



— 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.

Absolute vs Relative Imports in Python

by Mbithe Nzomo  Sep 19, 2018  30 Comments  best-practices intermediate python[Mark as Completed](#)[Tweet](#)[Share](#)[Email](#)

Table of Contents

- A Quick Recap on Imports
 - How Imports Work
 - Syntax of Import Statements
 - Styling of Import Statements
- Absolute Imports
 - Syntax and Practical Examples
 - Pros and Cons of Absolute Imports
- Relative Imports
 - Syntax and Practical Examples
 - Pros and Cons of Relative Imports
- Conclusion

Your Guide to the Python Programming Language and a Best Practices Handbook

python-guide.org

[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: [Absolute vs Relative Imports in Python](#)

If you've worked on a Python project that has more than one file, chances are you've had to use an import statement before.

Even for Pythonistas with a couple of projects under their belt, imports can be confusing! You're probably reading this because you'd like to gain a deeper understanding of imports in Python, particularly absolute and relative imports.

In this tutorial, you'll learn the differences between the two, as well as their pros and cons. Let's dive right in!

Free Bonus: 5 Thoughts On Python Mastery, a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

All Tutorial Topics

advanced api basics best-practices
community databases data-science
devops django docker flask front-end
gamedev gui intermediate
machine-learning projects python testing
tools web-dev web-scraping

Your Guide to the Python Programming Language

python-guide.org



Table of Contents

- A Quick Recap on Imports
- Absolute Imports
- Relative Imports
- Conclusion

[Mark as Completed](#)[Tweet](#) [Share](#) [Email](#)

 Recommended Video Course

Absolute vs Relative Imports in Python

**Pip, PyPI, Virtualenv:** How to Set It All Up

Avoid common Python packaging headaches with our free class:
[» Click here to get the first lesson](#)

A Quick Recap on Imports

You need to have a good understanding of [Python modules and packages](#) to know how imports work. A Python module is a file that has a `.py` extension, and a Python package is any folder that has modules inside it (or, in Python 2, a folder that contains an `__init__.py` file).

What happens when you have code in one module that needs to access code in another module or package? You import it!



A Python Best Practices Handbook

python-guide.org

Remove ads

How Imports Work

But how exactly do imports work? Let's say you import a module `abc` like so:

```
Python
```

```
import abc
```

The first thing Python will do is look up the name `abc` in `sys.modules`. This is a cache of all modules that have been previously imported.

If the name isn't found in the module cache, Python will proceed to search through a list of built-in modules. These are modules that come pre-installed with Python and can be found in the [Python Standard Library](#). If the name still isn't found in the built-in modules, Python then searches for it in a list of directories defined by `sys.path`. This list usually includes the current directory, which is searched first.

When Python finds the module, it binds it to a name in the local scope. This means that `abc` is now defined and can be used in the current file without throwing a `NameError`.

If the name is never found, you'll get a `ModuleNotFoundError`. You can find out more about imports in the Python documentation [here](#)!

Note: Security Concerns

Be aware that Python's import system presents some significant security risks. This is largely due to its flexibility. For example, the module cache is writable, and it is possible to override core Python functionality using the import system. Importing from third-party packages can also expose your application to security threats.

Here are a couple of interesting resources to learn more about these security concerns and how to mitigate them:

- [10 common security gotchas in Python and how to avoid them](#) by Anthony Shaw (Point 5 talks about Python's import system.)
- [Episode #168: 10 Python security holes and how to plug them](#) from the TalkPython podcast (The panelists begin talking about imports at around the 27:15 mark.)

Syntax of Import Statements

Now that you know how import statements work, let's explore their syntax. You can import both packages and modules. (Note that importing a package essentially imports the package's `__init__.py` file as a module.) You can also import specific objects from a package or module.

There are generally two types of import syntax. When you use the first one, you import the resource directly, like this:

```
Python
```

```
import abc
```

abc can be a package or a module.

When you use the second syntax, you import the resource from another package or module.

Here's an example:

Python

```
from abc import xyz
```

xyz can be a module, subpackage, or object, such as a class or function.

You can also choose to rename an imported resource, like so:

Python

```
import abc as other_name
```

This renames the imported resource abc to other_name within the script. It must now be referenced as other_name, or it will not be recognized.

Styling of Import Statements

[PEP 8](#), the official style guide for Python, has a few pointers when it comes to writing import statements. Here's a summary:

1. Imports should always be written at the top of the file, after any module comments and [docstrings](#).
2. Imports should be divided according to what is being imported. There are generally three groups:
 - standard library imports (Python's built-in modules)
 - related third party imports (modules that are installed and do not belong to the current application)
 - local application imports (modules that belong to the current application)
3. Each group of imports should be separated by a blank space.

It's also a good idea to order your imports alphabetically within each import group. This makes finding particular imports much easier, especially when there are many imports in a file.

Here's an example of how to style import statements:

Python

```
"""Illustration of good import statement styling.
```

```
Note that the imports come after the docstring.
```

```
"""

# Standard library imports
import datetime
import os

# Third party imports
from flask import Flask
from flask_restful import Api
from flask_sqlalchemy import SQLAlchemy

# Local application imports
from local_module import local_class
from local_package import local_function
```

The import statements above are divided into three distinct groups, separated by a blank space. They are also ordered alphabetically within each group.



[Remove ads](#)

Absolute Imports

You've gotten up to speed on how to write import statements and how to style them like a pro. Now it's time to learn a little more about absolute imports.

An absolute import specifies the resource to be imported using its full path from the project's root folder.

Syntax and Practical Examples

Let's say you have the following directory structure:

```
└── project
    ├── package1
    │   ├── module1.py
    │   └── module2.py
    └── package2
        ├── __init__.py
        ├── module3.py
        ├── module4.py
        └── subpackage1
            └── module5.py
```

There's a directory, `project`, which contains two sub-directories, `package1` and `package2`. The `package1` directory has two files, `module1.py` and `module2.py`.

The `package2` directory has three files: two modules, `module3.py` and `module4.py`, and an initialization file, `__init__.py`. It also contains a directory, `subpackage`, which in turn contains a file, `module5.py`.

Let's assume the following:

1. `package1/module2.py` contains a function, `function1`.
2. `package2/__init__.py` contains a class, `class1`.
3. `package2/subpackage1/module5.py` contains a function, `function2`.

The following are practical examples of absolute imports:

Python

```
from package1 import module1
from package1.module2 import function1
from package2 import class1
from package2.subpackage1.module5 import function2
```

Note that you must give a detailed path for each package or file, from the top-level package folder. This is somewhat similar to its file path, but we use a dot (.) instead of a slash (/).

Pros and Cons of Absolute Imports

Absolute imports are preferred because they are quite clear and straightforward. It is easy to tell exactly where the imported resource is, just by looking at the statement. Additionally, absolute imports remain valid even if the current location of the import statement changes. In fact, PEP 8 explicitly recommends absolute imports.

Sometimes, however, absolute imports can get quite verbose, depending on the complexity of the directory structure. Imagine having a statement like this:

Python

```
from package1.subpackage2.subpackage3.subpackage4.module5 import function6
```

That's ridiculous, right? Luckily, relative imports are a good alternative in such cases!

Relative Imports

A relative import specifies the resource to be imported relative to the current location—that is, the location where the import statement is. There are two types of relative imports: implicit and explicit. Implicit relative imports have been deprecated in Python 3, so I won't be covering them here.

Your Weekly Dose of All Things Python!

pycoders.com



[Remove ads](#)

Syntax and Practical Examples

The syntax of a relative import depends on the current location as well as the location of the module, package, or object to be imported. Here are a few examples of relative imports:

Python

```
from .some_module import some_class
from ..some_package import some_function
from . import some_class
```

You can see that there is at least one dot in each import statement above. Relative imports make use of dot notation to specify location.

A single dot means that the module or package referenced is in the same directory as the current location. Two dots mean that it is in the parent directory of the current location—that is, the directory above. Three dots mean that it is in the grandparent directory, and so on. This will probably be familiar to you if you use a Unix-like operating system!

Let's assume you have the same directory structure as before:

```
└── project
    ├── package1
    │   ├── module1.py
    │   └── module2.py
    └── package2
        ├── __init__.py
        ├── module3.py
        ├── module4.py
        └── subpackage1
            └── module5.py
```

Recall the file contents:

1. package1/module2.py contains a function, `function1`.
2. package2/__init__.py contains a class, `class1`.
3. package2/subpackage1/module5.py contains a function, `function2`.

You can import `function1` into the `package1/module1.py` file this way:

Python

```
# package1/module1.py

from .module2 import function1
```

You'd use only one dot here because `module2.py` is in the same directory as the current module, which is `module1.py`.

You can import `class1` and `function2` into the `package2/module3.py` file this way:

Python

```
# package2/module3.py

from . import class1
from .subpackage1.module5 import function2
```

In the first import statement, the single dot means that you are importing `class1` from the current package. Remember that importing a package essentially imports the package's `__init__.py` file as a module.

In the second import statement, you'd use a single dot again because `subpackage1` is in the same directory as the current module, which is `module3.py`.

Pros and Cons of Relative Imports

One clear advantage of relative imports is that they are quite succinct. Depending on the current location, they can turn the ridiculously long import statement you saw earlier to something as simple as this:

Python

```
from ..subpackage4.module5 import function6
```

Unfortunately, relative imports can be messy, particularly for shared projects where directory structure is likely to change. Relative imports are also not as readable as absolute ones, and it's not easy to tell the location of the imported resources.



A banner for "Learn Python Programming, By Example" from realpython.com. It features a cartoon illustration of a person sitting at a desk with a laptop, surrounded by books and a lightbulb, symbolizing programming and learning. A "Remove ads" button is located below the banner.

Conclusion

Good job for making it to the end of this crash course on absolute and relative imports! Now you're up to speed on how imports work. You've learned the best practices for writing import statements, and you know the difference between absolute and relative imports.

With your new skills, you can confidently import packages and modules from the Python standard library, third party packages, and your own local packages. Remember that you should generally opt for absolute imports over relative ones, unless the path is complex and would make the statement too long.

Thanks for reading!

[Mark as Completed](#) 

 **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: [Absolute vs Relative Imports in Python](#)



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}
```

Email Address

[Send Me Python Tricks »](#)

About **Mbithe Nzomo**



I'm an experienced software engineer with interests in artificial intelligence and machine learning. I'm currently a postgraduate student at The University of Manchester, studying Advanced Computer Science with a specialisation in AI.

[» More about Mbithe](#)

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

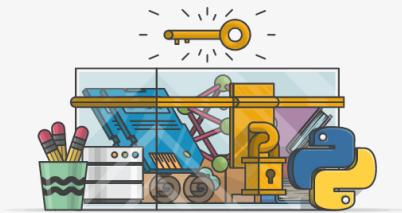


David



Joanna

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



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

[Level Up Your Python Skills »](#)

What Do You Think?

[Tweet](#) [Share](#) [Email](#)

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

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.

Keep Learning

Related Tutorial Categories: [best-practices](#) [intermediate](#) [python](#)

Recommended Video Course: [Absolute vs Relative Imports in Python](#)

Related Tutorial Categories: [best-practices](#) [intermediate](#) [python](#)

Recommended Video Course: [Absolute vs Relative Imports in Python](#)

Write Cleaner & More Pythonic Code

[realpython.com](#)



Help