Python → Modules and packages → Packages

Theory: Packages

15 minutes 0 / 0 problems solved

Start practicing

2937 users solved this topic. Latest completion was **about 3 hours ago**.

Current topic:

✓ Packages Stage 1

Table of contents:

§1. Package definition and structure

When our code is becoming bigger, it becomes very difficult to maintain and keep track of all the modules included. In order to make the code more organized, we can resort to packages. A **package** is a way of structuring modules **hierarchically** with the help of the so-called **"dotted module names"**. Thus the module name `sun.moon` designates a submodule named "moon" in a package named "sun".

The possible **structure** might be the following:

```
package/                           # first we name the main or top-level package
    __init__.py            # this directory should be treated as a package
    subpackage/            # we can add subpackage with extra modules
          __init__.py      # this directory should be treated as a subpackage
                   artificial.py
                   amateurs.py
                   ...
            subpackage2/
                  __init__.py
                  amazing.py
                  animate.py
                  barriers.py
                  ...
```

NB: it's necessary to create `__init__.py` files, that will make Python treat the directory as a package/subpackage. They can be empty or execute the initialization code for the package.

§2. Importing and referencing packages

Let us suppose we'd like to import a specific module from the package. There are two ways to import the "artificial" submodule from the subpackage:

```
from package.subpackage import artificial
```

This method allows to use the submodule content without naming the package and subpackage:

```
artificial.function(arg1, arg2)
```

The second method is more straightforward:

```
import package.subpackage.artificial
```

After we've loaded the submodule in such a way, its content should be referenced with its **full name**:

```
package.subpackage.artificial.function(arg1, arg2)
```

Besides, it's possible to import a particular function from the submodule:

```
from package.subpackage.artificial import function
```

After that, you can address the `function()` directly, without specifying the full path to a module.

The method of importing modules depends on your current program and needs. The main rule is readability!

§3. Import * from ...: advantages and disadvantages

You can also use `from package.subpackage import *`. This code will import all the submodules that your subpackage has, although you might not really need that. Moreover, it will be really time-consuming and considered to be a bad practice. How can we manage these side-effects?

The major thing to do is to provide the package with a particular **index** with the help of `__all__` statement that should be inserted into `__init__.py file`. There you want to list the submodules to be imported while `from package import *` operation is executed.

```
__all__ = ["submodule1", "submodule10"]
```

§4. Intra package references

Python is even more powerful than you could imagine: you can refer to the submodules of **siblings packages** if needed. For instance, if you use the `package.subpackage1.artificial` and there you need something from `package.subpackage2.amazing`, you can import it by `from package.subpackage2 import amazing` in the **artificial.py** file.

You can also carry out the so-called "**relative imports**" that use leading dots to indicate the current and parent packages involved. Thereby, for "amateurs" you can use the following:

```
from . import artificial    # one dot means addressing to a current package/subpac
kage

from .. import subpackage2  # two dots mean addressing to a parent package/subpack
age

from ..subpackage2 import module
```

§5. PEP Time!

Using **wildcard imports** (`from <module> import *`) is considered bad practice, as they make it unclear which names are present in the namespace, confusing both readers and many automated tools.

Absolute imports are recommended, as they are usually more readable. They also give better error messages if something goes wrong:

```
import package.subpackage.amateurs
from package.mypackage import amateurs
```

Explicit relative imports are also acceptable, especially when dealing with complex package layouts where using absolute imports would be unnecessarily verbose:

```
from . import animate            # in amazing.py, for example
from .barriers import function   # in animate.py, for example
```

Standard library code should avoid complex package layouts and always use absolute imports.

§6. Conclusion

- Using packages is a very good way to structure your code.

- Packages make your project simpler to perceive. They allow reusing code more easily.
- Different ways of importing have their own advantages and disadvantages. Remember one of the main rules of Python: readability counts!