# Absolute vs Relative Imports in Python

## How Imports Work

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

Python

```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`.

## Syntax of Import Statements:

Ex1:

import abc

Ex2:

from abc import xyz

Ex3:

import abc as other_name

**an example of how to style import**

```
"""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
```

## 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

```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

```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:

There are two types of relative imports: implicit and explicit. Implicit relative imports have been deprecated in Python 3

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:

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

```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

```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.