**Your first module: step 1**

In this section you're going to be working locally on your machine. Let's start from scratch. Create an empty file, just like this:

  
module.py

You will need two files to repeat these experiments. The first of them will be the module itself. It's empty now. Don't worry, you're going to fill it with actual code soon.

We've named the file module.py. Not very creative, but simple and clear.

**Your first module: step 2**

The second file contains the code using the new module. Its name is main.py. Its content is very brief so far:



main.py

Note: **both files have to be located in the same folder**. We strongly encourage you to create an empty, new folder for both files. Some things will be easier then.

It doesn't matter that the module is empty for now. The very first step has been done, but before you take the next step, we want you to take a look into the folder in which both files exist.

Do you notice something interesting?

A new subfolder has appeared - can you see it? Its name is \_\_pycache\_\_. Take a look inside. What do you see?

There is a file named (more or less) module.cpython-xy.pyc where x and y are digits derived from your version of Python (e.g., they will be 3 and 8 if you use Python 3.8).

The name of the file is the same as your module's name (module here). The part after the first dot says which Python implementation has created the file (CPython here) and its version number. The last part (pyc) comes from the words *Python* and *compiled*.

You can look inside the file - the content is completely unreadable to humans. It has to be like that, as the file is intended for Python's use only.

When Python imports a module for the first time, it **translates its contents into a somewhat compiled shape**.

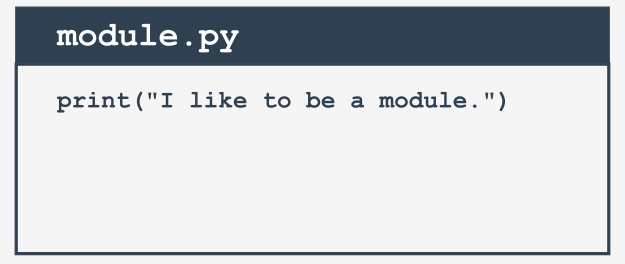
The file doesn't contain machine code - it's internal Python **semi-compiled code**, ready to be executed by Python's interpreter. As such a file doesn't require lots of the checks needed for a pure source file, the execution starts faster, and runs faster, too.

Thanks to that, every subsequent import will go quicker than interpreting the source text from scratch.

Python is able to check if the module's source file has been modified (in this case, the pyc file will be rebuilt) or not (when the pyc file may be run at once). As this process is fully automatic and transparent, you don't have to keep it in mind.

**Your first module: step 3**

Now we've put a little something into the module file:

  
Can you notice any differences between a module and an ordinary script? There are none so far.

It's possible to run this file like any other script. Try it for yourself.

What happens? You should see the following line inside your console:"

*“I like to be a module*.”

**Your first module: step 4**

Let's go back to the main.py file:



Run it. What do you see? Hopefully, you see something like this:

*“I like to be a module.”*

What does it actually mean?

When a module is imported, its content is **implicitly executed by Python**. It gives the module the chance to initialize some of its internal aspects (e.g., it may assign some variables with useful values).

Note: the **initialization takes place only once**, when the first import occurs, so the assignments done by the module aren't repeated unnecessarily.

Imagine the following context:

* there is a module named mod1;
* there is a module named mod2 which contains the import mod1 instruction;
* there is a main file containing the import mod1 and import mod2 instructions.

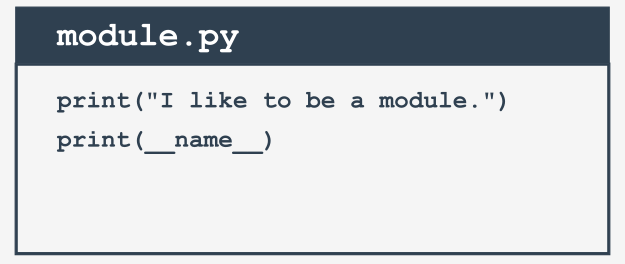
At first glance, you may think that mod1 will be imported twice - fortunately, **only the first import occurs**. Python remembers the imported modules and silently omits all subsequent imports.

**Your first module: step 5**

Python can do much more. It also creates a variable called \_\_name\_\_.

Moreover, each source file uses its own, separate version of the variable - it isn't shared between modules.

We'll show you how to use it. Modify the module a bit:



Now run the module.py file. You should see the following lines:

*I like to be a module*

*\_\_main\_\_*

Now run the main.py file. And? Do you see the same as us?

*I like to be a module*

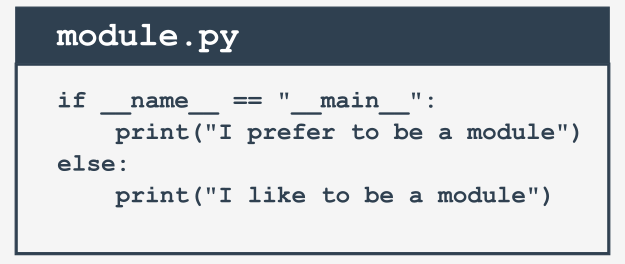
*module*

We can say that:

* when you run a file directly, its \_\_name\_\_ variable is set to \_\_main\_\_;
* when a file is imported as a module, its \_\_name\_\_ variable is set to the file's name (excluding .py)

**Your first module: step 6**

This is how you can make use of the \_\_main\_\_ variable in order to detect the context in which your code has been activated:



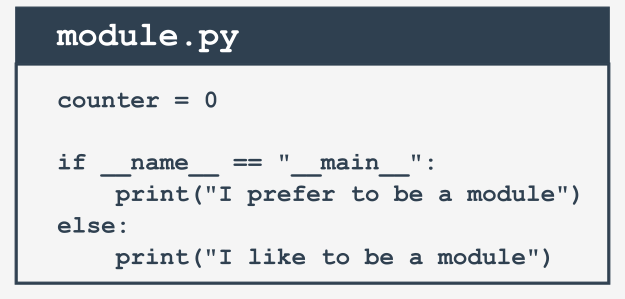
There's a cleverer way to utilize the variable, however. If you write a module filled with a number of complex functions, you can use it to place a series of tests to check if the functions work properly.

Each time you modify any of these functions, you can simply run the module to make sure that your amendments didn't spoil the code. These tests will be omitted when the code is imported as a module.

**Your first module: step 7**

This module will contain two simple functions, and if you want to know how many times the functions have been invoked, you need a counter initialized to zero when the module is being imported.

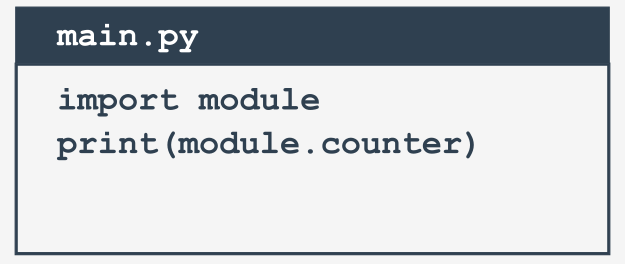
You can do it this way:



**Your first module: step 8**

Introducing such a variable is absolutely correct, but may cause important **side effects** that you must be aware of.

Take a look at the modified main.py file:



As you can see, the main file tries to access the module's counter variable. Is this legal? Yes, it is. Is it usable? It may be very usable. Is it safe?

That depends - if you trust your module's users, there's no problem; however, you may not want the rest of the world to see your **personal/private variable**.

Unlike many other programming languages, Python has no means of allowing you to hide such variables from the eyes of the module's users.

You can only inform your users that this is your variable, that they may read it, but that they should not modify it under any circumstances.

This is done by preceding the variable's name with \_ (one underscore) or \_\_ (two underscores), but remember, it's only a **convention**. Your module's users may obey it or they may not.

Of course, we'll follow the convention. Now let's put two functions into the module - they'll evaluate the sum and product of the numbers collected in a list.

In addition, let's add some ornaments there and remove any superfluous remnants.

**Your first module: step 9**

Okay. Let's write some brand new code in our module.py file. The updated module is ready here:

#!/usr/bin/env python3   
  
*""" module.py - an example of a Python module """*\_\_counter = 0  
  
  
def suml(the\_list):  
 global \_\_counter  
 \_\_counter += 1  
 the\_sum = 0  
 for element in the\_list:  
 the\_sum += element  
 return the\_sum  
  
  
def prodl(the\_list):  
 global \_\_counter   
 \_\_counter += 1  
 prod = 1  
 for element in the\_list:  
 prod \*= element  
 return prod  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 print("I prefer to be a module, but I can do some tests for you.")  
 my\_list = [i+1 for i in range(5)]  
 print(suml(my\_list) == 15)  
 print(prodl(my\_list) == 120)

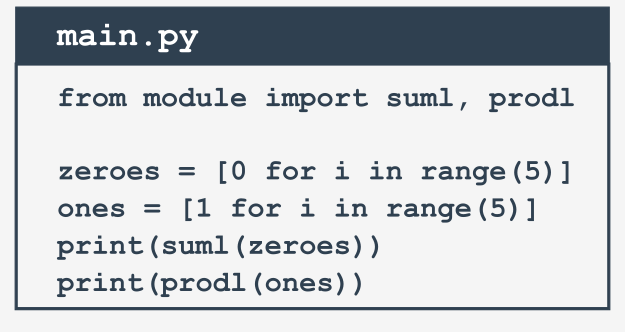
module.py

A few elements need some explanation, we think:

* the line starting with #! has many names - it may be called *shabang*, *shebang*, *hashbang*, *poundbang* or even *hashpling* (don't ask us why). The name itself means nothing here - its role is more important. From Python's point of view, it's just a **comment** as it starts with #. For Unix and Unix-like OSs (including MacOS) such a line **instructs the OS how to execute the contents of the file** (in other words, what program needs to be launched to interpret the text). In some environments (especially those connected with web servers) the absence of that line will cause trouble;
* a string (maybe a multiline) placed before any module instructions (including imports) is called the **doc-string**, and should briefly explain the purpose and contents of the module;
* the functions defined inside the module (suml() and prodl()) are available for import;
* we've used the \_\_name\_\_ variable to detect when the file is run stand-alone, and seized this opportunity to perform some simple tests.

**Your first module: step 10**

Now it's possible to use the updated module - this is one way:

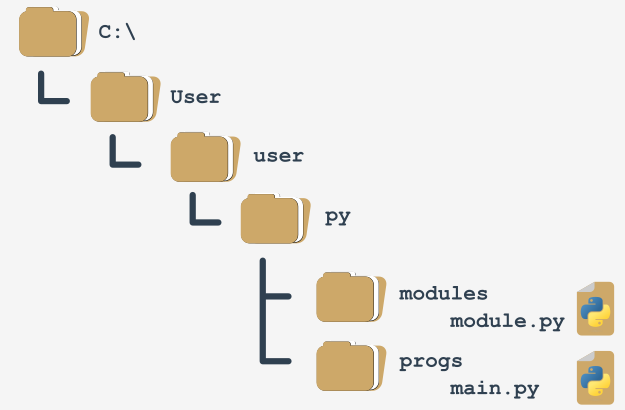


**Your first module: step 11**

It's time to make our example more complicated - so far we've assumed that the main Python file is located in the same folder/directory as the module to be imported.

Let's give up this assumption and conduct the following thought experiment:

* we are using Windows ® OS (this assumption is important, as the file name's shape depends on it)
* the main Python script lies in C:\Users\user\py\progs and is named main.py
* the module to import is located in C:\Users\user\py\modules



How to deal with it?

To answer this question, we have to talk about **how Python searches for modules**. There's a special variable (actually a list) storing all locations (folders/directories) that are searched in order to find a module which has been requested by the import instruction.

Python browses these folders in the order in which they are listed in the list - if the module cannot be found in any of these directories, the import fails.

Otherwise, the first folder containing a module with the desired name will be taken into consideration (if any of the remaining folders contains a module of that name, it will be ignored).

The variable is named path, and it's accessible through the module named sys. This is how you can check its regular value:

import sys  
  
for p in sys.path:  
 print(p)

We've launched the code inside the C:\User\user folder, and this is what we've got:

C:\Users\user

C:\Users\user\AppData\Local\Programs\Python\Python36-32\python36.zip

C:\Users\user\AppData\Local\Programs\Python\Python36-32\DLLs

C:\Users\user\AppData\Local\Programs\Python\Python36-32\lib

C:\Users\user\AppData\Local\Programs\Python\Python36-32

C:\Users\user\AppData\Local\Programs\Python\Python36-32\lib\site-packages

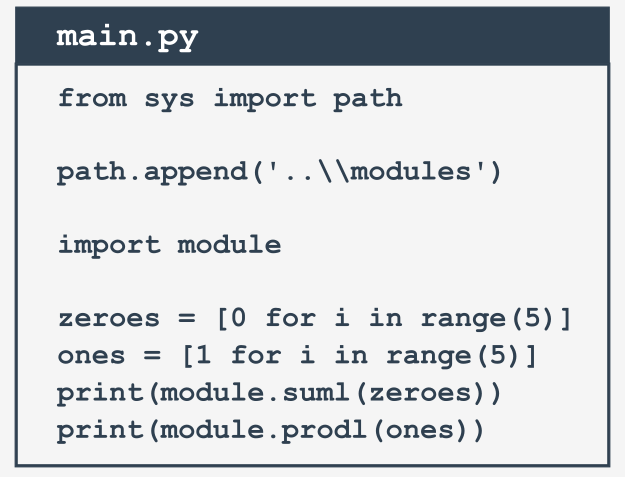
Note: the folder in which the execution starts is listed in the **first path's element**.

Note once again: there is a zip file listed as one of the path's elements - it's not an error. Python is able to treat zip files as ordinary folders - this can save lots of storage.

Can you figure out how we can solve our problem now? We can add a folder containing the module to the path variable (it's fully modifiable).

**Your first module: step 12**

One of several possible solutions looks like this:



Note:

* we've doubled the \ inside folder name - do you know why?  
  Because a backslash is used to escape other characters - if you want to get just a backslash, you have to escape it
* we've used the relative name of the folder - this will work if you start the main.py file directly from its home folder, and won't work if the current directory doesn't fit the relative path; you can always use an absolute path, like this:  
  path.append('C:\\Users\\user\\py\\modules')
* we've used the append() method - in effect, the new path will occupy the last element in the path list; if you don't like the idea, you can use insert() instead.