

Exception Handling in Python

The cause of an exception is often external to the program itself. For example, an incorrect input, a malfunctioning IO device etc. Because the program abruptly terminates on encountering an exception, it may cause damage to system resources, such as files. Hence, the exceptions should be properly handled so that an abrupt termination of the program is prevented.

Python uses `try` and `except` keywords to handle exceptions. Both keywords are followed by indented blocks.

Syntax:

```
try :  
    #statements in try block  
except :  
    #executed when error in try block
```

The `try:` block contains one or more statements which are likely to encounter an exception. If the statements in this block are executed without an exception, the subsequent `except:` block is skipped.

If the exception does occur, the program flow is transferred to the `except:` block. The statements in the `except:` block are meant to handle the cause of the exception appropriately. For example, returning an appropriate error message.

You can specify the type of exception after the `except` keyword. The subsequent block will be executed only if the specified exception occurs. There may be multiple `except` clauses with different exception types in a single `try` block. If the type of exception doesn't match any of the `except` blocks, it will remain unhandled and the program will terminate.

The rest of the statements after the `except` block will continue to be executed, regardless if the exception is encountered or not.

The following example will throw an exception when we try to divide an integer by a string.

Example: try...except blocks

[Copy](#)

```
try:  
    a=5  
    b='0'  
    print(a/b)  
except:  
    print('Some error occurred.')  
print("Out of try except blocks.")
```

Output

```
Some error occurred.  
Out of try except blocks.
```

You can mention a specific type of exception in front of the except keyword. The subsequent block will be executed only if the specified exception occurs. There may be multiple except clauses with different exception types in a single try block. If the type of exception doesn't match any of the except blocks, it will remain unhandled and the program will terminate.

Example: Catch Specific Error Type

Copy

```
try:
    a=5
    b='0'
    print (a+b)
except TypeError:
    print('Unsupported operation')
print ("Out of try except blocks")
```

Output

Unsupported operation
Out of try except blocks

As mentioned above, a single try block may have multiple except blocks. The following example uses two except blocks to process two different exception types:

Example: Multiple except Blocks

Copy


```
try:
    a=5
    b=0
    print (a/b)
except TypeError:
    print('Unsupported operation')
except ZeroDivisionError:
    print ('Division by zero not allowed')
print ('Out of try except blocks')
```

Output


Division by zero not allowed
Out of try except blocks

However, if variable b is set to '0', TypeError will be encountered and processed by corresponding except block.


NOW PLAYING



A Look Ahead at Xbox's Plans for 2022



Microsoft Gaming Executive, Says Netflix's Move Into...



GTA Publisher

'Baby Shark'...
Baby Shark Becomes 1st YouTube Video , To Surpass 10... [Watch Video](#)

A Look Ahead at...
A Look Ahead , at Xbox's Plans , for 2022.Kotaku reports that... [Watch Video](#)

Microsoft Gaming...
Microsoft Gaming Executive , Says Netflix's Move Into... [Watch Video](#)

GTA Publisher...
GTA Publisher

else and finally


In Python, keywords `else` and `finally` can also be used along with the `try` and `except` clauses. While the `except` block is executed if the exception occurs inside the `try` block, the `else` block gets processed if the `try` block is found to be exception free.

Syntax:

```
try:
    #statements in try block
except:
    #executed when error in try block
else:
    #executed if try block is error-free
finally:
    #executed irrespective of exception occurred or not
```

The `finally` block consists of statements which should be processed regardless of an exception occurring in the `try` block or not. As a consequence, the error-free `try` block skips the `except` clause and enters the `finally` block before going on to execute the rest of the code. If, however, there's an exception in the `try` block, the appropriate `except` block will be processed, and the statements in the `finally` block will be processed before proceeding to the rest of the code.

The example below accepts two numbers from the user and performs their division. It demonstrates the uses of `else` and `finally` blocks.

Example: try, except, else, finally blocks	 Copy
<pre>try: print('try block') x=int(input('Enter a number: ')) y=int(input('Enter another number: ')) z=x/y except ZeroDivisionError: print("except ZeroDivisionError block") print("Division by 0 not accepted") else: print("else block") print("Division = ", z) finally: print("finally block") x=0 y=0 print ("Out of try, except, else and finally blocks.")</pre>	

The first run is a normal case. The out of the `else` and `finally` blocks is displayed because the `try` block is error-free.

Output
<pre>try block Enter a number: 10 Enter another number: 2 else block Division = 5.0 finally block Out of try, except, else and finally blocks.</pre>

The second run is a case of division by zero, hence, the except block and the finally block are executed, but the else block is not executed.

Output
<pre>try block Enter a number: 10 Enter another number: 0 except ZeroDivisionError block Division by 0 not accepted finally block Out of try, except, else and finally blocks.</pre>

In the third run case, an uncaught exception occurs. The finally block is still executed but the program terminates and does not execute the program after the finally block.

Output
<pre>try block Enter a number: 10 Enter another number: xyz finally block Traceback (most recent call last): File "C:\python36\codes\test.py", line 3, in <module> y=int(input('Enter another number: ')) ValueError: invalid literal for int() with base 10: 'xyz'</pre>

Typically the finally clause is the ideal place for cleaning up the operations in a process. For example closing a file irrespective of the errors in read/write operations. This will be dealt with in the next chapter.

Raise an Exception

Python also provides the `raise` keyword to be used in the context of exception handling. It causes an exception to be generated explicitly. Built-in errors are raised implicitly. However, a built-in or custom exception can be forced during execution.

The following code accepts a number from the user. The try block raises a `ValueError` exception if the number is outside the allowed range.

Example: Raise an Exception	 Copy
<pre>try: x=int(input('Enter a number upto 100: ')) if x > 100: raise ValueError(x) except ValueError: print(x, "is out of allowed range") else: print(x, "is within the allowed range")</pre>	

Output
<pre>Enter a number upto 100: 200 200 is out of allowed range Enter a number upto 100: 50 50 is within the allowed range</pre>