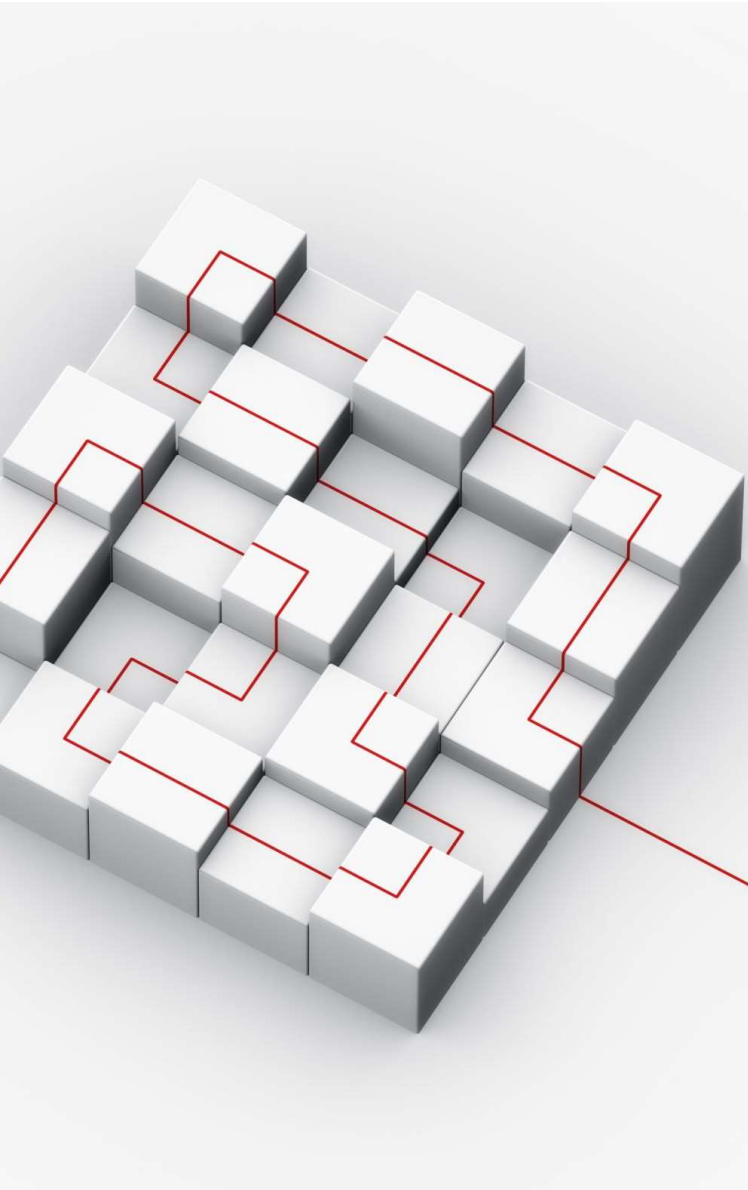


Module

Packages



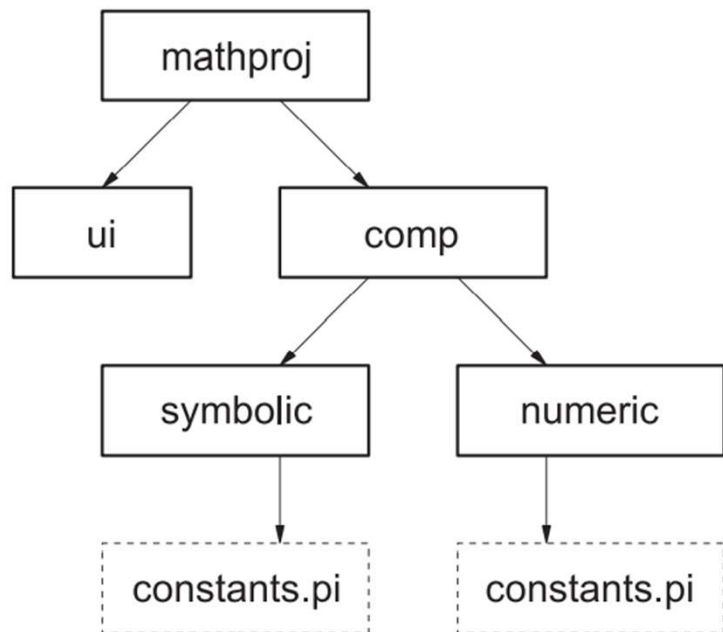
This chapter covers

- What is a package?
- A first example
- A concrete example
- The `__all__` attributes
- Absolute vs Relative Imports in Python
- The proper use of packages

What is a package?

- A module is a file containing code. A module defines a group of usually related Python functions or other objects. The name of the module is derived from the name of the file.
- When you understand modules, packages are easy, because a package is a directory containing code and possibly further subdirectories.
- A package contains a group of usually related code files (modules).
- The name of the package is derived from the name of the main package directory.

A first example



The constants.py file in the numeric part of the project defines pi as

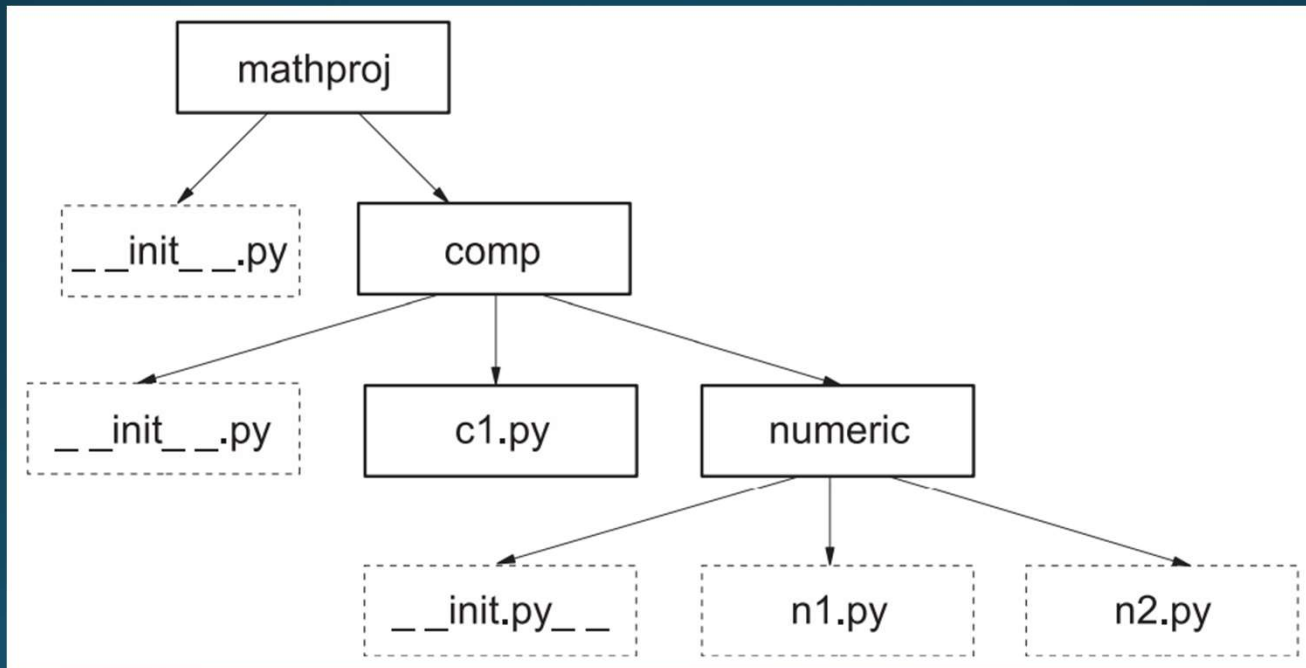
```
pi = 3.141592
```

whereas the constants.py file in the symbolic part of the project defines pi as

```
class PiClass:
    def __str__(self):
        return "PI"
pi = PiClass()
```

That's what packages are all about. They're ways of organizing very large collections of Python code into coherent wholes, by allowing the code to be split among different files and directories and imposing a module/submodule naming scheme based on the directory structure of the package files.

A concrete example - mathproj



`__init__.py` files in packages

- Python requires that a directory contain an `__init__.py` file before it can be recognized as a package. This requirement prevents directories containing miscellaneous Python code from being accidentally imported as though they defined a package.
- The `__init__.py` file is automatically executed by Python the first time a package or subpackage is loaded. This execution permits whatever package initialization you desire
- The first point is usually more important. For many packages, you won't need to put anything in the package's `__init__.py` file; just make sure that an empty `__init__.py` file is present.

The `__all__` attribute

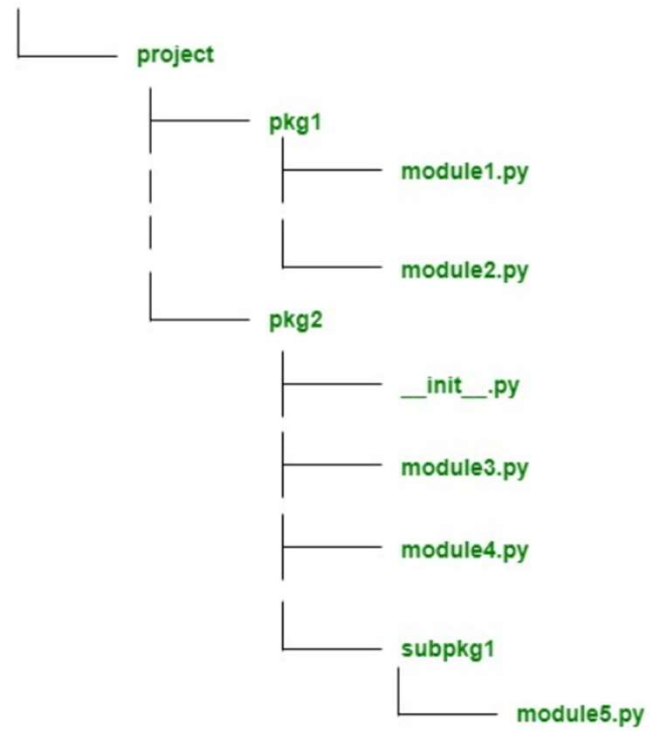
- If present in an `__init__.py` file, `__all__` should give a list of strings, defining those names that are to be imported when a `from ... import *` is executed on that particular package.
- If `__all__` isn't present, `from ... import *` on the given package does nothing.

Proper use of packages

- Packages shouldn't use deeply nested directory structures. Except for absolutely huge collections of code, there should be no need to do so. For most packages, a single top-level directory is all that's needed. A two-level hierarchy should be able to effectively handle all but a few of the rest. As written in The Zen of Python, by Tim Peters, "Flat is better than nested."
- Although you can use the `__all__` attribute to hide names from `from ... import *` by not listing those names, doing so probably is not a good idea, because it's inconsistent. If you want to hide names, make them private by prefacing them with an underscore.

Absolute imports in Python

- Absolute import involves a full path i.e., from the project's root folder to the desired module. An absolute import state that the resource is to be imported using its full path from the project's root folder.



```
1  # Python program showing
2  # practical example of
3  # absolute imports
4
5  # importing a fun1 from pkg1/module1
6  from pkg1.import module1 import fun1
7
8  from pkg1 import module2
9
10 # importing a fun2 from pkg2/module3
11 from pkg2 import module3 import fun2
12
13 # importing a fun3 from pkg2/subpkg1/module5
14 from pkg2.subpkg1.module5 import fun3
```

Pros and Cons of Absolute imports

Pros:

- Absolute imports are very useful because they are clear and straight to the point.
- Absolute import is easy to tell exactly from where the imported resource is, just by looking at the statement.
- Absolute import remains valid even if the current location of the import statement changes.

Cons:

If the directory structure is very big then usage of absolute imports is not meaningful. In such a case using relative imports works well.

Relative imports in Python

- Relative import specifies an object or module imported from its current location, that is the location where import statement resides.
- There two types of relative imports :
 - **Implicit** relative imports – Implicit relative import have been disapproved in Python(3.x).
 - **Explicit** relative imports – Explicit relative import have been approved in Python(3.x).

```
1  # Python program showing
2  # practical example of
3  # relative imports
4
5  # importing fun1 into pkg1/module1.py
6  from .module1 import fun1
7
8  # importing fun2 and fun3 into pkg2/module3.py
9  from .module3 import fun2
10 from .subpackage1.module5 import fun3
```

Pros and Cons of Relative imports

Pros:

- Working with relative imports is concise and clear.
- Based on the current location it reduces the complexity of an import statement.

Cons:

- Relative imports is not so readable as absolute ones.
- Using relative imports it is not easy because it is very hard to tell the location of a module.

The End