PACKAGES IN PYTHON

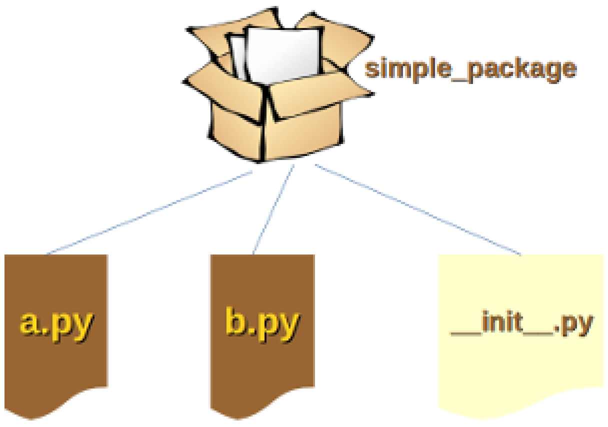
INTRODUCTION

We learned that modules are files containing Python statements and definitions, like function and class definitions. We will learn in this chapter how to bundle multiple modules together to form a package.

A package is basically a directory with Python files and a file with the name __init__.py. This means that every directory inside of the Python path, which contains a file named __init__.py, will be treated as a package by Python. It's possible to put several modules into a Package.

Packages are a way of structuring Python’s module namespace by using "dotted module names". A.B stands for a submodule named B in a package named A. Two different packages like P1 and P2 can both have modules with the same name, let's say A for example. The submodule A of the package P1 and the submodule A of the package P2 can be totally different.
A package is imported like a "normal" module.
We will start this chapter with a simple example.

A SIMPLE EXAMPLE



We will demonstrate with a very simple example how to create a package with some Python modules.
First of all, we need a directory. The name of this directory will be the name of the package, which we want to create. We will call our package "simple_package". This directory needs to contain a file with the name "__init__.py". This file can be empty, or it can contain valid Python code. This code will be executed when a package will be imported, so it can be used to initialize a package, e.g. to make sure that some other modules are imported or some values set. Now we can put into this directory all the Python files which will be the submodules of our module.

We create two simple files a.py and b.py just for the sake of filling the package with modules.
The content of a.py:

```
def bar():
    print("Hello, function 'bar' from module 'a' calling")
```

The content of b.py:

```
def foo():
    print("Hello, function 'foo' from module 'b' calling")
```

We will also add an empty file with the name __init__.py inside of simple_package directory.

Let's see whats happening, when we import simple_package from the interactive Python shell, assuming that the directory simple_package is either in the directory from which you call the shell or that it is contained in the search path or environment variable "PYTHONPATH":

```
>>> import simple_package
>>>
>>> simple_package
<module 'simple_package' from '/home/bernd/Dropbox
(Bodenseo)/websites/python-
course.eu/examples/simple_package/__init__.py'>
>>>
>>> simple_package/a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>>
>>> simple_package/b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
```

We can see that the package simple_package has been loaded but neither the module "a" nor the module "b"! We can import the modules a and b in the following way:

```
>>> from simple_package import a, b
>>> a.bar()
Hello, function 'bar' from module 'a' calling
>>> b.foo()
Hello, function 'foo' from module 'b' calling
>>>
```

As we have seen at the beginning of the chapter, we can't access neither "a" nor "b" by solely importing simple_package.

Yet, there is a way to automatically load these modules. We can use the file __init__.py for this purpose. All we have to do is add the following lines to the so far empty file __init__.py:

```
import simple_package.a
import simple_package.b
```

It will work now:

```
>>> import simple_package
>>>
>>> simple_package.a.bar()
Hello, function 'bar' from module 'a' calling
>>>
>>> simple_package.b.foo()
Hello, function 'foo' from module 'b' calling
```

A MORE COMPLEX PACKAGE

We will demonstrate in the following example how we can create a more complex package. We will use the hypothetical sound-Modul which is used in the official tutorial. (see https://docs.python.org/3/tutorial/modules.html)

```
sound
|-- effects
|   |-- echo.py
|   |-- __init__.py
|   |-- reverse.py
|   `-- surround.py
|-- filters
|   |-- equalizer.py
|   |-- __init__.py
|   |-- karaoke.py
|   `-- vocoder.py
|-- formats
|   |-- aiffread.py
|   |-- aiffwrite.py
|   |-- auread.py
|   |-- auwrite.py
|   |-- __init__.py
|   |-- wavread.py
|   `-- wavwrite.py
`-- __init__.py
```

You can download this example packages structure as a bzip-file. If we import the package "sound" by using the statement `import sound`, the package `sound` but not the subpackages `effects`, `filters` and `formats` will be imported, as we will see in the following example. The reason for this consists in the fact that the file `__init__.py` doesn't contain any code for importing subpackages:

```
>>> import sound
sound package is getting imported!
>>> sound
<module 'sound' from '/home/bernd/packages/sound/__init__.py'>
>>> sound.effects
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'sound' has no attribute 'effects'
```

If you also want to use the package `effects`, you have to import it explicitly with `import sound.effects`:

```
>>> import sound.effects
effects package is getting imported!
>>> sound.effects
<module 'sound.effects' from
'/home/bernd/packages/sound/effects/__init__.py'>
```

It is possible to have it done automatically when importing the sound module. For this purpose, we have to add the code line `import sound.effects` into the file `__init__.py` of the directory `sound`. The file should look like this now:

```
"""An empty sound package

This is the sound package, providing hardly anything!"""

import sound.effects
print("sound package is getting imported!")
```

If we import the package `sound` from the interactive Python shell, we will see that the subpackage `effects` will also be automatically loaded:

```
>>> import sound
effects package is getting imported!
sound package is getting imported!
```

Instead of using an absolute path we could have imported the `effects`-package relative to the `sound` package:

```
"""An empty sound package

This is the sound package, providing hardly anything!"""

from . import effects
print("sound package is getting imported!")
```

It is also possible to automatically import the package `formats`, when we are importing the `effects` package. We can also do this with a relative path, which we will include into the `__init__.py` file of the directory `effects`:

```
from .. import formats
```

Importing `sound` will also automatically import the modules `formats` and `effects`:

```
>>> import sound
formats package is getting imported!
effects package is getting imported!
sound package is getting imported!
```

You can download the sound package structure with all the changes we have made so far as a bzip-file.
To end this subchapter we want to show how to import the module `karaoke` from the package `filters` when we import the `effects` package. For this purpose we add the line
`from ..filters import karaoke`
into the `__init__.py` file of the directory `effects`. The complete file looks now like this:

```
"""An empty effects package

This is the effects package, providing hardly anything!"""

from .. import formats
from ..filters import karaoke
print("effects package is getting imported!")
```

Importing `sound` results in the following output:

```
>>> import sound
formats package is getting imported!
filters package is getting imported!
Module karaoke.py has been loaded!
effects package is getting imported!
sound package is getting imported!
```

We can access and use the functions of karaoke now:

```
>>> sound.filters.karaoke.func1()
Funktion func1 has been called!
>>>
```

IMPORTING A COMPLETE PACKAGE

For the next subchapter we will use again the initial example from the previous subchapter of our tutorial. We will add a module (file) `foobar` (filename: `foobar`) to the sound directory. The complete package can again be downloaded as a bzip-file. We want to demonstrate now, what happens, if we import the sound package with the star, i.e. `from sound import *`. Somebody might expect to import this way all the submodules and subpackages of the package. Let's see what happens:

```
>>> from sound import *
sound package is getting imported!
```

So we get the comforting message that the sound package has been imported. Yet, if we check with the `dir` function, we see that neither the module `foobar` nor the subpackages `effects`, `filters` and `formats` have been imported:

```
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']
```

Python provides a mechanism to give an explicit index of the subpackages and modules of a packages, which should be imported. For this purpose, we can define a list named __all__. This list will be taken as the list of module and package names to be imported when `from package import *` is encountered.

We will add now the line

```
__all__ = ["formats", "filters", "effects", "foobar"]
```

to the `__init__.py` file of the sound directory. We get a completely different result now:

```
>>> from sound import *
sound package is getting imported!
formats package is getting imported!
filters package is getting imported!
effects package is getting imported!
The module foobar is getting imported
>>>
```

Even though it is already apparent that all the modules have been imported, we can check with `dir` again:

```
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'effects', 'filters', 'foobar',
'formats']
>>>
```

The next question is what will be imported, if we use * in a subpackage:

```
>>> from sound.effects import *
sound package is getting imported!
effects package is getting imported!
>>> dir()
```

```
['__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']
>>>
```

Like expected the modules inside of `effects` have not been imported automatically. So we can add the following `__all__` list into the `__init__` file of the package `effects`:

```
__all__ = ["echo", "surround", "reverse"]
```

Now we get the intended result:

```
>>> from sound.effects import *
sound package is getting imported!
effects package is getting imported!
Module echo.py has been loaded!
Module surround.py has been loaded!
Module reverse.py has been loaded!
>>>
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'echo', 'reverse', 'surround']
>>>
```

Although certain modules are designed to export only names that follow certain patterns when you use import *, it is still considered bad practice. The recommended way is to import specific modules from a package instead of using *.