



# Python Exceptions: An Introduction

by Said van de Klundert  24m  35 Comments

 basics python

Mark as Completed



Share 

## Table of Contents

- Understanding Exceptions and Syntax Errors
- Raising an Exception in Python
- Debugging During Development With assert
- Handling Exceptions With the try and except Block

— FREE Email Series —

### Python Tricks

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

Email...

Get Python Tricks »

 No spam. Unsubscribe any time.

 Browse Topics  Guided Learning Paths

 Basics  Intermediate  Advanced

ai algorithms api best-practices career  
community databases data-science  
data-structures data-viz devops django  
docker editors flask front-end gamedev  
gui machine-learning news numpy

- Proceeding After a Successful Try With else
- Cleaning Up After Execution With finally
- Creating Custom Exceptions in Python
- Conclusion
- Frequently Asked Questions

projects python stdlib testing tools  
web-dev web-scraping

i Remove ads

**Watch Now** This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Raising and Handling Python Exceptions](#)

Python exceptions provide a mechanism for handling errors that occur during the execution of a program. Unlike syntax errors, which are detected by the parser, Python raises exceptions when an error occurs in syntactically correct code. Knowing how to raise, catch, and handle exceptions effectively helps to ensure your program behaves as expected, even when encountering errors.

### By the end of this tutorial, you'll understand that:

- Exceptions in Python occur when **syntactically correct code** results in an **error**.
- **The try ... except block** lets you execute code and handle exceptions that arise.
- You can use the `else`, and `finally` **keywords** for more refined **exception handling**.
- It's **bad practice** to **catch all exceptions** at once using `except Exception` or the bare `except` clause.
- Combining `try`, `except`, and `pass` allows your program to **continue silently** without handling the exception.

### Table of Contents

- Understanding Exceptions and Syntax Errors
- Raising an Exception in Python
- Debugging During Development With assert
- Handling Exceptions With the `try` and `except` Block
- Proceeding After a Successful Try With else
- Cleaning Up After Execution With finally
- Creating Custom Exceptions in Python
- Conclusion

In this tutorial, you'll get to know Python exceptions and all relevant keywords for exception handling by walking through a practical example of handling a platform-related exception. Finally, you'll also learn how to create your own custom Python exceptions.

- Frequently Asked Questions

**Get Your Code:** Click here to download the free sample code that shows you how exceptions work in Python.

 **Take the Quiz:** Test your knowledge with our interactive “Python Exceptions: An Introduction” quiz. You’ll receive a score upon completion to help you track your learning progress:

#### Interactive Quiz

### Python Exceptions: An Introduction



In this quiz, you'll test your understanding of Python exceptions. You'll cover the difference between syntax errors and exceptions and learn how to raise exceptions, make assertions, and use the try and except block.

Mark as Completed 



Share 

#### Recommended Video Course

Raising and Handling Python Exceptions

## Understanding Exceptions and Syntax Errors

Syntax errors occur when the parser detects an incorrect statement. Observe the following example:

#### Python Traceback

```
>>> print(0 / 0)
File "<stdin>", line 1
    print(0 / 0)
          ^
SyntaxError: unmatched ')'
```

The arrow indicates where the parser ran into the **syntax error**. Additionally, the error message gives you a hint about what went wrong. In this example, there was one bracket too many. Remove it and run your code again:

Python

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

This time, you ran into an **exception error**. This type of error occurs whenever syntactically correct Python code results in an error. The last line of the message indicates what type of exception error you ran into.

Instead of just writing *exception error*, Python details what *type* of exception error it encountered. In this case, it was a `ZeroDivisionError`. Python comes with [various built-in exceptions](#) as well as the possibility to create user-defined exceptions.

 Remove ads

## Raising an Exception in Python

There are scenarios where you might want to stop your program by raising an exception if a condition occurs. You can do this with the `raise` keyword:

Use raise to force an exception:



You can even complement the statement with a custom message. Assume that you're writing a tiny toy program that expects only numbers up to 5. You can raise an error when an unwanted condition occurs:

Python

low.py

```
number = 10
if number > 5:
    raise Exception(f"The number should not exceed 5. ({number=})")
print(number)
```

In this example, you raised an `Exception` object and passed it an informative custom message. You built the message using an [f-string](#) and a [self-documenting expression](#).

When you run `low.py`, you'll get the following output:

Python Traceback

```
Traceback (most recent call last):
File "./low.py", line 3, in <module>
    raise Exception(f"The number should not exceed 5. ({number=})")
Exception: The number should not exceed 5. (number=10)
```

The program comes to a halt and displays the exception to your [terminal](#) or [REPL](#), offering you helpful clues about what went wrong. Note that the final call to `print()` never executed, because Python raised the exception before it got to that line of code.

With the `raise` keyword, you can raise any exception object in Python and stop your program when an unwanted condition occurs.

## Debugging During Development With `assert`

Before moving on to the most common way of working with exceptions in Python using [the `try ... except` block](#), you'll take a quick look at an exception that's a bit different than the others.

Python offers a specific exception type that you should only use when debugging your program during development. This exception is the `AssertionError`. The `AssertionError` is special because you shouldn't ever raise it yourself using `raise`.

Instead, you use [the `assert` keyword](#) to check whether a condition is met and let Python raise the `AssertionError` if the condition isn't met.

The idea of an assertion is that your program should only attempt to run if certain conditions are in place. If Python checks your assertion and finds that the condition is `True`, then that is excellent! The program can continue. If the condition turns out to be `False`, then your program raises an `AssertionError` exception and stops right away:



Revisit your tiny script, `low.py`, from [the previous section](#). Currently, you're explicitly raising an exception when a certain condition isn't met:

Python

low.py

```
number = 1
if number > 5:
    raise Exception(f"The number should not exceed 5. ({number=})")
print(number)
```

Assuming that you'll handle this constraint safely for your production system, you could replace this [conditional statement](#) with an assertion for a quick way to retain this sanity check during development:

Python

low.py

```
number = 1
assert (number < 5), f"The number should not exceed 5. ({number=})"
print(number)
```

If the number in your program is below 5, then the assertion passes and your script continues with the next line of code. However, if you set number to a value higher than 5—for example, 10—then the outcome of the assertion will be False:

Python

low.py

```
number = 10
assert (number < 5), f"The number should not exceed 5. ({number=})"
print(number)
```

In that case, Python raises an `AssertionError` that includes the message you passed, and ends the program execution:

Shell

```
$ python low.py
Traceback (most recent call last):
  File "./low.py", line 2, in <module>
    assert (number < 5), f"The number should not exceed 5. ({number=})"
```

```
^^^^^^^^^
AssertionError: The number should not exceed 5. (number=10)
```

In this example, raising an `AssertionError` exception is the last thing that the program will do. The program will then come to halt and won't continue. The call to `print()` that follows the assertion won't execute.

Using assertions in this way can be helpful when you're debugging your program during development because it can be quite a fast and straightforward to add assertions into your code.

However, you shouldn't rely on assertions for catching crucial run conditions of your program in production. That's because Python globally disables assertions when you run it in optimized mode using the [-O and -OO command line options](#):

Shell

```
$ python -O low.py
10
```

In this run of your program, you used the `-O` command line option, which removes all `assert` statements. Therefore, your script ran all the way to the end and displayed a number that is *dreadfully* high!

**Note:** Alternatively, you can also disable assertions through the [PYTHONOPTIMIZE environment variable](#).

In production, your Python code may run using this optimized mode, which means that assertions aren't a reliable way to handle runtime errors in production code. They can be quick and useful helpers when you're debugging your code, but you should never use assertions to set crucial constraints for your program.

If `low.py` should reliably fail when `number` is above 5, then it's best to stick with [raising an exception](#). However, sometimes you might not want your program to fail when it encounters an exception, so how should you handle those situations?

 Remove ads

## Handling Exceptions With the `try` and `except` Block

In Python, you use the `try` and `except` block to catch and handle exceptions. Python executes code following the `try` statement as a normal part of the program. The code that follows the `except` statement is the program's response to any exceptions in the preceding `try` clause:



As you saw earlier, when syntactically correct code runs into an error, Python will raise an exception error. This exception error will crash the program if you don't handle it. In the `except` clause, you can determine how your program should respond to exceptions.

The following function can help you understand the try and except block:

```
Python          linux_interaction.py

def linux_interaction():
    import sys
    if "linux" not in sys.platform:
        raise RuntimeError("Function can only run on Linux systems.")
    print("Doing Linux things.")
```

The `linux_interaction()` can only run on a Linux system. Python will raise a `RuntimeError` exception if you call it on an operating system other than Linux.

**Note:** Picking the right exception type can sometimes be tricky. Python comes with many built-in exceptions that are hierarchically related, so if you browse the documentation, you're likely to find a fitting one.

Python even groups some of the exceptions into categories, such as `warnings` that you should use to indicate warning conditions, and `OS exceptions` that Python raises depending on system error codes.

If you still didn't find a fitting exception, then you can [create a custom exception](#).

You can give the function a try by adding the following code:

```
Python          linux_interaction.py

# ...

try:
    linux_interaction()
except:
    pass
```

The way you handled the error here is by handing out a pass. If you run this code on a macOS or Windows machine, then you get the following output:

Shell

```
$ python linux_interaction.py
```

You got nothing in response. The good thing here is that your program didn't crash. But letting an exception that occurred pass silently is bad practice. You should always at least know about and [log](#) if some type of exception occurred when you ran your code.

To this end, you can change pass into something that generates an informative message:

Python

linux\_interaction.py

```
# ...  
  
try:  
    linux_interaction()  
except:  
    print("Linux function wasn't executed.")
```

When you now execute this code on a macOS or Windows machine, you'll see the message from your except block printed to the console:

Shell

```
$ python linux_interaction.py  
Linux function wasn't executed.
```

When an exception occurs in a program that runs this function, then the program will continue as well as inform you about the fact that the function call wasn't successful.

What you didn't get to see was the type of error that Python raised as a result of the function call. In order to see exactly what went wrong, you'd need to catch the error that the function

raised.

The following code is an example where you capture the `RuntimeError` and output that message to your screen:

```
Python          linux_interaction.py

# ...

try:
    linux_interaction()
except RuntimeError as error:
    print(error)
    print("The linux_interaction() function wasn't executed.")
```

In the `except` clause, you assign the `RuntimeError` to the temporary variable `error`—often also called `err`—so that you can access the exception object in the indented block. In this case, you’re printing the object’s string representation, which corresponds to the error message attached to the object.

Running this function on a macOS or Windows machine outputs the following:

```
Shell

$ python linux_interaction.py
Function can only run on Linux systems.
The linux_interaction() function wasn't executed.
```

The first message is the `RuntimeError`, informing you that Python can only execute the function on a Linux machine. The second message tells you which function wasn’t executed.

In the example above, you called a function that you wrote yourself. When you executed the function, you caught the `RuntimeError` exception and printed it to your screen.

Here’s another example where you open a file and use a built-in exception:

```
try:  
    with open("file.log") as file:  
        read_data = file.read()  
except:  
    print("Couldn't open file.log")
```

If `file.log` doesn't exist, then this block of code will output the following:

Shell

```
$ python open_file.py  
Couldn't open file.log
```

This is an informative message, and your program will still continue to run. However, your `except` block will currently catch *any* exception, whether that's related to not being able to open the file or not. You could lead yourself onto a confusing path if you see this message even when Python raises a completely unrelated exception.

Therefore, it's always best to be *specific* when you're handling an exception.

In the [Python docs](#), you can see that there are a couple of built-in exceptions that you could raise in such a situation, for example:

`exception FileNotFoundError`

Raised when a file or directory is requested but doesn't exist. Corresponds to errno ENOENT. ([Source](#))

You want to handle the situation when Python can't find the requested file. To catch this type of exception and print it to screen, you could use the following code:

```
try:  
    with open("file.log") as file:  
        read_data = file.read()  
except FileNotFoundError as fnf_error:  
    print(fnf_error)
```

In this case, if `file.log` doesn't exist, then the output will be the following:

Shell

```
$ python open_file.py  
[Errno 2] No such file or directory: 'file.log'
```

You can have more than one function call in your `try` clause and anticipate catching various exceptions. Something to note here is that the code in the `try` clause will stop as soon as it encounters any one exception.

**Warning:** When you use a bare `except` clause, then Python catches any exception that inherits from `Exception`—which are most built-in exceptions! Catching the parent class, `Exception`, hides all errors—even those which you didn't expect at all. This is why you should avoid bare `except` clauses in your Python programs.

Instead, you'll want to refer to *specific exception classes* that you want to catch and handle. You can learn more about why this is a good idea [in this tutorial](#).

Look at the following code. Here, you first call `linux_interaction()` and then try to open a file:

Python

linux\_interaction.py

```
# ...  
  
try:  
    linux_interaction()
```

```
with open("file.log") as file:  
    read_data = file.read()  
except FileNotFoundError as fnf_error:  
    print(fnf_error)  
except RuntimeError as error:  
    print(error)  
print("Linux linux_interaction() function wasn't executed.")
```

If you run this code on a macOS or Windows machine, then you'll see the following:

Shell

```
$ python linux_interaction.py  
Function can only run on Linux systems.  
Linux linux_interaction() function wasn't executed
```

Inside the try clause, you ran into an exception immediately and didn't get to the part where you attempt to open file.log. Now look at what happens when you run the code on a Linux machine if the file doesn't exist:

Shell

```
$ python linux_interaction.py  
[Errno 2] No such file or directory: 'file.log'
```

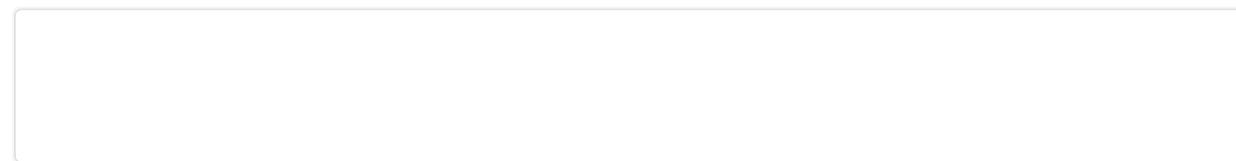
Note that if you're handling specific exceptions as you did above, then the order of the except clauses doesn't matter too much. It's all about which of the exceptions Python raises first. As soon as Python raises an exception, it checks the except clauses from top to bottom and executes the first matching one that it finds.

Here are the key takeaways about using Python's try ... except statements:

- Python executes a try clause up until the point where it encounters the first exception.
- Inside the except clause—the exception handler—you determine how the program responds to the exception.

- You can anticipate [multiple exceptions](#) and differentiate how the program should respond to them.
- [Avoid using bare except clauses](#), because they can hide unexpected exceptions.

While using `try` together with `except` is probably the most common error handling that you'll encounter, there's more that you can do to fine-tune your program's response to exceptions.



 Remove ads

## Proceeding After a Successful Try With `else`

You can use Python's `else` statement to instruct a program to execute a certain block of code only in the absence of exceptions:



Look at the following example:

```
Python           linux_interaction.py

# ...

try:
    linux_interaction()
except RuntimeError as error:
    print(error)
else:
    print("Doing even more Linux things.")
```

If you were to run this code on a Linux system, then the output would be the following:

Shell

```
$ python linux_interaction.py
Doing Linux things.
```

Doing even more Linux things.

Because the program didn't run into *any* exceptions, Python executed the code in the `else` clause. However, if you run this code on a macOS or Windows system, then you get a different output:

Shell

```
$ python linux_interaction.py  
Function can only run on Linux systems.
```

The `linux_interaction()` function raised a `RuntimeError`. You've handled the exception, so your program doesn't crash, and instead prints the exception message to the console. The code nested under the `else` clause, however, doesn't execute, because Python encountered an exception during execution.

Note that structuring your code like this is different from just adding the call to `print()` outside of the context of the `try ... except` block:

Python

linux\_interaction.py

```
# ...  
  
try:  
    linux_interaction()  
except RuntimeError as error:  
    print(error)  
print("Doing even more Linux things.")
```

If you don't nest the `print()` call under the `else` clause, then it'll execute even if Python encounters the `RuntimeError` that you handle in the `except` block above. On a Linux system, the output would be the same, but on macOS or Windows, you'd get the following output:

Shell

```
$ python linux_interaction.py
Function can only run on Linux systems.
Doing even more Linux things.
```

Nesting code under the `else` clause assures that it'll only run when Python doesn't encounter any exception when executing the `try ... except` block.

You can also create a nested `try ... except` block inside the `else` clause and catch possible exceptions there as well:

```
Python                               linux_interaction.py

# ...

try:
    linux_interaction()
except RuntimeError as error:
    print(error)
else:
    try:
        with open("file.log") as file:
            read_data = file.read()
    except FileNotFoundError as fnf_error:
        print(fnf_error)
```

If you were to execute this code on a Linux machine, then you'd get the following result:

```
Shell

$ python linux_interaction.py
Doing Linux things.
[Errno 2] No such file or directory: 'file.log'
```

From the output, you can see that `linux_interaction()` ran. Because Python encountered no exceptions, it attempted to open `file.log`. That file didn't exist, but instead of letting the

program crash, you caught the `FileNotFoundException` exception and printed a message to the console.

 Remove ads

## Cleaning Up After Execution With `finally`

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



Have a look at the following example:

```
Python          linux_interaction.py

# ...

try:
    linux_interaction()
except RuntimeError as error:
    print(error)
else:
    try:
        with open("file.log") as file:
            read_data = file.read()
    except FileNotFoundError as fnf_error:
        print(fnf_error)
finally:
    print("Cleaning up, irrespective of any exceptions.")
```

In this code, Python will execute everything in the `finally` clause. It doesn't matter if you encounter an exception somewhere in any of the `try ... except` blocks. Running the code on a macOS or Windows machine will output the following:

```
Shell

$ python linux_interaction.py
Function can only run on Linux systems.
Cleaning up, irrespective of any exceptions.
```

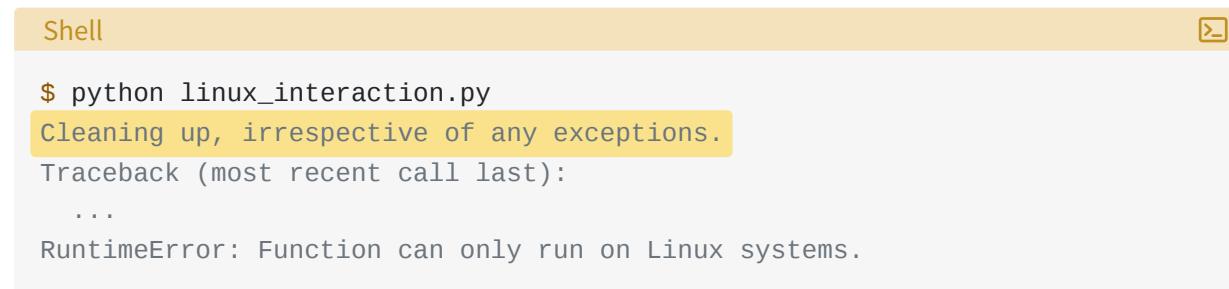
Note that the code inside the `finally` block will execute regardless of whether or not you're handling the exceptions:

```
Python          linux_interaction.py

# ...
```

```
try:  
    linux_interaction()  
finally:  
    print("Cleaning up, irrespective of any exceptions.")
```

You simplified the example code from above, but `linux_interaction()` still raises an exception on a macOS or Windows system. If you now run this code on an operating system other than Linux, then you'll get the following output:



```
Shell  
$ python linux_interaction.py  
Cleaning up, irrespective of any exceptions.  
Traceback (most recent call last):  
...  
RuntimeError: Function can only run on Linux systems.
```

Despite the fact that Python raised the `RuntimeError`, the code in the `finally` clause still executed and printed the message to your console.

This can be helpful because even code outside of a `try... except` block won't necessarily execute if your script encounters an unhandled exception. In that case, your program will terminate and the code *after* the `try ... except` block will never run. However, Python will still execute the code inside of the `finally` clause. This helps you make sure that resources like [file handles](#) and [database connections](#) are cleaned up properly.

## Creating Custom Exceptions in Python

With the large number of built-in exceptions that Python offers, you'll likely find a fitting type when deciding which exception to raise. However, sometimes your code won't fit the mold.

Python makes it straightforward to create custom exception types by inheriting from a built-in exception. Think back to your `linux_interaction()` function:

Python

linux\_interaction.py

```
def linux_interaction():
    import sys
    if "linux" not in sys.platform:
        raise RuntimeError("Function can only run on Linux systems.")
    print("Doing Linux things.")

# ...
```

Using a `RuntimeError` isn't a bad choice in this situation, but it would be nice if your exception name was a bit more specific. For this, you can create a custom exception:

Python

linux\_interaction.py

```
class PlatformException(Exception):
    """Incompatible platform."""

# ...
```

You generally create a custom exception in Python by inheriting from `Exception`, which is the base class for most built-in Python exceptions as well. You could also inherit from a different exception, but choosing `Exception` is usually the best choice.

That's really all that you need to do. In the code snippet above, you also added a `docstring` that describes the exception type and serves as the class body.

**Note:** Python requires some indented code in the body of your class. Alternatively to using the docstring, you could've also used `pass` or the ellipsis (`...`). However, adding a descriptive docstring adds the most value to your custom exception. To learn how to write effective docstrings, check out [How to Write Docstrings in Python](#).

While you can customize your exception object, you don't need to do that. It's often enough to give your custom Python exceptions a descriptive name, so you'll know what happened

when Python raises this exception in your code.

Now that you've defined the custom exception, you can raise it like any other Python exception:

```
Python          linux_interaction.py

class PlatformException(Exception):
    """Incompatible platform."""

def linux_interaction():
    import sys
    if "linux" not in sys.platform:
        raise PlatformException("Function can only run on Linux systems.")
    print("Doing Linux things.")

# ...
```

If you now call `linux_interaction()` on macOS or Windows, then you'll see that Python raises your custom exception:

```
Shell

$ python linux_interaction.py
Traceback (most recent call last):
...
PlatformException: Function can only run on Linux systems.
```

You could even use your custom `PlatformException` as a parent class for other custom exceptions that you could descriptively name for each of the platforms that users may run your code on.

# Conclusion

At this point, you're familiar with the basics of using Python exceptions. After seeing the difference between syntax errors and exceptions, you learned about various ways to raise, catch, and handle exceptions in Python. You also learned how you can create your own custom exceptions.

In this article, you gained experience working with the following exception-related keywords:

- `raise` allows you to raise an exception at any time.
- `assert` enables you to verify if a certain condition is met and raises an exception if it isn't.
- In the `try` clause, all statements are executed until an exception is encountered.
- `except` allows you to catch and handle the exception or exceptions that Python encountered in the `try` clause.
- `else` lets you code sections that should run only when Python encounters no exceptions in the `try` clause.
- `finally` enables you to execute sections of code that should always run, whether or not Python encountered any exceptions.

**Get Your Code:** Click here to download the free sample code that shows you how exceptions work in Python.

You now understand the basic tools that Python offers for dealing with exceptions. If you're curious about the topic and want to dive deeper, then take a look at the following tutorials:

- [Python's Built-in Exceptions: A Walkthrough With Examples](#)

- Exception Groups and `except*`
- Python `raise`: Effectively Raising Exceptions in Your Code
- How to Catch Multiple Exception in Python
- Understanding the Python Traceback
- LBYL vs EAFP: Preventing or Handling Errors in Python

What's your favorite aspect of exception handling in Python? Share your thoughts in the comments below.

## Frequently Asked Questions

Now that you have some experience with Python exceptions, you can use the questions and answers below to check your understanding and recap what you've learned.

These FAQs are related to the most important concepts you've covered in this tutorial. Click the *Show/Hide* toggle beside each question to reveal the answer.

**What are exceptions in Python?**

Show/Hide

**How are exceptions handled in Python?**

Show/Hide

**How do you catch all exceptions in Python?**

Show/Hide

**How does `try ... except` in Python work?**

Show/Hide

**What does try ... except pass do in Python?**

Show/Hide

**How do you raise an exception in Python?**

Show/Hide

**What is the purpose of using assert in Python?**

Show/Hide

**What is the role of the finally clause in exception handling?**

Show/Hide

 **Take the Quiz:** Test your knowledge with our interactive “Python Exceptions: An Introduction” quiz. You’ll receive a score upon completion to help you track your learning progress:

#### Interactive Quiz

#### Python Exceptions: An Introduction

In this quiz, you'll test your understanding of Python exceptions. You'll cover the difference between syntax errors and exceptions and learn how to raise exceptions, make assertions, and use the try and except block.



Mark as Completed



Share

 **Watch Now** This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Raising and Handling Python Exceptions](#)

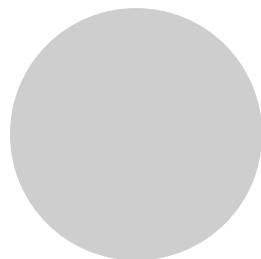


Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

[Send Me Python Tricks »](#)

## About Said van de Klundert



Said is a network engineer, Python enthusiast, and a guest author at Real Python.

[» More about Said](#)

*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:*



Adriana



Brenda



Bartosz



Geir Arne



Joanna



Kate



Martin

Master Real-World Python Skills  
With Unlimited Access to Real Python

**Join us and get access to thousands of  
tutorials, hands-on video courses, and a  
community of expert Pythonistas:**

[Level Up Your Python Skills »](#)

## What Do You Think?

Rate this article:



[LinkedIn](#)

[Twitter](#)

[Bluesky](#)

[Facebook](#)

[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 Topics: [basics](#) [python](#)

Recommended Video Course: [Raising and Handling Python Exceptions](#)

Related Tutorials:

- [Defining Your Own Python Function](#)
- [Python's raise: Effectively Raising Exceptions in Your Code](#)
- [Python's Built-in Exceptions: A Walkthrough With Examples](#)
- [Object-Oriented Programming \(OOP\) in Python](#)
- [Logging in Python](#)

Learn Python	Courses & Paths	Community	Membership	Company
<a href="#">Start Here</a>	<a href="#">Learning Paths</a>	<a href="#">Podcast</a>	<a href="#">Plans &amp; Pricing</a>	<a href="#">About Us</a>
<a href="#">Learning Resources</a>	<a href="#">Quizzes &amp; Exercises</a>	<a href="#">Newsletter</a>	<a href="#">Team Plans</a>	<a href="#">Team</a>
<a href="#">Code Mentor</a>	<a href="#">Browse Topics</a>	<a href="#">Community Chat</a>	<a href="#">For Business</a>	<a href="#">Mission &amp; Values</a>
<a href="#">Python Reference</a>	<a href="#">Live Courses</a>	<a href="#">Office Hours</a>	<a href="#">For Schools</a>	<a href="#">Editorial Guidelines</a>
<a href="#">Python Cheat Sheet</a>	<a href="#">Books</a>	<a href="#">Learner Stories</a>	<a href="#">Reviews</a>	<a href="#">Sponsorships</a>
<a href="#">Support Center</a>				<a href="#">Careers</a>
				<a href="#">Press Kit</a>
				<a href="#">Merch</a>



[Privacy Policy](#) • [Terms of Use](#) • [Security](#) • [Contact](#)

 Happy Pythoning!

© 2012–2026 DevCademy Media Inc. DBA Real Python. All rights reserved.

REALPYTHON™ is a trademark of DevCademy Media Inc.