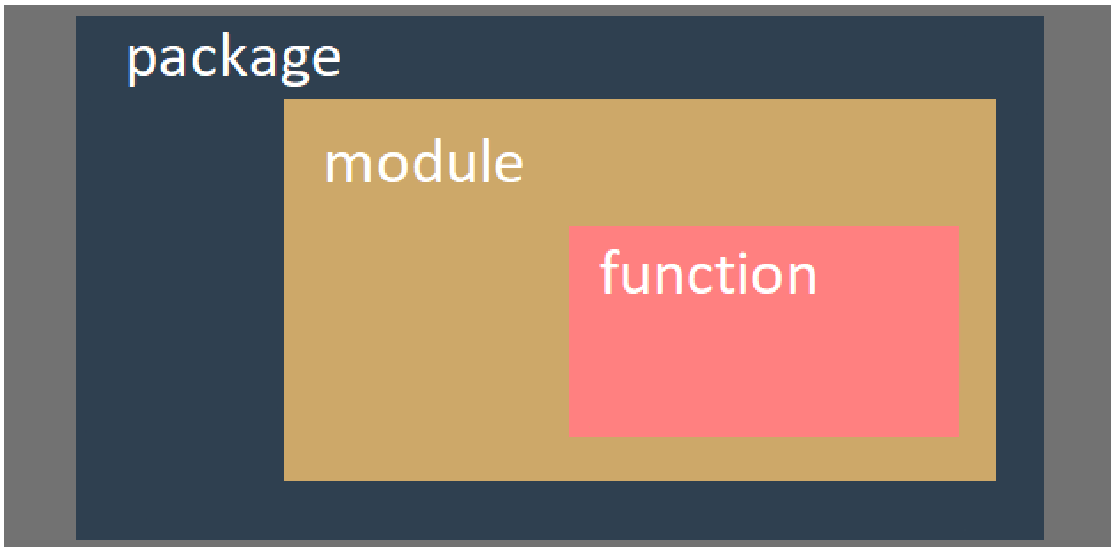
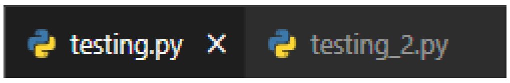
Packages in Python



In Modules in Python, we've talked about creating your own modules, we will dig deeper into that here.

Importing

First create two files with whatever name you'd like, make sure that they are in the **same folder**.



testing.py ×   testing_2.py

Import one from another, for example, in testing.py

```
import testing_2
```

You will notice that there is a new subfolder created, named __pychache__. The content inside is named `testing_2.cpython-310.pyc`, lets break it down. `testing_2` is the from the name of the second python file. `cpython` is the Python implementation that has created the file (CPython in this case). `310` says the version of your python, in this case, Python 3.10. The file extension `pyc` comes from the words Python and Compiled.

When Python imports a module for the first time, it translates its contents into a somewhat compiled shape, but the files doesn't contain any machine code, it's just internal Python **semi-compiled** code. It is made ready to be executed by Python's interpreter.

As such file doesn't require lots of the checks needed for a pure source file, the execution starts faster and the runtime is faster.

Testing

Now lets put this into `testing_2`.py.

```
print("Hello, I am a module")
```

Run testing (importing `testing_2`), the words "Hello, I am a module" will be shown in the console. When the module is imported, its contents is executed by Python. The module thus can initialize some variables or internal aspects.

The initialization process only takes place **once**, for example you imported `testing_2` in `testing.py`, and both `testing_2.py` and `testing.py` imported `testing_3`, `testing_3` will only be imported once, whichever is earlier.

Variables

Variables are **not** shared among different modules, each source code has its own.

Result running `testing.py`.

```
import testing_2
print("testing.py")
print(__name__)

# output:
# Hello, I am a module
# testing_2
# testing.py
# __main__
```

Result running `testing_2.py`.

```
print("Hello, I am a module")
print(__name__)

# output:
# Hello, I am a module
# __main__
```

Therefore, it can be shown that if you run a file directly, the __name__ variable is set to __main__. When a file is imported as a module, its __name__ variable is set to the file's name (excluding .py).

Put this into testing.py

```
import testing_2
# Called as module
```

And this into testing_2.py

```
if __name__ == "__main__":
    print("Local run")
else:
    print("Called as module")
```

As you can see from the result, the console shows, "Called as module", this shows how the __name__ variable can be used.

Unlike many other programming languages, Python does **not** hide personal/private variables, however, there is a convention by starting the variable name with _ (1 underscore) or __ (2 underscores), telling the user that they shouldn't modify it under. This is only a **convention**, the user may or may not obey it.

A few things to note when you're actually writing your module

```
#!/usr/bin/env python3
```

This has many names, "shebang", "shebang", "hashbang", "poundbang" or even "hashpling". The name doesn't matter as what it does is far more important. From Python's point of view, it's just a comment as it starts with #. For Unix and Unix-like OSs (including MacOS) such a line instructs the OS how to execute the contents of the file (in other words, what program needs to be launched to interpret the text). In some environments (especially those connected with web servers) the absence of that line will cause trouble.

```
""" module.py - an example of a Python module """
```

A string (maybe a multiline) placed before any module instructions (including imports) is called the doc-string, and should briefly explain the purpose and contents of the module.

Importing via file location/file path

The codes for the module and main aren't in the same folder?

You can use the sys module, as followed.

```
from sys import path
path.append('..\\modules') # file path goes here
import module
```

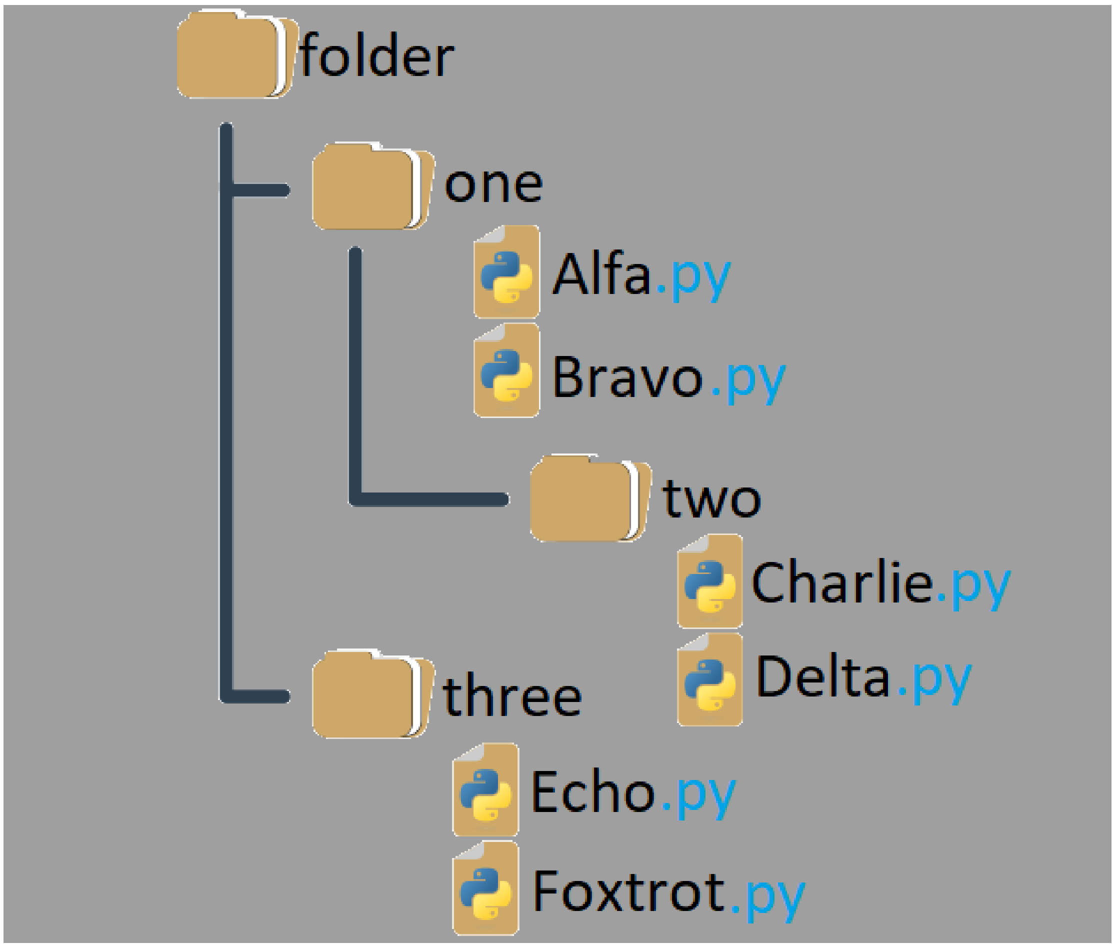
Important: remember to add an extra '\' character before the actual one from the file path, as the first '\' is an escape for the second one.

Suppose we are using Windows (file path format differ from OS)

After importing all the paths with the following code.

```
from sys import path
path.append('..\\modules')
```

And let's say that you've imported a directory of files and the tree looks like this



And you've put a function `algo()` in `Alfa.py`. You can call it by this line.

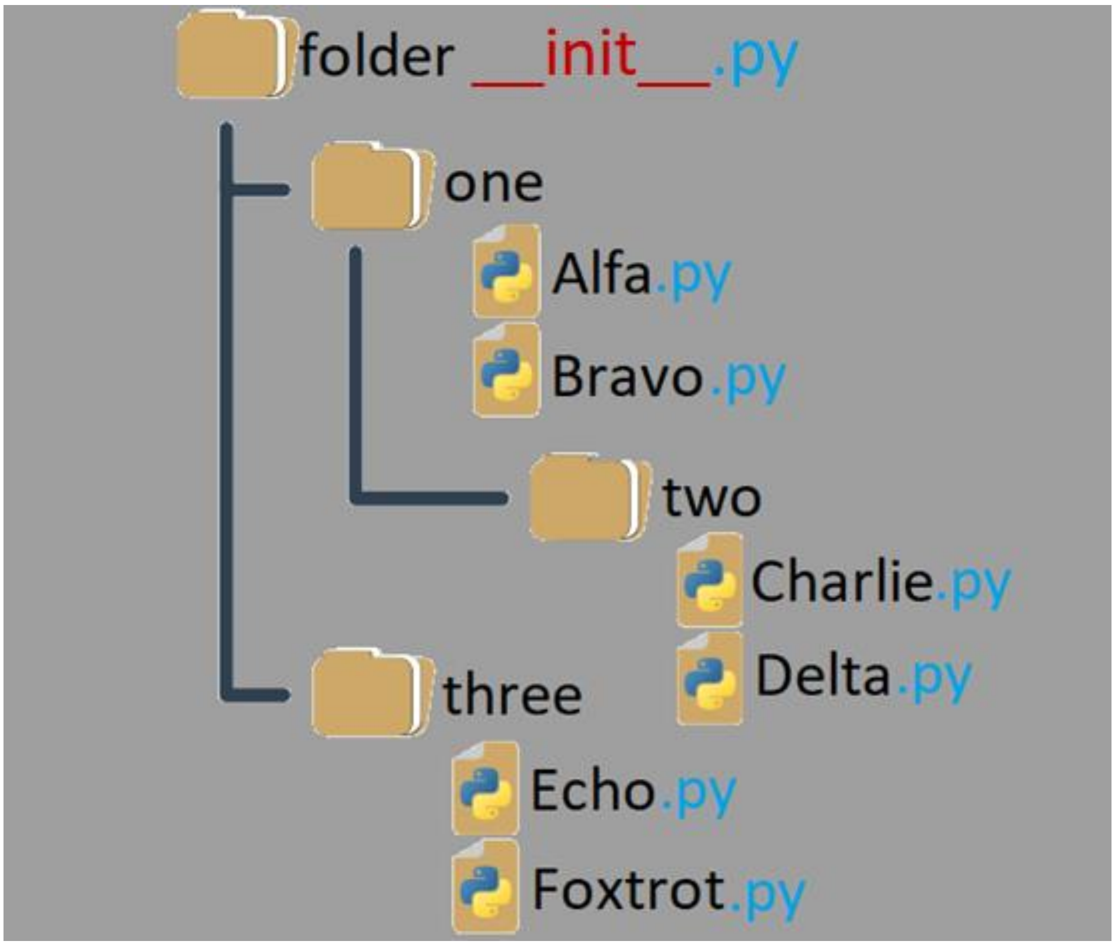
```
folder.one.Alfa.algo()
```

How do you transform such a tree into a real Python package?

You’ll need initialization, just like modules.

In modules you can do initialization by an unbound code (not a part of any function), however that’s not the case of packages.

You’ll need a very unique file called `__init__.py` in the package’s folder, the content inside is executed when any of the package’s modules is imported, if you don’t want to initialize anything, you can leave the file empty, but you **must** create such a file.



It’s not only the root folder that can contain __init__.py file, you can put it inside any of its subfolders too. It may be useful if some of the sub-packages (subfolders) need special initialization.

Key points (Personally I think that this is quite a hard area)

1. A module is designed to couple together some related entities (functions, variable, constants, etc.)
2. A package is a container which enables the coupling of several related modules under one common name. Such a container can be distributed as a batch of files deployed in a directory sub-tree or it can be packed inside a zip file.
3. During the very first import of the actual module, Python translates its source code into semi-compiled format (`.pyc`) files, and deploys these files into a folder called `__pycache__` under the module's home directory.
4. Private variables or entities does not exist in Python, however there is a convention saying that entities starting with _ (1 underscore) or __ (2 underscores) are private. Keep in mind that this is only a convention, users may or may not obey.
5. `#!`, shabang, shebang, hasbang, poundbang and hashpling (whatever it's called), is used to instruct Unix-like Oss how the Python source file should be launched. This convention has no effect under Microsoft Windows.
6. If you want to convince Python that it should take into account a non-standard package's directory, its name needs to be inserted/appended into the import directory list stored in the `path` variable, by the `sys` module.
7. A Python file named `__init__.py` is implicitly run when a package containing it is subject to import, and is used to initialize a package and/or its sub-packages (if there are any). This file **can** be empty, but **must** exist.