Exception Handling

Exception, Exception Handling, Except clause, Try? finally clause, User Defined Exceptions.

What is a Python Exception?

An **exception** can be defined as an **unusual condition** in a program resulting in the **interruption in the flow of the program.**
Whenever an exception occurs, the program stops the execution, and thus the further code is not executed. Therefore, an **exception is the run-time errors that** a Python script is unable to handle. An exception is a Python object that represents an error.
An exception may cause the program to stop if it is not properly "caught" (i.e. handled correctly). If you think that your program might raise an exception when executed, you will find it useful to use **`try/except`** to handle them.

Difference between Syntax Error and Exceptions

Syntax Error: As the name suggests this error is caused by wrong syntax in the code. It leads to the termination of the program.
Example:

```
# initialize the amount variable
amount = 10000
 # check that You are eligible to  purchase the item
if(amount>2999)
   print("You are eligible to purchase the item")
```

Output: if (a>2999)
^
SyntaxError: invalid syntax

Exceptions: Exceptions are raised when the program is syntactically correct but the code resulted in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.
Example:

```
# initialize the amount variable
marks = 10000
b=0
  # perform division with 0
a = marks / b
print(a)
```

Output: *Traceback (most recent call last):*
File "<stdin>", line 1, in <module>
***ZeroDivisionError**: division by zero*

Another example

In the example below, we prompt the user to enter a number, and after execution the program gives out the value of the given number squared. The program [**squ.py**] will work as long as you enter a number as your input.

```
while True:

      x = int(raw_input("Please enter a number: "))
      print("%s squared is %s" % (x, x**2))
```

When you enter a string, or any other character that is not a number, like "one", the program will raise a "Value Error" exception, as shown below:

Output:

```
python squ.py
Please enter a number: 3
3 squared is 9
Please enter a number: 4
4 squared is 16
Please enter a number: five
Traceback (most recent call last):
  File "squared.py", line 4, in <module>
 x = int(raw_input("Please enter a number: "))
ValueError: invalid literal for int() with base 10: 'five':
```

This is where handling exceptions comes in. You need to know how to properly handle exceptions, especially in production environments, because if they are not handled your program won't know what to do and will crash.

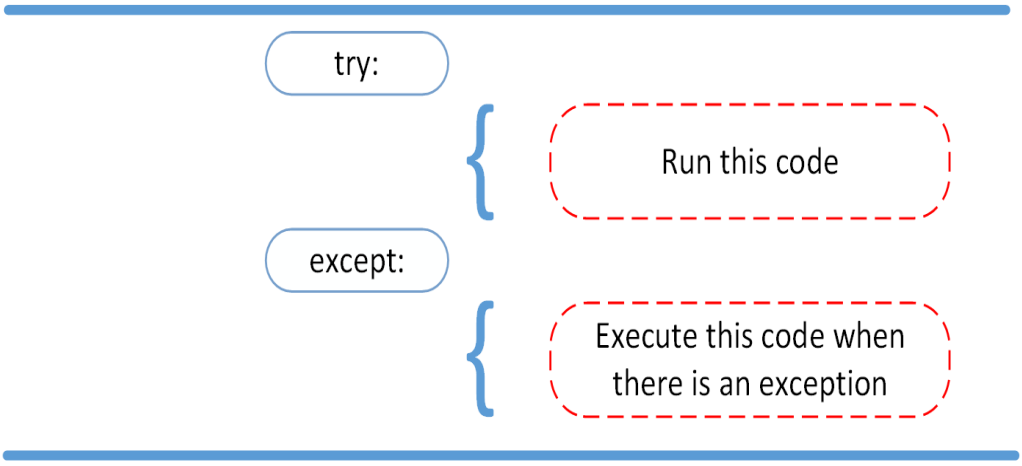
Python uses try and except keywords to handle exceptions. Both keywords are followed by indented blocks.

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 or may generate an exception. If the statements in this block are executed **without an exception**, the subsequent **except: block is skipped**.

If the **exception occurs**, 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 **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.

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

Let's rewrite the same program [**squ.py**], and this time take care of any exceptions that may arise.

```
while True:
      try:
           x = int(raw_input("Please enter a number: "))
            print("%s squared is %s" % (x, x**2))
      except ValueError:
            print("Please enter a valid number!")
```

As you can see from the above program, the code inside the try statement is executed first. If no error occurs, the rest of the code is executed and the except statement is skipped.
In the event that an exception is raised (like when entering a string instead of a number), the program executes the except statement that matches its try statement, followed by the rest of the code outside of the try/except block, if any.

Output:

```
python squ.py
Please enter a number: 3
3 squared is 9
Please enter a number: 4
4 squared is 16
Please enter a number: five
Please enter a valid number!
Please enter a number: 5
5 squared is 25
```

Common Exceptions

Python provides the number of built-in exceptions, but here we are describing the common standard exceptions so that you can get a perspective of what they are and how to use them.

KeyError Exception

This is an error that occurs when using a dictionary and you try to retrieve a key that is not present in the dictionary.

The following is a code used to authenticate users and it expects to receive a dictionary containing the keys "username" and "password". If any of the keys is not passed in the data, the program will raise a Key Error exception.

```
def authenticate_user(request):
      try:
            mail = request.data['username']
            word = request.data['password']
            user = authenticate(email=mail, password=word)
```

```
                if user and user.is_active:
                     return Response(user_details,status=HTTP_200_OK)
                 else:
                     res={'error':'cannot authenticate with the given credentials'}
                      return Response(res, status=status.HTTP_400_BAD_REQUEST)
          except KeyError:
                res = {'error': 'please provide a username and a password'}
```

By using **try/except** to catch the exception, our user will see the error instead of crashing our server.

IndentationError Exception

This kind of exception is raised when your code is not properly indented. Instead of using curly braces for specifying scope, Python uses indentation, so it is important to get the indentation correct, which is the purpose of this exception.

Let's write a function that checks if a number is even:

```
def is_even(number):
      if number % 2 == 0:
      print(" %s is an even number" % (number))
      else:
      print("%s is an odd number" % (number))

print(is_even(1))
print(is_even(2))
```

The function will result in an Indentation Error on both lines 2 and 4 because it expects the code to be indented with 8 spaces, not 4.
If you execute the program, it will give the following output:

```
$ python is_even.py
 File "is_even.py", line 3
   print(" %s is an even number" % (number))
      ^
IndentationError: expected an indented block
```

SyntaxError Exception

This is one of the most common types of exceptions in Python. It occurs when there is an error in the syntax of your code.

Let's look at the example below:

```
def is_odd(n):
       if n % 2 != 0:
            print(it's odd)
       else:
            print("not odd")

print(is_odd(7))
```

The output of this code will result in an invalid syntax at line 3 because the string "it's odd" is not enclosed in quotes.

```
$ python is_odd.py
  File "is_odd.py", line 3
 print(it's odd)
            ^
SyntaxError: EOL while scanning string literal
```

Keep in mind that there are many different types of syntax errors, and this is just one example.

TypeError Exception

This exception is raised when an operation or function is attempted that is invalid for the specified data type. In the example below, the function **sum_of_numbers** takes in 2 arguments and adds them together.

When you attempt to call the function with 3 arguments, it raises a TypeError exception because it expects only 2 arguments.

```
def sum_of_numbers(a, b):
       return a + b

print(sum_of_numbers(1, 2, 7))
```

Running this code yields the following result:

```
$ python sum_of_numbers.py
      Traceback (most recent call last):
       File "sum_of_numbers.py", line 4, in <module>
       print(sum_of_numbers(1, 2, 7))
      TypeError: sum_of_numbers() takes exactly 2 arguments (3 given)
```

1. **ZeroDivisionError:** Occurs when a number is divided by zero.
2. **NameError:** It occurs when a name is not found. It may be local or global.
3. **IndentationError:** If incorrect indentation is given.
4. **IOError:** It occurs when Input Output operation fails.
5. **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

List of Standard Exceptions –

Sr.No	***Exception Name & Description***
1	**Exception**: Base class for all exceptions
2	**StopIteration**: Raised when the next() method of an iterator does not point to any object.
3	**SystemExit**: Raised by the sys.exit() function.

4	**StandardError**: Base class for all built-in exceptions except StopIteration and SystemExit.
5	**ArithmeticError:** Base class for all errors that occur for numeric calculation.
6	**OverflowError:** Raised when a calculation exceeds the maximum limit for a numeric type.
7	**FloatingPointError:**Raised when a floating point calculation fails.
8	**ZeroDivisionError:** Raised when division or modulo by zero takes place for all numeric types.
9	**AssertionError:** Raised in case of failure of the Assert statement.
10	**AttributeError :**Raised in case of failure of attribute reference or assignment.
11	**EOFError:** Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
12	**ImportError:** Raised when an import statement fails.
13	**KeyboardInterrupt:** Raised when the user interrupts program execution, usually by pressing Ctrl+c.
14	**LookupError:** Base class for all lookup errors.
15	**IndexError:** Raised when an index is not found in a sequence.
16	**KeyError:** Raised when the specified key is not found in the dictionary.

17	**NameError:** Raised when an identifier is not found in the local or global namespace.
18	**UnboundLocalError:** Raised when trying to access a local variable in a function or method but no value has been assigned to it.
19	**EnvironmentError:** Base class for all exceptions that occur outside the Python environment.
20	**IOError:** Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
21	**IOError:** Raised for operating system-related errors.
22	**SyntaxError:** Raised when there is an error in Python syntax.
23	**IndentationError:** Raised when indentation is not specified properly.
24	**SystemError:** Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exist.
25	**SystemExit:** Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
26	**TypeError:** Raised when an operation or function is attempted that is invalid for the specified data type.
27	**ValueError :**Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
28	**RuntimeError:** Raised when a generated error does not fall into any category.
29	**NotImplementedError:** Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

Python Multiple Excepts

It is possible to have multiple except blocks for one try block. Let us see Python multiple exception handling examples.

```
>>> a,b=1,2
>>> try:
            print(a/b)
            print("This won't be printed")
            print('10'+10)
except TypeError:
             print("You added values of incompatible types")
except ZeroDivisionError:
            print("You divided by 0")
```

Output

```
You added values of incompatible types
```

When the interpreter encounters an exception, it checks the except blocks associated with that try block.
These except blocks may declare what kind of exceptions they handle. When the interpreter finds a matching exception, it executes that except block.
In our example, the first statement under the try block gave us a ZeroDivisionError.
We handled this in its except block, but the statements in try after the first one didn't execute.
This is because once an exception is encountered, the statements after that in the try block are skipped.
And if an appropriate except block or a generic except block isn't found, the exception isn't handled.
In this case, the rest of the program won't run. But if you handle the exception, the code after the excepts and the finally block will run as expected.
Let's try some code for this.

```
a,b=1,0
try:
      print(a/b)
except:
      print("You can't divide by 0")


print("Will this be printed?")
```

Output

```
You can’t divide by 0
```

```
>>>
```

2. Python Multiple Exception in one Except

You can also have one except block handle multiple exceptions. To do this, use parentheses. Without that, the interpreter will return a syntax error.

```
>>> try:
      print('10'+10)
      print(1/0)
except (TypeError,ZeroDivisionError):


      print("Invalid input")
```

Output

```
Invalid input
```

3. A Generic except After All Excepts

Finally, you can compliment all specific except blocks with a generic except at the end.

This block will serve to handle all exceptions that go undetected by the specific except blocks.

```
>>> try:
           print('1'+1)
           print(sum)
           print(1/0)
except NameError:
           print("sum does not exist")
except ZeroDivisionError:
           print("Cannot divide by 0")
except:


           print("Something went wrong")
```

Output

```
Something went wrong
```

Here, the first statement under the try block tries to concatenate a string to an int. This raises a TypeError.

As the interpreter comes across this, it checks for an appropriate except block that handles this.

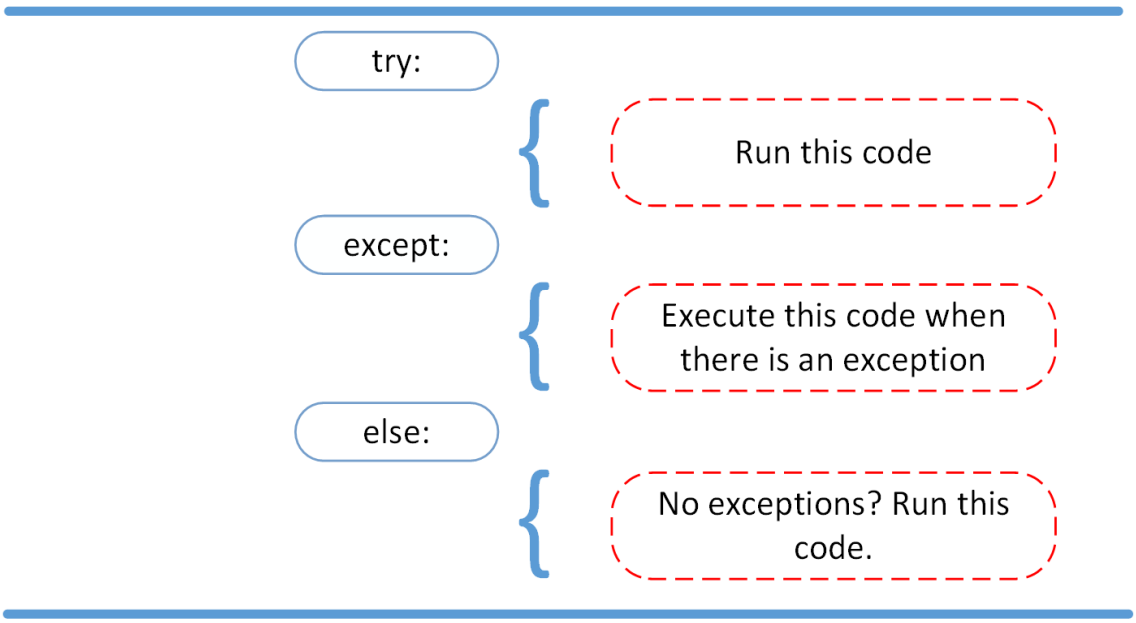
Also, you **can't put a statement between try and catch blocks.**

```
try:
   print("1")
print("2")
except:


   print("3")
```

This gives you a syntax error.

In Python, using the **`else`** statement, you can instruct a program to execute a certain block of code only in the absence of exceptions.



In some situations, you might want to run a certain block of code if the code block inside `try` ran without any errors. For these cases, you can use the optional `else` keyword with the `try` statement.

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

Let's look at an example:

```
# 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)
```

Output

If we pass an odd number:

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

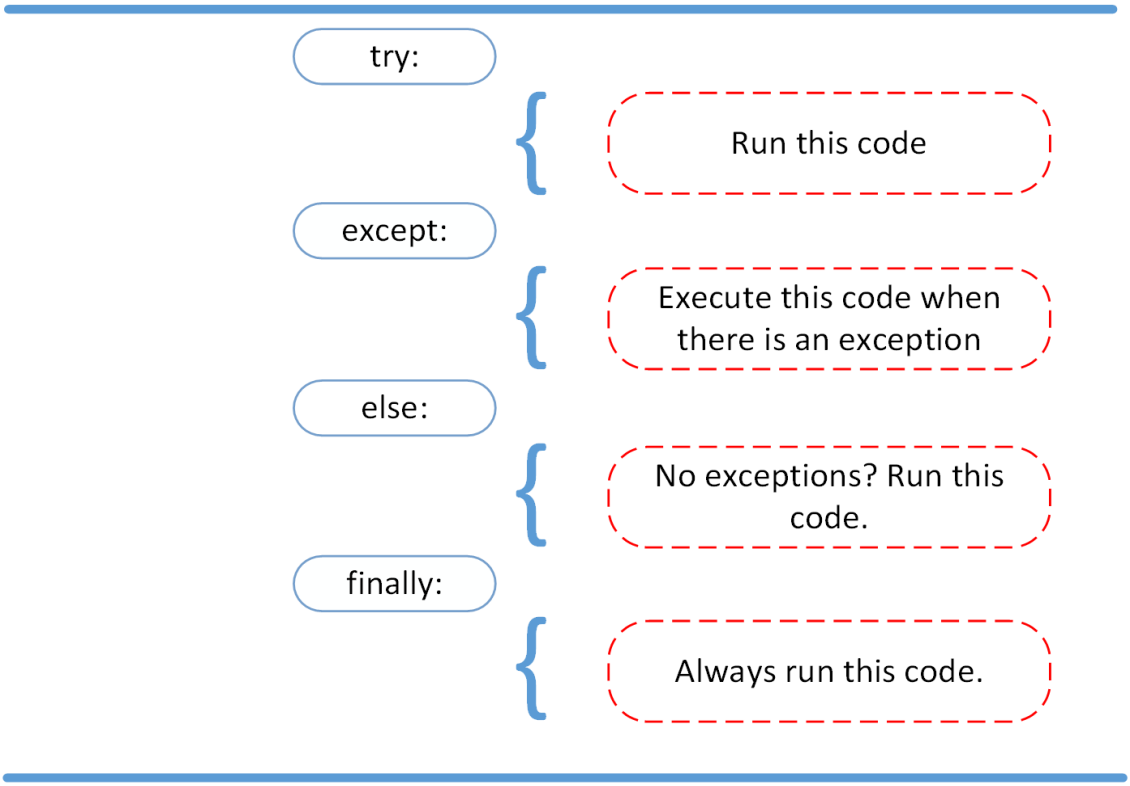
If we pass an even number, the reciprocal is computed and displayed.

```
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
```

Imagine that you always had to implement some sort of action to clean up after executing your code. Python enables you to do so using the `finally` clause.



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

```
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." )
```

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

```
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.
```

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

```
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.
```

In the **third run** case, an u**ncaught exception occur**s. The finally block is still executed but the program terminates and does not execute the program after the finally block.

Output

```
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'
```

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.

```
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")
```

```
Output
Enter a number upto 100: 200 200 is out of allowed range Enter a number upto 100: 50 50 is within the allowed range
```

Here, the raised exception is a `ValueError` type. However, you can define your custom exception type to be raised. Visit Python docs to know more about user defined exceptions.

Assertions in Python

It takes an expression as an argument and raises a python exception if the expression has a False Boolean value. Otherwise, it performs a No-operation (NOP).

```
>>> assert(True)
>>>
```

Now what if the expression was False?

```
>>> assert(False)
      Traceback (most recent call last):
        File "<stdin>", line 1, in <module>
      AssertionError
>>> assert(1==0)
```

Output

```
Traceback (most recent call last):File "<pyshell#157>", line 1, in <module>
assert(1==0)
AssertionError
```

Let's take another example, and let's create a .py file for that.

```
try:
        print(1)
        assert 2+2==4
        print(2)
```

```
        assert 1+2==4
        print(3)
except:
        print("An assert failed.")
        raise
finally:
        print("Okay")
```

Output

```
2
```

An assert failed.

Okay

Output

```
Traceback (most recent call last):File
"C:\Users\lifei\AppData\Local\Programs\Python\Python36-32\try2.py", line 5, in
<module>


assert 1+2==4


AssertionError
```

Interestingly, if you remove the raise from under the except block, this is the output:

```
= RESTART: C:\Users\lifei\AppData\Local\Programs\Python\Python36-32\try2.py =1


2
```

An assert failed.

Okay

Bye

This is because when we ‘raise’ an exception, we aren’t provisioning a handle for it.

We can use assertions to check for valid input and output to functions.

1. A Second Argument to assert

You may optionally provide a second argument to give out some extra information about the problem.

```
>>> assert False,"That's a problem"
```

Output

```
Traceback (most recent call last):File “<pyshell#173>”, line 1, in <module>


assert False,”That’s a problem”


AssertionError: That’s a problem
```

Defining Your Own Exceptions in Python

Finally, we’ll talk about creating our own exceptions. For this, we derive a new class from the Exception class.

Later, we call it like any other exception.

```
>>> class MyError(Exception):
print("This is a problem")


>>> raise MyError("MyError happened")
```

Output

```
Traceback (most recent call last):File “<pyshell#179>”, line 1, in <module>


raise MyError(“MyError happened”)


MyError: MyError happened
```