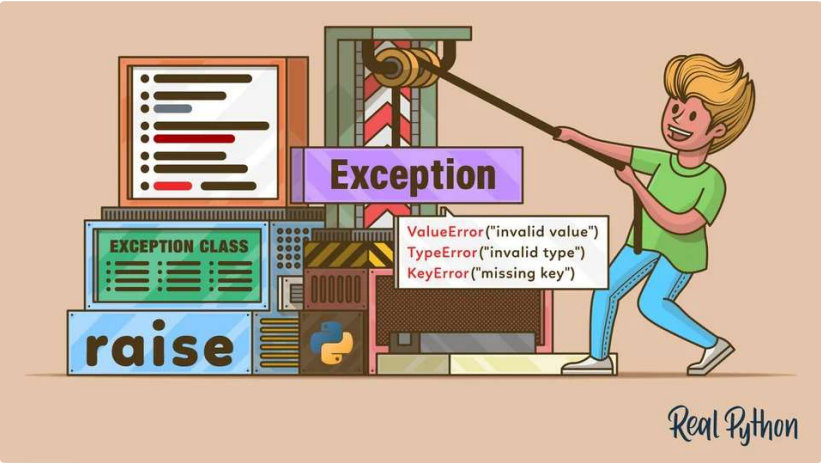

Python's raise: Effectively Raising Exceptions in Your Code

by Leodanis Pozo Ramos Jun 19, 2023 2 Comments

intermediate python

Mark as Completed

Tweet Share Email

Table of Contents

In your Python journey, you'll come across situations where you need to signal that something is going wrong in your code. For example, maybe a file doesn't exist, a network or database connection fails, or your code gets invalid input. A common approach to tackle these issues is to **raise an exception**, notifying the user that an error has occurred. That's what Python's `raise` statement is for.

Learning about the `raise` statement allows you to effectively handle errors and exceptional situations in your code. This way, you'll develop more robust programs and higher-quality code.

Table of Contents

- Handling Exceptional Situations in Python
- Raising Exceptions in Python: The raise Statement
- Choosing the Exception to Raise: Built-in vs Custom
- Deciding When to Raise Exceptions

In this tutorial, you'll learn how to:

- Raise exceptions in Python using the `raise` statement
- Decide **which exceptions** to raise and **when** to raise them in your code
- Explore common **use cases** for raising exceptions in Python
- Apply **best practices** for raising exceptions in your Python code

To get the most out of this tutorial, you should understand the fundamentals of Python, including variables, data types, conditional statements, exception handling, and classes.

> **Free Bonus: Click here to download the sample code** that you'll use to gracefully raise and handle exceptions in your Python code.

Handling Exceptional Situations in Python

Exceptions play a fundamental role in Python. They allow you to handle **errors** and **exceptional situations** in your code. But what is an exception? An exception represents an error or indicates that something is going wrong. Some programming languages, such as C, and Go, encourage you to return error codes, which you *check*. In contrast, Python encourages you to raise exceptions, which you *handle*.

> **Note:** In Python, not all exceptions are errors. The built-in `StopIteration` exception is an excellent example of this. Python internally uses this exception to terminate the iteration over iterators. Python exceptions that represent errors have the `Error` suffix attached to their names.
>
> Python also has a specific category of exceptions that represent warnings to the programmer. Warnings come in handy when you need to alert the user of some condition in a program. However, that condition may not warrant raising an exception and terminating the program. A common example of a warning is `DeprecationWarning`, which appears when you use deprecated features.

When a problem occurs in a program, Python automatically raises an exception. For example, watch what happens if you try to access a nonexistent index in a `list` object:

Python >>>

```
>>> colors = [
...     "red",
...     "orange",
...     "yellow",
...     "green",
...     "blue",
...     "indigo",
...     "violet"
... ]

>>> colors[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

In this example, your `colors` list doesn't have a `10` index. Its indices go from `0` to `6`, covering your seven colors. So, if you try to get index number `10`, then you get an `IndexError` exception telling you that your target index is out of range.

> **Note:** In the example above, Python raised the exception on its own, which it'll only do with built-in exceptions. You, as a programmer, have the option to raise built-in or custom exceptions, as you'll learn in the section Choosing the Exception to Raise: Built-in vs Custom.

Every raised exception has a traceback, also known as a **stack trace**, **stack traceback**, or **backtrace**, among other names. A traceback is a report containing the sequence of calls and operations that traces down to the current exception.

In Python, the traceback header is `Traceback (most recent call last)` in most situations. Then you'll have the actual call stack and the exception name followed by its error message.

> **Note:** Since the introduction of the new PEG parser in Python 3.9, there's been an ongoing effort to make the error messages in tracebacks more helpful and specific. This effort continues to produce new results, and Python 3.12 has incorporated even better error messages.

Exceptions will cause your program to terminate unless you handle them using a `try ... except` block:

Python >>>

```
>>> try:
...     colors[10]
... except IndexError:
...     print("Your list doesn't have that index :-(")
...
Your list doesn't have that index :-(
```

The first step in handling an exception is to *predict which exceptions can happen*. If you don't do that, then you can't handle the exceptions, and your program will crash. In that situation, Python will print the exception traceback so that you can figure out how to fix the problem. Sometimes, you must let the program fail in order to discover the exceptions that it raises.

In the above example, you know beforehand that indexing a list with an index beyond its range will raise an `IndexError` exception. So, you're ready to catch and handle that specific exception. The `try` block takes care of catching exceptions. The `except` clause specifies the exception that you're predicting, and the `except` code block allows you to take action accordingly. The whole process is known as **exception handling**.

If your code raises an exception in a function but doesn't handle it there, then the exception propagates to where you called function. If your code doesn't handle it there either, then it continues propagating until it reaches the main program. If there's no exception handler there, then the program halts with an exception traceback.

Exceptions are everywhere in Python. Virtually every module in the standard library uses them. Python will raise exceptions in many different circumstances. The Python documentation states that:

> Exceptions are a means of breaking out of the normal flow of control of a code block in order to handle errors or other exceptional conditions. An exception is *raised* at the point where the error is detected; it may be *handled* by the surrounding code block or by any code block that directly or indirectly invoked the code block where the error occurred. (Source)

In short, Python automatically raises exceptions when an error occurs during a program's execution. Python also allows you to raise exceptions on demand using the `raise` keyword. This keyword lets you handle your program's errors in a more controlled manner.

Raising Exceptions in Python: The `raise` Statement

In Python, you can raise either built-in or custom exceptions. When you raise an exception, the result is the same as when Python does it. You get an exception traceback, and your program crashes unless you handle the exception on time.

> **Note:** In Python's terminology, exceptions are *raised*, while in other programming languages, such as C++ and Java, exceptions are *thrown*.

To raise an exception by yourself, you'll use the `raise` statement, which has the following general syntax:

Python

```
raise [expression [from another_expression]]
```

A `raise` keyword with no argument reraises the active exception. Note that you can only use a bare `raise` in `except` code blocks with an active exception. Otherwise, you'll get a `RuntimeError` exception:

Python >>>

```
>>> raise
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: No active exception to reraise
```

In this example, you're in a context with no active exceptions. Therefore, Python can't reraise a previous exception. Instead, it raises a `RuntimeError` exception.

The bare `raise` statement is useful when you need to perform some actions after catching an exception, and then you want to reraise the original exception. You'll learn more about this use case of `raise` in the section Reraising a Previous Exception.

The `expression` object in the `raise` syntax must return an instance of a class that derives from `BaseException`, which is the base class for all built-in exceptions. It can also return the exception class itself, in which case Python will automatically instantiate the class for you.

Note that `expression` can be any Python expression that returns an exception class or instance. For example, your argument to `raise` could be a custom function that returns an exception:

Python >>>

```
>>> def exception_factory(exception, message):
...     return exception(message)
...

>>> raise exception_factory(ValueError, "invalid value")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid value
```

This `exception_factory()` function takes an exception class and an error message as arguments. Then the function instantiates the input exception using the message as an argument. Finally, it returns the exception instance to the caller. You can use this function as an argument to the `raise` keyword, as you do in the above example.

The `from` clause is also optional in the `raise` syntax. It allows you to chain exceptions together. If you provide the `from` clause, then `another_expression` must evaluate to another exception class or instance. You'll learn more about the `from` clause in the section Chaining Exceptions With the `from` Clause.

Here's another example of how `raise` works. This time, you create a new instance of the `Exception` class:

Python >>>

```
>>> raise Exception("an error occurred")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: an error occurred
```

For illustration purposes, you raise an instance of `Exception`, but raising this generic exception actually isn't a best practice. As you'll learn later, user-defined exceptions should derive from this class, although they can derive from other built-in exceptions too.

Note: In Python, starting the error messages of exceptions with a lowercase letter is common practice. Also, they don't end in a period.

To illustrate, consider the following examples:

Python >>>

```
>>> 42 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero

>>> [][0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

In the first example, you get a `ZeroDivisionError` exception because you're trying to divide `42` by `0`. Note how the default error message starts with a lowercase letter and doesn't have a period at its end.

In the second example, you try to access the `0` index in an empty list, and Python raises a `TypeError` exception for you. In this case, the error message follows the same described pattern.

Even though this practice isn't an explicit convention, you should consider sticking to this pattern to ensure consistency in your codebase.

The class constructor of exceptions like `Exception` can take multiple positional arguments or a tuple of arguments:

```
Python                                                >>>
>>> raise Exception("an error occurred", "unexpected value", 42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: ('an error occurred', 'unexpected value', 42)
```

You'll typically instantiate exceptions with a single argument: a string that provides an appropriate error message. However, you can also instantiate them with multiple arguments. In this example, you provide additional arguments to the `Exception` class constructor. These arguments allow you to give other developers more information about what caused the error and how to fix it.

The `.args` attribute gives you direct access to all the arguments passed to the `Exception` constructor:

```
Python                                                >>>
>>> error = Exception("an error occurred", "unexpected value", 42)
>>> error.args
('an error occurred', 'unexpected value', 42)
>>> error.args[1]
'unexpected value'
```

In this example, you access the arguments using the `.args` attribute. This attribute holds a tuple that you can index to access specific arguments.

Now that you've learned the basics of raising exceptions in Python, it's time to move on. An essential part of raising your own exceptions is deciding which exception is appropriate in a given moment.

Choosing the Exception to Raise: Built-in vs Custom

When it comes to manually raising exceptions in your code, deciding which exception to raise is an important step. In general, you should raise exceptions that clearly communicate the problem you're dealing with. In Python, you can raise two different kinds of exceptions:

- **Built-in exceptions**: These exceptions are built into Python. You can use them directly in your code without importing anything.
- **User-defined exceptions**: Custom exceptions are those that you create when no built-in exception fits your needs. You'll typically put them in a dedicated module for a specific project.

In the following sections, you'll find some guidelines for deciding which exception to raise in your code.

Raising Built-in Exceptions

Python has a rich set of built-in exceptions structured in a class hierarchy with the `BaseException` class at the top. One of the most frequently used subclasses of `BaseException` is `Exception`.

The `Exception` class is a fundamental part of Python's exception-handling scaffolding. It's the base class for most of the built-in exceptions that you'll find in Python. It's also the class that you'll typically use to create your custom exceptions.

Python has more than sixty built-in exceptions. You've probably seen some of the following concrete exceptions in your day-to-day coding:

Exception Class	Description
`ImportError`	Appears when an `import` statement has trouble loading a module
`ModuleNotFoundError`	Happens when `import` can't locate a given module
`NameError`	Appears when a global or local name isn't defined
`AttributeError`	Happens when an attribute reference or assignment fails
`IndexError`	Occurs when an indexing operation on a sequence uses an out-of-range index
`KeyError`	Occurs when a key is missing in a dictionary
`ZeroDivisionError`	Appears when the second operand in a division or modulo operation is `0`
`TypeError`	Happens when an operation, function, or method operates on an object of inappropriate type
`ValueError`	Occurs when an operation, function, or method receives the right type of argument but the wrong value

This table is just a small sample of Python's built-in exceptions. You can use these and all the other built-in exceptions when using the `raise` statement in your code.

In most cases, you'll likely find an appropriate built-in exception for your specific use case. If that's your case, then favor the built-in exception over a custom one. For example, say you're coding a function to compute the square of all the values in an input `list`, and you want to ensure that the input object is a list or tuple:

Python >>>

```
>>> def squared(numbers):
...     if not isinstance(numbers, list | tuple):
...         raise TypeError(
...             f"list or tuple expected, got '{type(numbers).__name__}'"
...         )
...     return [number**2 for number in numbers]
...

>>> squared([1, 2, 3, 4, 5])
[1, 4, 9, 16, 25]

>>> squared({1, 2, 3, 4, 5})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in squared
TypeError: list or tuple expected, got 'set'
```

In `squared()`, you use a conditional statement to check whether the input object is a list or a tuple. If it's not, then you raise a `TypeError` exception. That's an excellent exception choice because you want to ensure the correct *type* in the input. If the

type is wrong, then getting a `TypeError` is a logical response.

Coding and Raising Custom Exceptions

If you don't find a built-in exception that semantically suits your needs, then you can define a custom one. To do this, you must inherit from another exception class, typically `Exception`. For example, say that you're coding a gradebook app and need to calculate the students' average grades.

You want to ensure that all the grades are between 0 and 100. To handle this scenario, you can create a custom exception called `GradeValueError`. You'll raise this exception if a grade value is outside the target interval, meaning it's invalid:

Python

```
# grades.py

class GradeValueError(Exception):
    pass

def calculate_average_grade(grades):
    total = 0
    count = 0
    for grade in grades:
        if grade < 0 or grade > 100:
            raise GradeValueError(
                "grade values must be between 0 and 100 inclusive"
            )
        total += grade
        count += 1
    return round(total / count, 2)
```

In this example, you first create a custom exception by inheriting from `Exception`. You don't need to add new functionality to your custom exception, so you use the `pass` statement to provide a placeholder class body. This new exception is specific to your grade project. Note how the exception name helps communicate the underlying issue.

> **Note:** In Python, defining custom exceptions with a `pass` statement as the class body is a common practice. This is because the class's name is often the most important aspect of a custom exception.
>
> You'll also find situations where you may want to add new features to your custom exceptions. However, that topic is beyond the scope of this tutorial.

Inside the `calculate_average_grade()` function, you use a `for` loop to iterate over the input list of grades. Then you check if the current grade value falls outside the range of `0` to `100`. If that's the case, then you instantiate and raise your custom exception, `GradeValueError`.

Here's how your function works in practice:

Python >>>

```
>>> from grades import calculate_average_grade

>>> calculate_average_grade([98, 95, 100])
97.67

>>> calculate_average_grade([98, 87, 110])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in calculate_average_grade
GradeValueError: grade values must be between 0 and 100 inclusive
```

With an invalid grade value, your function raises `GradeValueError`, which is project-specific and shows an explicit message about the actual error.

Some developers may argue that, in this example, you can use the built-in `ValueError` instead of defining your own custom exception, and they might be right. In general, you'll define custom exceptions when you want to point out project-specific errors or exceptional situations.

If you decide to use custom exceptions, then remember that the naming conventions for classes apply. Additionally, you should add the suffix `Error` to custom exceptions that represent errors and no suffix for non-error exceptions. If you're defining custom warnings, then you should use the `Warning` suffix instead. All these naming conventions make your intentions clearer to other developers.

Deciding When to Raise Exceptions

Another important step in effectively using the `raise` statement is deciding when to raise a given exception in your code. You'll need to decide whether raising an exception is the best option to solve your current problem. In general, you should raise exceptions when you need to:

- **Signal errors and exceptional situations**: The most common use case of the `raise` statement is to signal that an error or exceptional situation has occurred. When an error or exceptional situation occurs in your code, you can raise an exception in response.

- **Reraise exception after doing some additional processing**: A common use case of `raise` is to reraise an active exception after performing some operations. A good example of this use case is when you need to log the error before raising the actual exception.

Some programming languages, such as C and Go, encourage you to return error codes from functions and methods. Then you'll have to check these codes in conditional statements and handle the underlying errors accordingly.

For example, Go programmers should be familiar with the following construct:

Go

```go
func SomeFunc(arg int) error {
    result, err := DoSomething(arg)
    if err != nil {
        log.Print(err)
        return err
    }
    return nil
}
```

Here, `DoSomething()` must return a result and an error. The conditional checks the error and takes action accordingly.

In contrast, Python encourages you to use exceptions to deal with errors and exceptional situations. Then you'll have to handle these errors using the `try` … `except` construct. This approach is pretty common in Python code. The standard library and the language itself have many examples of it.

An equivalent Python function would look something like this:

Python

```python
def some_func(arg):
    try:
        do_something(arg)
    except Exception as error:
        logging.error(error)
        raise
```

In `some_func()`, you use a `try` … `except` block to catch an exception that `do_something()` or any other code called from that function would raise. The `error` object represents the target exception in the example. Then you log the error and reraise the active exception, `error`.

> **Note:** The approach that encourages using error codes and conditionals is generally known as **look before you leap (LBYL)**. In contrast, the approach that encourages raising and handling exceptions is known as **easier to ask forgiveness than permission (EAFP)**. To dive deeper into how both approaches work, check out LBYL vs EAFP: Preventing or Handling Errors in Python.

In practice, you'll raise exceptions as soon as the error or exceptional situation occurs. On the other hand, when to catch and handle the raised exception will depend on your specific use case. Sometimes, capturing the exception close to where it first occurred makes sense. This way, you'll have enough context to correctly handle the exception and recover without terminating the entire program.

However, if you're writing a library, then you typically don't have enough context to handle the exception, so you leave the exception handling to the clients.

Now that you have a clearer idea of when to raise exceptions in your Python code, it's time to get your hands dirty and start raising exceptions.

Raising Exceptions in Practice

Raising exceptions is a popular way to deal with errors and exceptional situations in Python. For that reason, you'll find yourself using the `raise` statement in several situations.

For example, you'll use the `raise` statement to raise exceptions in response to a given condition in your code or to reraise active exceptions after performing some additional processing. You'll also use this statement to chain exceptions with the `from` clause so that you provide more context to anyone debugging your code or code that's built on top of it.

Before diving into this discussion, you'll learn a bit about the internal structure of an exception. Exception classes have an attribute called `.args`, which is a tuple containing the arguments that you provide to the exception constructor:

Python >>>

```
>>> error = Exception("an error occurred")

>>> error.args
('an error occurred',)
>>> error.args[0]
'an error occurred'
```

The `.args` attribute gives you dynamic access to all the arguments that you passed to the exception class constructor at instantiation time. You can take advantage of this attribute when you're raising your own exceptions.

Exception classes also have two methods:

1. `.with_traceback()` allows you to provide a new traceback for the exception, and it returns the updated exception object.
2. `.add_note()` allows you to include one or more notes in an exception's traceback. You can access the list of notes in a given exception by inspecting the `.__notes__` special attribute.

Both methods allow you to provide extra information about a given exception, which will help your users debug their code.

Finally, exceptions also have a bunch of special, or dunder, attributes like `.__notes__`. Two of the most useful dunder attributes are `.__traceback__` and `.__cause__`.

> **Note:** Special attributes are also known as **dunder attributes**. The word *dunder* comes from the double underscores at the beginning and end of the names.

The `.__traceback__` attribute holds the traceback object attached to an active exception:

Python >>>

```
>>> try:
...     result = 42 / 0
... except Exception as error:
...     tb = error.__traceback__
...
>>> tb
<traceback object at 0x102b150c0>
```

In this example, you access the `.__traceback__` attribute, which holds an exception traceback object. You can use traceback objects like this to customize a given exception using the `.with_traceback()` method.

Python automatically creates a traceback object when an exception occurs. Then it attaches the object to the exception's `.__traceback__` attribute, which is writable. You can create an exception and provide it with your own traceback using the

`.with_traceback()` method.

The `.__cause__` attribute will store the expression passed to the `from` class when you're chaining exceptions in your code. You'll learn more about this attribute in the section about `from`.

Now that you know the basics of how exceptions are built internally, it's time to continue learning about raising exceptions.

Raising Exceptions Conditionally

Raising an exception when you meet a given condition is a common use case of the `raise` statement. These conditions are usually related to possible errors and exceptional situations.

For example, say that you want to write a function to determine if a given number is prime. The input number must be an integer. It should also be greater than or equal to `2`. Here's a possible implementation that handles these situations by raising exceptions:

Python >>>

```
>>> from math import sqrt

>>> def is_prime(number):
...     if not isinstance(number, int):
...         raise TypeError(
...             f"integer number expected, got {type(number).__name__}"
...         )
...     if number < 2:
...         raise ValueError(f"integer above 1 expected, got {number}")
...     for candidate in range(2, int(sqrt(number)) + 1):
...         if number % candidate == 0:
...             return False
...     return True
...
```

This function checks if the input number isn't an instance of `int`, in which case it raises a `TypeError`. Then the function checks if the input number is less than `2`, raising a `ValueError` if the condition is true. Note that both `if` statements check for conditions that would cause errors or conditional situations if they passed silently.

Then the function iterates through the integers between `2` and the square root of `number`. Inside the loop, the conditional statement checks if the current number is divisible by any other in the interval. If so, then the function returns `False` because the number isn't prime. Otherwise, it returns `True` to signal that the input number is prime.

Finally, it's important to note that you've raised both exceptions early in the function, right before doing any computation. Raising exceptions early, as you did in this function, is considered a best practice.

Reraising a Previous Exception

You can use the `raise` statement without any arguments to reraise the last exception that occurred in your code. The classic situation where you'd want to use `raise` this way is when you need to log the error after it happens:

Python >>>

```
>>> import logging

>>> try:
...     result = 42 / 0
... except Exception as error:
...     logging.error(error)
...     raise
...
ERROR:root:division by zero
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
```

In this example, you use `Exception` to catch any exceptions that happen in the `try` code block. If an exception of any type occurs, then you log the actual error using the `logging` module from the standard library and finally reraise the active exception using a bare `raise` statement.

> **Note:** Catching a generic `Exception`, as in the example above, is considered a bad practice in Python. You should always try to find a specific exception to catch. This way, you'll prevent hiding unknown errors.
>
> However, in this specific use case of `raise`, you may want to specify a broad or generic exception in the `except` clause so that you can catch several different errors, log them, and then reraise the original exception for higher-level code to handle.

Note that you'll get a similar effect if you use `raise` with a reference to the active exception as an argument:

Python >>>

```
1  >>> try:
2  ...     result = 42 / 0
3  ... except Exception as error:
4  ...     logging.error(error)
5  ...     raise error
6  ...
7  ERROR:root:division by zero
8  Traceback (most recent call last):
9    File "<stdin>", line 5, in <module>
10   File "<stdin>", line 2, in <module>
11 ZeroDivisionError: division by zero
```

If you use the current exception as an argument to `raise`, then you get an extra piece of traceback. The second line in the traceback tells you that line 5 has reraised an exception. That's the exception that you manually raised. The third traceback line tells you what the original exception in the code was. That exception occurred on line 2.

Another common scenario where you'd need to reraise an exception is when you want to wrap one exception in another or intercept one exception and translate it into a different one. To illustrate, say that you're writing a math library, and you have several external math-related libraries as dependencies. Each external library has its own exceptions, which may end up confusing you and your users.

In this case, you can catch those libraries' exceptions, wrap them with a custom exception and then raise it. For example, the following function captures a `ZeroDivisionError` and wraps it in a custom `MathLibraryError`:

```
Python                                                    >>>
>>> class MathLibraryError(Exception):
...     pass
...

>>> def divide(a, b):
...     try:
...         return a / b
...     except ZeroDivisionError as error:
...         raise MathLibraryError(error)
...

>>> divide(1, 0)
Traceback (most recent call last):
  File "<stdin>", line 3, in divide
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in divide
MathLibraryError: division by zero
```

In this example, you catch the concrete exception, `ZeroDivisionError`, and wrap it in your own custom exception, `MathLibraryError`. This technique can be useful in a few situations, including the following:

- **Abstracting away external exceptions:** When you're writing a library that interacts with multiple external components, you may want to abstract away external exceptions and raise custom exceptions so that your users don't depend on the former. That's what you did in the example above.
- **Unifying handling actions:** When you have multiple exception types with the same handling action, it might make sense to catch all those exceptions, raise a single custom exception, and handle it according to your planned actions. This practice can simplify your error-handling logic.
- **Augmenting the context of a caught exception:** When you're handling an exception that doesn't have enough context or behavior at first, you can add those features and then reraise the exception manually.

The above use cases of reraising exceptions are neat and can facilitate your life when you're handling exceptions in your code. However, changing exceptions with the `from` clause often offers a better alternative.

For example, if you use the `from None` syntax in the above example, then you'll suppress the `ZeroDivisionError` and only get information about `MathLibraryError`. In the following section, you'll learn about the `from` clause and how it works.

Chaining Exceptions With the `from` Clause

The `raise` statement has an optional `from` clause. This clause allows you to chain the raised exception with a second exception provided as an argument to `from`. Note that if you use the `from` clause, then its argument must be an expression that returns an exception class or instance. You'll typically use `from` in an `except` code block to chain the raised exception with the active one.

If the argument to `from` is an exception instance, then Python will attach it to the raised exception's `.__cause__` attribute. If it's an exception class, then Python will instantiate it before attaching it to `.__cause__`.

The effect of `from` is that you'll have both exception tracebacks on your screen:

Python >>>

```
>>> try:
...     result = 42 / 0
... except Exception as error:
...     raise ValueError("operation not allowed") from error
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ValueError: operation not allowed
```

In this example, you use `Exception` in the `except` clause to catch any exception in the `try` block. Then you raise a `ValueError` from the concrete exception, which is a `ZeroDivisionError` in this case.

The `from` clause chains both exceptions, providing complete context for the user to debug the code. Note how Python presents the first exception as the direct cause of the second one. This way, you'll be in a better position to track the error down and fix it.

This technique is pretty handy when you're processing a piece of code that can raise multiple types of exceptions. Consider the following `divide()` function:

Python >>>

```
>>> def divide(x, y):
...     for arg in (x, y):
...         if not isinstance(arg, int | float):
...             raise TypeError(
...                 f"number expected, got {type(arg).__name__}"
...             )
...     if y == 0:
...         raise ValueError("denominator can't be zero")
...     return x / y
...
```

This function raises a `TypeError` if the input argument isn't a number. Similarly, it raises a `ValueError` if the `y` argument is equal to `0` because this will cause the zero division error.

Here are two examples of how the `from` clause can be helpful:

Python >>>

```
>>> try:
...     divide(42, 0)
... except Exception as error:
...     raise ValueError("invalid argument") from error
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 6, in divide
ValueError: denominator can't be zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ValueError: invalid argument

>>> try:
...     divide("One", 42)
... except Exception as error:
...     raise ValueError("invalid argument") from error
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 4, in divide
TypeError: number expected, got str

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ValueError: invalid argument
```

In the first example, the traceback shows the `ValueError` exception that happens when the second argument to `divide()` is `0`. This traceback helps you track the actual error in your code. In the second example, the traceback directs your eyes toward the `TypeError` exception, which is due to using the wrong argument type.

It's important to note that if you don't use `from`, then Python will raise both exceptions, but the output will be a bit different:

Python >>>

```
>>> try:
...     divide(42, 0)
... except Exception as error:
...     raise ValueError("invalid argument")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 6, in divide
ValueError: denominator can't be zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ValueError: invalid argument
```

Now the traceback doesn't state that the the first exception is the root cause of the second one.

Another way to use `from` is when you use `None` as its argument. Using `from None` allows you to suppress or hide the original exception's traceback when it's not necessary or informative. You can also use this syntax to suppress the traceback of built-in exceptions when raising your own exception.

To illustrate how `from None` works, say that you're coding a package to consume an external REST API. You've decided to use the `requests` library to access the API. However, you don't want to expose the exceptions that this library provides. Instead, you want to use a custom exception.

> **Note:** For the code below to work, you must first install the `requests` library in your current Python environment using `pip` or a similar tool.

Here's how you can achieve that behavior:

Python >>>

```
>>> import requests

>>> class APIError(Exception):
...     pass
...

>>> def call_external_api(url):
...     try:
...         response = requests.get(url)
...         response.raise_for_status()
...         data = response.json()
...     except requests.RequestException as error:
...         raise APIError(f"{error}") from None
...     return data
...

>>> call_external_api("https://api.github.com/events")
[
    {
        'id': '29376567903',
        'type': 'PushEvent',
    ...
]
```

The `call_external_api()` function takes a URL as an argument and makes a GET request to it. If an error occurs during the API call, then the function raises the `APIError` exception, which is your custom exception. The `from None` clause will hide the original exception's traceback, replacing it with your own.

To check how this function works, say that you make a spelling mistake while providing the API endpoint:

Python >>>

```
>>> call_external_api("https://api.github.com/event")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in call_external_api
__main__.APIError: 404 Client Error: Not Found for url:
  https://api.github.com/event
```

In this example, you misspell the target URL. This mistake raises an exception that Python automatically catches in the `except` clause using the generic `RequestException` exception. Note how you've completely suppressed the original exception in the traceback. Instead, you only get the custom `APIError` exception.

The `from None` construct is useful when you want to provide a custom traceback and error message and suppress the original exception. This can be helpful in cases where the original exception message isn't very informative or useful for users, and you want to provide a more helpful message.

Following Best Practices When Raising Exceptions

When it comes to raising exceptions in Python, you can follow a few practices and recommendations that will make your life more pleasant. Here's a summary of some of these practices and recommendations:

- **Favor specific exceptions over generic ones**: You should raise the most specific exception that suits your needs. This practice will help you track down and fix problems and errors.
- **Provide informative error messages and avoid exceptions with no message**: You should write descriptive and explicit error messages for all your exceptions. This practice will provide a context for those debugging the code.
- **Favor built-in exceptions over custom exceptions**: You should try to find an appropriate built-in exception for every error in your code before writing your own exception. This practice will ensure consistency with the rest of the Python ecosystem. Most experienced Python developers will be familiar with the most common built-in exceptions, making it easier for them to understand and work with your code.
- **Avoid raising the `AssertionError` exception**: You should avoid raising the `AssertionError` in your code. This exception is specifically for the `assert` statement, and it's not appropriate in other contexts.
- **Raise exceptions as soon as possible**: You should check error conditions and exceptional situations early in your code. This practice will make your code more efficient by avoiding unnecessary processing that a delayed error check could throw away. This practice fits the fail-fast design.
- **Explain the raised exceptions in your code's documentation**: You should explicitly list and explain all the exceptions that a given piece of code could raise. This practice helps other developers understand which exceptions they should expect and how they can handle them appropriately.

To illustrate some of these recommendations, consider the following example, where you raise a generic exception:

Python >>>

```
>>> age = -10

>>> if age < 0:
...     raise Exception("invalid age")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: invalid age
```

In this example, you use the `Exception` class to point out an issue that's closely related to the input value. Using a more specific exception like `ValueError` is more appropriate in this example. Improving the error message a bit will also help:

Python >>>

```
>>> if age < 0:
...     raise ValueError("age must not be negative")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: age must not be negative
```

In this example, the very name of the raised exception helps you communicate the actual issue. Additionally, the error message is more precise and helpful. Again, you should provide informative error messages regardless of whether you're using a built-in or custom exception.

In general, the error message in an exception should concisely describe what went wrong. The error message should be specific enough to help other developers identify, diagnose, and debug the error. However, it shouldn't reveal too many details of your code's internals because this practice may lead to security flaws.

In any case, remember that these are just recommended practices. They're not strict rules. You'll face situations in which you would want to raise and handle generic exceptions.

Finally, if you're writing a library for other developers, then you must document the exceptions that your functions and methods can raise. You should list the types of exceptions that your code may raise and briefly describe what each exception means and how your users can handle them in their code.

Raising Exceptions and the `assert` Statement

The `raise` statement isn't the only statement that raises exceptions in Python. You also have the **`assert`** statement. However, the goal of `assert` is different, and it can only raise one type of exception, `AssertionError`.

The `assert` statement is a debugging and testing tool in Python. It allows you to write sanity checks that are known as assertions. You can use these checks to verify whether certain assumptions remain true in your code. If any of your assertions become false, then the `assert` statement raises an `AssertionError` exception. Getting this error may mean that you have a bug in your code.

> **Note:** To dive deeper into writing assertions in your code, check out Python's `assert`: Debug and Test Your Code Like a Pro.

You shouldn't use the `assert` statement to handle user input or other kinds of input errors. Why? Because assertions can, and most likely will, be disabled in production code. So, it's best to use them during development as a tool for debugging and testing.

In Python, `assert` has the following syntax:

Python

```
assert expression[, assertion_message]
```

In this construct, `expression` can be any valid Python expression or object that you need to test for truthiness. If `expression` is false, then the statement raises an `AssertionError`. The `assertion_message` parameter is optional but encouraged because it adds more context for those debugging and testing the code. It can hold an error message describing the issue that the statement is supposed to catch.

Here's an example that shows how you write an `assert` statement with a descriptive error message:

Python >>>

```
>>> age = -10

>>> assert age >= 0, f"expected age to be at least 0, got {age}"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: expected age to be at least 0, got -10
```

The error message in this assertion clearly communicates what's making the condition fail. In this example, the condition is false, so `assert` raises an `AssertionError` in response. Again, you should use something other than the `assert` statement to validate input because assertions can be disabled in production code, making the input skip the validation.

It's also important to note that your code shouldn't explicitly raise the `AssertionError` exception with a `raise` statement. This exception should occur as a result of a failing assertion while you're testing and debugging your code during development.

Raising Exception Groups

If you're in Python 3.11 or greater, then you'll have the option to use the new `ExceptionGroup` class and the associated `except*` syntax. These new Python features are useful when you need to handle multiple errors simultaneously. For example, you might reach for them when an asynchronous program has several concurrent tasks that could fail at the same time. But in general, you probably won't raise an `ExceptionGroup` very often.

Apart from the usual error message, the `ExceptionGroup` constructor takes an additional argument consisting of a non-empty sequence of exceptions. Here's a toy example that shows how you can raise an exception group and what its traceback looks like:

Python >>>

```
>>> raise ExceptionGroup(
...     "several errors",
...     [
...         ValueError("invalid value"),
...         TypeError("invalid type"),
...         KeyError("missing key"),
...     ],
... )
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  | ExceptionGroup: several errors (3 sub-exceptions)
  +-+---------------- 1 ----------------
    | ValueError: invalid value
    +---------------- 2 ----------------
    | TypeError: invalid type
    +---------------- 3 ----------------
    | KeyError: 'missing key'
    +------------------------------------
```

You raise an `ExceptionGroup` as you'd raise any other exception in Python. However, the traceback of an exception group is quite different from the traceback of a regular exception. You'll get information about the exception group and its grouped exceptions.

Once you've wrapped several exceptions in an exception group, then you can catch them with the `except*` syntax, like in the code below:

Python >>>

```
>>> try:
...     raise ExceptionGroup(
...         "several errors",
...         [
...             ValueError("invalid value"),
...             TypeError("invalid type"),
...             KeyError("missing key"),
...         ],
...     )
... except* ValueError:
...     print("Handling ValueError")
... except* TypeError:
...     print("Handling TypeError")
... except* KeyError:
...     print("Handling KeyError")
...
Handling ValueError
Handling TypeError
Handling KeyError
```

Note that this construct behaves differently from either multiple `except` clauses that catch different exceptions or an `except` clause that catches multiple exceptions. In those latter cases, the code will catch the first exception that occurs. With this new syntax, your code will raise all the exceptions, so it can catch all of them.

> **Note:** For a deep dive into the various ways to catch one or all of multiple exceptions, check out How to Catch Multiple Exceptions in Python.

Finally, when you raise an `ExceptionGroup`, Python will try it as a regular exception because it's a subclass of `Exception`. For example, if you remove the asterisk from the `except` clause, then Python won't catch any of the listed exceptions:

Python >>>

```
>>> try:
...     raise ExceptionGroup(
...         "several errors",
...         [
...             ValueError("invalid value"),
...             TypeError("invalid type"),
...             KeyError("missing key"),
...         ],
...     )
... except ValueError:
...     print("Handling ValueError")
... except TypeError:
...     print("Handling TypeError")
... except KeyError:
...     print("Handling KeyError")
...
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  | ExceptionGroup: several errors (3 sub-exceptions)
  +-+---------------- 1 ----------------
    | ValueError: invalid value
    +---------------- 2 ----------------
    | TypeError: invalid type
    +---------------- 3 ----------------
    | KeyError: 'missing key'
    +------------------------------------
```

In this new version of your code, you removed the asterisk from the `except` clause. So, your code won't catch any individual exceptions in the group. The idea is that to catch any of the subexceptions in an exception group, you must use the `except*` syntax.

However, you can use a plain `except` if you want to catch the `ExceptionGroup` itself:

Python >>>

```
>>> try:
...     raise ExceptionGroup(
...         "several errors",
...         [
...             ValueError("invalid value"),
...             TypeError("invalid type"),
...             KeyError("missing key"),
...         ],
...     )
... except ExceptionGroup:
...     print("Got an exception group!")
...
Got an exception group!
```

In the context of a regular `except` clause, Python tries `ExceptionGroup` as it would try any other exception. It catches the group and runs the handling code.

Conclusion

Now you have a solid understanding of how to **raise exceptions** in Python using the `raise` statement. You also learned when to raise exceptions in your code and how to decide which exception to raise depending on the error or issue that you're dealing with. Additionally, you've dove into some best practices and recommendations to improve your code's ability to deal with errors and exceptions.

In this tutorial, you've learned how to:

- Use Python's **`raise`** statement to raise exceptions in your code

- Decide **which exceptions** to raise and **when** to raise them in your code
- Explore common **use cases** for raising exceptions in Python
- Apply **best practices** for raising exceptions in your Python code

You're equipped with the required knowledge and skills to effectively handle errors and exceptional situations in your code.

With these new skills, you can write reliable and maintainable code that can gracefully handle errors and exceptional situations. Overall, efficiently handling exceptions is an essential skill in Python programming, so keep practicing and refining it to become a better developer!

Free Bonus: **Click here to download the sample code** that you'll use to gracefully raise and handle exceptions in your Python code.

About **Leodanis Pozo Ramos**

Leodanis is an industrial engineer who loves Python and software development. He's a self-taught Python developer with 6+ years of experience. He's an avid technical writer with a growing number of articles published on Real Python and other sites.

» More about Leodanis

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Aldren

Bartosz

Geir Arne

Kate

What Do You Think?

Rate this article:

Tweet Share Share Email

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Keep Learning

Related Tutorial Categories: intermediate python