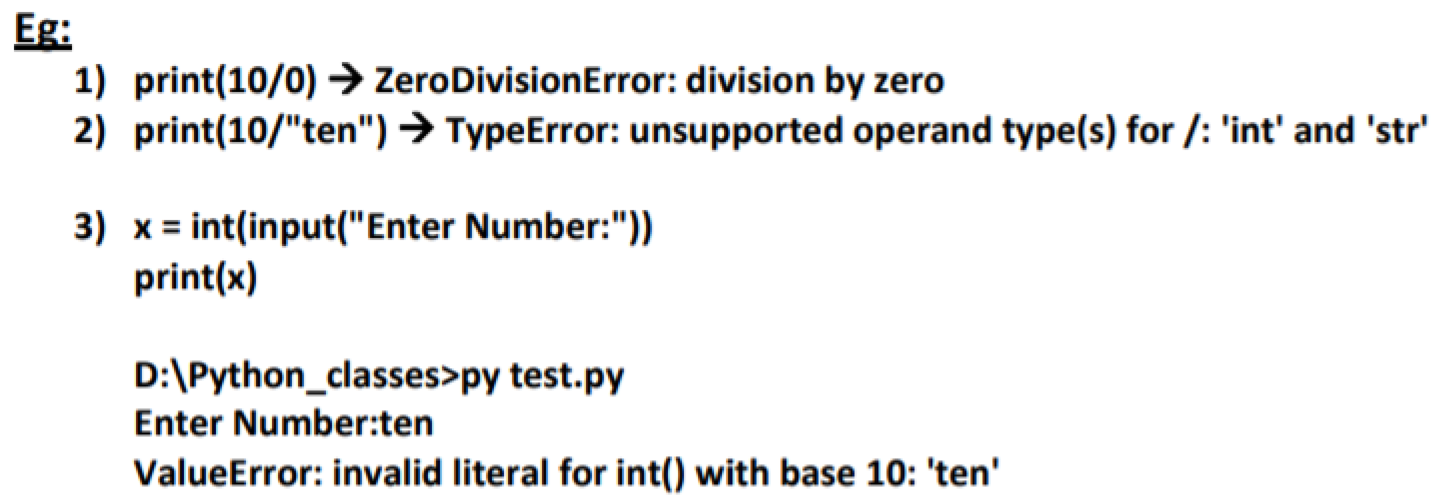
Eg 2:

```
print "Hello"
SyntaxError: Missing parentheses in call to 'print'
```

Note: Programmer is responsible to correct these syntax errors. Once all syntax errors are corrected the program execution will proceed.

2. Runtime Errors:

- Also known as exceptions.
- While executing the program if something goes wrong because of end user input or programming logic or memory problems etc then we get Runtime Errors.



Eg:

1) `print(10/0)` → ZeroDivisionError: division by zero
2) `print(10/"ten")` → TypeError: unsupported operand type(s) for /: 'int' and 'str'

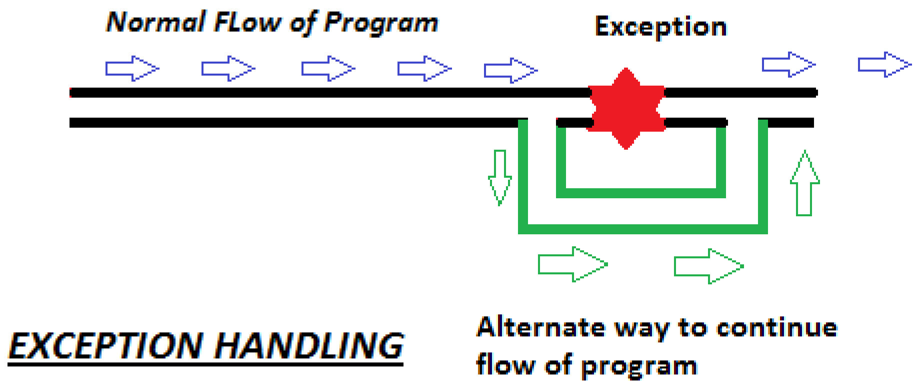
3)
```
x = int(input("Enter Number:"))
print(x)

D:\Python_classes>py test.py
Enter Number:ten
ValueError: invalid literal for int() with base 10: 'ten'
```

Purpose of exception handling

- The main **purpose** is to prevent abnormal termination of the program and to customise the **exception** message.

- Imagine that we order a product online, but while en-route, there's a failure in delivery. A good company can handle this problem and gracefully re-route our package so that it still arrives on time.

- Likewise, in Python, the code can experience errors while executing our instructions. Good *exception handling* can handle errors and gracefully re-route the program to give the user still a positive experience.



Default Exception Handing in Python

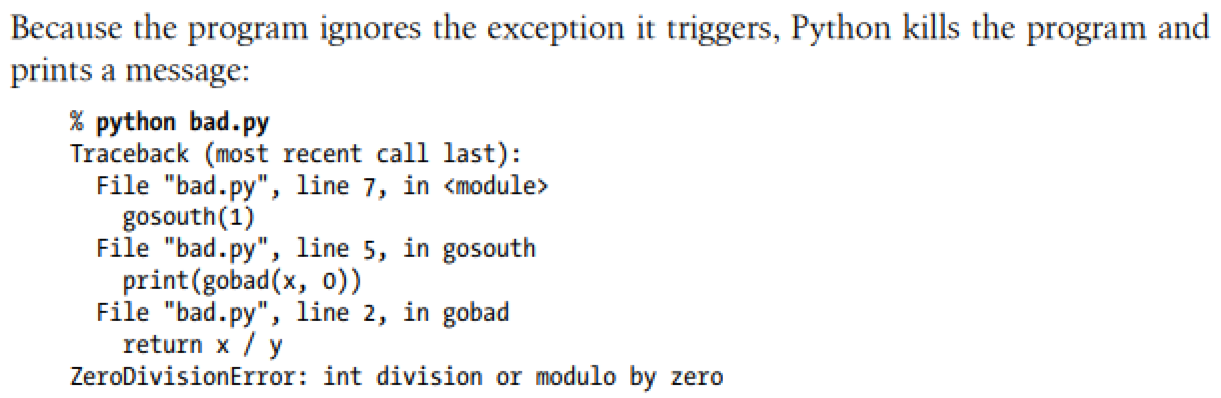
- Every exception in Python is an object. For every exception object the corresponding classes are available.
- Whenever an exception occurs, the corresponding exception object will be thrown, and the Python Virtual Machine(PVM) will check for handling code. If the handling code is not available, then Python interpreter terminates the program abnormally and prints corresponding exception information to the console.

Default Exception Handing in Python

- Exceptions not caught percolate up to the top level of the Python process and run Python's default exception-handling logic (i.e., Python terminates the running program and prints a standard error message).

- Let's look at an example. Running the following module file, bad.py, generates a divide-by-zero exception

```
def gobad(x, y):
          return x / y

def gosouth(x):
          print(gobad(x, 0))

gosouth(1)
```



Because the program ignores the exception it triggers, Python kills the program and prints a message:

```
% python bad.py
Traceback (most recent call last):
  File "bad.py", line 7, in <module>
    gosouth(1)
  File "bad.py", line 5, in gosouth
    print(gobad(x, 0))
  File "bad.py", line 2, in gobad
    return x / y
ZeroDivisionError: int division or modulo by zero
```

Default Exception Handing in Python

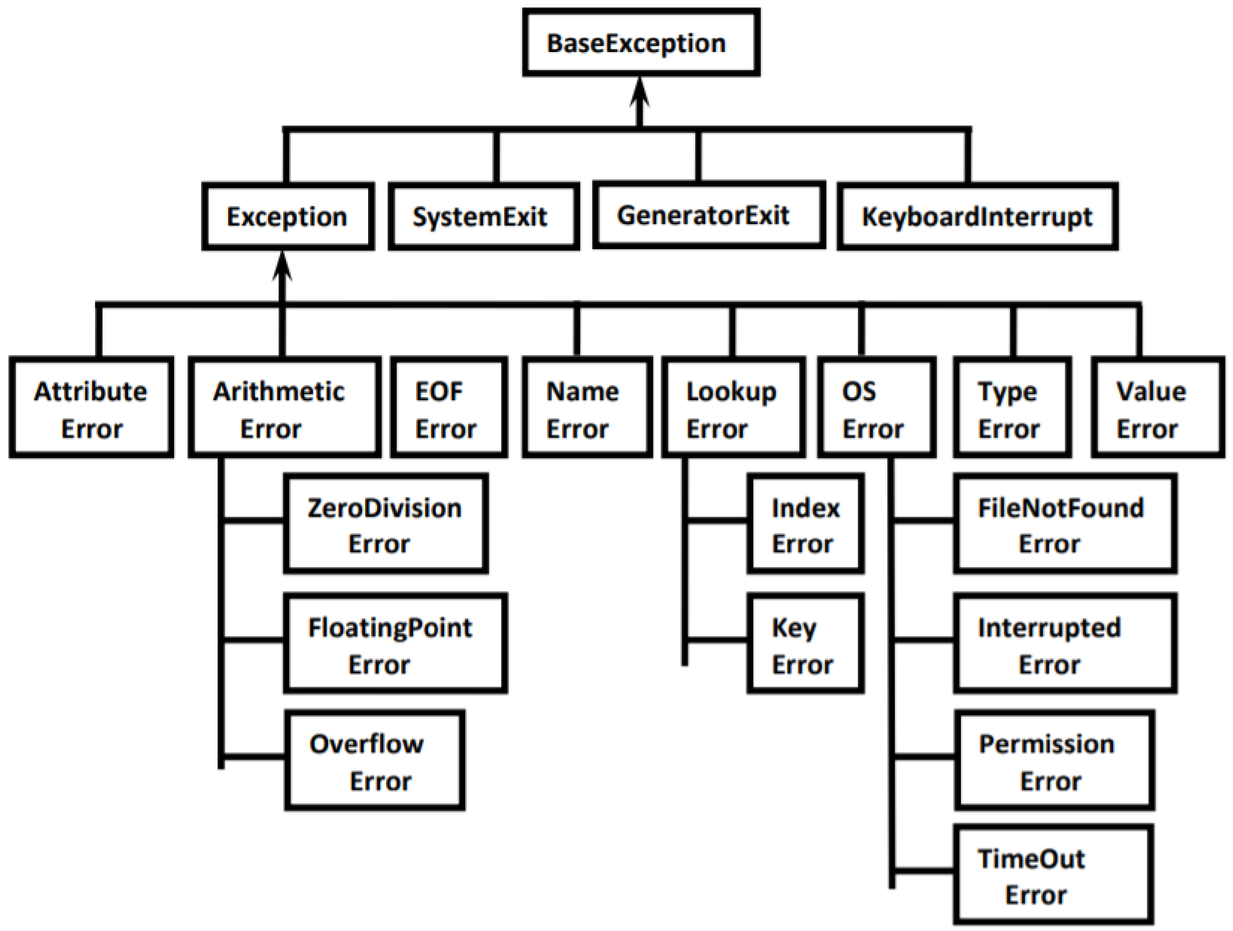
- The message consists of a stack trace (“Traceback”) and the name of and details about the exception that was raised. The stack trace lists all lines active when the exception occurred, from oldest to newest.

- The file and line number information is more useful. For example, here we can see that the bad divide happens at the last entry in the trace—line 2 of the file bad.py, a return statement.

Exception Hierarchy

- Every Exception in Python is a class.
- All exception classes are child classes of BaseException .i.e., every exception class extends **BaseException** either directly or indirectly.
- The Exception class contains many direct child subclasses that handle most Python errors.
- Hence BaseException acts as root for Python Exception Hierarchy.

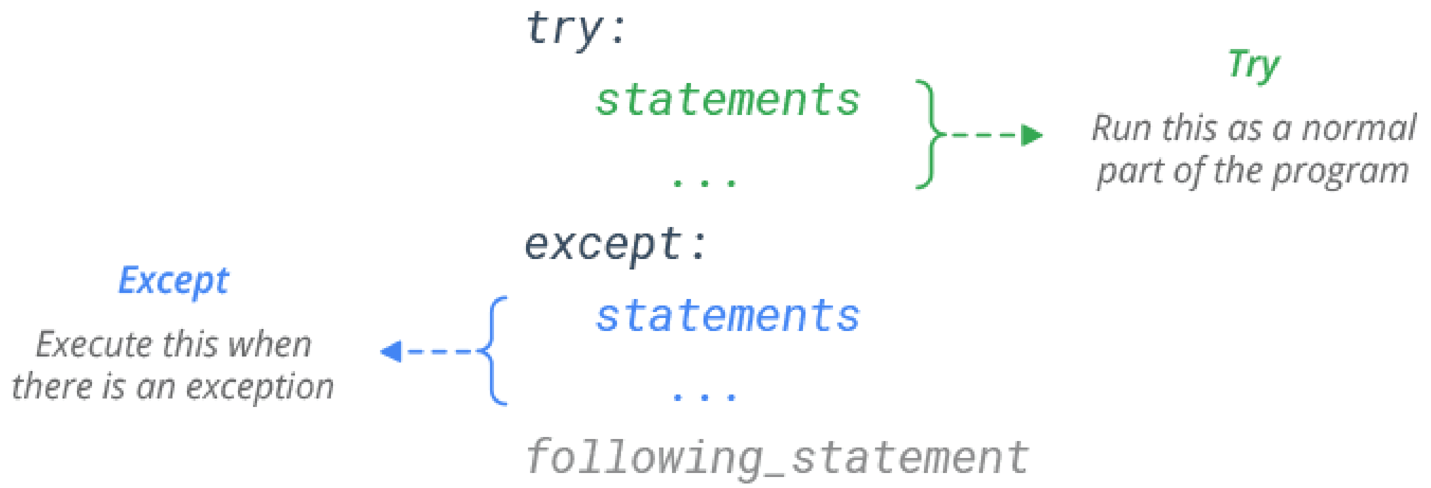
Exception Hierarchy



Customized Exception Handling by using try-except

- In Python, exceptions can be handled using a `try-except` statement.
- When a runtime error is raised, it can be caught or uncaught: “try” lets you catch a raised error, and “except” lets you do something about it.
- The critical operation which can raise an exception is placed inside the try clause. The code that handles the exceptions is written in the except clause.

try:
    Risky Code
except XXX:
    Handling code/Alternative Code



```
try:
    statements
        ...
except:
    statements
        ...
following_statement
```

Try
Run this as a normal part of the program
Except
Execute this when there is an exception

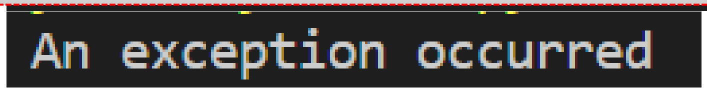
Example

- The try block will generate an exception, because x is not defined.

- Since the try block raises an error, the except block will be executed.

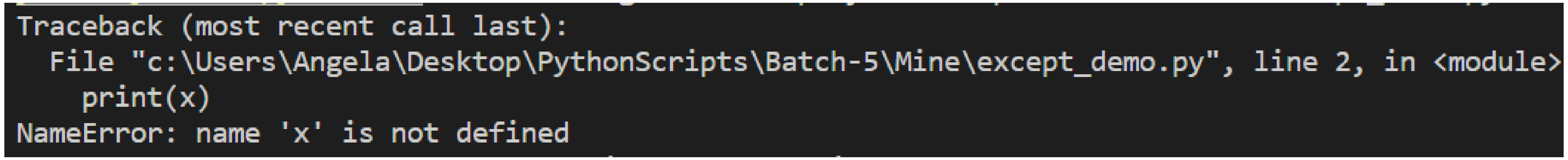
- Without the try block, the program will crash and raise an error.

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

print(" more code")
```



```
An exception occurred
```

```
#try:
print(x)
#except:
#print("An exception occurred")
```

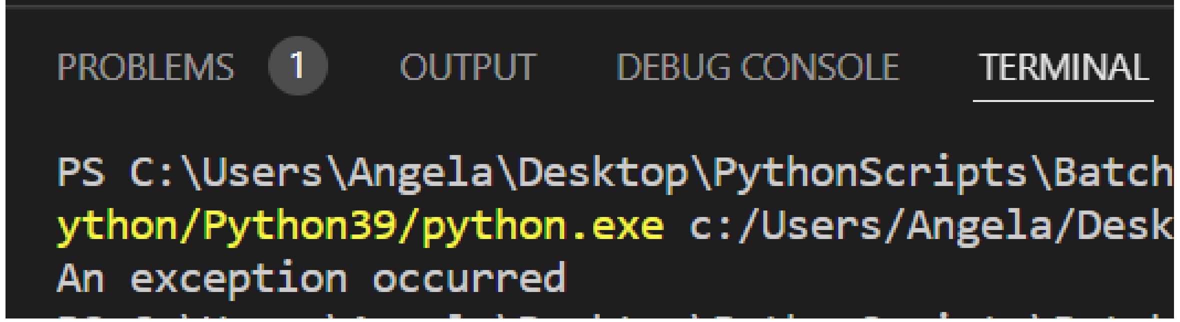


```
Traceback (most recent call last):
  File "c:\Users\Angela\Desktop\PythonScripts\Batch-5\Mine\except_demo.py", line 2, in <module>
    print(x)
NameError: name 'x' is not defined
```

Catching Specific Exceptions in Python

- In the above example, we did not mention any specific exception in the except clause.
- This is not a good programming practice as it will catch all exceptions and handle every case in the same way.
- We can specify which exceptions an except clause should catch.
- If the type of exception doesn't match the except blocks, it will remain un-handled and the program will terminate.

```
try:
        print(x)
except NameError:
        print("An exception occurred")
```



```
PS C:\Users\Angela\Desktop\PythonScripts\Batch
ython/Python39/python.exe c:/Users/Angela/Desk
An exception occurred
```

Customized Exception Handling by using try-except

- Without try/except:

```
print("stmt-1")
print(10/0)
print("stmt-3")
```

Output

```
stmt-1
ZeroDivisionError: division by zero
Abnormal termination/Non-Graceful Termination
```

- With try-except:

```
print("stmt-1")
try:
        print(10/0)
except ZeroDivisionError:
        print(10/2)
print("stmt-3")
```

Output

```
stmt-1
5.0
stmt-3
Normal termination/Graceful Termination
```

Control flow in try-except

Assume you have the following code

- Let’s investigate some possible cases to analyze the control flow in try – except

```
try:
            stmt-1
            stmt-2
            stmt-3
 except XXX:
        stmt-4
 stmt-5
```

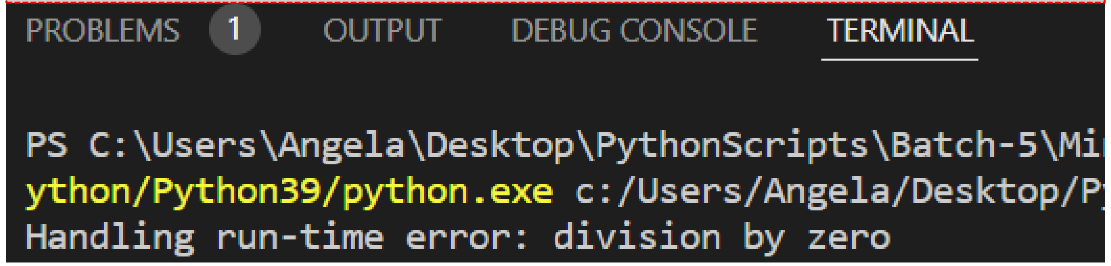
- Case-1: If there is no exception
  - 1,2,3,5 and Normal Termination
- Case-2: If an exception was raised at stmt-2 and corresponding except block matched
  - 1,4,5 Normal Termination
- Case-3: If an exception was raised at stmt-2 and corresponding except block not matched
  - 1, Abnormal Termination
- Case-4: If an exception was raised at stmt-4 or at stmt-5
  - always abnormal termination.

Control –flow in try…except

- Exception handlers don’t just handle exceptions if they occur immediately in the try clause, but also if they occur inside functions that are called (even indirectly) in the try clause.

- In this example, err is an object of type ZeroDivisionError. It stores the exception object. err is an instance of type ZeroDivisionError Exception class.

```
# Handling run-time error
def this_fails():
        x = 1/0

try:
        this_fails()
except ZeroDivisionError as err:
        print('Handling run-time error:', err)
```



```
PS C:\Users\Angela\Desktop\PythonScripts\Batch-5\Mi
ython/Python39/python.exe c:/Users/Angela/Desktop/P
Handling run-time error: division by zero
```

Conclusion

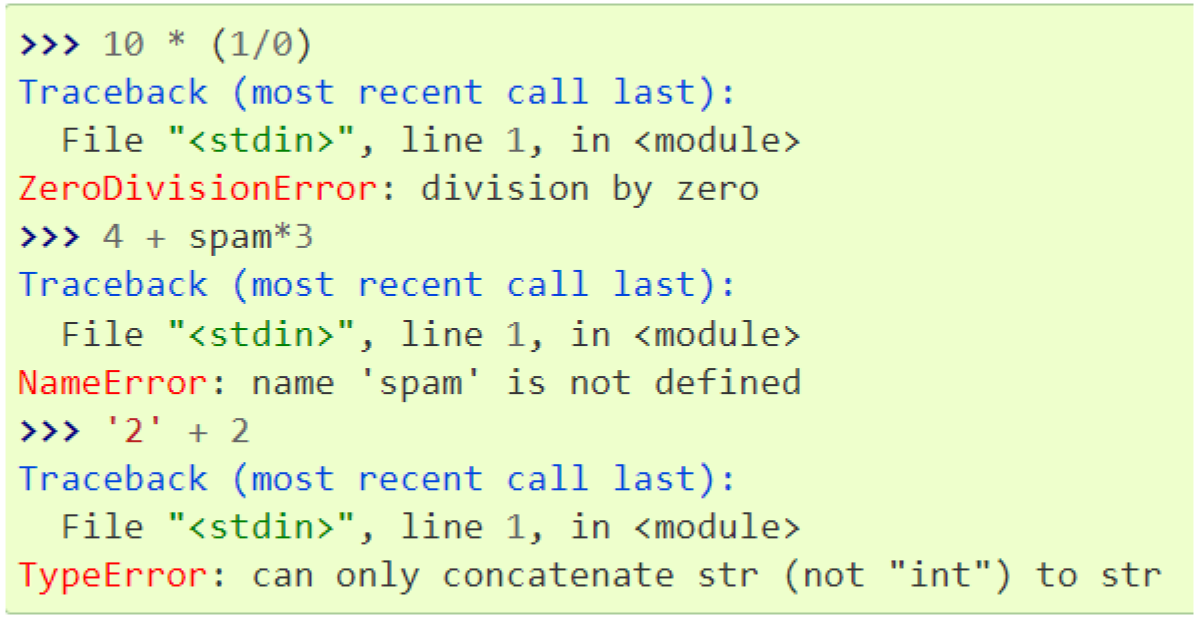
1) If an exception is raised anywhere within the try block, then rest of the try block won’t be executed, even though we handled that exception. Hence, we must take only risky code inside try block and length of the try block should be as less as possible.

2) In addition to try block, there may be a chance of raising exceptions inside except and finally blocks also.

3) If any statement which is not part of try block raises an exception, then it is always abnormal termination

Exception Messages

- Most exceptions that are not handled by programs, result in error messages as shown here:



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

```
10 *(1/0)

4+ spam*3

'2'  + 2
```

Exception Messages

- The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message.

- The string printed as the exception type(ZeroDivisionError, NameError and TypeError) is the name of the built-in exception that occurred.

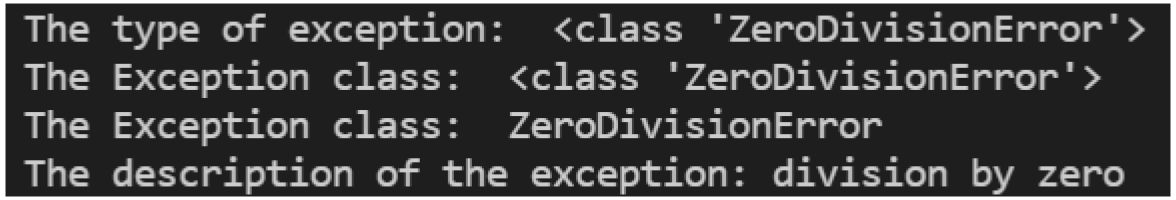
- The rest of the line provides detail based on the type of exception and what caused it.

- The preceding part of the error message shows the context where the exception occurred, in the form of a stack traceback. In general, it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

Printing the exception message

- Exception messages can be captured and printed in different ways as shown the following example

```
try:
        a = 1/0

except Exception as e:
        print("The type of exception class: ", type(e))
        print("The Exception class: ",e.__class__)
        print("The Exception class name: ",e.__class__.__name__)
        print("The description of the exception:",e)
```

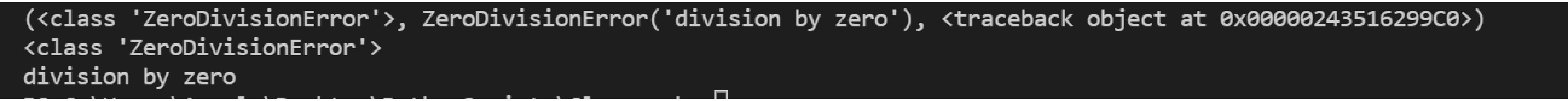


```
The type of exception:  <class 'ZeroDivisionError'>
The Exception class:  <class 'ZeroDivisionError'>
The Exception class:  ZeroDivisionError
The description of the exception: division by zero
```

Capture and print Python exception message

- Another way is to import the sys module and use the sys.exc_info() method to capture and print the exception message.

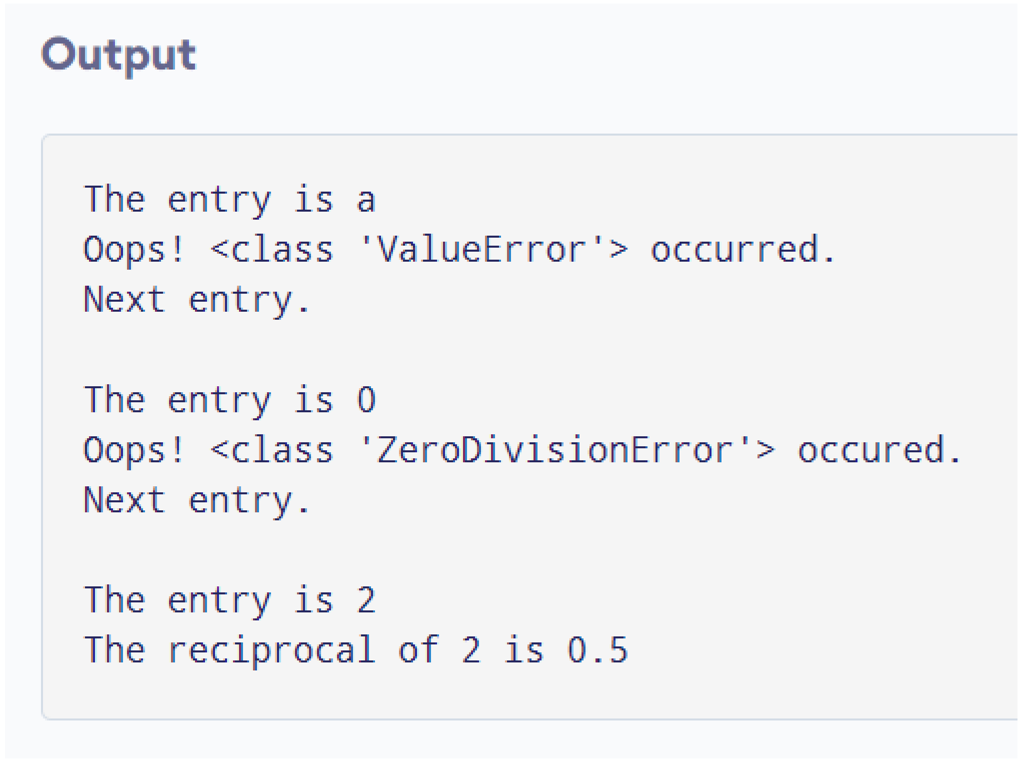
```
import sys
try:
        a = 1/0
except Exception as e:
        print(sys.exc_info())
        print(sys.exc_info()[0])
        print(sys.exc_info()[1])
```



```
(<class 'ZeroDivisionError'>, ZeroDivisionError('division by zero'), <traceback object at 0x0000024351629 9C0>)
<class 'ZeroDivisionError'>
division by zero
```

Example

```
# import module sys to get the type of exception

import sys
randomList = ['a', 0, 2]

for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except:
        print("Oops!", sys.exc_info()[0], "occurred.")
        print("Next entry.")
        print()

print("The reciprocal of", entry, "is", r)
```

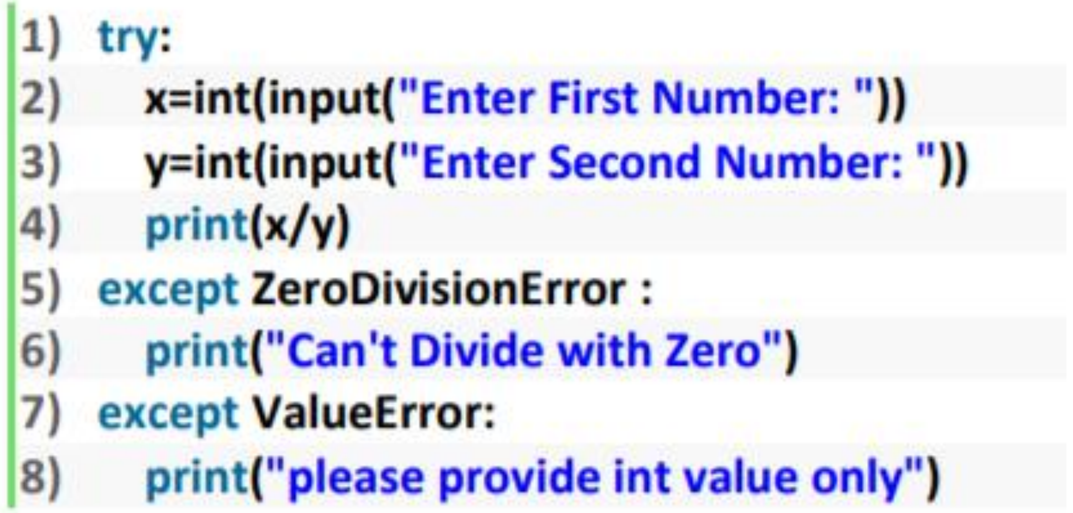


Output

```
The entry is a
Oops! <class 'ValueError'> occurred.
Next entry.

The entry is 0
Oops! <class 'ZeroDivisionError'> occured.
Next entry.

The entry is 2
The reciprocal of 2 is 0.5
```

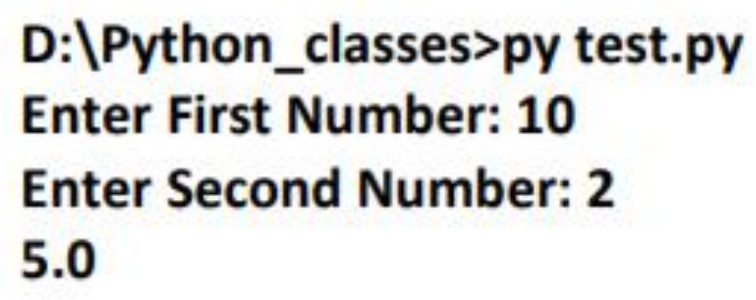
try with multiple except blocks

- You can define as many except blocks as you want, to catch and handle specific exceptions.
- Each except block can have the name of the exception class that it can handle.

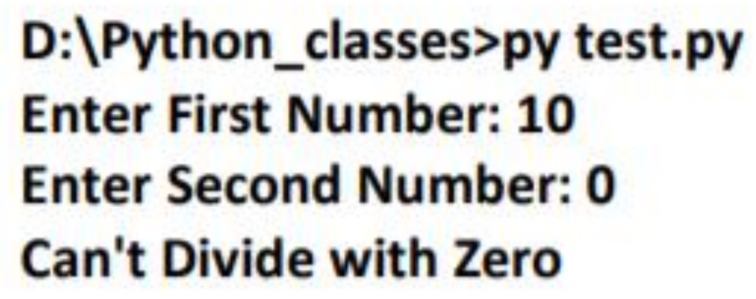


```
try:
    x=int(input("Enter First Number: "))
    y=int(input("Enter Second Number: "))
    print(x/y)
except ZeroDivisionError :
    print("Can't Divide with Zero")
except ValueError:
    print("please provide int value only")
```

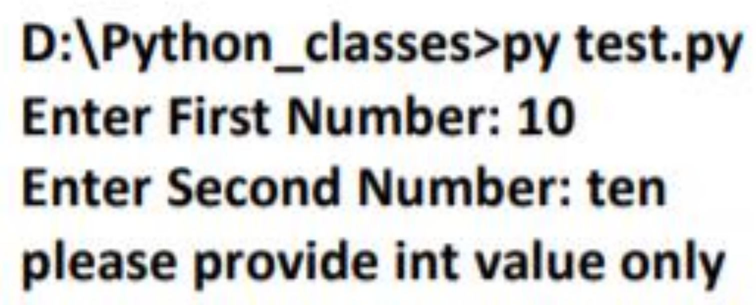
```
try:
        x=int(input("Enter First Number: "))
        y=int(input("Enter Second Number: "))
        print(x/y)
except ZeroDivisionError :
        print("Can't Divide with Zero")
except ValueError:
        print("Please provide int value only")
```



```
D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: 2
5.0
```



```
D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: 0
Can't Divide with Zero
```



```
D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: ten
please provide int value only
```

Default except Block or the Catch-All Exception Handler

- We can use default except block to handle any type of exceptions.
- In default except block generally we can print normal error messages.

Syntax:

```
try:
        x=int(input(" Enter First Number: "))
        y=int(input("Enter Second Number. "))
        print(x/y)
except ZeroDivisionError :
        print(" ZeroDivisionError :Can't divide with zero")
except:
        print("Default Except:Plz provide valis input only")
```

```
D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: 0
ZeroDivisionError:Can't divide with zero

D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: ten
Default Except:Plz provide valid input only
```

Catching multiple Exceptions in one except block

- We can use a tuple of values to specify multiple exceptions in an except clause. Here is an example pseudo code.

```
try:
        # do something
        pass
except ValueError:
        # handle ValueError exception
        pass

except (TypeError, ZeroDivisionError):
        # handle multiple exceptions
        # TypeError and ZeroDivisionError
        pass
except:
        # handle all other exceptions
        pass
```

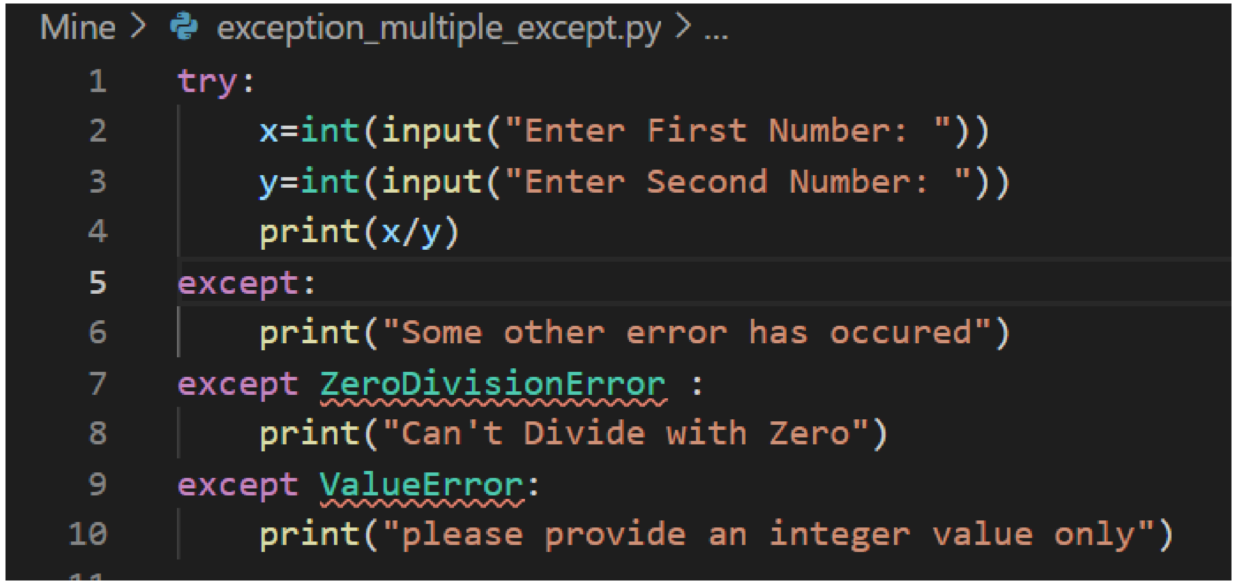
Catch Multiple Exceptions

- Example

```
# Execute same block of code for multiple exceptions
try:
    x = 1/0
except (ZeroDivisionError, ValueError):
    print('ZeroDivisionError or ValueError is raised')
except:
    print('Something else went wrong')

# Prints ZeroDivisionError or ValueError is raised
```

try with Multiple except Blocks

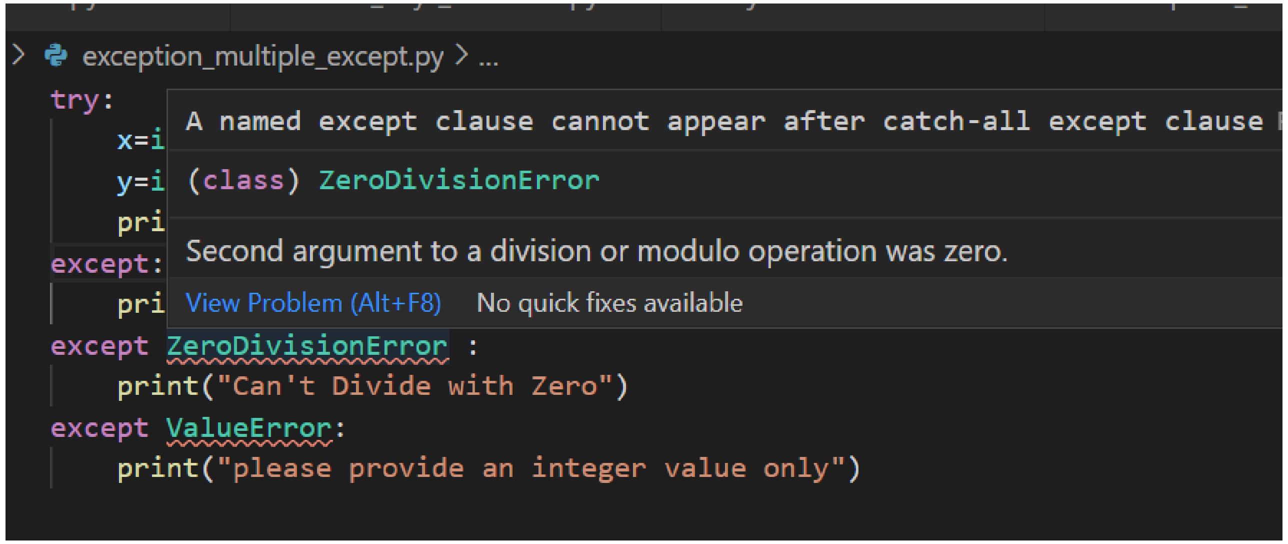
- If **try** with multiple except blocks is used, then the order of these except blocks is important.
- Python interpreter will always consider from top to bottom until matched except block identified.
- In a try with multiple except blocks the default except block should be last, else we’ll get a SyntaxError, as there’s no use of placing the specific except blocks at the end, as all the exceptions will be caught by the default/catch-all except clause.



Mine > exception_multiple_except.py > ...

```python
try:
    x=int(input("Enter First Number: "))
    y=int(input("Enter Second Number: "))
    print(x/y)
except:
    print("Some other error has occured")
except ZeroDivisionError :
    print("Can't Divide with Zero")
except ValueError:
    print("please provide an integer value only")
```

```
try:
        x=int(input("Enter First Number: "))
        y=int(input("Enter Second Number: "))
        print(x/y)
except:
        print("Some other error has occured")
except ZeroDivisionError:
        print("Can't Divide with Zero")
except ValueError:
        print("please provide an integer value only")
```



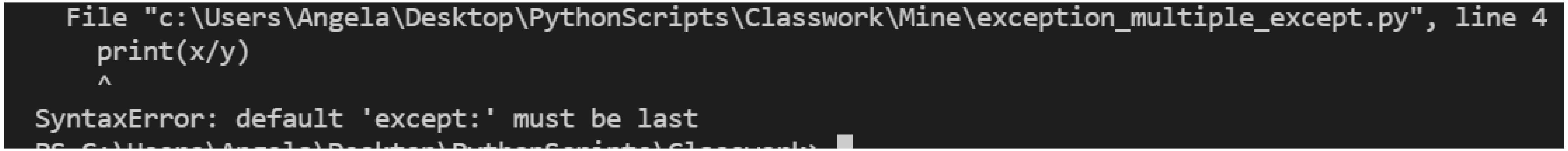
> exception_multiple_except.py > ...

```
try:
    x=i
    y=i
    pri
except:
    pri
except ZeroDivisionError :
    print("Can't Divide with Zero")
except ValueError:
    print("please provide an integer value only")
```

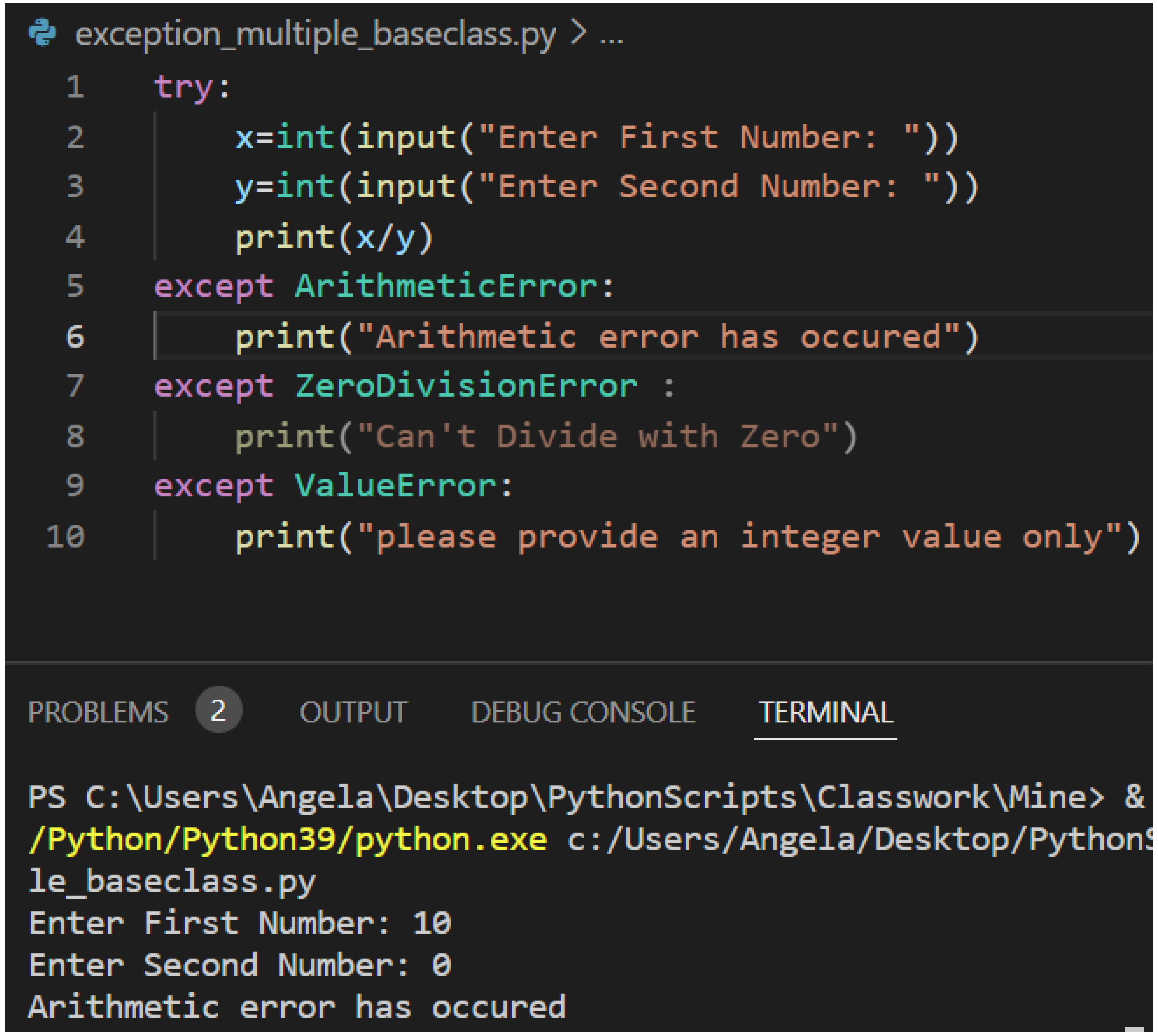
```
A named except clause cannot appear after catch-all except clause
(class) ZeroDivisionError
```

Second argument to a division or modulo operation was zero.

View Problem (Alt+F8)   No quick fixes available



```
  File "c:\Users\Angela\Desktop\PythonScripts\Classwork\Mine\exception_multiple_except.py", line 4
    print(x/y)
    ^
SyntaxError: default 'except:' must be last
PS C:\Users\Angela\Desktop\PythonScripts\Classwork>
```



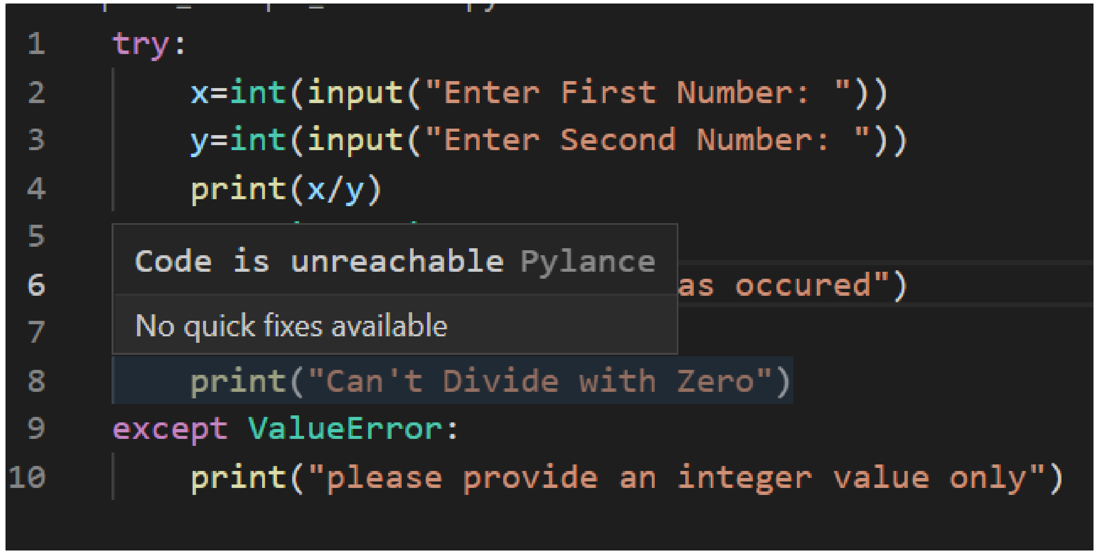
exception_multiple_baseclass.py > ...

```python
try:
    x=int(input("Enter First Number: "))
    y=int(input("Enter Second Number: "))
    print(x/y)
except ArithmeticError:
    print("Arithmetic error has occured")
except ZeroDivisionError :
    print("Can't Divide with Zero")
except ValueError:
    print("please provide an integer value only")
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\Angela\Desktop\PythonScripts\Classwork\Mine> &
/Python/Python39/python.exe c:/Users/Angela/Desktop/PythonS
le_baseclass.py
Enter First Number: 10
Enter Second Number: 0
Arithmetic error has occured
```

```
try:
        x=int(input("Enter First Number: "))
        y=int(input("Enter Second Number: "))
        print(x/y)
except ArithmeticError:
        print(" Arithmetic Error has occured")
except ZeroDivisionError:
        print("Can't Divide with Zero")
except ValuesError:
        print("please provide an integer value only")
```



```python
try:
    x=int(input("Enter First Number: "))
    y=int(input("Enter Second Number: "))
    print(x/y)
```

Code is unreachable Pylance

No quick fixes available

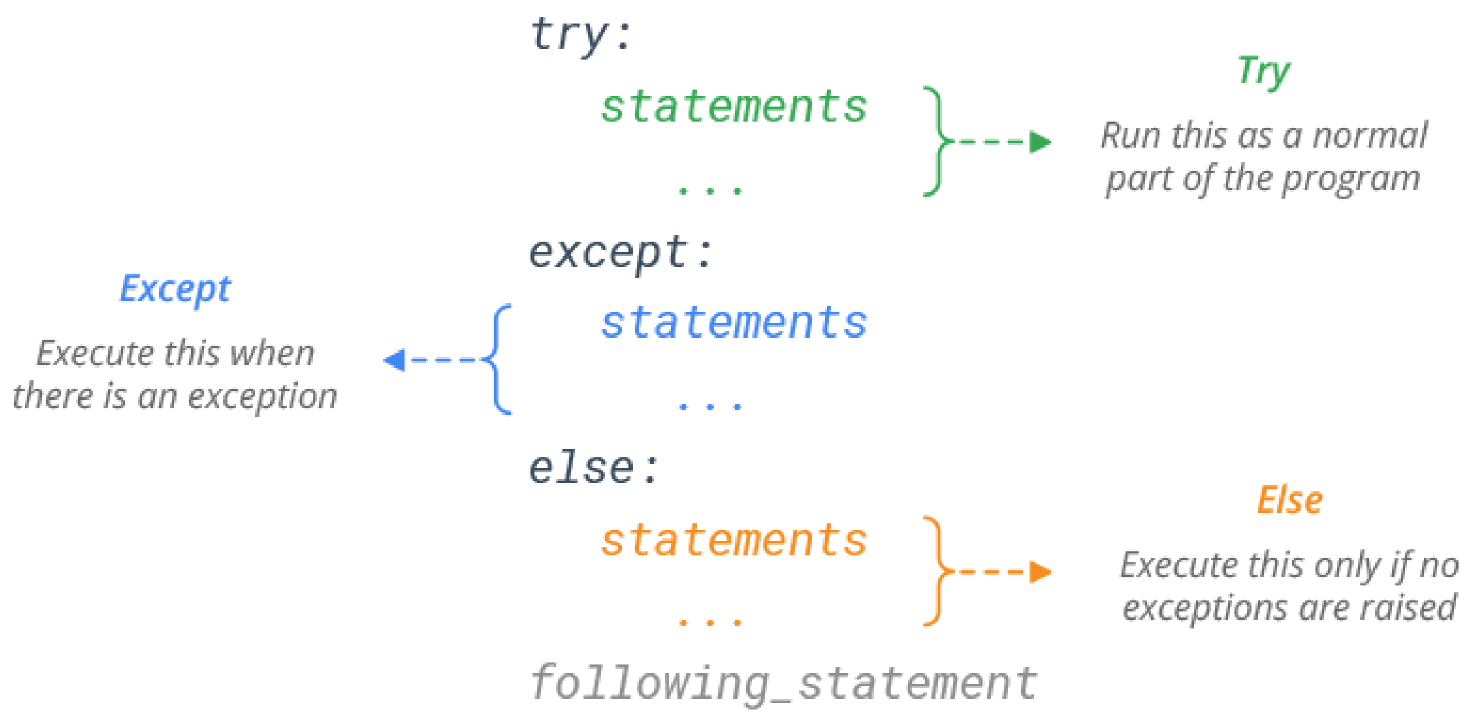
```python
                                    as occured")

    print("Can't Divide with Zero")
except ValueError:
    print("please provide an integer value only")
```

Python try with else clause

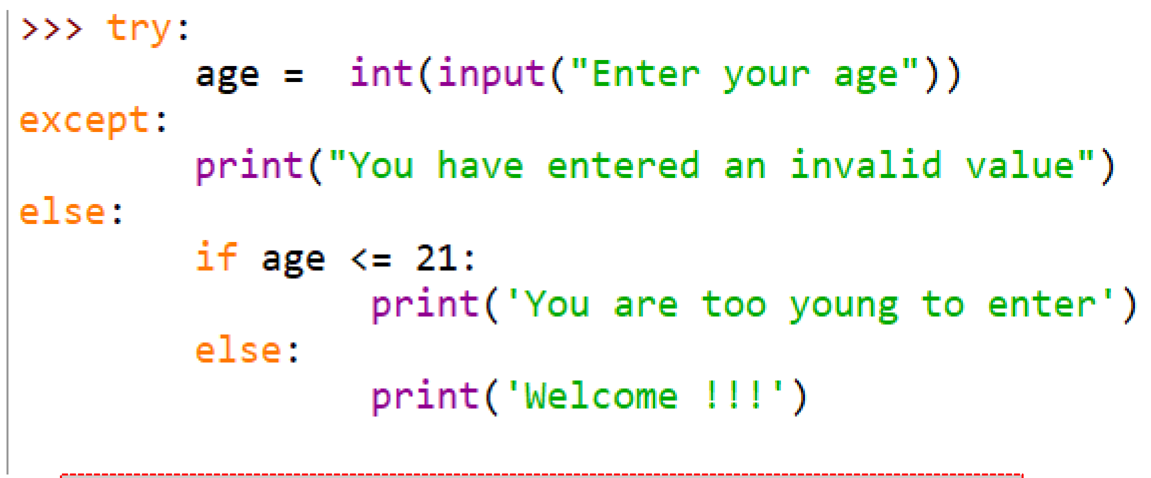
- You can include an **else** clause when catching exceptions with a **try** statement.

- The statements inside the **else** block will be executed only if the code inside the **try** block doesn't generate an exception.

- **Note:** Exceptions in the else clause are not handled by the preceding except clauses.

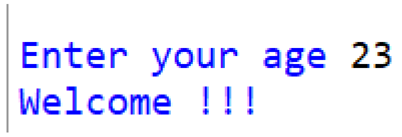

try:
statements
...
Try
Run this as a normal part of the program
except:
statements
...
Except
Execute this when there is an exception
else:
statements
...
Else
Execute this only if no exceptions are raised
following_statement

Example

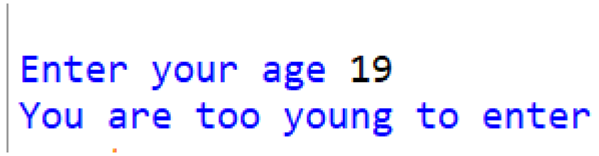


```
>>> try:
        age =  int(input("Enter your age"))
except:
        print("You have entered an invalid value")
else:
        if age <= 21:
                print('You are too young to enter')
        else:
                print('Welcome !!!')
```

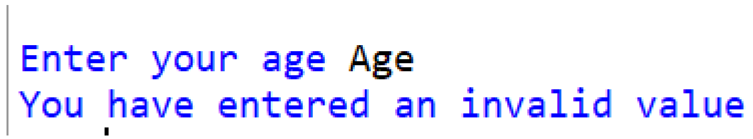
```
try:
        age = int(input("Enter your age :"))
except:
        print("You have entered an invalid value")
else:
         if age <= 21:
                print('You are too young to enter')
        else:
                print("welcome")
```



```
Enter your age 23
Welcome !!!
```



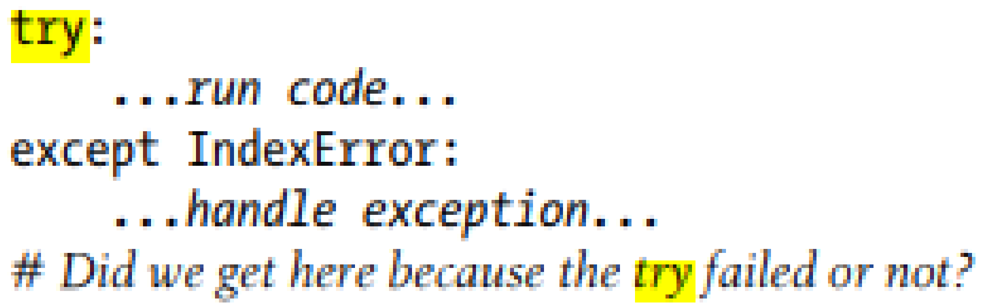
```
Enter your age 19
You are too young to enter
```



```
Enter your age Age
You have entered an invalid value
```

Purpose of try…else clause

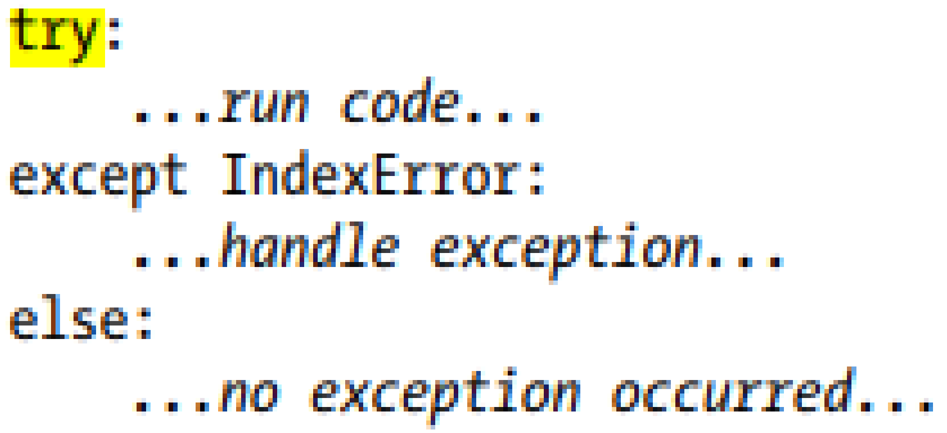
- The purpose of the else clause is not always immediately obvious.
- Without it, though, there is no way to tell (without setting and checking Boolean flags) whether the flow of control has proceeded past a try statement because no exception was raised, or because an exception occurred and was handled:



```
try:
    ...run code...
except IndexError:
    ...handle exception...
# Did we get here because the try failed or not?
```

Purpose of try…else clause

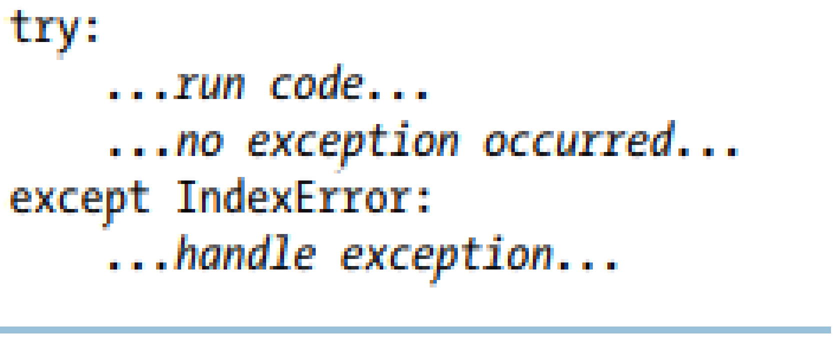
- Much like the way else clauses in loops make the exit cause more apparent, the else clause provides syntax in a try that makes what has happened obvious and unambiguous:



```
try:
    ...run code...
except IndexError:
    ...handle exception...
else:
    ...no exception occurred...
```

Purpose of try…else clause

- You can almost emulate an else clause by moving its code into the try block. This can lead to incorrect exception classifications, though.



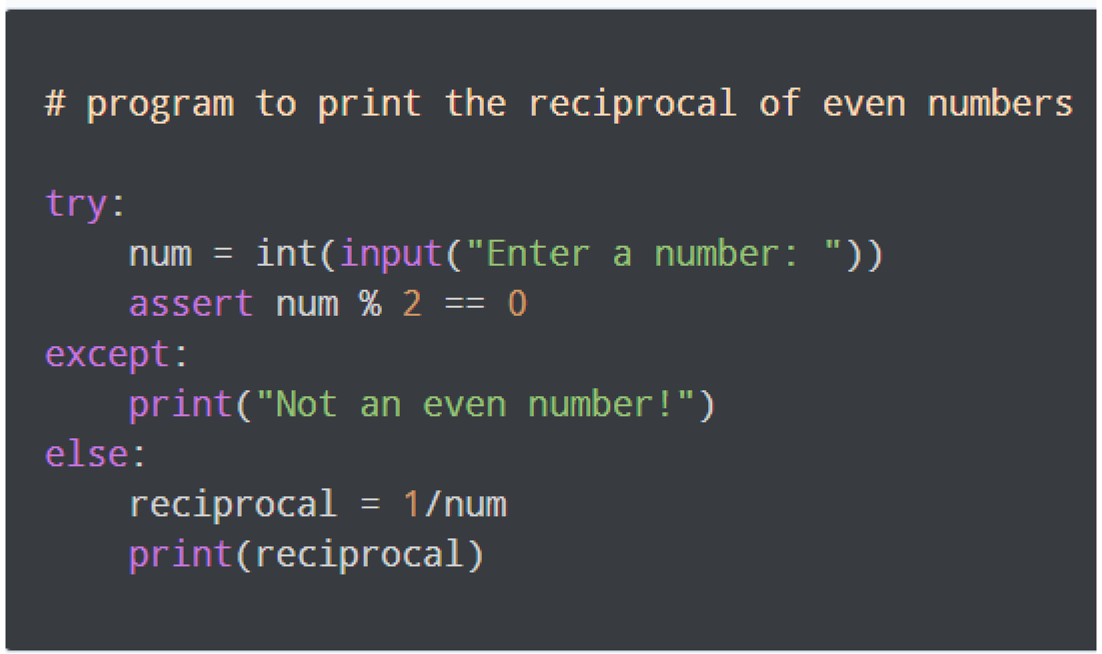
```
try:
    ...run code...
    ...no exception occurred...
except IndexError:
    ...handle exception...
```

If the “no exception occurred” action triggers an IndexError, it will register as a failure of the try block and erroneously trigger the exception handler below the try (subtle, but true!).

By using an explicit else clause instead, you make the logic more obvious and guarantee that except handlers will run only for real failures in the code you’re wrapping in a try, not for failures in the else case’s action.

try…else - Example

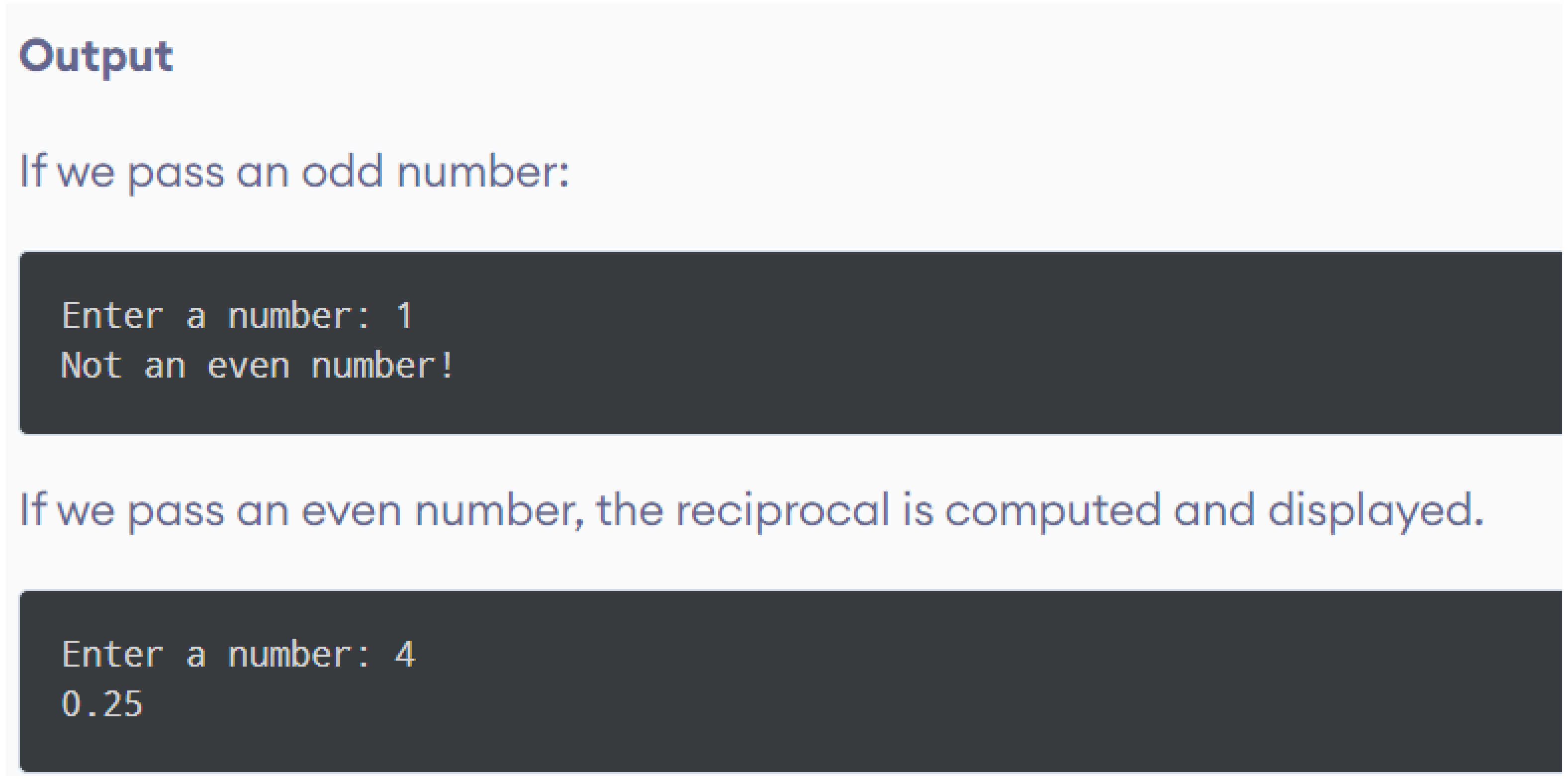
- Exceptions in the else clause are not handled by the preceding except clauses.



```python
# program to print the reciprocal of even numbers

try:
    num = int(input("Enter a number: "))
    assert num % 2 == 0
except:
    print("Not an even number!")
else:
    reciprocal = 1/num
    print(reciprocal)
```

```
# program to print the reciprocal of even numbers

try:
        num = int(input("Enter a number: "))
        assert num % 2 == 0
except:
        print("Not an even number!")
else:
        reciprocal = 1/num
        print( reciprocal)
```

try...else - Example



Output

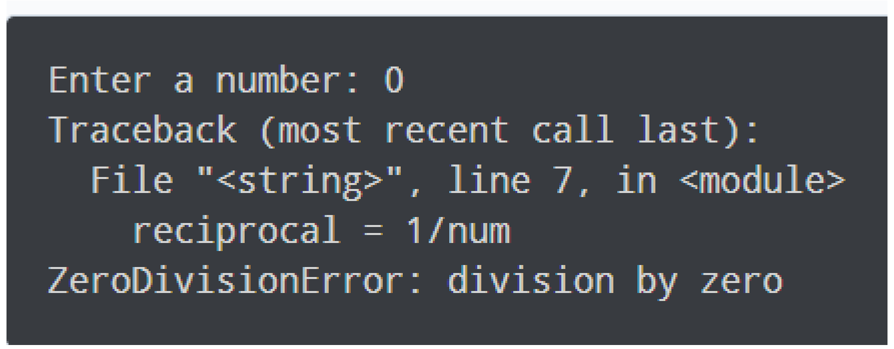
If we pass an odd number:

```
Enter a number: 1
Not an even number!
```

If we pass an even number, the reciprocal is computed and displayed.

```
Enter a number: 4
0.25
```

However, if we pass 0, we get ZeroDivisionError as the code block inside else is not handled by preceding except.



```
Enter a number: 0
Traceback (most recent call last):
  File "<string>", line 7, in <module>
    reciprocal = 1/num
ZeroDivisionError: division by zero
```

Finally: for the things you want to do *no matter what.*

If you try to cook something, you start by turning on the oven.

If the thing you try is a complete **failure**,
you have to turn off the oven.

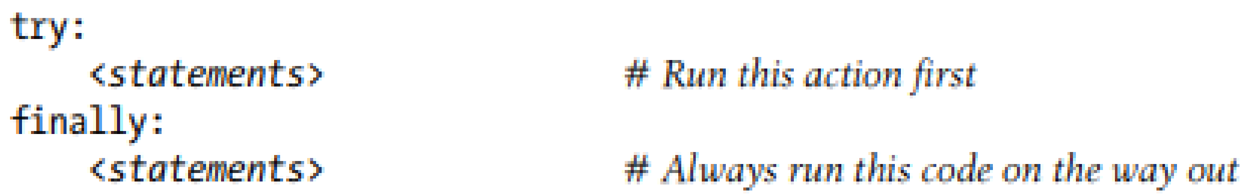
If the thing you try **succeeds**,
you have to turn off the oven.

You have to turn off the oven no matter what!

Are you sure you want to try this?
No matter what, do NOT let me forget to turn off the oven! Last time I torched half the neighborhood.

try---finally clause

- If a finally clause is included in a **try**, Python will always run its block of statements “on the way out” of the try statement, whether an exception occurred while the try block was running or not. Its general form is



```
try:
    <statements>                    # Run this action first
finally:
    <statements>                    # Always run this code on the way out
```

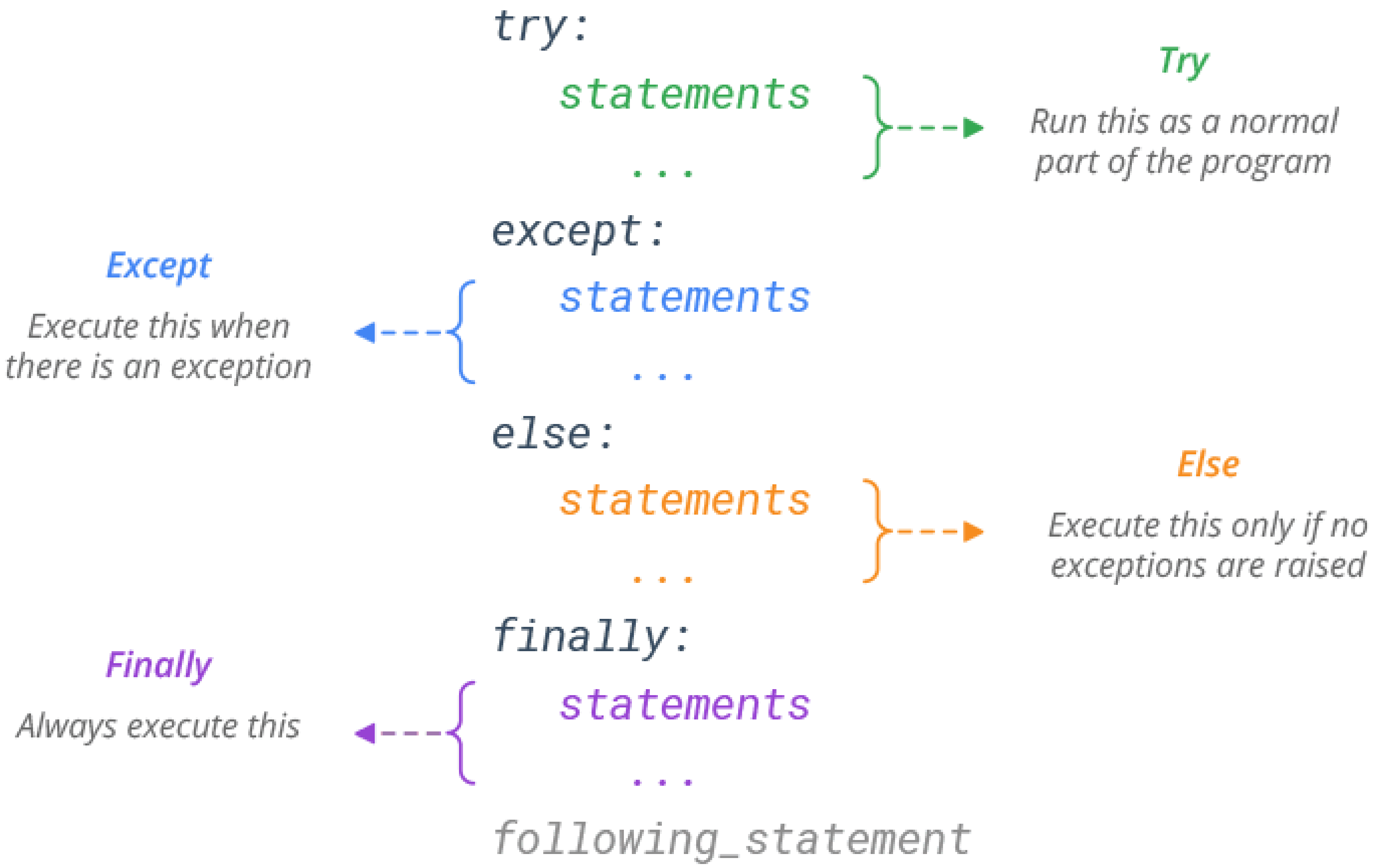
try---finally clause

With this variant, Python begins by running the statement block associated with the try header line. What happens next depends on whether an exception occurs during the try block:

- If no exception occurs while the try block is running, Python jumps back to run the finally block and then continues execution past below the try statement.

- If an exception does occur during the try block's run, Python still comes back and runs the finally block, but it then propagates the exception up to a higher try or the top-level default handler; the program does not resume execution below the try statement. That is, the finally block is run even if an exception is raised, but unlike an except, the finally does not terminate the exception—it continues being raised after the finally block runs.

The try/finally form is useful when you want to be completely sure that an action will happen after some code runs, regardless of the exception behavior of the program. In practice, it allows you to specify cleanup actions that always must occur, such as file closes and server disconnects.



```
try:
    statements
        ...
except:
    statements
        ...
else:
    statements
        ...
finally:
    statements
        ...
following_statement
```

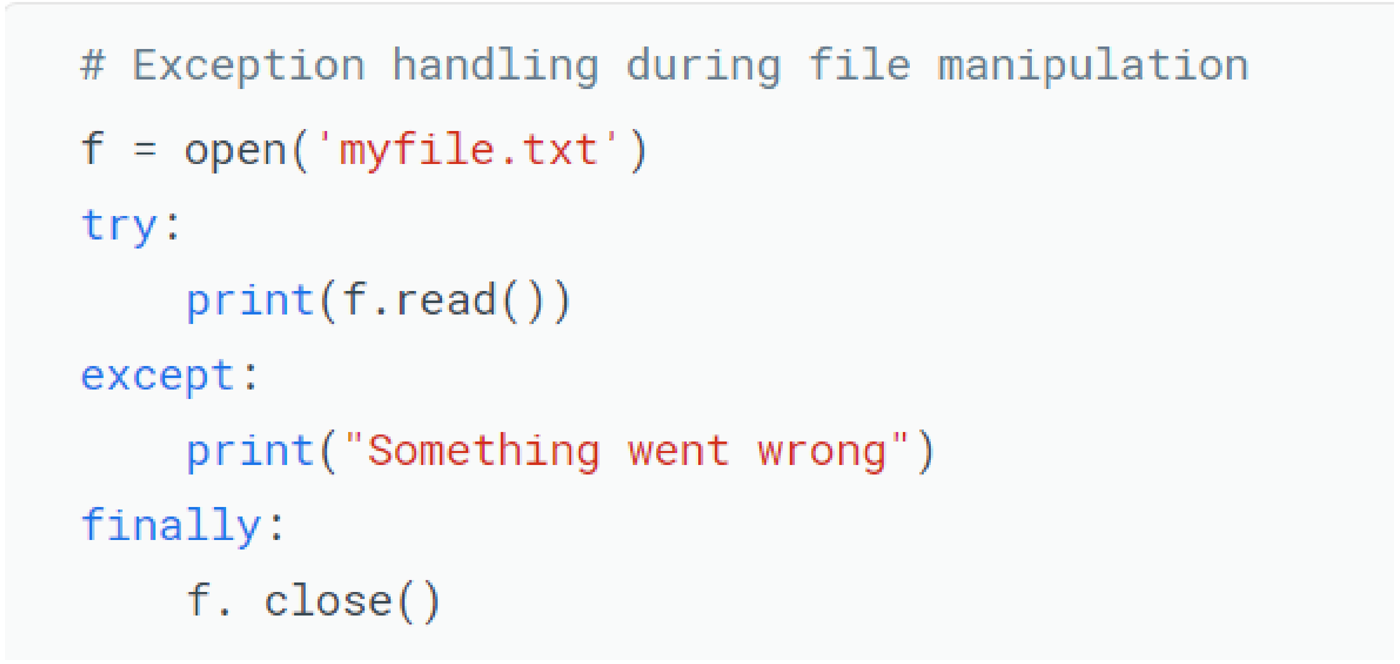
Try
Run this as a normal part of the program

Except
Execute this when there is an exception

Else
Execute this only if no exceptions are raised

Finally
Always execute this

Example



```python
# Exception handling during file manipulation
f = open('myfile.txt')
try:
    print(f.read())
except:
    print("Something went wrong")
finally:
    f. close()
```

```
# Exception handling during file
manipulation
f = open('myfile.txt')
try:
        print(f.read())
except:
        print("Something went wrong")
finally:
        f.close()
```

Use finally clause to define clean-up actions that must be executed under all circumstances e.g. closing a file.

The need for the finally block

- It is not recommended to maintain clean up code(Resource Deallocating Code or Resource Releasing code) inside try block because there is no guarantee for the execution of every statement inside try block always.

- It is not recommended to maintain clean up code inside except block, because if there is no exception then except block won't be executed.

- Hence, it’s required that there should some block to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether exception handled or not handled. Such a place is the finally block.

- Hence the main purpose of finally block is to maintain clean up code. The finally block always executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs.

Case-1: **If there is no exception**

```
try:
        print("try")
except:
        print("except")
finally:
        print("finally")
```

Output

```
try
finally
```

Case-2: **If there is an exception raised but handled**

```
try:
        print("try")
        print(10/0)
except ZeroDivisionError:
        print("except")
finally:
        print("finally")
```

Output

```
try
except
finally
```

Case-3: **If there is an exception raised but not handled**

```
try:
        print("try")
        print(10/0)
except NameError:
        print("except")
finally:
        print("finally")
```

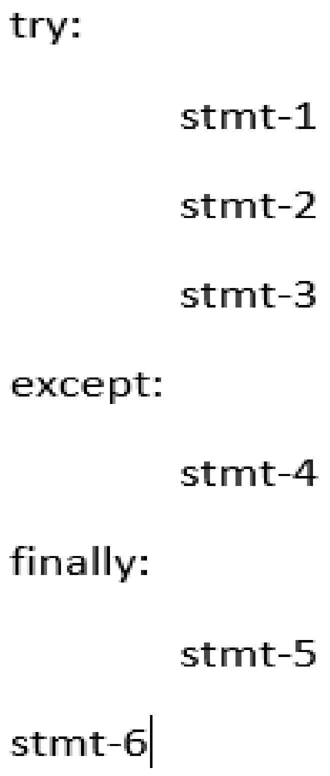
Output

```
try
finally
ZeroDivisionError: division by zero(Abnormal Termination)
```

The speciality of finally block is it will be executed always whether exception raised or not raised and whether exception handled or not handled

Control Flow in try-except-finally

- Case-1: If there is no exception
  - 1,2,3,5,6 Normal Termination
- Case-2: If an exception raised at stmt2 and the corresponding except block matched
  - 1,4,5,6 Normal Termination
- Case-3: If an exception raised at stmt2 but the corresponding except block not matched
  - 1,5 Abnormal Termination
- Case-4:If an exception raised at stmt4
  - then it is always abnormal termination but before that finally block will be executed.
- Case-5: If an exception raised at stmt-5 or at stmt-6
  - then it is always abnormal termination

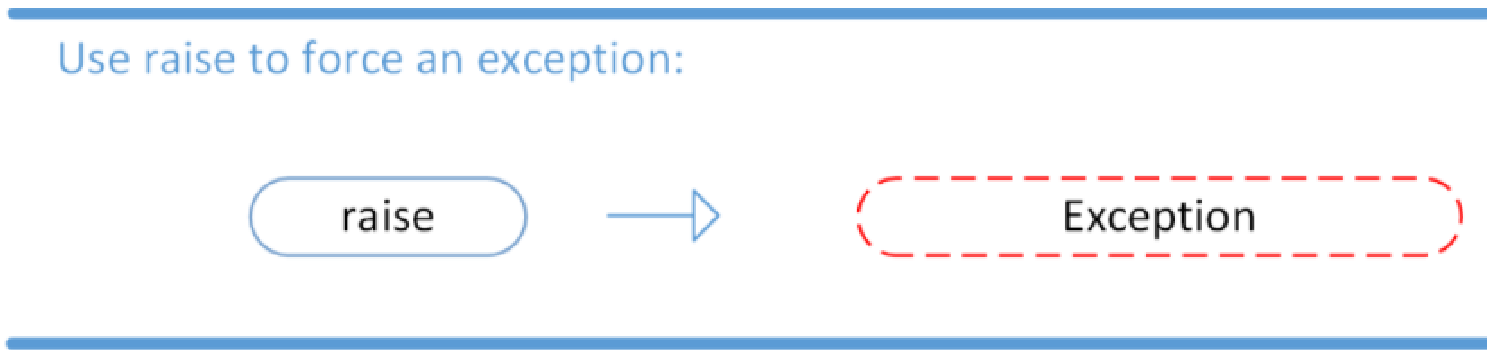


```
try:
            stmt-1
            stmt-2
            stmt-3
except:
            stmt-4
finally:
            stmt-5
stmt-6
```

Raising Exceptions in Python

- In Python programming, exceptions are raised when errors occur at runtime. We can also manually raise exceptions using the raise keyword.

- This is usually done for the purpose of error-checking.

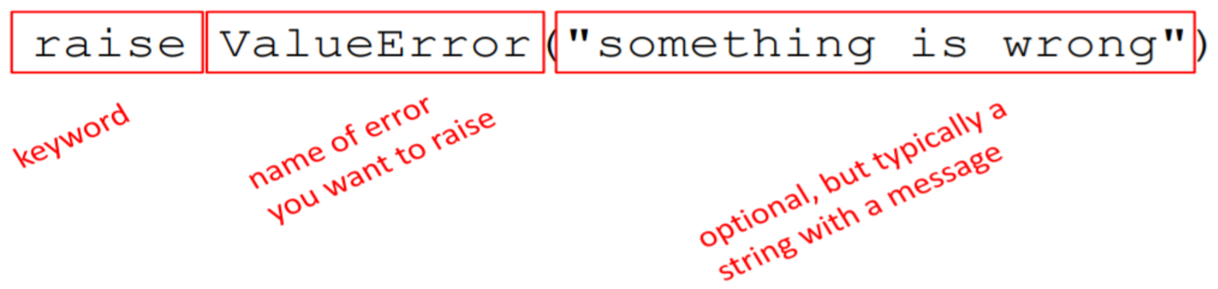


Raising Exceptions

- The effect when you raise an exception is the same as when Python raises one: it prints a traceback and an error message.

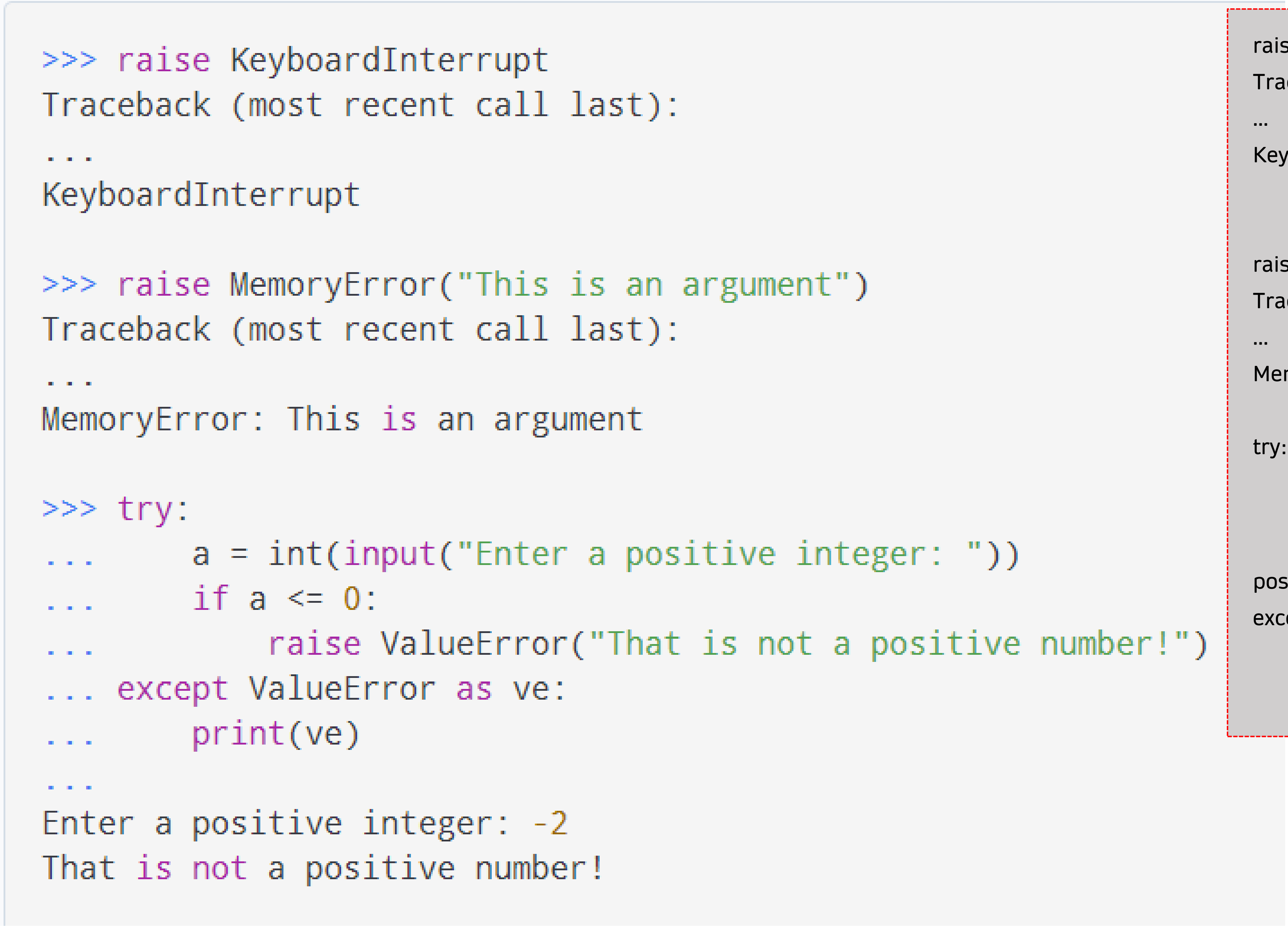
- When you raise an exception, you can provide a detailed error message as an optional argument

```
raise <exceptionName>(<arguments>)
```



```
raise ValueError("something is wrong")
```

keyword
name of error you want to raise
optional, but typically a string with a message



```
>>> raise KeyboardInterrupt
Traceback (most recent call last):
...
KeyboardInterrupt

>>> raise MemoryError("This is an argument")
Traceback (most recent call last):
...
MemoryError: This is an argument

>>> try:
...     a = int(input("Enter a positive integer: "))
...     if a <= 0:
...         raise ValueError("That is not a positive number!")
... except ValueError as ve:
...     print(ve)
...
Enter a positive integer: -2
That is not a positive number!
```

```
raise KeyboardInterrupt
Traceback (most recent call last):
...
KeyboardInterrupt


raise MemoryError("This is an argument")
Traceback (most recent call last):
...
MemoryError: This is an argument


try:
        a = int(input("Enter a positive integer: "))
        if a <= 0:
                raise ValueError("That is not a
positive number!")
except ValueError as ve:
        print(ve)
```

Valid combinations of try-except-else-finally

1) A try block should be followed by an except or finally block.

2) An except block, should have a try block. i.e except without try is always invalid.

3) A finally block, without try block is invalid.

4) We can write multiple except blocks for the same try block, but we cannot write multiple finally blocks for the same try.

5) An else block can be there only if there is an except block

6) In try-except/s-else-finally order is important.

7) We can define try-except-else-finally inside try, except, else and finally blocks. i.e nesting of try-except-else-finally is always possible.

Types of Exceptions:

In Python there are 2 types of exceptions

1) Predefined Exceptions
2) User Defined Exceptions

Predefined Exceptions:

Also known as inbuilt exceptions. The exceptions which are raised automatically by Python virtual machine whenever a particular event occurs are called pre-defined exceptions.

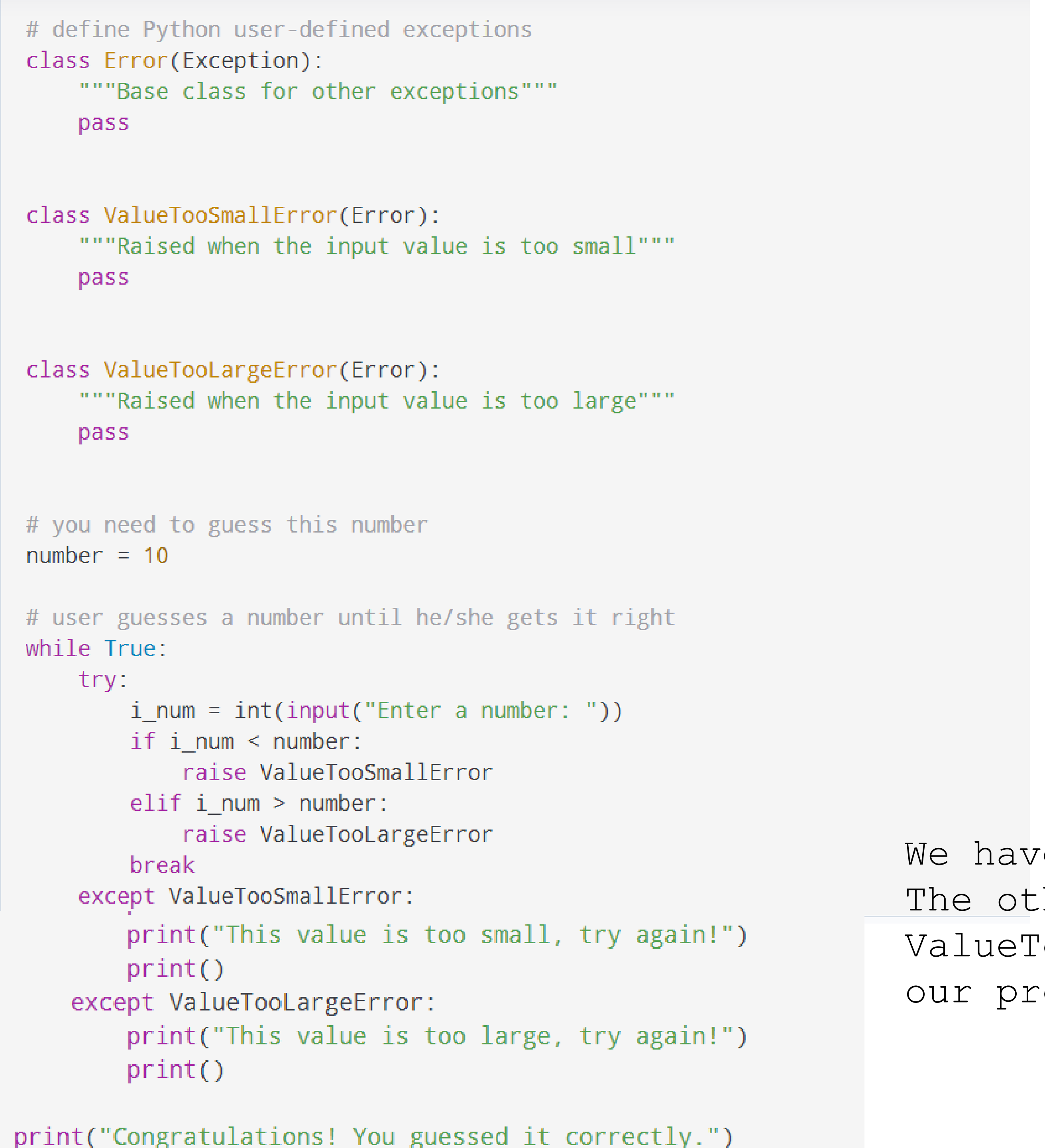
User Defined Exceptions:

Also known as Customized Exceptions or Programmatic Exceptions
Sometimes, we must define and raise exceptions explicitly to indicate that something goes wrong, such type of exceptions are called User Defined Exceptions or Customized Exceptions

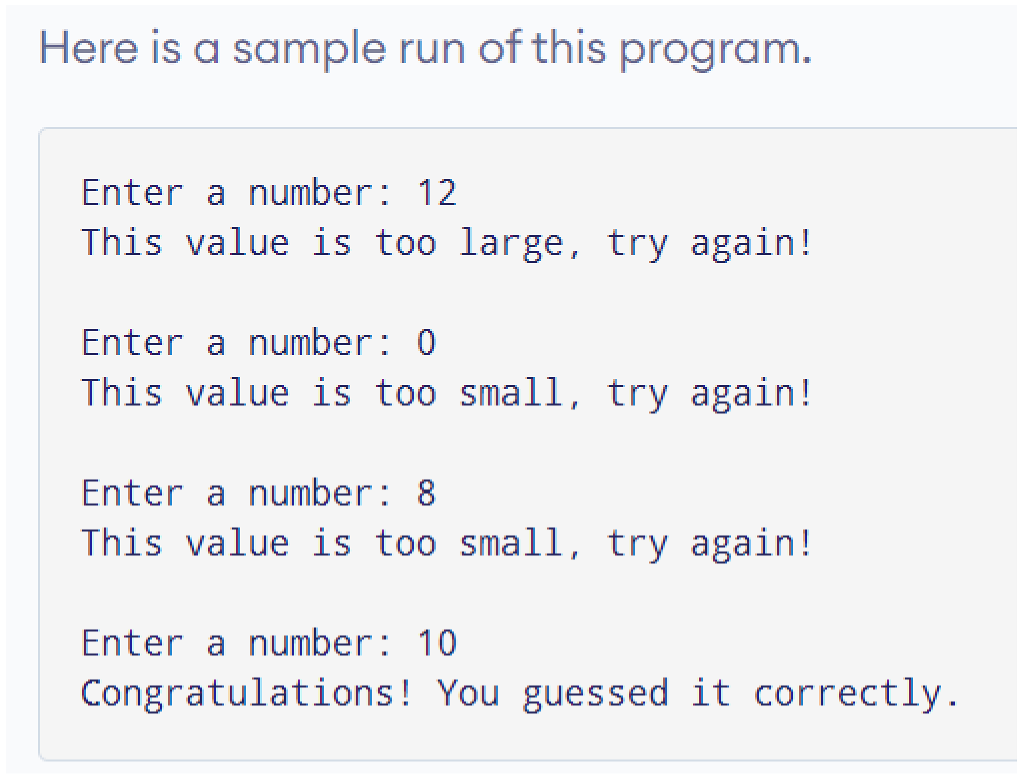
User Defined Exceptions

- In this example, we will illustrate how user-defined exceptions can be used in a program to raise and catch errors.

- This program will ask the user to enter a number until they guess a stored number correctly. To help them figure it out, a hint is provided whether their guess is greater than or less than the stored number.



```python
# define Python user-defined exceptions
class Error(Exception):
    """Base class for other exceptions"""
    pass


class ValueTooSmallError(Error):
    """Raised when the input value is too small"""
    pass


class ValueTooLargeError(Error):
    """Raised when the input value is too large"""
    pass


# you need to guess this number
number = 10

# user guesses a number until he/she gets it right
while True:
    try:
        i_num = int(input("Enter a number: "))
        if i_num < number:
            raise ValueTooSmallError
        elif i_num > number:
            raise ValueTooLargeError
        break
    except ValueTooSmallError:
        print("This value is too small, try again!")
        print()
    except ValueTooLargeError:
        print("This value is too large, try again!")
        print()

print("Congratulations! You guessed it correctly.")
```



Here is a sample run of this program.

```
Enter a number: 12
This value is too large, try again!

Enter a number: 0
This value is too small, try again!

Enter a number: 8
This value is too small, try again!

Enter a number: 10
Congratulations! You guessed it correctly.
```

We have defined a base class called Error.
The other two exceptions (ValueTooSmallError and ValueTooLargeError) that are actually raised by our program are derived from this class.