Learning Objectives: Modules and Packages

Learners will be able to...

- **Define a module**
- **Differentiate between a module and a package**
- **Import a package and its modules**
- **Describe the importance of `__init__.py`**
- **Use packages from the standard library**

info

Make Sure You Know

You are familiar with writing and importing modules. You also have experience writing functions and classes.

Limitations

We will only be working with modules written in Python. Subpackages are referenced, but not covered in-depth.

Reviewing Modules

What is a Module?

In the most simple terms, a module is nothing more than a valid Python file imported by another Python file. Copy the code below and paste it into the **top** panel. This panel will be the module.

```
# top panel

def my_func():
  print('I am a function')

class MyClass():
  @classmethod
  def hello(self):
    print('I am a class')
```

▼ **Did you notice?**
There are no special indicators that the file in the top panel is a module. It is just regular Python code. You do not need to do anything special when creating a module.

In the **bottom** panel, import `example_module`. Using the module name and dot notation, call `my_func` and the `hello()` method. You should see the two strings from their respective print statements in the module (top panel).

```
# bottom panel

import example_module

example_module.my_func()
example_module.MyClass.hello()
```

People create modules (and packages) with a specific purpose in mind:

- Reuse - creating modules means they can be reused in many other projects
- Naming - different modules means Python can differentiate between functions and modules with the same name

- Maintenance - maintaining code is easier when modules are independent from one another

So while any Python file can be a module, modules tend to have a different look and feel when compared to generic Python script.

challenge

Try these variations:

- In the module (**top** panel), add a string and list variables.

```
# top panel

my_string = 'I am a string'
my_list = ['I', 'am', 'a', 'list']
```

In the **bottom** panel, print both variables from the module.

```
# bottom panel
import example_module

print(example_module.my_string)
print(example_module.my_list)
```

▼ **What is happening?**
While modules are often classes or functions, any variable can be imported.

- Import the module using the `from` keyword.

```
# bottom panel
from example_module import my_func, MyClass, my_string,
        my_list

my_func()
MyClass.hello()
print(my_string)
print(my_list)
```

▼ **What is happening?**

The `from` keyword means that you can directly access the contents of the module without invoking the module name and dot notation. However, be aware that this can cause name collisions. For example, assume you already defined a function called `my_func`. Keeping the module name means you can differentiate between the two.

```
import example_module

def my_func():
  print('Local function definition')

my_func()
example_module.my_func()
```

Running Code on Import

While you do not need any special code to signify if a file is a module, you do need to be aware of some behavior specific to the importing process. Update the module (**top** panel) so that it calls `my_func`.

```
# top panel

def my_func():
  print('I am a function')

class MyClass():
  @classmethod
  def hello(self):
    print('I am a class')

my_func()
```

In the **bottom** panel, simply import `example_module`; do not do anything else. Click the button below to run your script.

```
# bottom panel

import example_module
```

Why did Python print that string? We often think of modules as being definitions for classes or functions. That is, modules are passive until the end user decides to implement them. However, importing a module runs **all** of the code in the file.

It is possible to keep code from being executed during the import process. Using a conditional that tests if `__name__` is equal to `__main__` will only execute associated code if Python is directly running the module file. If Python is importing the module file, then the code in the conditional is not executed.

```
# top panel

def my_func():
  print('I am a function')

class MyClass():
  @classmethod
  def hello(self):
    print('I am a class')

if __name__ == '__main__':
  my_func()
```

▼ **Did you notice?**

After making the last changes to the module, you should see the following:

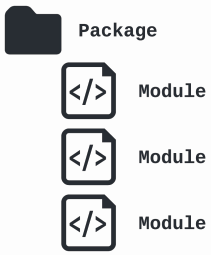
```
Command was successfully executed.
```

This is how Codio says your program ran without crashing and there was no output.

Packages

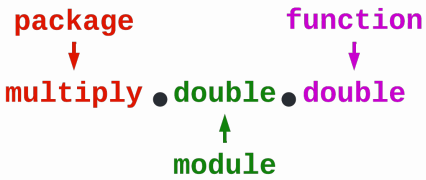
What is a Package?

Packages allow you to organize and provide structure to many modules. If a module is a Python file that contains classes, functions, and other code, a package is a directory that contains modules.



Depicted is a folder icon labeled as a package. Three file icons labeled as modules are grouped together underneath the package. The modules are indented to show the hierarchical nature between packages and modules.

Let's assume there exists a package called `multiply` with the modules `double` and `triple`. The `double` module contains the function `double` which takes a number and returns the number multiplied by 2. The module `triple` has the function `triple`. This function takes a number and returns that number multiplied by 3. Accessing a module within a package uses dot notation to separate the package name from the module name and from the function name.



Depicted are the words “multiply”, “double”, and “double”. These words are color coded to show that each one has a specific representation. “Multiply” is the name of the package, the first “double” is the name of the module, and the second “double” is the name of the function.

Import both the `double` and `triple` modules from the `multiply` package. Then call the `double` and `triple` functions.

```
import multiply.double, multiply.triple

print(multiply.double.double(3))
print(multiply.triple.triple(3))
```

Importing Packages

You can see how the hierarchical structure of packages leads to lengthy calls to functions and classes. In fact, you can even have subpackages in a Python package. This adds another layer of structure, but also lengthens calls to code within each module. Thankfully, you can import packages in much the same way you can import modules.

```
from multiply.double import double
from multiply.triple import triple

print(double(3))
print(triple(3))
```

challenge

Try these variations:

- Import the functions with an alias.

```
from multiply.double import double as d
from multiply.triple import triple as t

print(d(3))
print(t(3))
```

- Import just the package.

```
import multiply

print(multiply.double.double(3))
print(multiply.triple.triple(3))
```

▼ **Why is there an error?**

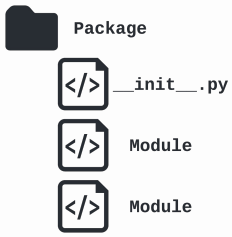
Technically speaking, `import multiply` is valid Python. The problem lies in that importing a package, by default, does not import any of the modules. To do that, we need to talk about our next topic — initializing packages.

Package Initialization

Package Initialization

Previously, packages were described as a collection of modules. In reality, it is a little more complicated than that. Packages often include an initialization file entitled `__init__.py`. This file is run whenever a package is imported.

▼ **Did you notice?**
The initialization file is **not** a requirement for creating a package. However, there was a time when Python required a **`init`**`.py` file for every package. The convention of including an initialization file has continued. So even though it is not required, it is a good idea to include one with your package.



Depicted is a folder icon labeled as a package. Three file icons are grouped together under the package. The first file is **init**.py, the initialization file. The other files are modules.

The **top** panel is the initialization file for the `multiply` package. Add the code below to `__init__.py`.

```
# top panel

print('The multiply package is now initialized!')
```

Next, in the **bottom** panel, import the `multiply` package. Do not add any other code. Now run your script.

```
# bottom panel

import multiply
```

Importing and Initializing

While the specific code that goes into a package's initialization file varies from case to case, initialization files can aid in the importing of packages. On the previous page, importing just the package and then calling the modules resulted in an error. That is because importing a module didn't do anything; there was no code in the initialization file.

Modify `__init__.py` so that importing the `multiply` module automatically imports the `double` and `triple` modules.

```
# top panel

import multiply.double, multiply.triple
```

In the **bottom** panel, import just the `multiply` module. Then call both the `double` and `triple` functions. Your script should work as expected now. This is because the initialization file for the package imported each module.

```
# bottom panel

import multiply

print(multiply.double.double(3))
print(multiply.triple.triple(3))
```

Another use case for the initialization file is when importing all of the contents of a package with the `*` wild card. First, let's start with the basic state of a Python script. Erase the code from the **top** panel.

```
# top panel
```

In the **bottom** panel, print the results of the `dir` function. This built-in function returns all available attributes and methods for all objects. Notice that even with no variable, function, or class declarations there are still attributes.

```
# bottom panel

print(dir())
```

Now, import the entirety of the `multiply` package with `*`. Then print `dir` one more time. What do you think will happen?

```
# bottom panel

from multiply import *

print(dir())
```

Nothing changed. The functions from the modules did not appear in the list. Unlike a module, the `*` operator does nothing when importing a package. This too can be changed in the initialization file.

In the **top** panel define the variable `__all__` to be a list of strings. Each element should be the name of a module in the `multiply` package.

```
# top panel

__all__ = [
  'double',
  'triple'
]
```

You do not need to make any changes to the code in the bottom panel. Running your script again should show you the `double` and `triple` functions.

▼ **Using the `*` operator.**

Using the `*` operator when importing either modules or packages is not encouraged. Instead, explicitly name the modules or packages you wish to import.

Built-In Packages

Standard Library

While you can write your own packages, there are built-in packages you can use as well. Every programming language has something called a standard library, which is a collection of many packages to help the end user more effectively write code.

You know that you are using the standard library when you import a package without having to download the package first. For example, the `math` package is already installed on your system as it is a part of the standard library. Print the value of Pi.

```
print(pi)
```

You get an error because `pi` is not defined. While the `math` package is already installed on your system, you need to import it from the standard library before you can use it.

```
import math

print(math.pi)
```

challenge

Try these variations:

- Print the contents of the `math` package.

```
import math

print(dir(math))
```

- Print all of the modules in Python's standard library.

```
print(help('modules'))
```

▼ **What is happening?**

The output from the code above is wider than the output area, so you do not see nicely formatted columns for all of the modules. The output should be more legible if you run it in the terminal.

Power of the Standard Library

Python is considered to be a "batteries included" language, which means their standard library is quite large. Without having to install a third-party package, you can do the following in Python:

- Parse strings (`re` and `string`)
- Perform cryptographic tasks (`hashlib` and `secrets`)
- Work with other file formats (`csv` and `json`)
- Create GUIs (`tkinter`)
- Communicate with the OS (`os` and `io`)

Python has a large standard library, which means you can perform many tasks without having to install any third-party packages.

challenge

Try these variations:

- Load and print a JSON file.

```
import json

with open('code/star_wars.json') as data:
  luke = json.load(data)

output = json.dumps(luke, indent=2)
print(output)
```

- Encrypt a string.

```
import hashlib

secret = 'Shhhh... this is very important'
byte_string = secret.encode()
hashed_secret = hashlib.sha256(byte_string)

print(hashed_secret.hexdigest())
```

Formative Assessment 1

Formative Assessment 2