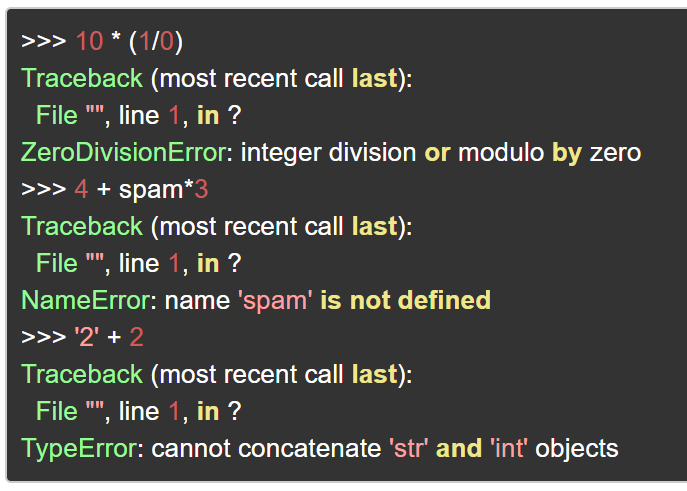
**Exceptions in Python**

## Errors and Exceptions: In Python, there are two kinds of errors: syntax errors and exceptions.

**Syntax Errors**: (also known as parsing errors). The parser repeats the offending line and displays an 'arrow' pointing at the earliest point in the line where the error was detected.

The error is caused by (or at least detected at) the token preceding the arrow: in the example, the error is detected at the keyword print, since a colon (':') is missing before it.

File name and line number are printed so you know where to look in case the input came from a script.

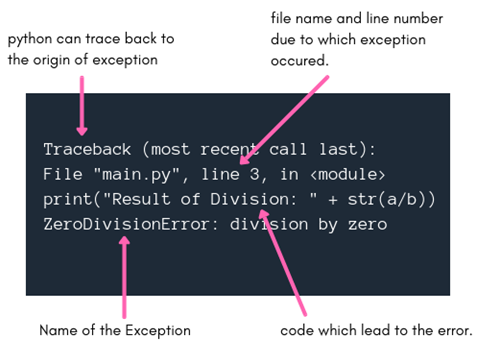


## Exceptions: The other kind of errors in Python are exceptions. Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.

## Errors detected during execution are called exceptions. Exceptions come in different types, and the type is printed as part of themessage

## The types in the example are ZeroDivisionError, NameError and TypeError.

### Decoding the Exception Message in Python

The term Traceback in the exception message means that python has traced back the code to the point from where the exception occured and will be showing the related messages after this line.

The second line in the exception message, as you can see above, tells us the name of the python file and the exact line number for the code due to which exception was generated.

If that is still not helpful for someone, in the third line of exception message the complete code statement which lead to the exception is printed.

And then in the last line, python tells us which exception/error occured, which in our example above is ZeroDivisionError.

|  |  |
| --- | --- |
| **Python Built-in Exceptions** | |
| **Exception** | **Cause of Error** |
| **AssertionError** | Raised when assert statement fails. |
| **AttributeError** | Raised when attribute assignment or reference fails. |
| **EOFError** | Raised when the input() functions hits end-of-file condition. |
| **FloatingPointError** | Raised when a floating point operation fails. |
| **GeneratorExit** | Raise when a generator's close() method is called. |
| **ImportError** | Raised when the imported module is not found. |
| **IndexError** | Raised when index of a sequence is out of range. |
| **KeyError** | Raised when a key is not found in a dictionary. |
| **KeyboardInterrupt** | Raised when the user hits interrupt key (Ctrl+c or delete). |
| **MemoryError** | Raised when an operation runs out of memory. |
| **NameError** | Raised when a variable is not found in local or global scope. |
| **NotImplementedError** | Raised by abstract methods. |
| **OSError** | Raised when system operation causes system related error. |
| **OverflowError** | Raised when result of an arithmetic operation is too large to be represented. |
| **ReferenceError** | Raised when a weak reference proxy is used to access a garbage collected referent. |
| **RuntimeError** | Raised when an error does not fall under any other category. |
| **StopIteration** | Raised by next() function to indicate that there is no further item to be returned by iterator. |
| **SyntaxError** | Raised by parser when syntax error is encountered. |
| **IndentationError** | Raised when there is incorrect indentation. |
| **TabError** | Raised when indentation consists of inconsistent tabs and spaces. |
| **SystemError** | Raised when interpreter detects internal error. |
| **SystemExit** | Raised by sys.exit() function. |
| **TypeError** | Raised when a function or operation is applied to an object of incorrect type. |
| **ValueError** | Raised when a function gets argument of correct type but improper value. |
| **ZeroDivisionError** | Raised when second operand of division or modulo operation is zero. |

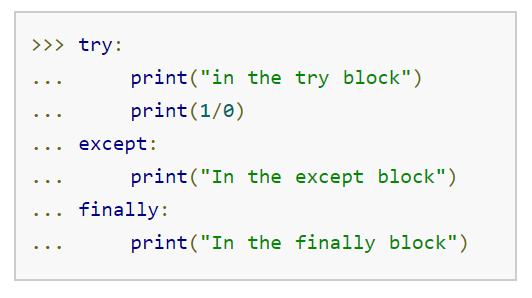
**Handling Exceptions with Try/Except/Finally**

Errors and Exceptions in Python are handled with the Try: Except: Finally construct.

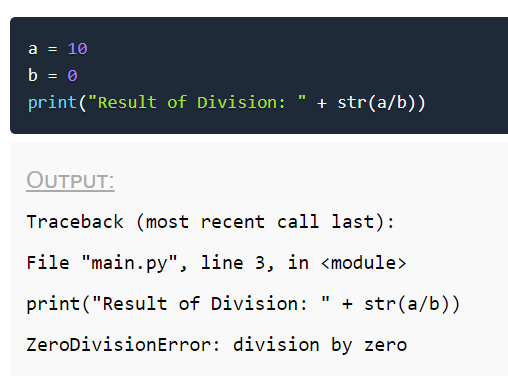
If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a try: block. After the try: block, include an except: statement, followed by a block of code which handles the problem as elegantly as possible.

You put the unsafe code in the try: block. You put the fall-back code in the Except: block. The final code is kept in the Finally: block.

For example, look at the code below.

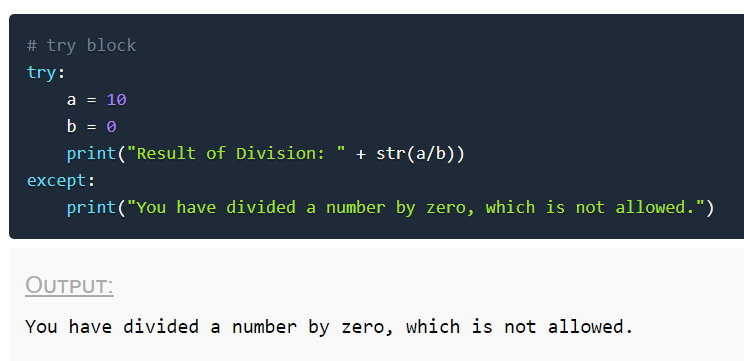


Below we have the code performing **division by zero**:



The above code leads to exception and the exception message is printed as output on the console.

If we use the try and except block, we can handle this exception gracefully.



### The try block

As you can see in the code example above, the try block is used to put the whole code that is to be executed in the program(which you think can lead to exception), if any exception occurs during execution of the code inside the try block, then it causes the execution of the code to be directed to the except block and the execution that was going on in the try block is interrupted. But, if no exception occurs, then the whole try block is executed and the except block is never executed.

### The except block

The try block is generally followed by the except block which holds the exception cleanup code(exception has occured, how to effectively handle the situation) like some print statement to print some message or may be trigger some event or store something in the database etc.

In the except block, along with the keyword except we can also provide the name of exception class which is expected to occur. In case we do not provide any exception class name, it catches all the exceptions, otherwise it will only catch the exception of the type which is mentioned.

Here is the syntax:

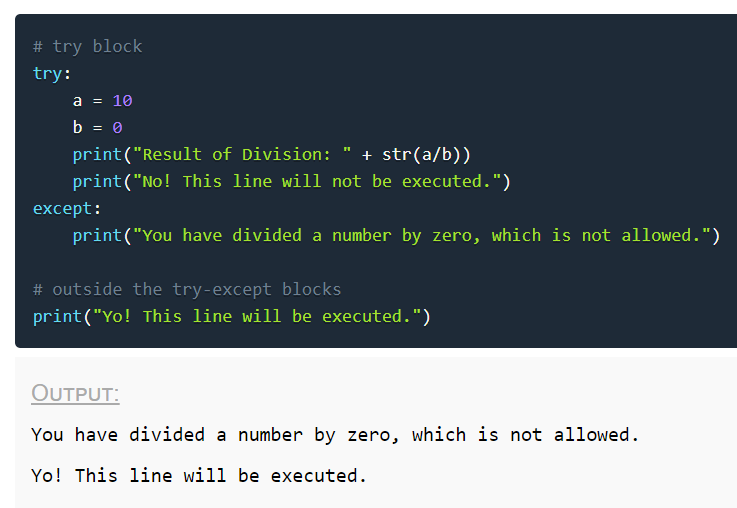
If you notice closely, we have mentioned types of exceptions, yes, we can even provide names of multiples exception classes separated by comma in the except statement.

### Code Execution continues after except block

Another important point to note here is that code execution is interrupted in the try block when an exception occurs, and the code statements inside the try block after the line which caused the exception are not executed.

The execution then jumps into the except block. And after the execution of the code statements inside the except block the code statements after it are executed, just like any other normal execution.

Let's take an example:



## Catching Multiple Exceptions in Python

There are multiple ways to accomplish this. Either we can have multiple except blocks with each one handling a specific exception class or we can handle multiple exception classes in a single except block.

**Way-1**

**# try block**

**try**:

a **=** int(input("**Enter numerator number:** "))

b **=** int(input("**Enter denominator number:** "))

**print**("**Result of Division:** " + str(a/b))

**# except block handling division by zero**

**except**(ZeroDivisionError):

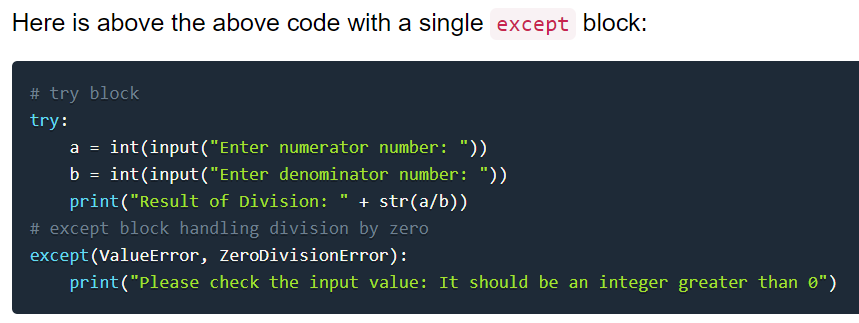
**print**("**You have divided a number by zero, which is not allowed.**")

**# except block handling wrong value type**

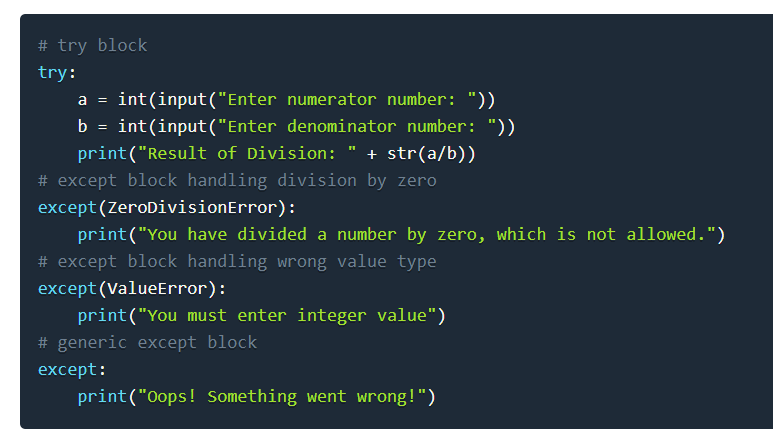
**except**(ValueError):

print**("You must enter integer value")**

**Way-2**

****

**Way-3**

****

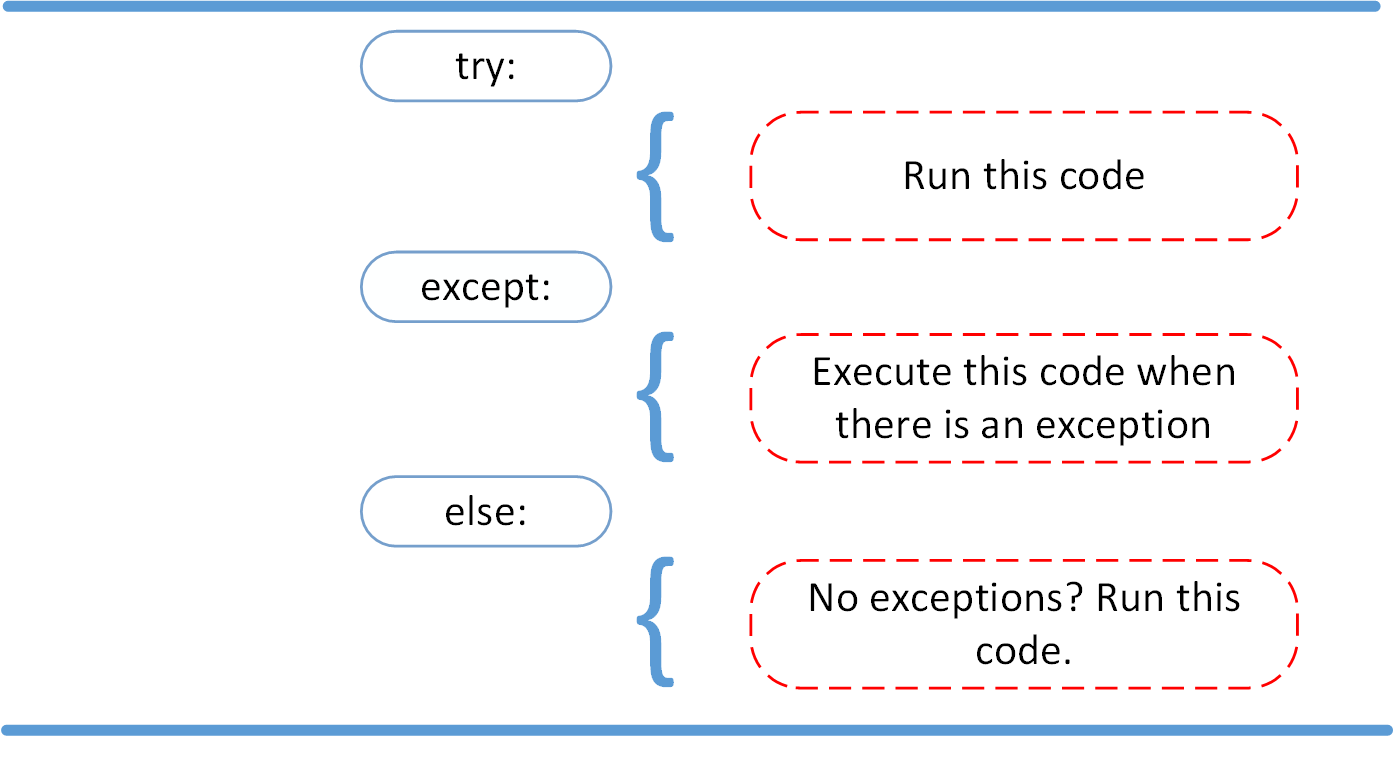
Here are few important points about the above-mentioned syntax −

* A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
* You can also provide a generic except clause, which handles any exception.
* After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
* The else-block is a good place for code that does not need the try: block's protection.

**The else Clause**

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

Diagram of try, except, and else statements in Python



Look at the following example:

a = int(input())

try:

if a > 5:

raise Exception

except:

print("There is an exception")

#raise

else:

print(f"value of a is {a}")

## Raising Exceptions

In Python programming, exceptions are raised when corresponding errors occur at run time, but we can forcefully raise it using the keyword raise. We can also optionally pass in value to the exception to clarify why that exception was raised.

>>> raise KeyboardInterrupt

Traceback (most recent call last):

...

KeyboardInterrupt

>>> raise MemoryError("This is an argument")

Traceback (most recent call last):

...

MemoryError: This is an argument

>>> try:

... a = int(input("Enter a positive integer: "))

... if a <= 0:

... raise ValueError("That is not a positive number!")

... except ValueError as ve:

... print(ve)

...

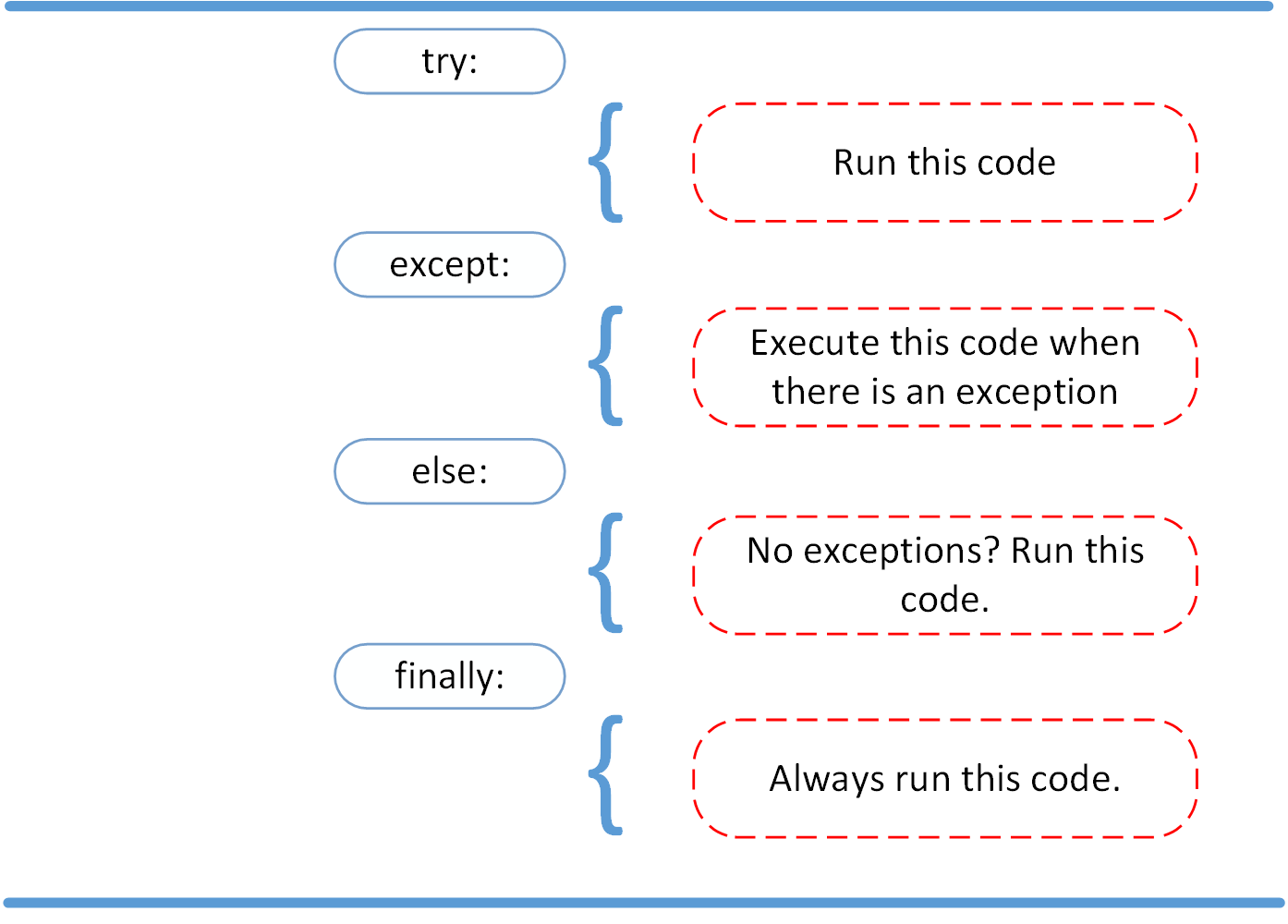
**Output:**

**Enter a positive integer: -2**

**That is not a positive number!**

## try...finally

The try statement in Python can have an optional finally clause. This clause is executed no matter what, and is generally used to release external resources.



For example, we may be connected to a remote data center through the network or working with a file or working with a Graphical User Interface (GUI).

In all these circumstances, we must clean up the resource once used, whether it was successful or not. These actions (closing a file, GUI or disconnecting from network) are performed in the finally clause to guarantee execution.

Here is an example of file operations to illustrate this.

try:

f = open("test.txt",encoding = 'utf-8')

# perform file operations

finally:

f.close()

This type of construct makes sure the file is closed even if an exception occurs.