# Python\_How to Best Use Try Except

In Python programming, exception handling allows a programmer to enable flow control. And it has no. of built-in exceptions to catch errors in case your code breaks. Using try-except is the most common and natural way of handling unexpected errors along with many more exception handling constructs. In this tutorial, you’ll get to explore some of the best techniques to use try-except in Python.

**Error Handling or Exception Handling** in Python can be enforced by setting up exceptions. Using a try block, you can implement an exception and handle the error inside an except block. Whenever the code breaks inside a try block, the regular code flow will stop and the control will get switched to the except block for handling the error.

**Why use Try-Except/Try-Except-else Clause?** With the help of try-except and try-except-else, you can avoid many unknown problems that could arise from your code. For example, the Python code using LBYL (Look before you leap) style can lead to race conditions. Here, the try-except clause can come to rescue you. Also, there are cases where your code depends critically on some information which could get outdated till the time you receive it. For example, the code making calls to os.path.exists or Queue.full may fail as these functions might return data that become stale by the time you use it. The wiser choice here would be to follow the try-except-else style in your code to manage the above cases more reliably.

**Raising exceptions** is also permissible in Python. It means you can throw or raise an exception whenever it is needed. You can do it simply by calling [raise Exception(‘Test error!’)] from your code. Once raised, the exception will stop the current execution as usual and will go further up in the call stack until handled.

**Why use Exceptions?** They not only help solve popular problems like race conditions but are also very useful in controlling errors in areas like loops, file handling, database communication, network access and so on.

Hence, we’ll cover broader problems and provide solutions in this post. Please note that exception handling is an art which brings you immense powers to write robust and quality code. So, brace to read some keynotes on exceptions along with the best ways to handle them.

### **1. How To Handle An Arbitrary Exception**

Sometimes, you may need a way to allow any arbitrary exception and also want to be able to display the error or exception message.

It is easily achievable using the Python exceptions. Check the below code. While testing, you can place the code inside the try block in the below example.

try:

#your code

except Exception as ex:

print(ex)

### **2. Catch Multiple Exceptions In One Except Block**

You can catch multiple exceptions in a single except block. See the below example.

except (Exception1, Exception2) as e:

pass

Please note that you can separate the exceptions from the variable with a comma which is applicable in Python 2.6/2.7. But you can’t do it in Python 3. So, you should prefer to use the [as] keyword.

### **3. Handling Multiple Exceptions With One Except Block**

There are many ways to handle multiple exceptions. The first of them requires placing all the exceptions which are likely to occur in the form of a tuple. Please see from below.

try:

file = open('input-file', 'open mode')

except (IOError, EOFError) as e:

print("Testing multiple exceptions. {}".format(e.args[-1]))

The next method is to handle each exception in a dedicated except block. You can add as many except blocks as needed. See the below example.

try:

file = open('input-file', 'open mode')

except EOFError as ex:

print("Caught the EOF error.")

raise ex

except IOError as e:

print("Caught the I/O error.")

raise ex

The last but not the least is to use the except without mentioning any exception attribute.

try:

file = open('input-file', 'open mode')

except:

# In case of any unhandled error, throw it away

raise

This method can be useful if you don’t have any clue about the exception possibly thrown by your program.

### **4. Re-Raising Exceptions In Python**

Exceptions once raised keep moving up to the calling methods until handled. Though you can add an except clause which could just have a [raise] call without any argument. It’ll result in reraising the exception.

See the below example code.

try:

# Intentionally raise an exception.

raise Exception('I learn Python!')

except:

print("Entered in except.")

# Re-raise the exception.

raise

#### **Output:**

Entered in except.

Traceback (most recent call last):

File "python", line 3, in <module>

Exception: I learn Python!

### **5. When To Use The Else Clause**

Use an else clause right after the try-except block. The else clause will get hit only if no exception is thrown. The else statement should always precede the except blocks.

In else blocks, you can add code which you wish to run when no errors occurred.

See the below example. In this sample, you can see a while loop running infinitely. The code is asking for user input and then parsing it using the built-in [int()] function. If the user enters a zero value, then the except block will get hit. Otherwise, the code will flow through the else block.

while True:

# Enter integer value from the console.

x = int(input())

# Divide 1 by x to test error cases

try:

result = 1 / x

except:

print("Error case")

exit(0)

else:

print("Pass case")

exit(1)

### **6. Make Use Of [Finally Clause]**

If you have a code which you want to run in all situations, then write it inside the [finally block]. Python will always run the instructions coded in the [finally block]. It is the most common way of doing clean up tasks. You can also make sure the clean up gets through.

An error is caught by the try clause. After the code in the except block gets executed, the instructions in the [finally clause] would run.

Please note that a [finally block] will ALWAYS run, even if you’ve returned ahead of it.

See the below example.

try:

# Intentionally raise an error.

x = 1 / 0

except:

# Except clause:

print("Error occurred")

finally:

# Finally clause:

print("The [finally clause] is hit")

#### **Output:**

Error occurred

The [finally clause] is hit

### **7. Use The As Keyword To Catch Specific Exception Types**

With the help of as <identifier>, you can create a new object. And you can also the exception object. Here, the below example, we are creating the IOError object and then using it within the clause.

try:

# Intentionally raise an error.

f = open("no-file")

except IOError as err:

# Creating IOError instance for book keeping.

print("Error:", err)

print("Code:", err.errno)

#### **Output:**

('Error:', IOError(2, 'No such file or directory'))

('Code:', 2)

### **8. Best Practice For Manually Raising Exceptions**

Avoid raising generic exceptions because if you do so, then all other more specific exceptions have to be caught also. Hence, the best practice is to raise the most specific exception close to your problem.

#### **Bad Example.**

def bad\_exception():

try:

raise ValueError('Intentional - do not want this to get caught')

raise Exception('Exception to be handled')

except Exception as error:

print('Inside the except block: ' + repr(error))

bad\_exception()

**Output:**

Inside the except block: ValueError('Intentional - do not want this to get caught',)

#### **Best Practice:**

Here, we are raising a specific type of exception, not a generic one. And we are also using the args option to print the incorrect arguments if there is any. Let’s see the below example.

try:

raise ValueError('Testing exceptions: The input is in incorrect order', 'one', 'two', 'four')

except ValueError as err:

print(err.args)

**Output:**

('Testing exceptions: The input is in incorrect order', 'one', 'two', 'four')

**9. How To Skip Through Errors And Continue Execution**

Ideally, you shouldn’t be doing this. But if you still want to do, then follow the below code to check out the right approach.

try:

assert False

except AssertionError:

pass

print('Welcome to Prometheus!!!')

#### **Output:**

Welcome to Prometheus!!!

Now, have a look at some of the most common Python exceptions and their examples.

### **Most Common Exception Errors**

* **IOError –** It occurs on errors like a file fails to open.
* **ImportError –** If a python module can’t be loaded or located.
* **ValueError –** It occurs if a function gets an argument of right type but an inappropriate value.
* **KeyboardInterrupt –** It gets hit when the user enters the interrupt key (i.e. Control-C or Del key)
* **EOFError –** It gets raised if the input functions (input()/raw\_input()) hit an end-of-file condition (EOF) but without reading any data.

### **Examples Of Most Common Exceptions**

except IOError:

print('Error occurred while opening the file.')

except ValueError:

print('Non-numeric input detected.')

except ImportError:

print('Unable to locate the module.')

except EOFError:

print('Identified EOF error.')

except KeyboardInterrupt:

print('Wrong keyboard input.')

except:

print('An error occurred.')

## Summary – How To Best Use Try-Except In Python

while programming, errors are bound to happen. It’s a fact which no one can ignore. And there could be many reasons for errors like bad user input, insufficient file permission, the unavailability of a network resource, insufficient memory or most likely the programmer’s mistake.

Anyways, all of this can be handled if your code use exception handling and implement it with constructs like try-except, or tr-except-else, try-except-finally. Hope, you would have enjoyed reading the above tutorial.

If you liked the post, then please don’t miss to share it with friends and on social media (**[facebook](https://www.facebook.com/TechBeamersWorld/)**/[**twitter**](https://twitter.com/TechBeamers)).

<https://www.techbeamers.com/use-try-except-python/>