**Complete Guide to Imports in Python: Absolute, Relative, and More**

How to plan your code so imports are clear and clean

by Aquiles CarattinoOct. 4, 2019Packageabsoluteimportingimportrelative

Importing is not only a matter of using external libraries, but it also allows you to keep your code clean and organized. In this tutorial, we are going to discuss from the very basics of importing to complex topics such as lazy loading of modules in your packages. You are free to skip ahead to the section that compels you the most.

**Introduction to importing**

In Python, whenever you want to import a module, you do the following:

import sys

Python

In this case, sys is a Python Standard Library that provides functions to interact with the interpreter itself. Python comes bundled with plenty of libraries for different tasks. You can find them all [here](https://docs.python.org/3/library/index.html). If you would like to use a function defined in this module, you can simply do:

sys.exit()

Python

Which will terminate your Python session. sys can do many more things than just quitting the interpreter. It can also tell you if you are running a script with some arguments. For example, you can write the following to a file **test\_argv.py**:

import sys

print(sys.argv)

Python

Now, you can run the file and see what the output is:

python test\_argv.py -b 1

Bash

Importing the entire sys module may not be what we want since we are only using one of its functions. In this case, we can also be very selective with what we want to import:

from sys import argv

print(argv)

Python

And the output will be the same. Understanding when you will use the full import or you will select just what you need depends a lot on your personal preference, on what do you want to achieve, and on what the library requires.

**Importing \***

On the examples above, we have seen that we were importing only one module from sys. If we would want to import more modules, we can specify them on the same line, for example:

from sys import argv, exit

print(argv)

exit()

Python

You can import as many packages as you want. A common practice, to avoid having lines that become too long, is to stack them vertically. For example, you could have something like this:

from sys import (api\_version,

argv,

base\_exec\_prefix,

exit,

)

Python

Note the use of the (, ) to make a clear list of imports. As you may imagine, if you need to import a lot of modules from a package, it becomes troublesome to make a list of all you need. Therefore, you may want to import all of the available modules at once. Python allows you to do it like this:

from sys import \*

print(api\_version)

print(argv)

exit()

Python

However, this is a highly discouraged practice. Since you are importing modules without control, some functions may get overwritten. Let's see it with the following example:

from time import \*

from asyncio import \*

Python

Perhaps you are aware that time has a function called sleep, which halts the execution of the program for a given number of seconds. If you write and run a script like the following:

print('Here')

sleep(1)

print('After')

Python

You will notice that there is no delay between the lines 'Here' and 'After'. What is happening is that both time and asyncio define a function sleep which behaves in very different ways. The amount of knowledge that you need to keep in your head to understand what is going on is so large, that most developers avoid using the \* when importing.

The case of time and asyncio is special, because both of them belong to the standard Python library. When you start using libraries defined by others, sometimes it is hard to know and remember all the modules and functions defined. Moreover, some names are so handy (like sleep), that you may find them defined in different packages.

Unless you know exactly what and why you would need to import \*, it is very wise to use the first syntax that we saw in the article:

import time

import asyncio

print('Here')

asyncio.sleep(1)

print('After')

time.sleep(1)

print('Finally')

Python

And now you know exactly what is going on, even if you haven't used the asyncio library before. When we discuss importing your modules, it will become much clearer how the Python importing machinery works.

**Importing As**

We say that when importing modules, sometimes we will find ourselves in the situation in which two packages define different functions with the same name. Such is the case of time and asyncio which both define sleep. To avoid this name clash when importing, Python allows us to change the name of what we are importing. We can do the following:

from asyncio import sleep as async\_sleep

from time import sleep as time\_sleep

print('Here')

async\_sleep(1)

print('After')

time\_sleep(1)

print('Finally')

Python

In this way, we can use either the sleep from asyncio or from time avoiding name clashes. With this, we import just the modules we want, and not the entire package, but still maintain our options open.

The example above is only one case in which the import as is handy. If you are used to generating plots with Python, probably you have encountered lines like this:

import matplotlib.pyplot as plt

import numpy as np

import pandas as pd

Python

The three lines above are ubiquitous in many scientific programs. They are so common that editors such as Pycharm can suggest you to import numpy if they see a line that includes something like np.. In the examples above, the import as is not to prevent name clashes, but to make the notation handier. Instead of typing:

matplotlib.pyplot.plot(x, y)

Python

You can simply type:

plt.plot(x, y)

Python

Different packages have different shortcuts. For example PyQtGraph is normally shortened as pg, and for sure different fields use different abbreviations. Importing Numpy as np or Pandas as pd is not mandatory. However, since it is what the community does, it will make your code much more readable.

**Note**

If you go through StackOverflow, you will see that more often than not, the line in which numpy is imported is omitted and you just see the use of np.

**Importing your code**

So far, we have seen how to import packages and modules developed by other people. Importing, however, is a great tool to structure different parts of your code into different files, making it much handier to maintain. Therefore, sooner or later you are going to find yourself importing your code. Let's start very simple and build up in complexity. In a file called **first.py** let's place the following code:

def first\_function():

print('This is the first function')

Python

In another file, let's call it **second.py**, let's put the following code:

from first import first\_function

first\_function()

Python

And you can run it:

$ python second.py

This is the first function

Bash

That is as easy as it gets. You define a function in a file, but you use that function in another file. Bear in mind that what we discussed in the previous sections still holds. You can do from first import first\_function as ff, for example. Having only scripts is just the beginning. At some point, you will also organize your code into folders. Let's create a folder called **module\_a**, within it, a new file, called **third.py**. So the folder structure is like this:

$ tree

.

├── first.py

├── module\_a

│ └── third.py

└── second.py

Bash

Let's add a new function in **third**. Bear in mind that the examples are incredibly basic in order not to lose the important concepts from sight:

def third\_function():

print('This is the third function')

Python

Now, let's edit **second.py** to import this new function:

from first import first\_function

from module\_a.third import third\_function

first\_function()

third\_function()

Python

If you run it as before, you will get the following output:

This is the first function

This is the third function

Bash

Pay attention to the notation we used to import the third\_function. We specified the folder, in this case, module\_a and then we referred to the file with a dot: .. We ended up having module\_a.third, and we stripped the .py. This already allows you to improve a lot your code and its structure, but it is just the tip of the iceberg.

Sometimes, when you start installing libraries, they have dependencies and you can easily lose track of every package installed. Let's see a very simple example. I will assume you have **numpy** already installed (however, the examples below will work with several packages). Create a new folder, called **numpy**, with a file called **sleep.py** the folder structure will end up looking like this:

.

├── first.py

├── module\_a

│ └── third.py

├── numpy

│ └── sleep.py

└── second.py

Bash

Within the file **sleep.py**, write the following lines of code:

def sleep():

print('Sleep')

Python

It is a very simple example. Now we can update **second.py** to use our new function sleep:

from numpy.sleep import sleep

sleep()

Python

The main question now is, how does Python know that it should import the sleep you just defined and not a module from the real *numpy*? If you go ahead and run the code, you should get the following error:

Traceback (most recent call last):

File "second.py", line 3, in <module>

from numpy.sleep import sleep

ModuleNotFoundError: No module named 'numpy.sleep'

Bash

This exception is utterly hard to understand. It is telling you that Python tried to look for a module called sleep in the *numpy* package, and not in our folder. The quick solution to this problem is to create an empty file called **\_\_init\_\_.py** in the numpy folder:

.

├── first.py

├── module\_a

│ └── third.py

├── numpy

│ ├── \_\_init\_\_.py

│ └── sleep.py

└── second.py

Bash

If you run the code again, you won't see any problems:

$ python second.py

Sleep

Bash

To explain what is going on, you need to understand how Python looks for packages on your computer. The topic is complex, and Python allows you a great deal of customization. The [official documentation](https://docs.python.org/3/reference/import.html) shines some light into the matter once you have experience. In short, Python will first look at whether what you are trying to import belongs to the standard library. This means that if we would have called the folder time instead of numpy, the behavior would have been different.

If Python doesn't find the module in its standard library, it will check for external modules. In it does this also in a very special order. It will first start by searching in the current directory, and then it will move to the directories where packages are installed (for example, when you do pip install numpy. Therefore, it is fair to ask yourself why in the first example it didn't work and in the second, after adding the empty **\_\_init\_\_.py** it did.

For Python to realize that a directory is a module, it must contain an **\_\_init\_\_.py** file. This is exactly so to prevent unintended name clashes, such as what happens with *numpy* in our example. Imagine you start developing another program in which you need to use numpy. How can you be sure you will import the proper numpy and not the one we have just developed? Python allows you to check which directories it will look into for importing:

import sys

for path in sys.path:

print(path)

Python

The code above will list all the directories that belong to the path. Probably you will see a list of around 4 or 6 folders, most of them quite logical: where Python is installed, your virtual environment folders, etc.

**Adding Folder to the Path**

The next logical question one can ask is whether the path in which Python looks for modules can be modified, and the answer is absolutely yes. The first option is to do it at runtime. You can easily append a directory to the variable sys.path. One relatively common practice is to add the current directory to the list of paths:

import os

import sys

CURR\_DIR = os.path.dirname(os.path.abspath(\_\_file\_\_))

print(CURR\_DIR)

sys.path.append(CURR\_DIR)

for path in sys.path:

print(path)

Python

The code above is straightforward if you go through it. You can add any path you want, not necessarily the current directory. One of the advantages of this approach is that you modify the system path only while your program runs. If you run two different programs, each will have its own path.

Another option is to modify the **PYTHONPATH** environment variable. Environment variables are available on every operating system, the only difference is how you can set and modify them. Many programs are designed in such a way that you can modify their behavior by setting some *global* variables, which are stored and handled by the operating system itself.

If you are on **Linux** or **Mac**, the command to set these variables is export, you would do the following:

export PYTHONPATH=$PYTHONPATH':/home/user/'

echo $PYTHONPATH

Bash

The first line appends the folder /home/user to the variable PYTHONPATH. Note that we have used : as a directory separator.

If you are on **Windows**, you need to right-click on "Computer", select "Properties". Check in the "Advanced System Settings" for the option "Environment variables". If PYTHONPATH exists, you can modify it, if it does not exist, you can create it by clicking on "New". Bear in mind that on Windows, you have to use ; to separate directories, since : is part of the folder path (e.g.: C:\Users\Test\...).

Once you modified your Python Path, you can run the following code:

import sys

for path in sys.path:

print(path)

Python

You will see that /home/user appears at the top of the list of directories. You can add another directory, for example:

export PYTHONPATH=$PYTHONPATH':/home/user/test'

Bash

And you will see it also appearing. Adding information to the Python Path is a great way of developing a structure on your own computer, with code in different folders, etc. It can also become hard to maintain. As a quick note, Python allows you to read environment variables at runtime:

import os

print(os.environ.get('PYTHONPATH'))

Python

Note that on Windows, the changes to environment variables are permanent, but on Linux and Mac you need to follow [extra steps](https://stackoverflow.com/questions/3402168/permanently-add-a-directory-to-pythonpath) if you want them to be kept.

**PYTHONPATH and Virtual Environment**

There is a very handy trick when you work with virtual environments which is to modify environment variables when you activate or deactivate them. This works seamlessly on Linux and Mac, but Windows users may require some tinkering to adapt the examples below.

If you inspect the **activate** script (located in the folder *venv/bin*) you can get inspiration about what is done with the PATH variable, for example. The first step is to store the old variable, before modifying it, then we append whatever we want. When we deactivate the virtual environment, we set the old variable back.

Virtual Environment has three hooks to achieve exactly this. Next to the **activate** script, you will see three more files, called *postactivate*, *postdeactivate* and *predeactivate*. Let's modify *postactivate*, which should be empty if you never used it before. Add the following:

PYTHONPATH\_OLD="$PYTHONPATH"

PYTHONPATH=$PYTHONPATH":/home/user"

export PYTHONPATH

export PYTHONPATH\_OLD

Bash

Next time you activate your virtual environment, you will have the directory /home/user added to the PYTHONPATH. It is a good practice to go back to the original version of the python path once you deactivate your environment. You can do it editing the **predeactivate** file:

PYTHONPATH="$PYTHONPATH\_OLD"

unset $PYTHONPATH\_OLD

Bash

With this, we set the variable to the status it had before activating and we remove the extra variable we created. Note that in case you don't deactivate the environment, but simply close the terminal, the changes to the PYTHONPATH won't be saved. The *predeactivate* script is important if you switch from one environment to another and keep using the same terminal.

In the examples of the previous sections, we imported a function *downstream* in the file system. This means, that the function was inside of a folder next to the main script file. What happens if we want to import from a sibling module? Imagine we have the following situation:

├── \_\_init\_\_.py

├── mod\_a

│ ├── file\_a.py

│ └── \_\_init\_\_.py

├── mod\_b

│ ├── file\_b.py

│ └── \_\_init\_\_.py

└── start.py

Bash

We have a **start** file at the top-level directory, we have two modules, **mod\_a** and **mod\_b**, each with its own **\_\_init\_\_** file. Now, imagine that the function you are developing inside of **file\_b** needs something defined in **file\_a**. Following what we saw earlier, it is easy to import from **start**, we would do just:

from mod\_a import file\_a

from mod\_b import file\_b

Python

To have a concrete example, let's create some dummy code. First, in the file **file\_a**, let's develop a simple function:

def simple():

print('This is simple A')

Python

Which, from the **start** file we can use as follows:

from mod\_a.fila\_a import simple

simple()

Python

If we want to use the same function within the **file\_b**, the first thing we can try is to simply copy the same line. Thus, open **file\_b** and add the following:

from mod\_a.file\_a import simple

def bsimple():

print('This is simple B')

simple()

Python

And we can edit **start** to look as follows:

from mod\_b import file\_b

file\_b.bsimple()

Python

If we run start, we will get the output we were expecting:

$ python start

This is simple B

This is simple

Bash

However, and this is very big, HOWEVER, sometimes we don't want to run **start**, we want to run directly **file\_b**. If we run it as it is, we are expecting no output, but we can try it anyway:

$ python file\_b.py

Traceback (most recent call last):

File "file\_b.py", line 1, in <module>

from mod\_a.file\_a import simple

ModuleNotFoundError: No module named 'mod\_a'

Bash

And here you start to realize the headaches that the importing in Python can generate as soon as your program gets a bit more sophisticated. What we are seeing is that depending on where in the file system we run Python, it will understand what mod\_a is. If you go back to the previous sections and see what we discussed about the Path used for searching modules, you will see that the first path is the current directory. When we run **start**, we are triggering Python from the root of our project and therefore it will find **mod\_a**. If we enter to a sub-directory, then it will no longer find it.

The same happens if we trigger python from any other folder:

$ python /path/to/project/start.py

Bash

Based on what we have discussed earlier, can you think of a solution to prevent the errors?

What we are doing in the examples above is called **absolute imports**. This means that we specify the full path to the module we want to import. What you have to remember is that the folder from which you trigger Python is the first place where the program looks for modules. Then it goes to the paths stored in sys.path. So, if we want the code above to work, we need to be sure that Python knows where **mod\_a** and **mod\_b** are stored.

The proper way would be to include the folder in the **PYTHONPATH** environment variable, as we explained earlier. A *dirtier* way would be to append the folder at runtime, we can add the following lines to **file\_by.py**:

import os

import sys

BASE\_PATH = os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)))

sys.path.append(BASE\_PATH)

from mod\_a.file\_a import simple

Python

This is very similar to what we have done earlier. The important line is the highlighted, it shows you a way of getting the full path to the folder one level above where the current file (**file\_b.py**) is. Note that you need to append to the sys.path before you try to import mod\_a, or it will fail such as before.

If you think about this approach, you can quickly notice that it has several drawbacks. The most obvious one is that you should add those lines to every single file you are working with. Imagine that later you develop **mod\_c** which depends also on **mod\_a**, you will need to append the folder to the path again, etc. This quickly becomes a nightmare.

Another problem with our current approach is that we are specifying the name of the module, but not the package to which it belongs. This connects back to what we did at the beginning of the article. Modules that belong to packages sometimes have the same names even if they are very different. Imagine you would like to develop a module called string. Perhaps you are a theoretical physicist working on string theory. If you have code that looks like this:

from string import m\_theory

Python

It will give you problems because string belongs to Python's standard library. It can also happen quite often that you develop two different packages and both have some modules with the same name. In the end, it is hard to come up with unique names, and things like *config*, *lib*, *util*, etc. are quite descriptive.

A better approach is to develop projects always in their own folder. The structure would be like this:

code

├── pckg\_a

│ └── file\_a.py

├── pckg\_b

│ ├── docs

│ │ ├── conf\_complete.py

│ │ └── output

│ │ └── output.txt

│ └── mod\_a

│ ├── factorial.py

│ ├── \_\_init\_\_.py

│ └── people.py

└── pckg\_c

├── another.py

├── \_\_init\_\_.py

├── mod\_a

│ ├── file\_a.py

│ └── \_\_init\_\_.py

├── mod\_b

│ ├── file\_b.py

│ └── \_\_init\_\_.py

└── start.py

Bash

In the folder tree above, you can see a base folder called **code**. Inside there are different packages, *a*, *b*, and *c*. If you check **pckg\_c** you will notice that it contains the code we were discussing in this tutorial. There are several advantages to working in this way. First, you can add just the folder **code** to the PYTHONPATH and you will have all your packages immediately available. The other advantage is that now you can import modules without risking mistakes:

from pckg\_b import mod\_a as one\_module

from pckg\_c import mod\_a as two\_module

Python

Now you see that it is very clear what module you are importing, even though they are both called mod\_a. Remember, absolute imports mean that you define the full path of what you want to import. However, in Python, full is *relative*. You are not specifying a path in the file system, but rather an import path. Therefore, it is impossible to think about absolute imports without also considering the PYTHONPATH.

**Relative Imports**

Another option for importing modules is to define the relative path. Let's continue building on the example from the previous section. Imagine you have a folder structure like this:

code

├── mod\_a

│ ├── file\_a.py

│ └── \_\_init\_\_.py

├── mod\_b

│ ├── file\_b.py

│ ├── \_\_init\_\_.py

│ └── mod\_a

│ ├── file\_c.py

│ └── \_\_init\_\_.py

└── start.py

Bash

Note that in this example, we are placing the files within a folder called **code**, which will be relevant later on. Each file\_X.py defines a function called function\_X (where X is the letter of the file). The function simply prints the name of the function. It is a very simple example to show how this structure works. By not it should be clear that if you would like to import function\_c from file\_c in the start.py file, you would simply do the following:

from mod\_b.mod\_a.file\_c import function\_c

Python

The situation becomes more interesting when you want to import function\_a into file\_b. It is important to pay attention because there are two different mod\_a defined. If we add the following to file\_b:

from mod\_a.file\_c import function\_c

Python

It would work, regardless of how you run the script:

$ python mod\_b/file\_b.py

$ cd mod\_b

$ python file\_b.py

Bash

But this is not what we wanted! We want function\_a from file\_a. If we, however, add the following to file\_b:

from mod\_a.file\_a import function\_a

Python

We would get the following error:

$ python mod\_b/file\_b.py

Traceback (most recent call last):

File "mod\_b/file\_b.py", line 1, in <module>

from mod\_a.file\_a import function\_a

ImportError: No module named file\_a

Bash

So, now is where relative imports come into play. From file\_b, the module we want to import is one folder up. In principle, it would be enough to write the following:

from ..mod\_a.file\_a import function\_a

def function\_b():

print('This is simple B')

function\_a()

function\_b()

Python

Most tutorials end at this point. They explain that the first . means this directory, while the second means going one level up, etc. However, if you run the file, there will be problems. If you are still using Python 2 (STRONGLY discouraged!), you would get the following error:

$ cd mod\_b/

$ python file\_b.py

Traceback (most recent call last):

File "file\_b.py", line 1, in <module>

from ..mod\_a.file\_a import function\_a

ValueError: Attempted relative import in non-package

Bash

This error means that you can't simply run the file as if it would be a script. You have to tell Python that it is a package. The way of running a script as if it would be a package is to add a -m, you need to be one folder up to work:

$ python -m mod\_b.file\_b

Bash

**Python 3** users can simply run the file as always, and you will get the following error:

$ python3 file\_b.py

Traceback (most recent call last):

File "file\_b.py", line 1, in <module>

from ..mod\_a.file\_a import function\_a

ValueError: attempted relative import beyond top-level package

Bash

It doesn't matter if you change folders, if you move one level up, you will get the same problem:

$ python3 mod\_b/file\_b.py

Traceback (most recent call last):

File "mod\_b/file\_b.py", line 1, in <module>

from ..mod\_a.file\_a import function\_a

ValueError: attempted relative import beyond top-level package

Bash

At some point, this becomes nerve-wracking. It doesn't matter if you add folders to the PATH, create **\_\_init\_\_.py** files, etc. It all boils down to the fact that you are not treating your files as a package. **Python 2** was showing a different error message that could point into the direction of solving the problem, but for **Python 3** it became slightly more cryptical. To instruct Python to run your file as part of a package, you would need to do the following:

$ python3 -m code.mod\_b.file\_b

This is function\_b

This is function\_a

Bash

Bear in mind that the only way of running the code like this is if python knows where to find the folder code. And this brings us back to the discussion of the PYTHONPATH variables. If you are in the folder that contains code and run Python from there, you won't see any problems. If you, however, are in any other folder on your computer, Python will follow to usual rules to try to understand where code is.

There is one more detail with relative imports. Imagine that **file\_c** has the following:

from ..file\_b import function\_b

def function\_c():

print('This is function c')

function\_b()

function\_c()

Python

Since **file\_c** is deeper, we can try to run it in different ways:

$ python -m code.mod\_b.mod\_a.file\_c

$ python -m mod\_b.mod\_a.file\_c

Bash

However, the second option is going to fail. **file\_c** is importing **file\_b** which in turn is importing **file\_a**. Therefore, Python needs to be able to go all the way to the root of code. This is, however, not always the case. It depends on how the code was developed. Bear in mind that imports work equally well if you import code into another project, provided that Python knows where to find it:

>>> from code.mod\_b.mod\_a.file\_c import function\_c

Pycon

The last detail is that you can't mix relative imports and what we have done at the beginning of this section. If you add the following to **file\_b**:

from ..mod\_a.file\_a import function\_a

from mod\_a.file\_c import function\_c

def function\_b():

print('This is function\_b')

function\_a()

function\_b()

Python

You will get the following error:

$ python -m code.mod\_b.file\_b

Traceback (most recent call last):

[...]

ModuleNotFoundError: No module named 'mod\_a'

Bash

When you decide to run your code as a module (using the -m), then you have to make all the imports explicit. One way of solving the problem with our code above would be to change one line:

from .mod\_a.file\_c import function\_c

Python

Then it is clear that we want to import from mod\_a which is in the same folder than the **file\_b**.

**Mixing Absolute and Relative**

This is possible. There is no secret to it, for example, we can change **file\_b.py** to look like this:

from ..mod\_a.file\_a import function\_a

from code.mod\_b.mod\_a.file\_c import function\_c

def function\_b():

print('This is function\_b')

function\_c()

function\_b()

Python

And if we run the file, there won't be any issues:

$ python -m code.mod\_b.file\_b

Bash

So, now you see that you can easily mix relative and absolute imports.

**Absolute or Relative: Conclusions**

Deciding whether you want to use absolute imports or relative imports is up to your own taste. If you are developing a package that has a lot of modules nested to each other, using relative imports can make your code clearer. For example, this is how the same import would look like in the two different cases:

from package.module\_1.module\_2.module\_3.file import my\_function

from .file import my\_function

Python

If you would be developing **file\_2** next to **file**, importing my\_function can become very different depending on whether you go the absolute or the relative path.

However, it is important to consider that typing less is not the only factor at play here. Remember, that once you use relative imports, you can't simply run your files using python file\_2.py, you will need to use the python -m command, and it will mean that you will need to write the full path to the file within the package.

Sometimes bigger programs can be executed in parts, to test the behavior and give a quick feeling of what the code does without running a full-fledged solution. Therefore, it is really up to the developer to have sensitivity and decide whether the absolute import or the relative import makes the code clearer and the execution easier.

The example code for this article can be found [on Github](https://github.com/PFTL/website/tree/master/example_code/37_imports)