Modules and packages

Mojo provides a packaging system that allows you to organize and compile code libraries into importable files. This page introduces the necessary concepts about how to organize your code into modules and packages (which is a lot like Python), and shows you how to create a packaged binary with the `mojo package` command.

Mojo modules

To understand Mojo packages, you first need to understand Mojo modules. A Mojo module is a single Mojo source file that includes code suitable for use by other files that import it. For example, you can create a module to define a struct such as this one:

mymodule.mojo

```
struct MyPair:
    var first: Int
    var second: Int

    fn __init__(inout self, first: Int, second: Int):
        self.first = first
        self.second = second

    fn dump(self):
        print(self.first, self.second)
```

Notice that this code has no `main()` function, so you can't execute `mymodule.mojo`. However, you can import this into another file with a `main()` function and use it there.

For example, here's how you can import `MyPair` into a file named `main.mojo` that's in the same directory as `mymodule.mojo`:

main.mojo

```
from mymodule import MyPair

fn main():
    var mine = MyPair(2, 4)
    mine.dump()
```

Alternatively, you can import the whole module and then access its members through the module name. For example:

main.mojo

```mojo
import mymodule

fn main():
    var mine = mymodule.MyPair(2, 4)
    mine.dump()
```

You can also create an alias for an imported member with `as`, like this:

main.mojo

```mojo
import mymodule as my

fn main():
    var mine = my.MyPair(2, 4)
    mine.dump()
```

In this example, it only works when `mymodule.mojo` is in the same directory as `main.mojo`. Currently, you can't import `.mojo` files as modules if they reside in other directories. That is, unless you treat the directory as a Mojo package, as described in the next section.

Note

A Mojo module may include a `main()` function and may also be executable, but that's generally not the practice and modules typically include APIs to be imported and used in other Mojo programs.

Mojo packages

A Mojo package is just a collection of Mojo modules in a directory that includes an `__init__.mojo` file. By organizing modules together in a directory, you can then import all the modules together or individually. Optionally, you can also compile the package into a `.mojopkg` or `.📦` file that's easier to share and still compatible with other system architectures.

You can import a package and its modules either directly from source files or from a compiled `.mojopkg`/`.📦` file. It makes no real difference to Mojo which way you import a package. When importing from source files, the directory name works as the package name, whereas when importing from a compiled package, the filename is the package name (which you specify with the `mojo package` command—it can differ from the directory name).

For example, consider a project with these files:

```
main.mojo
mypackage/
    __init__.mojo
    mymodule.mojo
```

`mymodule.mojo` is the same code from examples above (with the `MyPair` struct) and `__init__.mojo` is empty.

In this case, the `main.mojo` file can now import `MyPair` through the package name like this:

```
main.mojo

from mypackage.mymodule import MyPair

fn main():
    var mine = MyPair(2, 4)
    mine.dump()
```

Notice that the `__init__.mojo` is crucial here. If you delete it, then Mojo doesn't recognize the directory as a package and it cannot import `mymodule`.

Then, let's say you don't want the `mypackage` source code in the same location as `main.mojo`. So, you can compile it into a package file like this:

```
$ mojo package mypackage -o mypack.mojopkg
```

Note

A `.mojopkg` file contains non-elaborated code, so you can share it across systems. The code becomes an architecture-specific executable only after it's imported into a Mojo program that's then compiled with `mojo build`.

Now, you can move the `mypackage` source somewhere else, and the project files now look like this:

```
main.mojo
mypack.mojopkg
```

Because we named the package file different from the directory, we need to fix the import statement and it all works the same:

```
main.mojo

from mypack.mymodule import MyPair
```

Note

If you want to rename your package, you cannot simply edit the `.mojopkg` or `.📦` filename, because the package name is encoded in the file. You must instead run `mojo package` again to specify a new name.

The `__init__` file

As mentioned above, the `__init__.mojo` file is required to indicate that a directory should be treated as a Mojo package, and it can be empty.

Currently, top-level code is not supported in `.mojo` files, so unlike Python, you can't write code in `__init__.mojo` that executes upon import. You can, however, add structs and functions, which you can then import from the package name.

However, instead of adding APIs in the `__init__.mojo` file, you can import module members, which has the same effect by making your APIs accessible from the package name, instead of requiring the `<package_name>.<module_name>` notation.

For example, again let's say you have these files:

```
main.mojo
mypackage/
    __init__.mojo
    mymodule.mojo
```

Let's now add the following line in `__init__.mojo`:

```
__init__.mojo

from .mymodule import MyPair
```

That's all that's in there. Now, we can simplify the import statement in `main.mojo` like this:

```
main.mojo

from mypackage import MyPair
```

This feature explains why some members in the Mojo standard library can be imported from their package name, while others required the `<package_name>.<module_name>` notation. For example, the `functional` module resides in the `algorithm` package, so you can import members of that module (such as the `map()` function) like this:

```
from algorithm.functional import map
```

However, the `algorithm/__init__.mojo` file also includes these lines:

```
algorithm/__init__.mojo

from .functional import *
from .reduction import *
```

So you can actually import anything from `functional` or `reduction` simply by naming the package. That is, you can drop the `functional` name from the import statement, and it also works:

```
from algorithm import map
```

Note

Which modules in the standard library are imported to the package scope varies, and is subject to change. Refer to the documentation for each module to see how you can import its members.