# **Python Exception Handling**

Error in Python can be of two types i.e. [Syntax errors and Exceptions](https://www.geeksforgeeks.org/errors-and-exceptions-in-python/). Errors are problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which change the normal flow of the program.

### **DiffError in Python can be of two types i.e.** [**Syntax errors and Exceptions**](https://www.geeksforgeeks.org/errors-and-exceptions-in-python/)**. Errors are problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which change the normal flow of the program.**

### **Different types of exceptions in python:**

### **In Python, there are several built-in Python exceptions that can be raised when an error occurs during the execution of a program. Here are some of the most common types of exceptions in Python:**

### **SyntaxError: This exception is raised when the interpreter encounters a syntax error in the code, such as a misspelled keyword, a missing colon, or an unbalanced parenthesis.**

### **TypeError: This exception is raised when an operation or function is applied to an object of the wrong type, such as adding a string to an integer.**

### **NameError: This exception is raised when a variable or function name is not found in the current scope.**

### **IndexError: This exception is raised when an index is out of range for a list, tuple, or other sequence types.**

### **KeyError: This exception is raised when a key is not found in a dictionary.**

### **ValueError: This exception is raised when a function or method is called with an invalid argument or input, such as trying to convert a string to an integer when the string does not represent a valid integer.**

### **AttributeError: This exception is raised when an attribute or method is not found on an object, such as trying to access a non-existent attribute of a class instance.**

### **IOError: This exception is raised when an I/O operation, such as reading or writing a file, fails due to an input/output error.**

### **ZeroDivisionError: This exception is raised when an attempt is made to divide a number by zero.**

### **ImportError: This exception is raised when an import statement fails to find or load a module.**

### **These are just a few examples of the many types of exceptions that can occur in Python. It’s important to handle exceptions properly in your code using try-except blocks or other error-handling techniques, in order to gracefully handle errors and prevent the program from crashing.**

## **Difference between Syntax Error and Exceptions**

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

### **Example:**

### **There is a syntax error in the code . The ‘if' statement should be followed by a colon (:), and the ‘print' statement should be indented to be inside the ‘if' block.**

### **amount = 10000**

### **if(amount > 2999)**

### **print("You are eligible to purchase Dsa Self Paced")**

### **Output:**

### 

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

### **Example:**

### **Here in this code a s we are dividing the ‘marks’ by zero so a error will occur known as ‘ZeroDivisionError’**

### 

### **marks = 10000**

### **a = marks / 0**

### **print(a)**

### 

### 

### **Output:**

### 

### **In the above example raised the ZeroDivisionError as we are trying to divide a number by 0.**

### **Note: Exception is the base class for all the exceptions in Python. You can check the exception hierarchy** [**here**](https://docs.python.org/2/library/exceptions.html#exception-hierarchy)**.**

### **Example:**

### **1) TypeError: This exception is raised when an operation or function is applied to an object of the wrong type. Here’s an example: Here a ‘TypeError’ is raised as both the datatypes are different which are being added.**

### 

### **x = 5**

### **y = "hello"**

### **z = x + y**

### **z**

### **output:**

### **Traceback (most recent call last):**

### **File "7edfa469-9a3c-4e4d-98f3-5544e60bff4e.py", line 4, in <module>**

### **z = x + y**

### **TypeError: unsupported operand type(s) for +: 'int' and 'str'**

#### **try catch block to resolve it:**

### **The code attempts to add an integer (‘x') and a string (‘y') together, which is not a valid operation, and it will raise a ‘TypeError'. The code used a ‘try' and ‘except' block to catch this exception and print an error message.**

### **x = 5**

### **y = "hello"**

### **try:**

### **z = x + y**

### **except TypeError:**

### **print("Error: cannot add an int and a str")**

### 

### **Output**

### **Error: cannot add an int and a str**

### 

## **Try and Except Statement – Catching Exceptions**

### **Try and except statements are used to catch and handle exceptions in Python. Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause.**

### **Example: Here we are trying to access the array element whose index is out of bound and handle the corresponding exception.**

The try block lets you test a block of code for errors.

The except block lets you handle the error.

The else block lets you execute code when there is no error.

The finally block lets you execute code, regardless of the result of the try- and except blocks.

### **Python3**

| **a = [1, 2, 3]****try:** **print ("Second element = %d" %(a[1]))** **print ("Fourth element = %d" %(a[3]))****except:** **print ("An error occurred")** |
| --- |

### **Output**

### **Second element = 2**

### **An error occurred**

### 

### **In the above example, the statements that can cause the error are placed inside the try statement (second print statement in our case). The second print statement tries to access the fourth element of the list which is not there and this throws an exception. This exception is then caught by the except statement.**

## **Catching Specific Exception**

### **A try statement can have more than one except clause, to specify handlers for different exceptions. Please note that at most one handler will be executed. For example, we can add IndexError in the above code. The general syntax for adding specific exceptions are –**

### **try:**

### **# statement(s)**

### **except IndexError:**

### **# statement(s)**

### **except ValueError:**

### **# statement(s)**

### **Example: Catching specific exceptions in the Python**

### **The code defines a function ‘fun(a)' that calculates b based on the input a. If a is less than 4, it attempts a division by zero, causing a ‘ZeroDivisionError'. The code calls fun(3) and fun(5) inside a try-except block. It handles the ZeroDivisionError for fun(3) and prints “ZeroDivisionError Occurred and Handled.” The ‘NameError' block is not executed since there are no ‘NameError' exceptions in the code.**

### 

### **Output**

### **ZeroDivisionError Occurred and Handled**

### 

### **If you comment on the line fun(3), the output will be**

### **NameError Occurred and Handled**

### **The output above is so because as soon as python tries to access the value of b, NameError occurs.**

## **Try with Else Clause**

### **In Python, you can also use the else clause on the try-except block which must be present after all the except clauses. The code enters the else block only if the try clause does not raise an exception.**

### **Try with else clause**

### **The code defines a function AbyB(a, b) that calculates c as ((a+b) / (a-b)) and handles a potential ZeroDivisionError. It prints the result if there’s no division by zero error. Calling AbyB(2.0, 3.0) calculates and prints -5.0, while calling AbyB(3.0, 3.0) attempts to divide by zero, resulting in a ZeroDivisionError, which is caught and “a/b results in 0” is printed.**

### 

### **Output:**

### **-5.0**

### **a/b result in 0**

## **Finally Keyword in Python**

### **Python provides a keyword** [**finally**](https://www.geeksforgeeks.org/finally-keyword-in-python/)**, which is always executed after the try and except blocks. The final block always executes after the normal termination of the try block or after the try block terminates due to some exception.**

### **Syntax:**

### **try:**

### **# Some Code....**

### 

### **except:**

### **# optional block**

### **# Handling of exception (if required)**

### 

### **else:**

### **# execute if no exception**

### 

### **finally:**

### **# Some code .....(always executed)**

### **Example:**

### **The code attempts to perform integer division by zero, resulting in a ZeroDivisionError. It catches the exception and prints “Can’t divide by zero.” Regardless of the exception, the finally block is executed and prints “This is always executed.”**

### **Output:**

### **Can't divide by zero**

### **This is always executed**

## **Raising Exception**

### **The** [**raise statement**](https://www.geeksforgeeks.org/python-raising-an-exception-to-another-exception/) **allows the programmer to force a specific exception to occur. The sole argument in raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception).**

### **This code intentionally raises a NameError with the message “Hi there” using the raise statement within a try block. Then, it catches the NameError exception, prints “An exception,” and re-raises the same exception using raise. This demonstrates how exceptions can be raised and handled in Python, allowing for custom error messages and further exception propagation.**

### **Python3**

| **try:** **raise NameError("Hi there")****except NameError:** **print ("An exception")** **raise** |
| --- |

### **The output of the above code will simply line printed as “An exception” but a Runtime error will also occur in the last due to the raise statement in the last line. So, the output on your command line will look like**

### **Traceback (most recent call last):**

### **File "/home/d6ec14ca595b97bff8d8034bbf212a9f.py", line 5, in <module>**

### **raise NameError("Hi there") # Raise Error**

### **NameError: Hi there**

### **Advantages of Exception Handling:**

### **Improved program reliability: By handling exceptions properly, you can prevent your program from crashing or producing incorrect results due to unexpected errors or input.**

### **Simplified error handling: Exception handling allows you to separate error handling code from the main program logic, making it easier to read and maintain your code.**

### **Cleaner code: With exception handling, you can avoid using complex conditional statements to check for errors, leading to cleaner and more readable code.**

### **Easier debugging: When an exception is raised, the Python interpreter prints a traceback that shows the exact location where the exception occurred, making it easier to debug your code.**

### **Disadvantages of Exception Handling:**

### **Performance overhead: Exception handling can be slower than using conditional statements to check for errors, as the interpreter has to perform additional work to catch and handle the exception.**

### **Increased code complexity: Exception handling can make your code more complex, especially if you have to handle multiple types of exceptions or implement complex error handling logic.**

### **Possible security risks: Improperly handled exceptions can potentially reveal sensitive information or create security vulnerabilities in your code, so it’s important to handle exceptions carefully and avoid exposing too much information about your program.**

### **Overall, the benefits of exception handling in Python outweigh the drawbacks, but it’s important to use it judiciously and carefully in order to maintain code quality and program reliability.**

### **Different types of exceptions in python:**

In Python, there are several built-in Python exceptions that can be raised when an error occurs during the execution of a program. Here are some of the most common types of exceptions in Python:

* **SyntaxError:** This exception is raised when the interpreter encounters a syntax error in the code, such as a misspelled keyword, a missing colon, or an unbalanced parenthesis.
* **TypeError**: This exception is raised when an operation or function is applied to an object of the wrong type, such as adding a string to an integer.
* **NameError**: This exception is raised when a variable or function name is not found in the current scope.
* **IndexError**: This exception is raised when an index is out of range for a list, tuple, or other sequence types.
* **KeyError**: This exception is raised when a key is not found in a dictionary.
* **ValueError**: This exception is raised when a function or method is called with an invalid argument or input, such as trying to convert a string to an integer when the string does not represent a valid integer.
* **AttributeError**: This exception is raised when an attribute or method is not found on an object, such as trying to access a non-existent attribute of a class instance.
* **IOError**: This exception is raised when an I/O operation, such as reading or writing a file, fails due to an input/output error.
* **ZeroDivisionError**: This exception is raised when an attempt is made to divide a number by zero.
* **ImportError**: This exception is raised when an import statement fails to find or load a module.

These are just a few examples of the many types of exceptions that can occur in Python. It’s important to handle exceptions properly in your code using try-except blocks or other error-handling techniques, in order to gracefully handle errors and prevent the program from crashing.

## **Difference between Syntax Error and Exceptions**

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

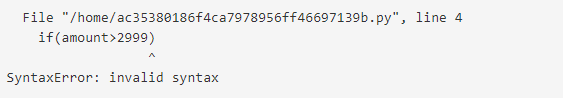
**Example:**

There is a syntax error in the code . The **‘if'** statement should be followed by a colon (:), and the **‘print'** statement should be indented to be inside the **‘if'** block.

* Python3

| amount **=** 10000  **if**(amount > 2999)  print("You are eligible to purchase Dsa Self Paced") |
| --- |

**Output:**

****

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

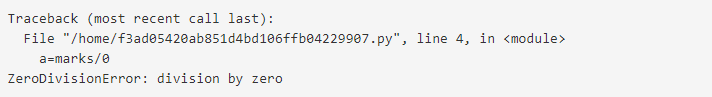
**Example:**

Here in this code a s we are dividing the **‘marks’** by zero so a error will occur known as **‘ZeroDivisionError’**

* Python3

| marks **=** 10000  a **=** marks **/** 0  print(a) |
| --- |

**Output:**

****

In the above example raised the ZeroDivisionError as we are trying to divide a number by 0.

**Note:** Exception is the base class for all the exceptions in Python. You can check the exception hierarchy [here](https://docs.python.org/2/library/exceptions.html#exception-hierarchy).

### **Example:**

1) TypeError: This exception is raised when an operation or function is applied to an object of the wrong type. Here’s an example:  
 Here a **‘TypeError’** is raised as both the datatypes are different which are being added.

* Python3

| x **=** 5  y **=** "hello"  z **=** x **+** y |
| --- |

output:

Traceback (most recent call last):

File "7edfa469-9a3c-4e4d-98f3-5544e60bff4e.py", line 4, in <module>

z = x + y

TypeError: unsupported operand type(s) for +: 'int' and 'str'

#### **try catch block to resolve it:**

The code attempts to add an integer (**‘x'**) and a string (**‘y'**) together, which is not a valid operation, and it will raise a **‘TypeError'**. The code used a **‘try'** and **‘except'** block to catch this exception and print an error message.

* Python3

| x **=** 5  y **=** "hello"  **try**:  z **=** x **+** y  **except** TypeError:  print("Error: cannot add an int and a str") |
| --- |

**Output**

Error: cannot add an int and a str

## **Try and Except Statement – Catching Exceptions**

Try and except statements are used to catch and handle exceptions in Python. Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause.

**Example:** Here we are trying to access the array element whose index is out of bound and handle the corresponding exception.

* Python3

| a **=** [1, 2, 3]  **try**:  **print** ("Second element = %d" **%**(a[1]))    **print** ("Fourth element = %d" **%**(a[3]))    **except**:  print ("An error occurred") |
| --- |

**Output**

Second element = 2

An error occurred

In the above example, the statements that can cause the error are placed inside the try statement (second print statement in our case). The second print statement tries to access the fourth element of the list which is not there and this throws an exception. This exception is then caught by the except statement.

## **Catching Specific Exception**

A try statement can have more than one except clause, to specify handlers for different exceptions. Please note that at most one handler will be executed. For example, we can add IndexError in the above code. The general syntax for adding specific exceptions are –

try:

# statement(s)

except IndexError:

# statement(s)

except ValueError:

# statement(s)

**Example:** Catching specific exceptions in the Python

The code defines a function **‘fun(a)'** that calculates b based on the input a. If a is less than 4, it attempts a division by zero, causing a **‘ZeroDivisionError'**. The code calls fun(3) and fun(5) inside a try-except block. It handles the ZeroDivisionError for fun(3) and prints **“ZeroDivisionError Occurred and Handled.”** The **‘NameError'** block is not executed since there are no **‘NameError'** exceptions in the code.

* Python3

| **def** fun(a):  **if** a < 4:    b **=** a**/**(a**-**3)  **print**("Value of b = ", b)    **try**:  fun(3)  fun(5)  **except** ZeroDivisionError:  print("ZeroDivisionError Occurred and Handled")  **except** NameError:  print("NameError Occurred and Handled") |
| --- |

**Output**

ZeroDivisionError Occurred and Handled

If you comment on the line fun(3), the output will be

NameError Occurred and Handled

The output above is so because as soon as python tries to access the value of b, NameError occurs.

## **Try with Else Clause**

In Python, you can also use the else clause on the try-except block which must be present after all the except clauses. The code enters the else block only if the try clause does not raise an exception.

**Try with else clause**

The code defines a function **AbyB(a, b)** that calculates c as ((a+b) / (a-b)) and handles a potential ZeroDivisionError. It prints the result if there’s no division by zero error. Calling **AbyB(2.0, 3.0)** calculates and prints -5.0, while calling **AbyB(3.0, 3.0)** attempts to divide by zero, resulting in a **ZeroDivisionError**, which is caught and **“a/b results in 0”** is printed.

* Python3

| **def** AbyB(a , b):  **try**:  c **=** ((a**+**b) **/** (a**-**b))  **except** ZeroDivisionError:  print ("a/b result in 0")  **else**:  print (c)  AbyB(2.0, 3.0)  AbyB(3.0, 3.0) |
| --- |

**Output:**

-5.0

a/b result in 0

## **Finally Keyword in Python**

Python provides a keyword [finally](https://www.geeksforgeeks.org/finally-keyword-in-python/), which is always executed after the try and except blocks. The final block always executes after the normal termination of the try block or after the try block terminates due to some exception.

**Syntax:**

try:

# Some Code....

except:

# optional block

# Handling of exception (if required)

else:

# execute if no exception

finally:

# Some code .....(always executed)

**Example:**

The code attempts to perform integer division by zero, resulting in a **ZeroDivisionError**. It catches the exception and prints **“Can’t divide by zero.”** Regardless of the exception, the finally block is executed and prints **“This is always executed.”**

* Python3

| **try**:  k **=** 5**//**0  print(k)    **except** ZeroDivisionError:  **print**("Can't divide by zero")    **finally**:  print('This is always executed') |
| --- |

**Output:**

Can't divide by zero

This is always executed

## **Raising Exception**

The [raise statement](https://www.geeksforgeeks.org/python-raising-an-exception-to-another-exception/) allows the programmer to force a specific exception to occur. The sole argument in raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception).

This code intentionally raises a NameError with the message **“Hi there”** using the raise statement within a try block. Then, it catches the NameError exception, prints **“An exception,”** and re-raises the same exception using raise. This demonstrates how exceptions can be raised and handled in Python, allowing for custom error messages and further exception propagation.

* Python3

| **try**:  **raise** NameError("Hi there")  **except** NameError:  print ("An exception")  **raise** |
| --- |

The output of the above code will simply line printed as “An exception” but a Runtime error will also occur in the last due to the raise statement in the last line. So, the output on your command line will look like

Traceback (most recent call last):

File "/home/d6ec14ca595b97bff8d8034bbf212a9f.py", line 5, in <module>

raise NameError("Hi there") # Raise Error

NameError: Hi there

### **Advantages of Exception Handling:**

* **Improved program reliability**: By handling exceptions properly, you can prevent your program from crashing or producing incorrect results due to unexpected errors or input.
* **Simplified error handling**: Exception handling allows you to separate error handling code from the main program logic, making it easier to read and maintain your code.
* **Cleaner code:** With exception handling, you can avoid using complex conditional statements to check for errors, leading to cleaner and more readable code.
* **Easier debugging**: When an exception is raised, the Python interpreter prints a traceback that shows the exact location where the exception occurred, making it easier to debug your code.

### **Disadvantages of Exception Handling:**

* **Performance overhead:** Exception handling can be slower than using conditional statements to check for errors, as the interpreter has to perform additional work to catch and handle the exception.
* **Increased code complexity**: Exception handling can make your code more complex, especially if you have to handle multiple types of exceptions or implement complex error handling logic.
* **Possible security risks:** Improperly handled exceptions can potentially reveal sensitive information or create security vulnerabilities in your code, so it’s important to handle exceptions carefully and avoid exposing too much information about your program.

Overall, the benefits of exception handling in Python outweigh the drawbacks, but it’s important to use it judiciously and carefully in order to maintain code quality and program reliability.

<https://www.w3schools.com/python/python_try_except.asp>