Shri Vile Parle Kelavani Mandal's

SHRI BHAGUBHAI MAFATLAL POLYTECHNIC

Computer Engineering Department

Exception Handling

Course: Programming in Python

Course Code: PRP228918

Name of Staff: Mr. Pratik H. Shah

SEMESTER : IV

DIVISION : A & B

Types of Error:

In any programming language, there are 2 types of errors possible. They are:

1. **Syntax Errors**
2. **Runtime Errors**

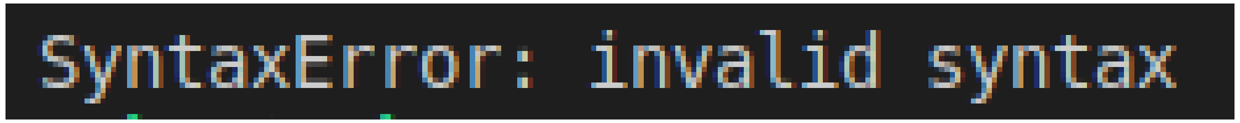
Syntax Errors

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

Program: Syntax Error (demo1.py)

```
x=123
if x==123
   print("Hello")
```

Output



```
SyntaxError: invalid syntax
```

Here we missed placing a colon in the if condition, which is violating the syntax.
Hence, syntax error.

Runtime Errors in Python

While executing the program if something goes wrong then we will get Runtime Errors.
They might be caused due to,

1.End-User Input
2.Programming Logic
3.Memory Problems etc.

Such types of errors are called exceptions.

Program: Runtime Error (demo2.py)

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

Output: Runtime Error

Program: Runtime Error (demo3.py)

```
print(10/”two”)
```

Output: Runtime Error in Python

Note: Exception Handling concept applicable for Runtime Errors but not for syntax errors.

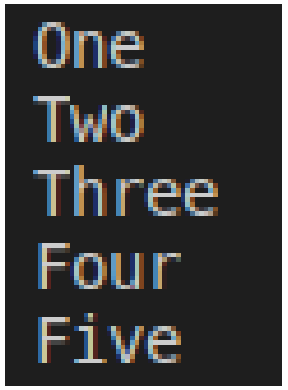
Normal Flow of Program Execution in Python:

In a program, if all statements are executed as per the conditions successfully and if we get the output as expected then that flow is called the normal flow of the Program Execution. The below program will get executed successfully from start to end.

Program: Normal Flow (demo4.py)

```
print('One')
print('Two')
print('Three')
print('Four')
print('Five')
```

Output:



```
One
Two
Three
Four
Five
```

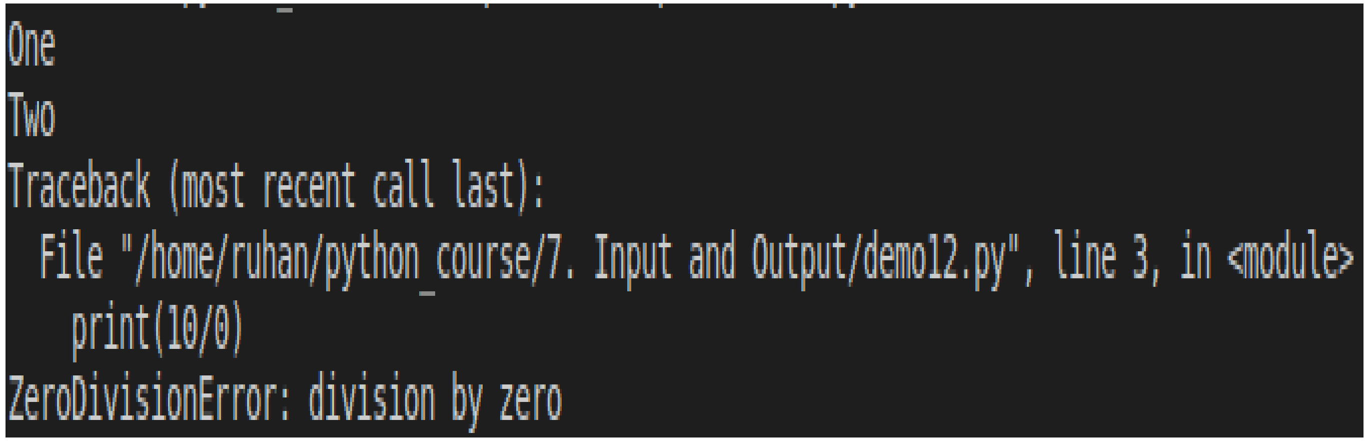
Abnormal Flow of Program Execution in Python:

While executing statements in a program, if any error occurs at runtime, then immediately program flow gets terminated abnormally, without executing the other statements. This kind of termination is called an abnormal flow of execution. The following example terminated abnormally.

Program: Abnormal flow (demo5.py)

```
print('One')
print('Two')
print(10/0)
print('Four')
print('Five')
```

Output:



```
One
Two
Traceback (most recent call last):
  File "/home/ruhan/python_course/7. Input and Output/demo12.py", line 3, in <module>
    print(10/0)
ZeroDivisionError: division by zero
```

Above program terminated in the middle of the execution where a run time error occurred. As discussed, if a runtime error occurs it won’t execute the remaining statements.

What is an Exception in Python?

An unwanted or unexpected event that disturbs the normal flow of the program is called an exception. Whenever an exception occurs, immediately the program will terminate abnormally. In order to get our program executed normally, we need to handle those exceptions on high priority.

Exception Handling in Python:

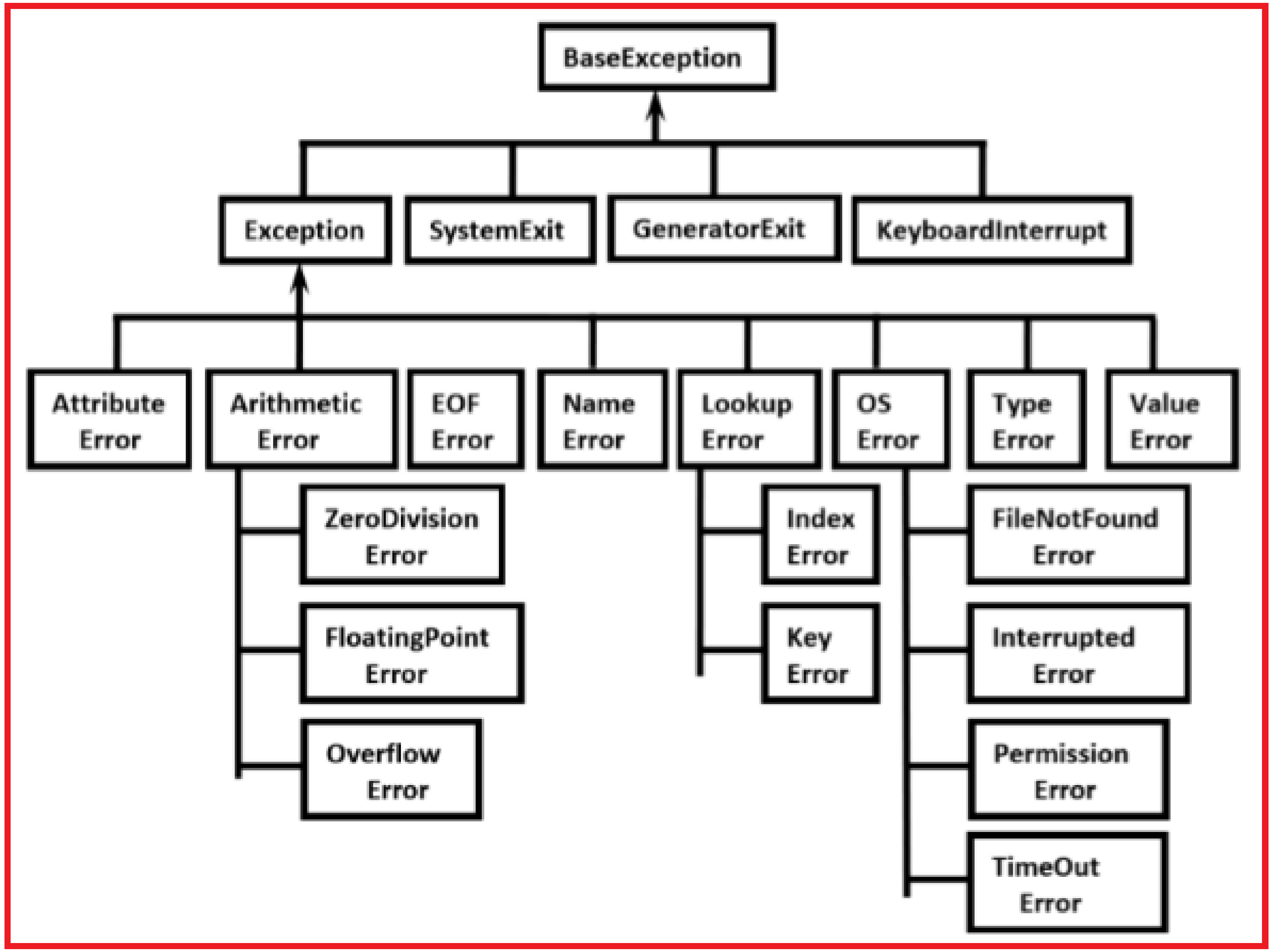
Exception handling does not mean repairing or correcting the exception. Instead, it is the process in which we define a way so that the program doesn’t terminate abnormally due to the exceptions.

Default Exception Handing in Python:

In python, for every exception type, a corresponding class is available and every exception is an object to its corresponding class. Whenever an exception occurs, Python Virtual Machine (PVM) will create the corresponding exception object and will check for handling code.

If handling code is not available, then the Python interpreter terminates the program abnormally and prints corresponding exception information to the console. The rest of the program won’t be executed.

Exception Hierarchy:



Note:

1. Every Exception in Python is a class.
2. The BaseException class is the root class for all exception classes in the python exception hierarchy and all the exception classes are child classes of BaseException.
3. The Programmers need to focus and understand clearly the Exception and child classes.

How to Handle Exceptions in Python?

Using Try-Except statements we can handle exceptions in python.

1. **Try block:** try is a keyword in python. The code which may be expected to raise an exception should be written inside the try block.

2. **Except block:** except is a keyword in python. The corresponding handling code for the exception, if occurred, needs to be written inside the except block.

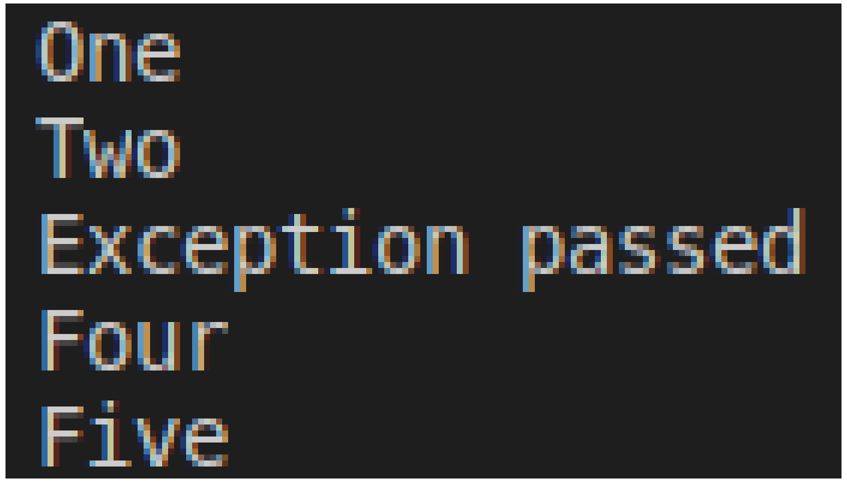
How does it work?

If the code in the try block raises an exception, then only the execution flow goes to the except block for handling code. If there is no exception raised by the code in the try block, then the execution flow won’t go to the except block.

Program: Handling exception by using try and except (demo6.py)

```
print('One')
print('Two')
try:
   print(10/0)
except ZeroDivisionError:
   print("Exception passed")
print('Four')
print('Five')
```

Output:



```
One
Two
Exception passed
Four
Five
```

Let’s see how the execution flow is, in different scenarios using ‘try except’

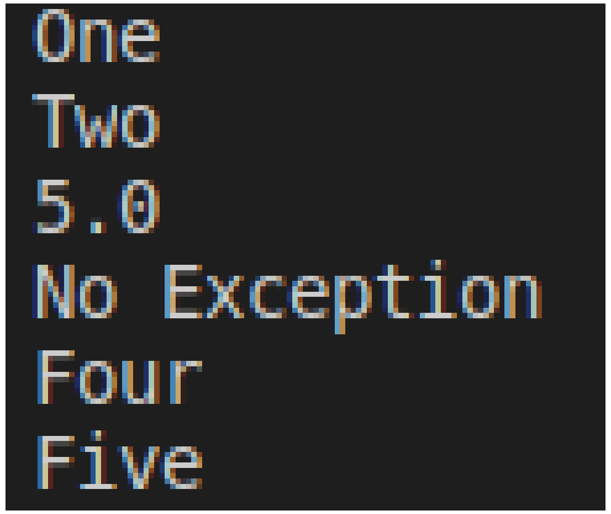
Case1:

In our program, if there is no exception, then should be normal flow with normal termination. In this case, the except block won’t execute.

Program: without Exception (demo7.py)

```
print('One')
print('Two')
try:
   print(10/2)
   print("No Exception")
except ZeroDivisionError:
   print("Exception passed")
print('Four')
print('Five')
```

Output:



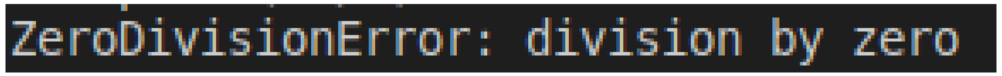
```
One
Two
5.0
No Exception
Four
Five
```

Case 2:
In our program, if an exception occurs before the try block then the program terminates abnormally. Only the code inside the try block will be checked for the exception and its handling code. But, if there is any exception for the code before the try block, then abnormal termination

Program: program terminates abnormally (demo8.py)

```
print(10/0)
try:
  print(10/2)
  print("No Exception")
except ZeroDivisionError:
  print("Exception passed")
print('Four')
print('Five')
```

Output:



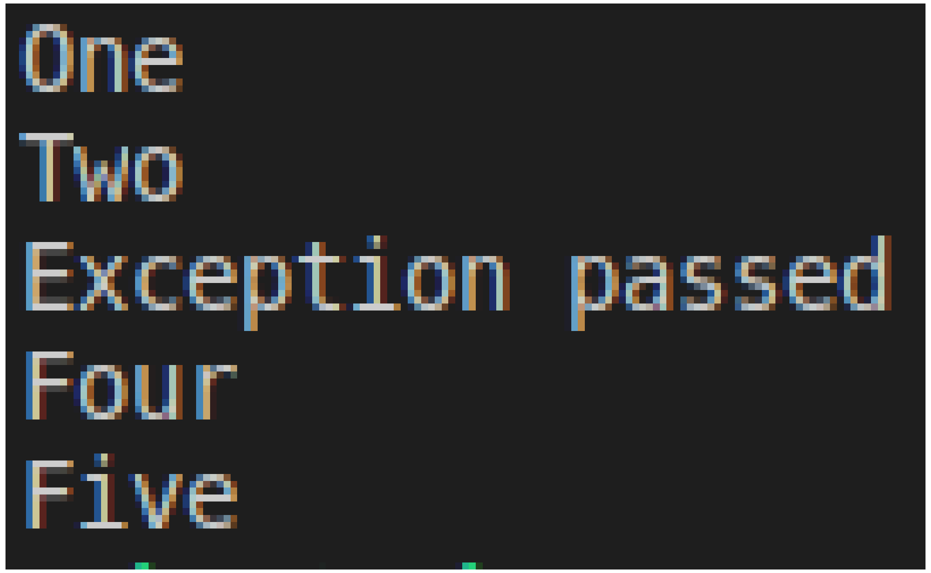
```
ZeroDivisionError: division by zero
```

Case 3:
If the code inside the try block raises an exception, then the execution flow goes to the except block. In the except block, we should have handled the corresponding exception that occurred in the try block, only then the except block will execute leading to normal termination.

Program: demo9.py

```
print('One')
print('Two')
try:
  print(10/0)
  print("No Exception")
except ZeroDivisionError:
  print("Exception passed")
print('Four')
print('Five')
```

Output:



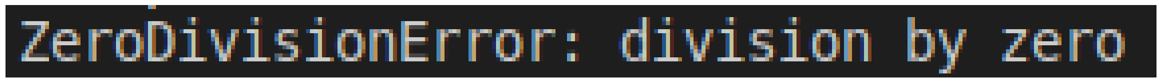
```
One
Two
Exception passed
Four
Five
```

Case 4:
If the code inside the try block raises an exception, then the execution flow goes to the except block. In the except block we should have handled for the exception other than what occurred in the try block, then the except block will not execute leading to abnormal termination.

Program: demo10.py

```
try:
  print(10/0)
  print("No Exception")
except TypeError:
  print("Exception passed")
print('Four')
print('Five')
```

Output:



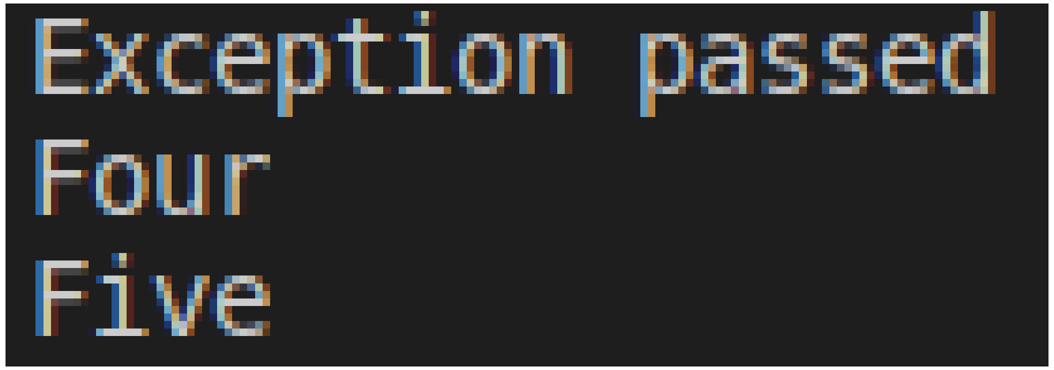
```
ZeroDivisionError: division by zero
```

Case 5:
If there are 3 lines of code inside the try block and an exception is raised when executing the first line, then the execution flow goes to the except block without executing the remaining two lines.

Program: demo11.py

```
try:
  print(10/0)
  print("line 2")
  print("line 3")
  print("No Exception")
except ZeroDivisionError:
  print("Exception passed")
print('Four')
print('Five')
```

Output:



```
Exception passed
Four
Five
```

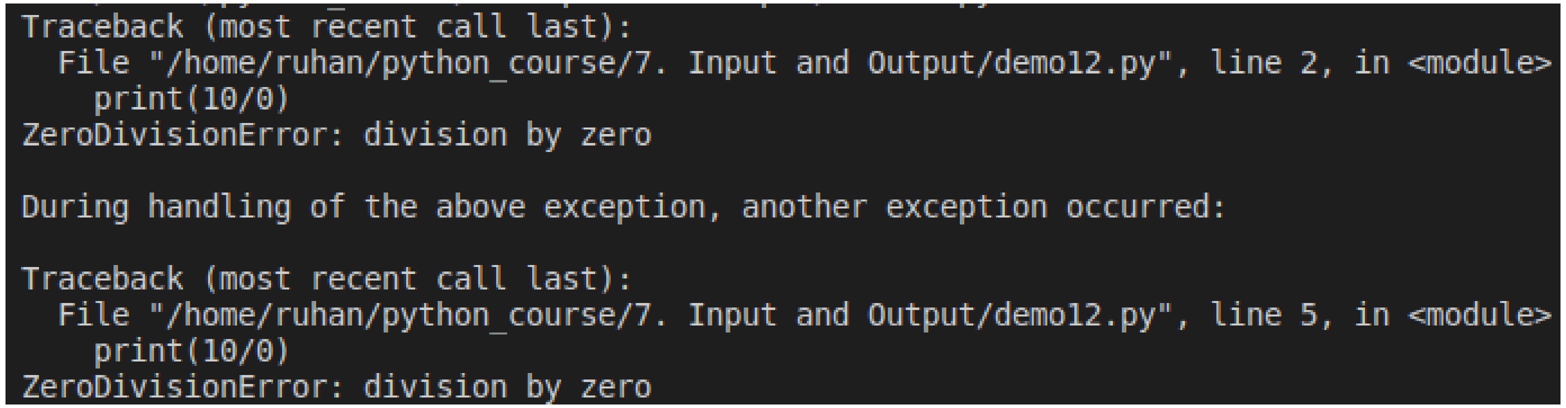
Case 6:

If an exception is raised inside the try block, the execution goes to the except block. If the code inside the except block also raises an exception then it leads to abnormal termination

Program: demo12.py

```
try:
  print(10/0)
  print("No Exception")
except ZeroDivisionError:
  print(10/0)
print('Four')
print('Five')
```

Output:



```
Traceback (most recent call last):
  File "/home/ruhan/python_course/7. Input and Output/demo12.py", line 2, in <module>
    print(10/0)
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/home/ruhan/python_course/7. Input and Output/demo12.py", line 5, in <module>
    print(10/0)
ZeroDivisionError: division by zero
```

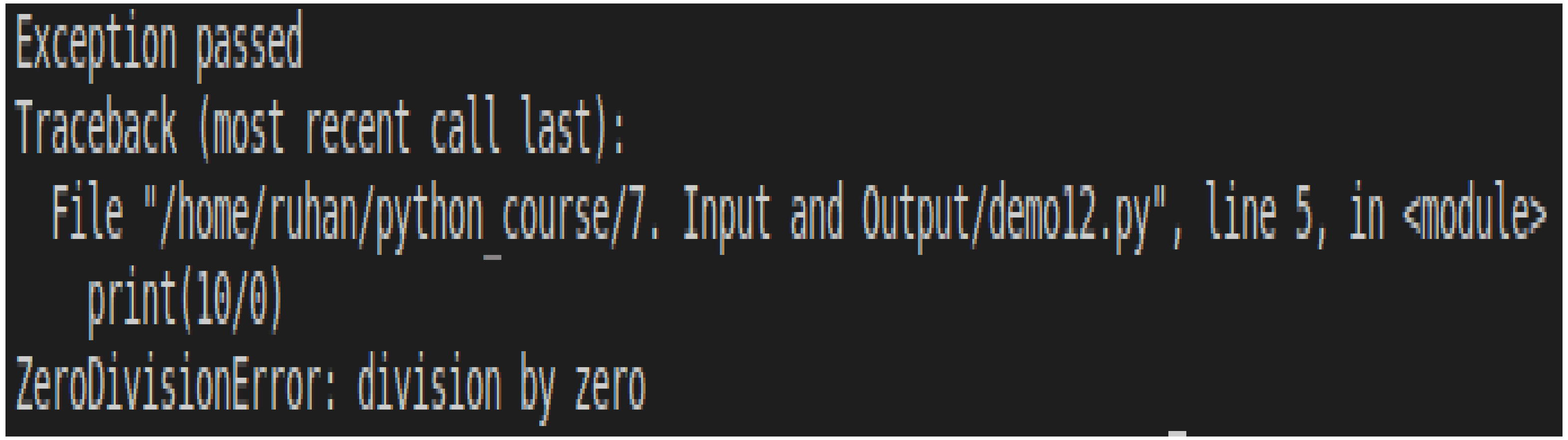
Case 7:

If an exception is raised inside the try block, the execution goes to the except block. After the execution of the code inside the except block, the execution comes out of it and continues with the other statements that follow. If any exceptions occur, when executing those following statements, then it leads to abnormal termination.

Program: demo13.py

```python
try:
    print(10/0)
except ZeroDivisionError:
    print("Exception passed")
print(10/0)
print('Four')
print('Five')
```

Output:



```
Exception passed
Traceback (most recent call last):
  File "/home/ruhan/python_course/7. Input and Output/demo12.py", line 5, in <module>
    print(10/0)
ZeroDivisionError: division by zero
```

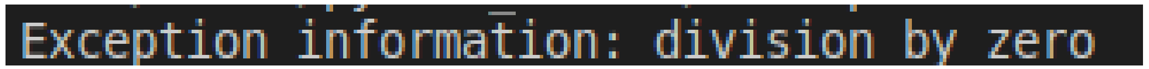
Printing Exception Information in Python:

In the above examples, in the exception handling code, we are printing our custom message like “Exception passed” etc. Rather than that, we can also print exception information by creating a reference to the exception.

Program: Printing exception information (demo14.py)

```
try:
    print(10/0)
except ZeroDivisionError as z:
    print("Exception information:", z)
```

Output:

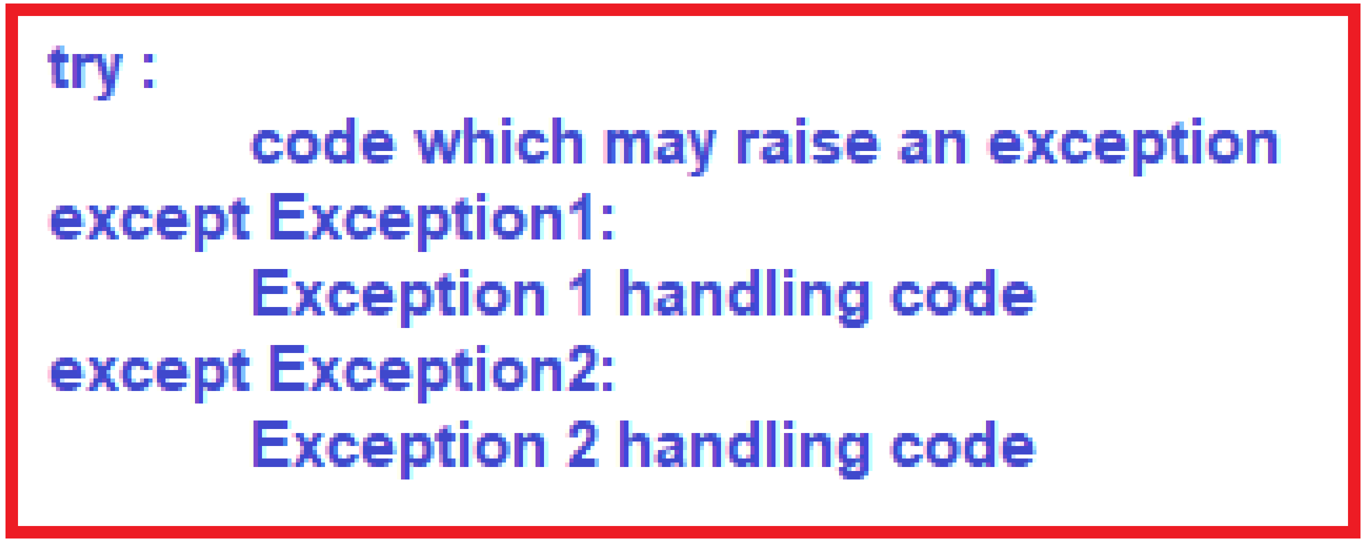


```
Exception information: division by zero
```

try with multiple except Blocks in python:

In python, try with multiple except blocks are allowed. Sometimes, there may be the possibility that a piece of code can raise different exceptions in different cases. In such cases, in order to handle all the exceptions, we can use multiple except blocks. So, for every exception type a separate except block we have to write.

Syntax:

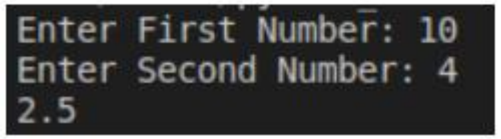


```
try :
        code which may raise an exception
except Exception1:
        Exception 1 handling code
except Exception2:
        Exception 2 handling code
```

Program: try with Multiple Except Blocks (demo15.py)

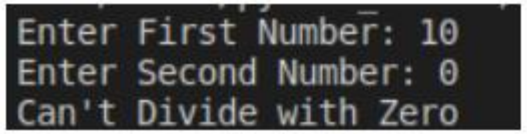
```
try:
   x=int(input("Enter First Number: "))
   y=int(input("Enter Second Number: "))
   print(x/y)
except ZeroDivisionError:
   print("Can't Divide with Zero")
except ValueError:
   print("please provide int value only")
```

Output1:



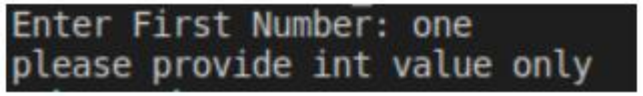
```
Enter First Number: 10
Enter Second Number: 4
2.5
```

Output2:



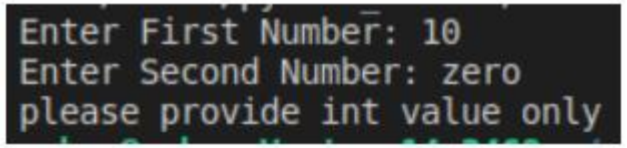
```
Enter First Number: 10
Enter Second Number: 0
Can't Divide with Zero
```

Output3:



```
Enter First Number: one
please provide int value only
```

Output4:

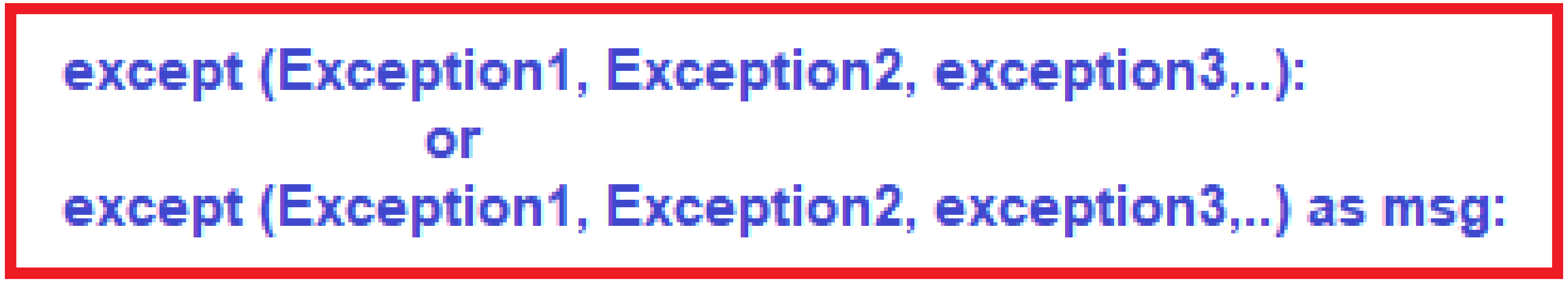


```
Enter First Number: 10
Enter Second Number: zero
please provide int value only
```

ONE Except BLOCK – MULTIPLE EXCEPTIONS

Rather than writing different except blocks for handling different exceptions, we can handle all of them in one except block. The only thing is we will not have the flexibility of customizing the message for each exception. Rather, we can take the reference to the exception and print its information as shown in one of the examples above.

Syntax:



```
except (Exception1, Exception2, exception3,..):
                        or
except (Exception1, Exception2, exception3,..) as msg:
```

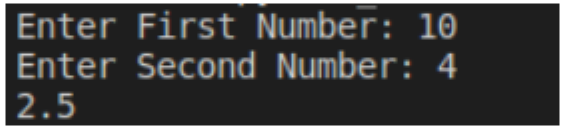
Parenthesis are mandatory, and this group of exceptions is internally considered as a tuple.

Program: single except block handling multiple exceptions (demo16.py)

```
try:
   x=int(input("Enter First Number: "))
   y=int(input("Enter Second Number: "))
   print(x/y)
except (ZeroDivisionError,ValueError) as e:
   print("Please Provide valid numbers only and problem is: ", e)
```

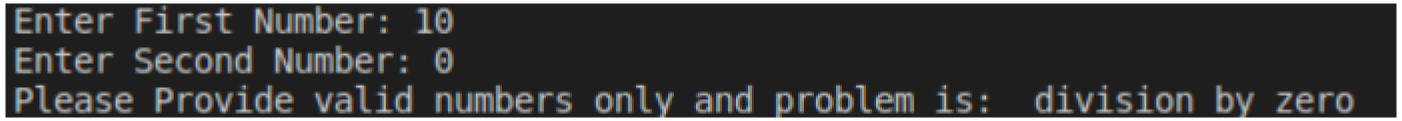
Output is in next slide

Output1:



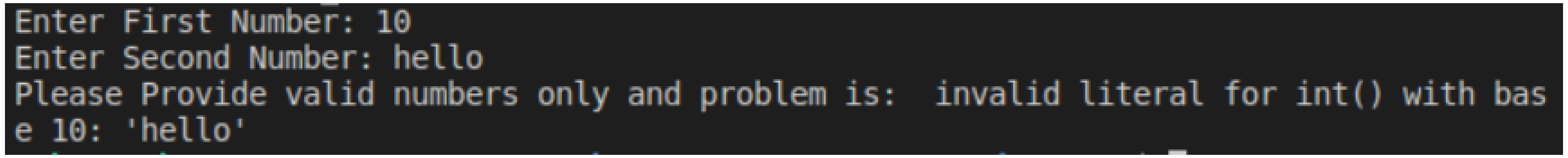
```
Enter First Number: 10
Enter Second Number: 4
2.5
```

Output2:



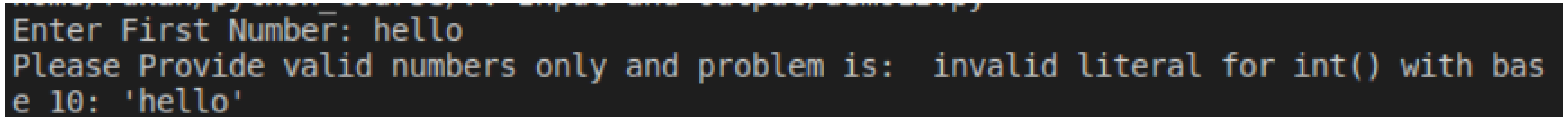
```
Enter First Number: 10
Enter Second Number: 0
Please Provide valid numbers only and problem is:  division by zero
```

Output3:



```
Enter First Number: 10
Enter Second Number: hello
Please Provide valid numbers only and problem is:  invalid literal for int() with bas
e 10: 'hello'
```

Output4:



```
Enter First Number: hello
Please Provide valid numbers only and problem is:  invalid literal for int() with bas
e 10: 'hello'
```

Default Except Block in Python:

We can use default except block to handle any type of exceptions. It’s not required to mention any exception type for the default block. Whenever we don’t have any idea of what expectation the code could raise, then we can go for default except block.

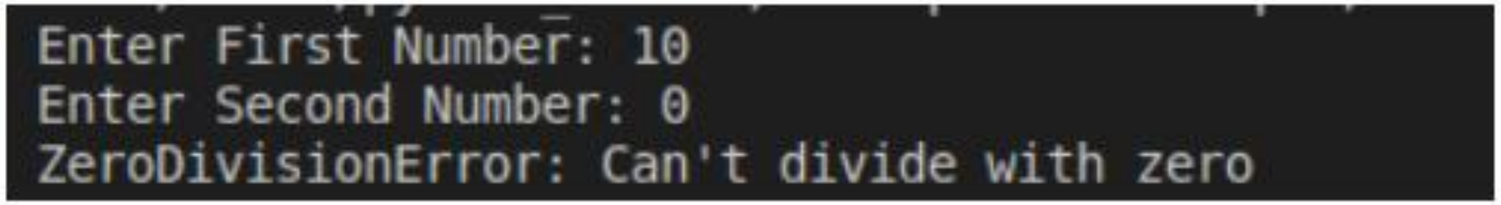
We know that we can have multiple except blocks for a single try block. If the default except block is one among those multiple ones, then it should be at last, else we get Syntax Error.

Program: Default except block (demo17.py)

```
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: Please provide valid input only")
```

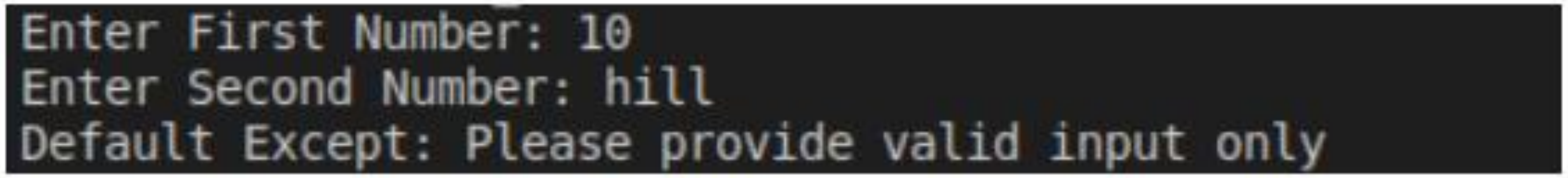
Output is in next slide

Output1:



```
Enter First Number: 10
Enter Second Number: 0
ZeroDivisionError: Can't divide with zero
```

Output2:

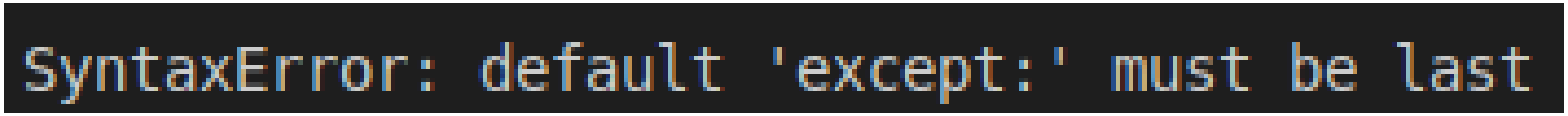


```
Enter First Number: 10
Enter Second Number: hill
Default Except: Please provide valid input only
```

Program: Default block (demo18.py)

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

Output



```
SyntaxError: default 'except:' must be last
```

Finally Block in Python

Why do we need Finally Block?

1. In any project after completing the tasks, it is recommended to destroy all the unwanted things that are created for the completion of the task.

2. For example, if I have opened a database connection in my code and done some tasks related to the database. After the tasks are completed, it is recommended to close the connection.

3. These clean-up activities are very important and should be handled in our code.

4. finally is a keyword in python that can be used to write a finally block to do clean-up activities.

Why not ‘try except’ block for clean-up activities?

Try block: There is no guarantee like every statement will be executed inside the try block. If an exception is raised at the second line of code in the try block, then the remaining lines of code in that block will not execute. So, it is not recommended to write clean-up code inside the try block.

Except block: There is no guarantee that an except block will execute. If there is no exception then except block won’t be executed. Hence, it is also not recommended to write clean-up code inside except block

Conclusion:

So, a separate block is required to write clean-up activities. This block of code should always execute irrespective of the exceptions. If no exception, then the clean-up code should execute. If an exception is raised and is also handled, then also clean-up code should execute. If an exception is raised, but not handled, then also clean-up code should execute.

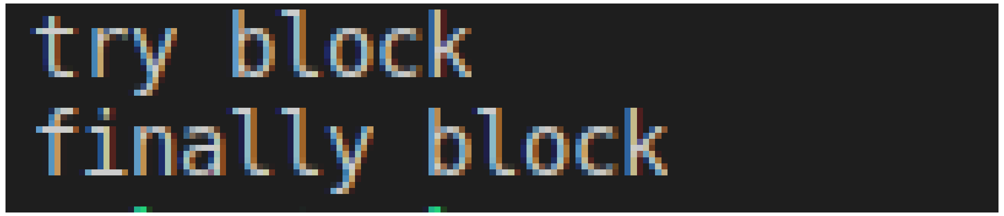
All the above requirements, which can’t be achieved by try-except block, can be achieved by finally block. The finally block will be executed always, irrespective of the exception raised or not, exception handled or not.

Case 1: If there is no exception, then finally block will execute

Program: demo19.py

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

Output:



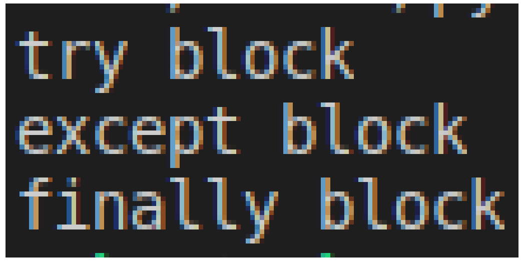
```
try block
finally block
```

Case 2: If an exception is raised and handled with except block, then also finally block will be executed

Program: demo20.py

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

Output:



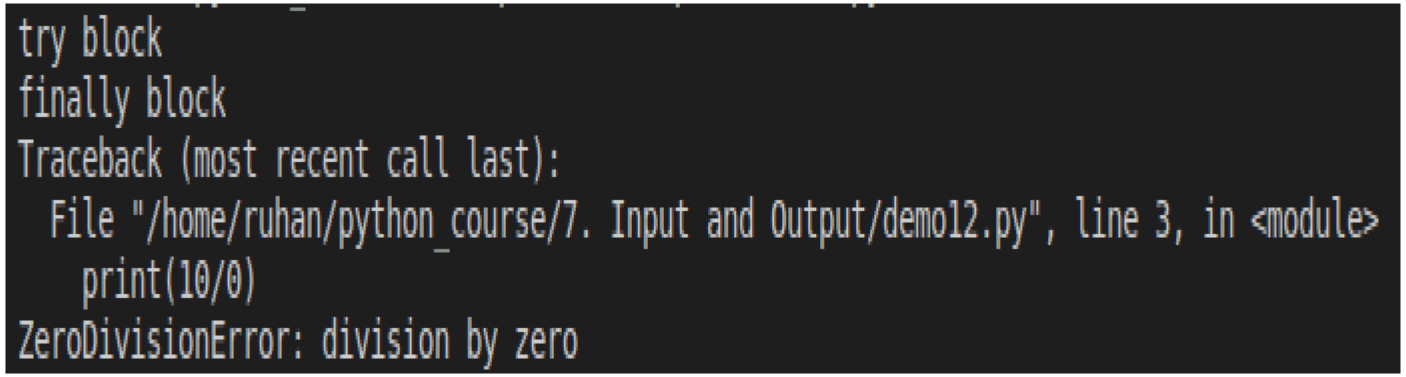
```
try block
except block
finally block
```

Case 3: If an exception is raised inside the try block but except block doesn’t handle that particular exception, then also finally block will be executed.

Program: demo21.py

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

Output:



```
try block
finally block
Traceback (most recent call last):
  File "/home/ruhan/python_course/7. Input and Output/demo12.py", line 3, in <module>
    print(10/0)
ZeroDivisionError: division by zero
```

If an exception is raised but is not handled, then the program is supposed to terminate abnormally. But before terminating, the finally block will be executed.

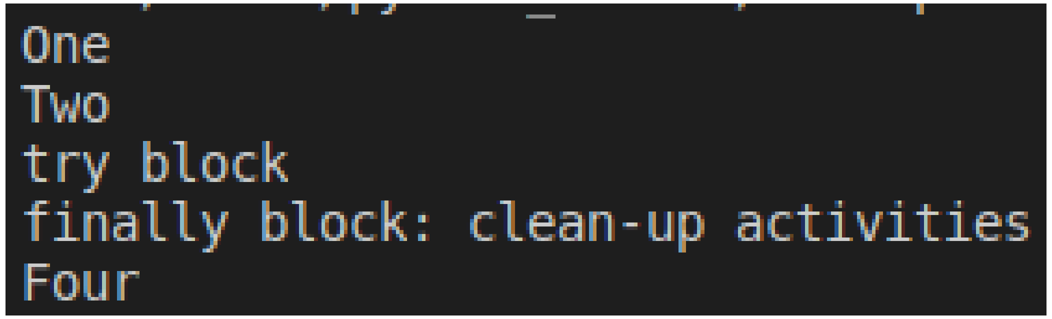
Different control flow cases of try except finally in python:

Case 1:
If there is no exception, then try, and finally blocks will execute, and except block won’t execute, leading to normal termination.

Program: demo22.py

```
print("One")
print("Two")
try:
  print("try block")
except ZeroDivisionError:
  print("except block: Handling code")
finally:
  print("finally block: clean-up activities")
print("Four")
```

Output:



```
One
Two
try block
finally block: clean-up activities
Four
```

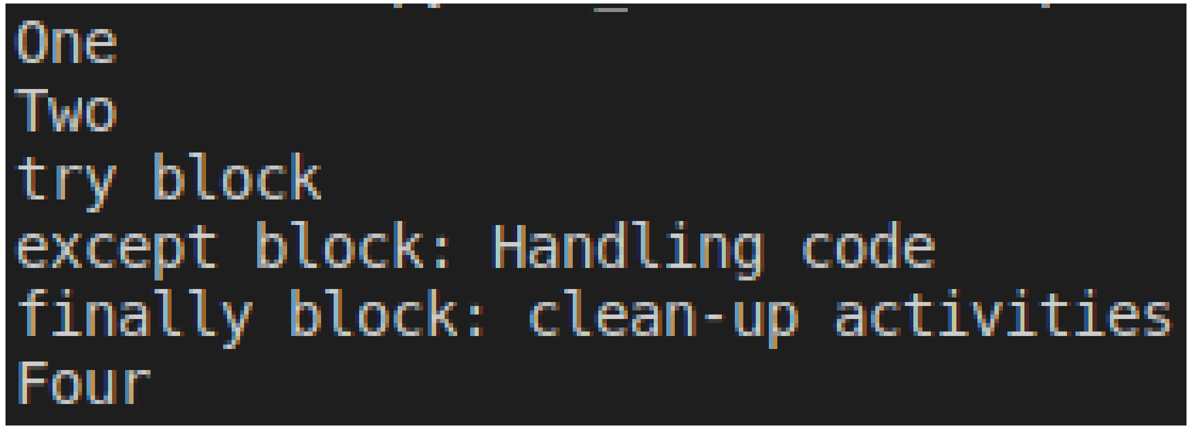
Case 2:

If an exception is raised inside the try block and the except block is handling that corresponding exception then try, except, and finally blocks will execute, leading to normal termination.

Program: demo23.py

```
print("One")
print("Two")
try:
  print("try block")
  print(10/0)
except ZeroDivisionError:
  print("except block: Handling code")
finally:
  print("finally block: clean-up activities")
print("Four")
```

Output:



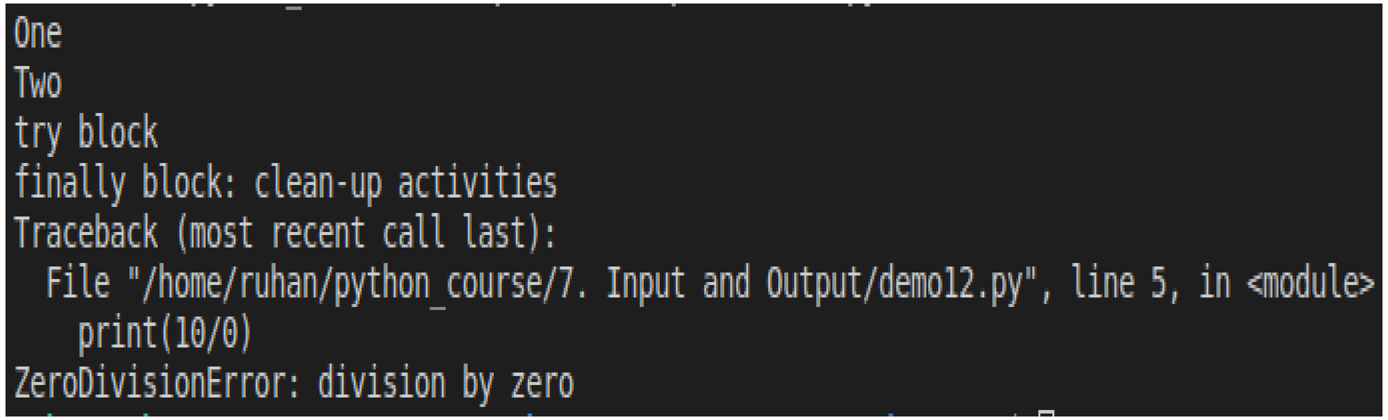
```
One
Two
try block
except block: Handling code
finally block: clean-up activities
Four
```

Case 3:
If an exception is raised inside the try block and the except block is not handling that corresponding exception then try and finally blocks will execute, leading to abnormal termination.

Program: demo24.py

```
print("One")
print("Two")
try:
    print("try block")
    print(10/0)
except NameError:
    print("except block: Handling code")
finally:
    print("finally block: clean-up activities")
print("Four")
```

Output:



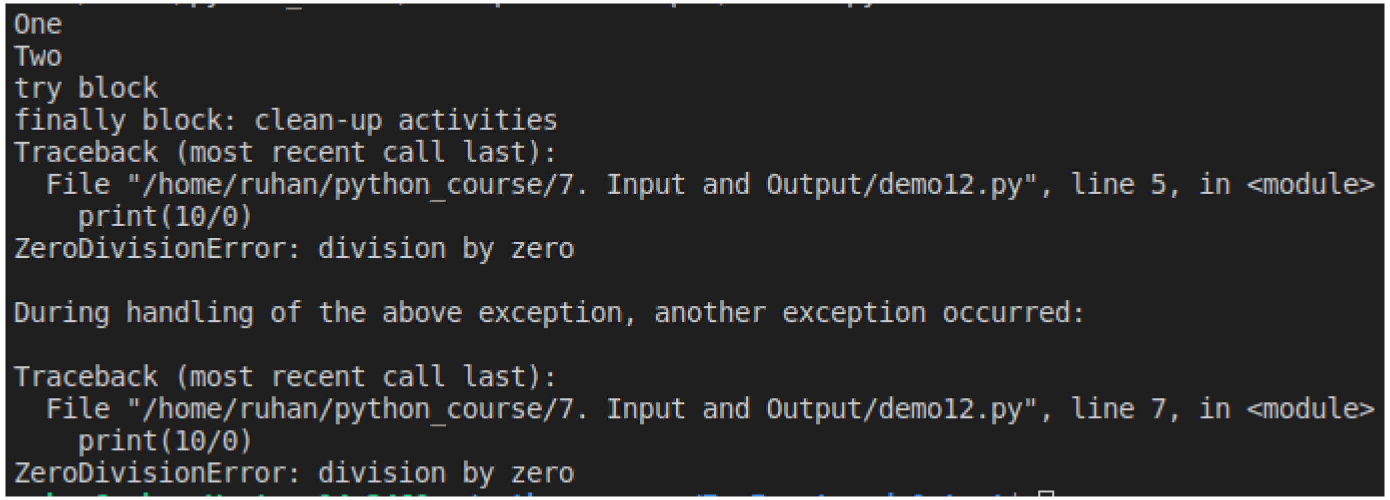
```
One
Two
try block
finally block: clean-up activities
Traceback (most recent call last):
  File "/home/ruhan/python_course/7. Input and Output/demo12.py", line 5, in <module>
    print(10/0)
ZeroDivisionError: division by zero
```

Case 4:
If an exception is raised inside the try block, then flow goes to the except block. And in the except block, assume, the particular exception is handled. Instead of handling exceptions, if the block itself raises another exception then the program will terminate abnormally. In this case also, before the program termination finally block will be executed.

Program: demo25.py

```
print("One")
print("Two")
try:
    print("try block")
    print(10/0)
except ZeroDivisionError:
    print(10/0)
finally:
    print("finally block: clean-up activities")
print("Four")
```

Output:



```
One
Two
try block
finally block: clean-up activities
Traceback (most recent call last):
  File "/home/ruhan/python_course/7. Input and Output/demo12.py", line 5, in <module>
    print(10/0)
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/home/ruhan/python_course/7. Input and Output/demo12.py", line 7, in <module>
    print(10/0)
ZeroDivisionError: division by zero
```

Case 5:

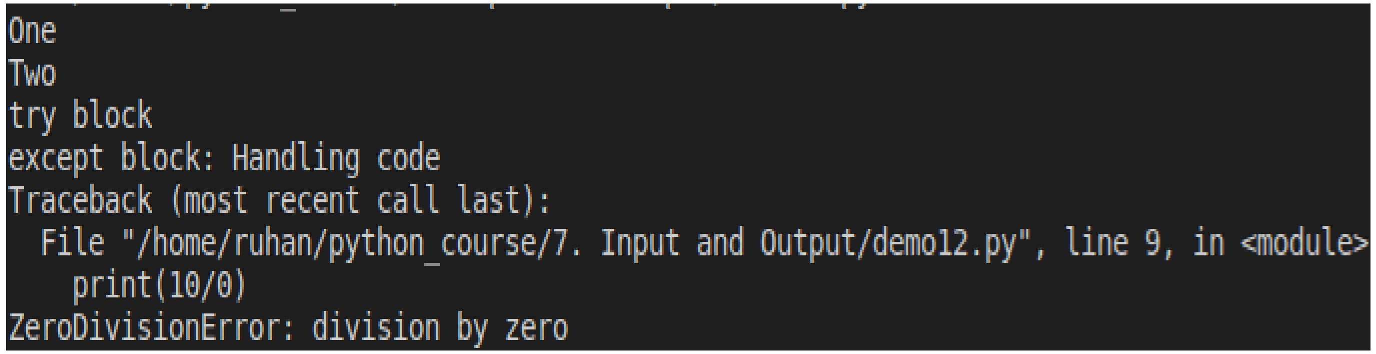
If an exception is raised inside the finally block, then it’s always abnormal termination.

Program: demo26.py

Output:

```
print("One")
print("Two")
try:
   print("try block")
   print(10/0)
except ZeroDivisionError:
   print("except block: Handling code")
finally:
   print(10/0)

print("Four")
```



```
One
Two
try block
except block: Handling code
Traceback (most recent call last):
  File "/home/ruhan/python_course/7. Input and Output/demo12.py", line 9, in <module>
    print(10/0)
ZeroDivisionError: division by zero
```

Nested try-except-finally blocks in Python

In python nested try except finally blocks are allowed. We can take try-except-finally blocks inside try block. We can take try-except-finally blocks inside except block. We can take try-except-finally blocks inside finally block

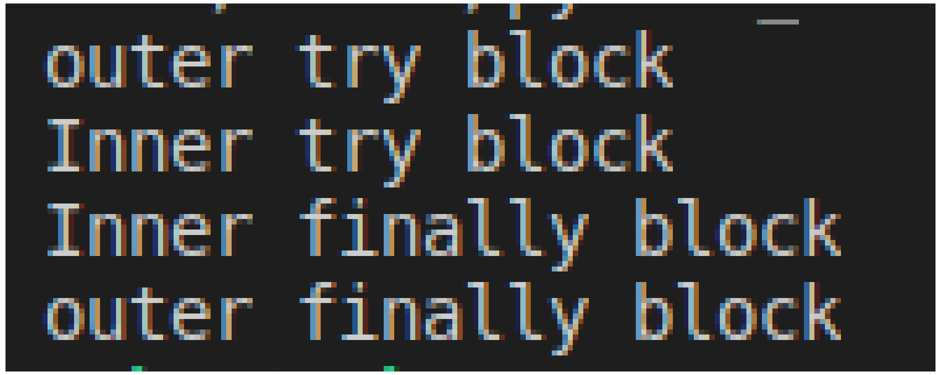
Different cases and scenarios

Case 1: If no exception raised then outer try, inner try, inner finally, outer finally blocks will get executed

Program: demo27.py

```
try:
  print("outer try block")
  try:
     print("Inner try block")
  except ZeroDivisionError:
     print("Inner except block")
  finally:
     print("Inner finally block")
except:
  print("outer except block")
finally:
  print("outer finally block")
```

Output:



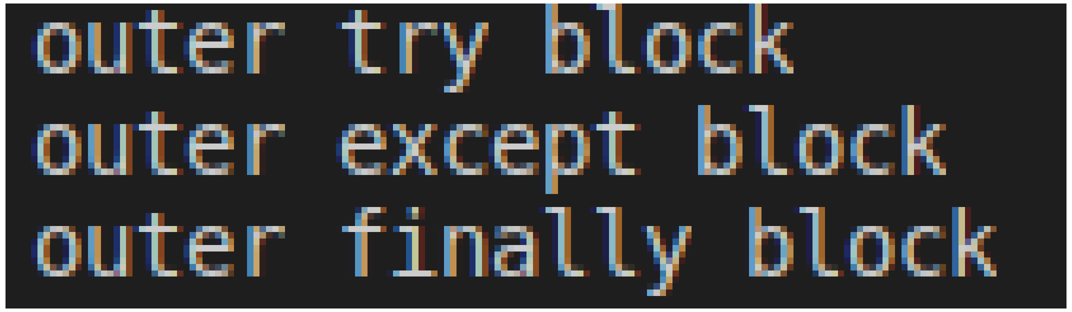
```
outer try block
Inner try block
Inner finally block
outer finally block
```

Case 2:
If an exception is raised in an outer try, then outer except blocks are responsible to handle that exception.

Program: demo28.py

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

Output:



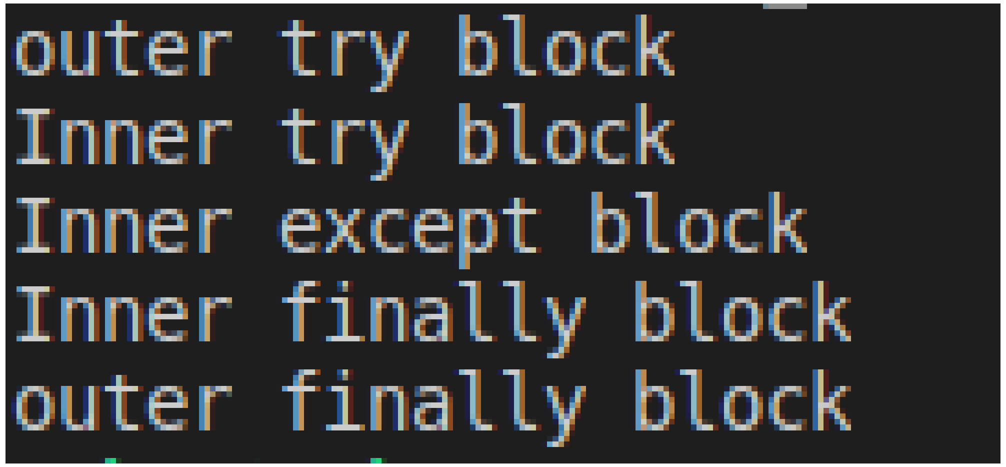
```
outer try block
outer except block
outer finally block
```

Case 3:
If an exception raised in inner try block then inner except block is responsible to handle, if it is unable to handle then outer except block is responsible to handle.

Program: Exception handled by inner except block (demo29.py)

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

Output:

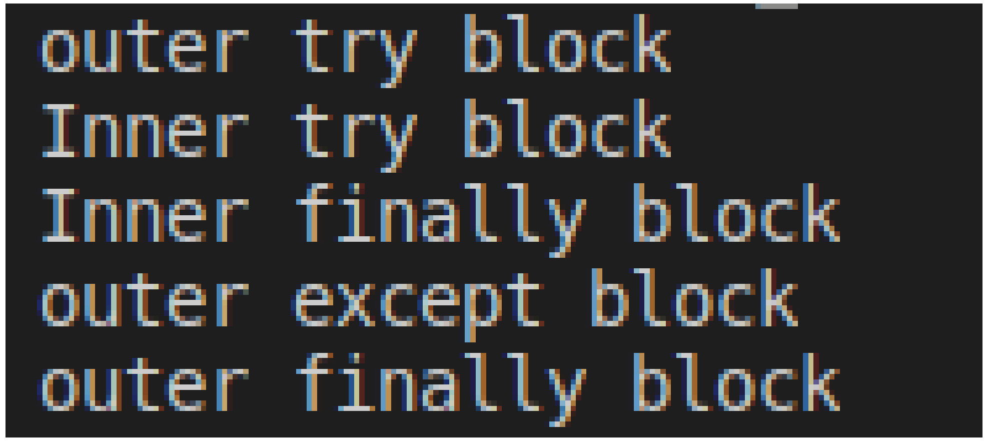


```
outer try block
Inner try block
Inner except block
Inner finally block
outer finally block
```

Program: Exception handled by outer except block (demo30.py)

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

Output:



```
outer try block
Inner try block
Inner finally block
outer except block
outer finally block
```

Else Block in Python:

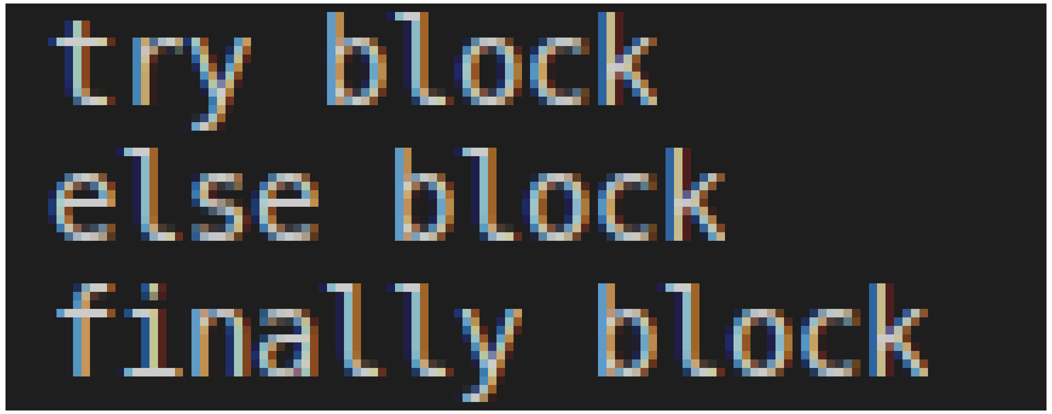
We can use else blocks with try-except-finally blocks. The else block will be executed if and only if there are no exceptions inside the try block.

Note: If no exception then try, else and finally blocks will get executed. If an exception is raised inside the try block, then the except block will get executed but the else block won’t.

Program: else block (demo31.py)

```
try:
   print("try block")
except:
   print("except: Handling code")
else:
   print("else block")
finally:
   print("finally block")
```

Output:

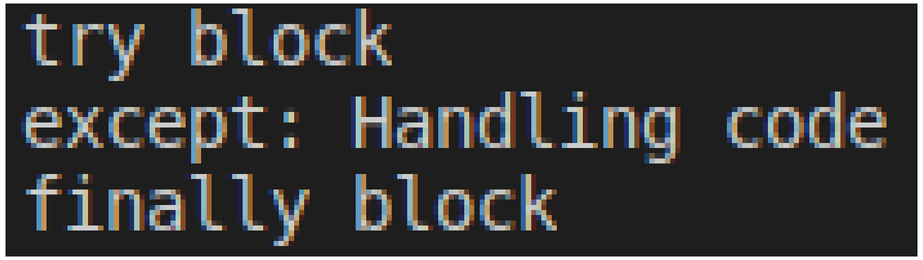


```
try block
else block
finally block
```

Program: else block (demo32.py)

```
try:
   print("try block")
   print(10/0)
except:
   print("except: Handling code")
else:
   print("else block")
finally:
   print("finally block")
```

Output:



```
try block
except: Handling code
finally block
```

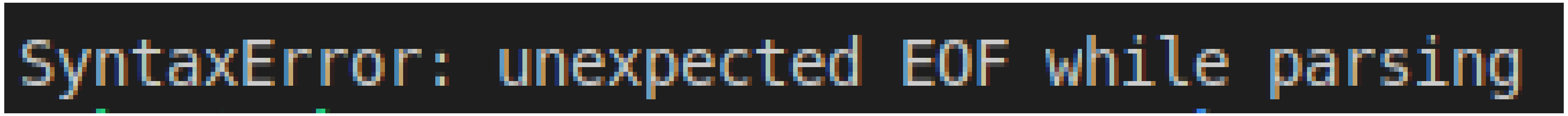
Possible Combinations with try-except-else-finally

Only try block is invalid, only except block is invalid, only finally block is invalid

Program: only try block (demo33.py)

```
try:
  print("try block")
```

Output:

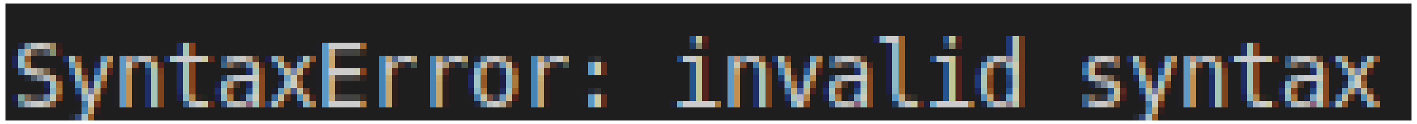


```
SyntaxError: unexpected EOF while parsing
```

Program: only except block (demo34.py)

```
except:
    print("except: Handling code")
```

Output:

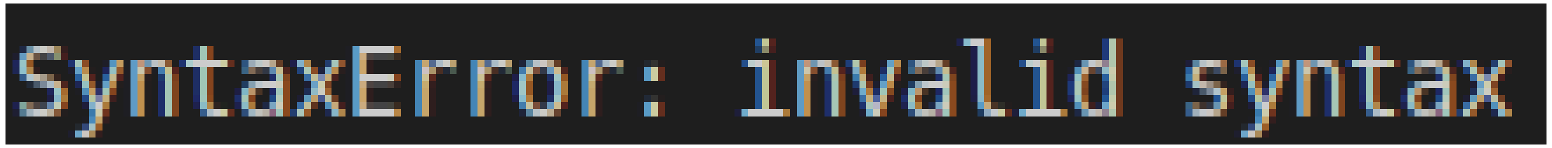


```
SyntaxError: invalid syntax
```

Program: Only Finally block (demo35.py)

```
finally:
    print("finally block")
```

Output:



```
SyntaxError: invalid syntax
```

Note:

1. try with except combination is valid → try block follows except block

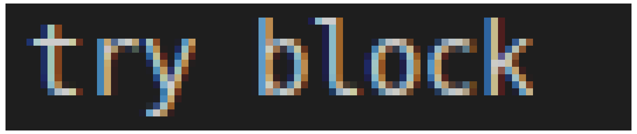
2. try with finally combination is valid → try block follows finally block

3. try, except and else combination is valid → try, except blocks follows else block

Program: try with except combination (demo36.py)

```
try:
    print("try block")
except:
    print("except: Handling code")
```

Output:

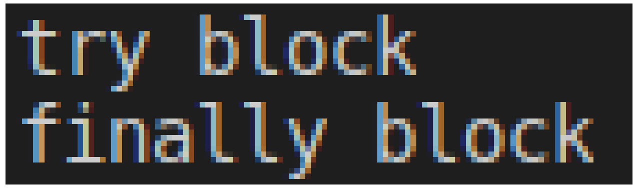


```
try block
```

Program: try with finally combination (demo37.py)

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

Output:

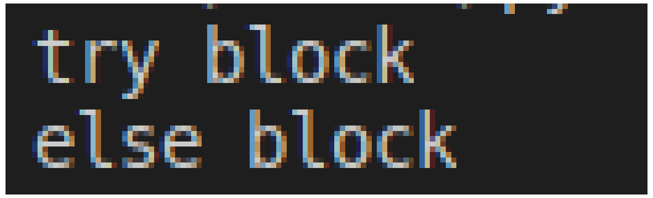


```
try block
finally block
```

Program: try, except and else combination (demo38.py)

```
try:
   print("try block")
except:
   print("except: Handling code")
else:
   print("else block")
```

Output:



```
try block
else block
```

Custom Exception in Python

Predefined Exceptions in Python

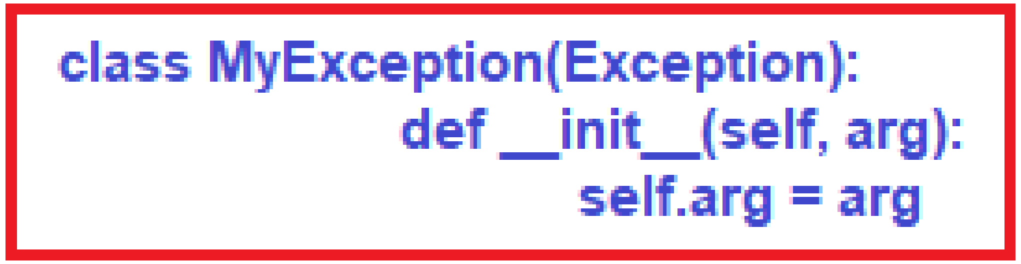
The exceptions which we handled in the examples till now are predefined ones. They are already available built-in exceptions. If any exception occurs, then Python Virtual Machine will handle those exceptions using the predefined exceptions. Examples of predefined exceptions are ZeroDivisionError, ValueError, NameError etc.

Custom Exceptions in Python:

Sometimes based on project requirement, a programmer needs to create his own exceptions and raise explicitly for corresponding scenarios. Such types of exceptions are called customized Exceptions or Programmatic Exceptions. We can raise a custom exception by using the keyword ‘raise’.

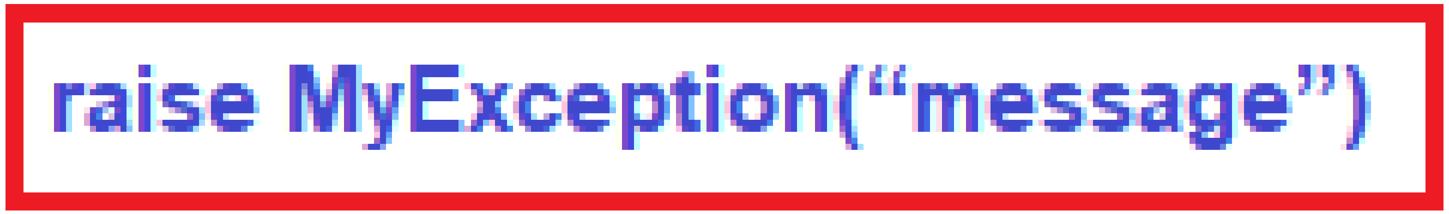
Steps to create Custom Exception in python:

The **first step** is to create a class for our exception. Since all exceptions are classes, the programmer is supposed to create his own exception as a class. The created class should be a child class of in-built “Exception” class.



```
class MyException(Exception):
        def __init__(self, arg):
            self.arg = arg
```

In the **second step** raise the exception where it required. When the programmer suspects the possibility of exception, then he can raise his own exception using raise keyword

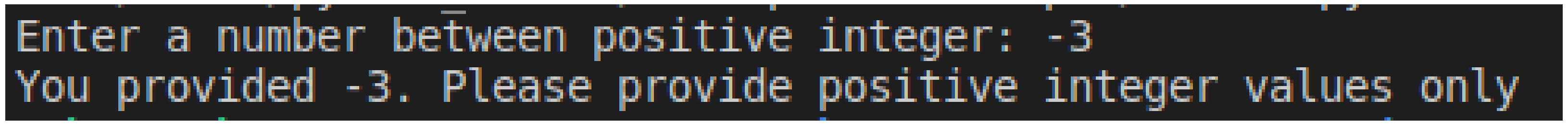


```
raise MyException(“message”)
```

Program: Creating Custom Exception in Python (demo39.py)

```
class NegativeError(Exception):
   def __init__(self, data):
      self.data = data
try:
   x = int(input("Enter a number between positive integer: "))
   if x < 0:
      raise NegativeError(x)
except NegativeError as e:
   print("You provided {}. Please provide positive integer values only".format(e))
```

Output:

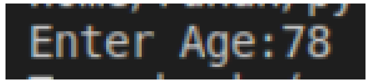


```
Enter a number between positive integer: -3
You provided -3. Please provide positive integer values only
```

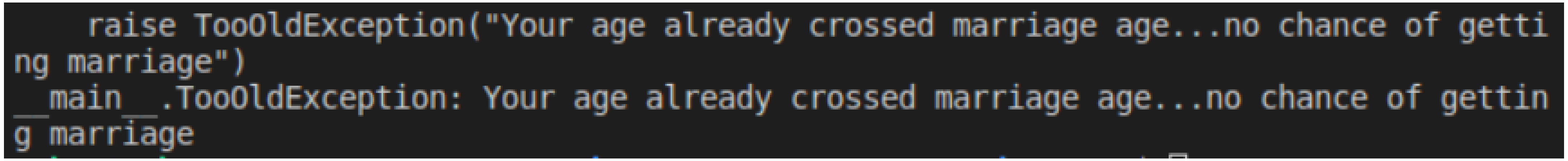
Program: Custom Exceptions in Python (demo40.py)

```
class TooYoungException(Exception):
   def __init__(self, arg):
       self.msg=arg
class TooOldException(Exception):
   def __init__(self, arg):
       self.msg=arg
age=int(input("Enter Age:"))
if age>60:
   raise TooOldException("Your age already crossed marriage age...no chance of getting marriage")
elif age<18:
   raise TooYoungException("Plz wait some more time you will get best match soon!!!")
else:
   print("You will get match details soon by email!!!")
```

Output:



```
Enter Age:78
```



```
    raise TooOldException("Your age already crossed marriage age...no chance of getti
ng marriage")
__main__.TooOldException: Your age already crossed marriage age...no chance of gettin
g marriage
```

THANK YOU !!!!