# Relative Imports

PEP 328 describes how relative imports came about and what specific syntax was chosen. The idea behind it was to use periods to determine how to relatively import other packages / modules. The reason was to prevent the accidental shadowing of standard library modules. Let's use the example folder structure that PEP 328 suggests and see if we can get it to work:

```
my_package/
    __init__.py
    subpackage1/
        __init__.py
        module_x.py
        module_y.py
    subpackage2/
        __init__.py
        module_z.py
    module_a.py
```

Create the files and folders above somewhere on your hard drive. In the top-level __init__.py, put the following code in place:

```
from . import subpackage1
from . import subpackage2
```

Next navigate down in subpackage1 and edit its __init__.py to have the following contents:

```
from . import module_x
from . import module_y
```

Now edit module_x.py such that is has the following code:

```
from .module_y import spam as ham

def main():
    ham()
```

```
    ham()
```

Finally edit module_y.py to match this:

```
def spam():
    print('spam ' * 3)
```

Open a terminal and cd to the folder that has my_package, but not into my_package. Run the Python interpreter in this folder. I'm using iPython below mainly because its auto-completion is so handy:

```
In [1]: import my_package

In [2]: my_package.subpackage1.module_x
Out[2]: <module 'my_package.subpackage1.module_x' from 'my_package/subpackage1/module_x.py'>

In [3]: my_package.subpackage1.module_x.main()
spam spam spam
```

Relative imports are great for creating code that you turn into packages. If you have created a lot of code that is related, then this is probably the way to go. You will find that relative imports are used in many popular packages on the Python Packages Index (PyPI). Also note that if you need to go more than one level, you can just use additional periods. However, according to PEP 328, you really shouldn't go above two.

Also note that if you were to add an "if __name__ == '__main__'" portion to the module_x.py and tried to run it, you would end up with a rather confusing error. Let's edit the file and give it a try!

```
from . module_y import spam as ham

def main():
    ham()

if __name__ == '__main__':
    # This won't work!
    main()
```

Now navigate into the subpackage1 folder in your terminal and run the following command:

following command:

```
python module_x.py
```

You should see the following error on your screen for Python 2:

```
Traceback (most recent call last):
  File "module_x.py", line 1, in <module>
    from . module_y import spam as ham
ValueError: Attempted relative import in non-package
```

And if you tried to run it with Python 3, you'd get this:

```
Traceback (most recent call last):
  File "module_x.py", line 1, in <module>
    from . module_y import spam as ham
SystemError: Parent module '' not loaded, cannot perform relative import
```

What this means is that module_x.py is a module inside of a package and you're trying to run it as a script, which is incompatible with relative imports.

If you'd like to use this module in your code, you will have to add it to Python's import search path. The easiest way to do that is as follows:

```
import sys
sys.path.append('/path/to/folder/containing/my_package')
import my_package
```

Note that you want the path to the folder right above my_package, not my_package itself. The reason is that my_package is THE package, so if you append that, you'll have issues using the package. Let's move on to optional imports!