Chapter 2:
Python Functions, Modules, and Virtual Environments

In this chapter, you'll first learn how to use the built-in functions in Python and how to import modules in the Python standard library. You'll then learn three different ways of importing modules and the pros and cons of each way of importing.

You'll also learn how functions work and how to define your own functions. Many modules we use in this book are not in the Python standard library, and you'll learn different ways of installing these modules on your computer.

We'll discuss how modules work and how to create your own self-made modules. You'll then learn what is a virtual environment, why it's useful, and how to create and activate a virtual environment.

Why functions?

Functions are useful tools in Python (and in programming in general). A function takes inputs, executes the specified tasks, and returns the output. We create functions in Python so that we don't have to repeat the same process every time we need certain jobs done. Instead, we can just call the function and enter the inputs. The function will take inputs and return the output for us to use.

Another benefit of using functions is to make your program more organized, less cluttered, and less error-prone.

Imagine you are writing a program to complete ten sequential tasks, and each task involves 20 steps. If you put all 200 steps in one large block of code, it's difficult to organize and easy to make mistakes. On the other hand, you can define each task as a function, and you only need to deal with 20 steps when defining each function. After that, you can call those ten functions and organize them neatly to complete the tasks.

Built-in functions in Python

Python comes with many built-in functions that you can readily use. We have already seen some of them in Chapter 1. For example, `print()` is a built-in function to print outputs for you to see, `round()` is a function to round a float number to the specified number of digits after the decimal point, and `int()` is a function to covert other types of variables into an integer.

Below, you'll learn a few built-in functions that are frequently used in this book.

The `range()` function

The function `range()` is used to produce a list of integers. For example, `range(5)` produces the list of the following five values [0, 1, 2, 3, 4]. The default start value is 0 because Python is using 0 indexing. You can specify the start value if you want the list not to start with 0. For example, `range(3,6)` produces the list of the following three values [3, 4, 5].

The default increment value is 1, but you can also specify the increment in the function as the optional third argument. Here is one example: if you run the following lines of code in the Spyder editor (use the tab key on your keyboard to indent):

```
for x in range(-5, 6, 2):
    print(x)
```

you'll have the following output:

```
-5
-3
-1
1
3
5
```

The third argument in `range(-5, 6, 2)` tells the program to increase the value by 2 from one element to the next in the produced list.

The increment can also be a negative integer, and it means that the values in the list decrease one by one. For example, `range(9,0,-3)` produces this list: [9, 6, 3].

The `range()` function produces a list of integers, but not float numbers. However, we can easily get around this problem. For example, if you want to print out the list of values from 0.1 to 1 with increments of 0.1, the following lines of code will do the trick:

```
for x in [i*0.1+0.1 for i in range(10)]:
    print(x)
```

The command inside the square brackets first generates the list of integers 0 to 9 (through code `range(10)`). It then transforms each element in the list by first multiplying it by 0.1 and

then adding 0.1 to it (through code `i*0.1+0.1 for i in`). The second line of command prints out each value inside the generated list.

The output for the above two lines of code is:

```
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
```

The `input()` function

Speech recognition is the process of converting human voice into written text. To prepare you for the details of speech recognition later in this book, we'll first discuss how Python takes written text inputs from the user through the keyboard. Python does this using a built-in function called `input()`.

Suppose you have the following program:

```
color = input('What is your favorite color?')
print('I see, so your favorite color is {}'.format(color))
```

If you run the program in Spyder, you'll see a screen similar to Figure 2-1.

INSERT FIGURE 2-1 HERE

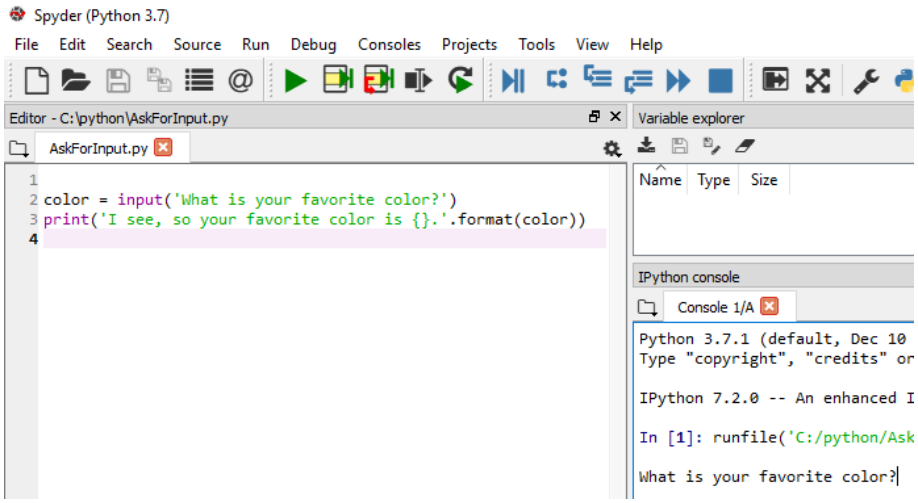

Figure 2-1: A screenshot of what happens when Python is asking for input

As you can see from the Figure 2-1, the program is asking for your input at the lower right panel (i.e., the IPython console). If you type in your answer after the question mark (?) and press the ENTER key on your keyboard, the program will continue to run. Suppose you put **blue** as your answer, the program will have the following output (the user input is in bold):

```
What is your favorite color? blue
I see, so your favorite color is blue.
```

The program can ask for multiple inputs. For example, the program can ask for your first name and last name separately, as in the following program:

```
FirstName = input('What is your first name?\n')
LastName = input('What is your last name?\n')
print('Nice to meet you, {0} {1}.'.format(FirstName, LastName))
```

The program asks for your input twice. The sequence `\n` is an escape character, indicating a new line. In this case, the cursor will be at the line below the question "What is your first name?" Table 2-1 provides a list of commonly used escape characters in Python.

Table 2-1: Commonly used escape characters in Python.

Code	**Representation**
\n	New line
\\	The backslash

\'	Single quotation mark '
\"	Double quotation mark "
\t	Horizontal tab

In the example above, we have used two placeholders, `{0}` and `{1}`, and their values are characters in the string variables *FirstName* and *LastName*, respectively. Another way to achieve the same outcome is to replace the last line of command with the following:

```
print(f'Nice to meet you, {FirstName } {LastName }.')
```

One interaction with the program produces the following output (user input is in bold):

```
What is your first name?
John
What is your last name?
Smith
Nice to meet you, John Smith.
```

If you want to find out what a particular built-in function does, you can use the command line `help()`. For example

```
help(abs)
```

produces the following output

```
abs(x, /)
Return the absolute value of the argument.
```

And you'll know that the built-in function `abs()` returns the absolute value of the argument.

Import Modules from the Python Standard Library

You are not limited to use just the Python built-in functions. The Python standard library has many modules to provide us with other functions to use. We'll discuss three different ways of importing a function from a module, and the pros and cons of each method.

The "Import Module" Method

The first way is to import the module only, without specifying any function in the module when importing.

For example, if you want to find out the value of the cosine of a 30 degree angle, you can first import the *math* module. Then you can use the `cos()` function in the module by calling both the module name and the function name: `math.cos()`.

Enter the following two lines of code in Spyder:

```
import math
print(math.cos(30))
```

and you'll have an output of 0.15425144988758405.

Note that you need to have the code `import math` in your program before you call the function `math.cos()`. If you do not import the *math* module first, and just run this command

```
print(math.cos(30))
```

Python will give you an error message as follows

```
NameError: name 'math' is not defined
```

Another point to keep in mind is that you still need to put the module name in front of the function name when you call the function.

Enter the following two lines of code in Python:

```
import math
print(cos(30))
```

and you'll still get an error message as follows

```
NameError: name 'cos' is not defined
```

This is because Python doesn't know where to find the `cos()` function, even though you have imported the *math* module.

The "from Module Import Function" Method

If you must use one or two functions in a certain module many times in your program, you can save time by importing just those one or two functions.

Enter the following two lines of code in Python:

```
from math import cos, log # ❶
print(cos(30)+log(100))
```

and you'll have the correct output 4.759421635875676. Note that you don't need to put `math.` in front of the functions `cos()` or `log()`. This is because in line ❶, we have told the program where to look for the two functions.

Imagine you have to use `cos()` and `log()` 100 times in the program. By using `from math import cos, log` instead of `import math`, you reduced the appearance of `math.` 200 times in the program.

The "Star Import" Method

In case your program relies heavily on many different functions in a module, it saves you time to simply import all functions from the module.

For example, later in this book, we often use many different functions in the *turtle* module in a program, so we start the program by the following line of code:

```
from turtle import *
```

This way, we don't have to put the module name in front of all *turtle* functions in the program.

You may ask, "Why don't we use `import *` for all modules so we don't need to type in any module name in front of functions in the program?"

Using `import *` can pollute function names in the program. Imagine you import ten different modules in a program using `import *` and some modules may have the same function names. The program doesn't know whether a function is from module A or from module B.

To give you a concrete example, suppose you have the following three lines of code:

```
from math import *     #❶
from numpy import *    #❷
print(log(100))
```

You import all functions from the *math* module in line ❶, which has defined a `log()` function. In line ❷, you import all functions from the *numpy* module, which has its own `log()` function. When the function `log()` is called, the program doesn't know which `log()` function to use.

However, if you import just one or two modules, and you are sure they don't share the same function names, you can use the `import *` method. To be certain, you can use `dir(module)` to see all the functions in an imported module.

For example, if you run the following two lines of code:

```
import math
print(dir(math))
```

You'll have the following output:

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot',
'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

Ignore names that start and end with the underscore, such as *_name_* or *_loader_* because every module has them.

BEGIN BOX

SUMMARY OF THE THREE WAYS OF IMPORTING MODULES

1. The "import module" method. For example, `import math`. You need to put the module name in front of the function every time you call a function, e.g., `math.log()` or `math.cos()`.
2. The "from module import function" method. For example, `from math import cos, log`. You do not need to put the module name in front of the two functions `cos()` or `log()`. But you still need to put the module name in front of the other functions, e.g., `math.sin()` or `math.tan()`.
3. The "star import" method. Import all functions in the module. For example, `from math import *`. You do not need to put the module name in front of any function from the *math* module. However, if you are using many different modules and you use *import* * on all of them, they may interfere with each other because different modules may have the same function name. This is a good method if you are using just one or two modules in your program.

END BOX

Define Your Own Functions

We mentioned above that functions make repeating tasks easy. They also make your program more organized and less error-prone.

So far, we have treated functions like black boxes. In this section, you'll "look under the hood" and see how functions work.

Functions can take one or more inputs (or arguments), or no input at all.

A Function with No Argument

Here is a sample program with a function that takes no input (use the tab key on your keyboard to indent):

```
def TheEnd(): #❶
    print('Finished printing')
for i in (1,2,3):  #❷
    print(i)
TheEnd() #❸
```

At ❶, we start to define the function `TheEnd()` that has no argument. We use `def` to initiate the function definition. We need to put the colon sign (":") after the name of the function. When the function is called, it will print a message that says, "Finished printing." The message will not be printed during the definition stage because the function is not called yet. Further, all command lines inside a definition must be indented, so that the program knows which commands are included in the function.

At ❷, the program prints three numbers. Finally, the function `TheEnd()` is called. As a result, the message "Finished printing" is printed ❸. The output from the above program is as follows:

```
1
2
3
Finished printing
```

As you can see, the command line in the function is executed only when the function is called, not when it is defined.

A Function with One Argument

Here is an example of a function that takes one input.

Imagine you want to write a thank-you note to 50 different people. The message is the same except the recipient's name. You can define a function to print the message, and we only need to supply the name for each message instead of the whole message.

We first define a function called `msgs()` as follows:

```
def msgs(name):
    print(f"Thank you {name}, I appreciate your help!")
```

The name of the function is `msgs`, with the variable *name* as its only input. If we call the function twice as follows:

```
msgs("Mary")
msgs("Bob")
```

The output will be a thank-you note for Mary and Bob, respectively.

```
Thank you Mary, I appreciate your help!
Thank you Bob, I appreciate your help!
```

To write the 50 thank-you notes, you can just call the function 50 times.

A Function with More Than One Argument

In many cases, functions have two or more arguments as inputs. Consider the following program, *Price.py*, in Listing 2-1, which defines a function that needs three inputs.

```
def StockPrice(div, g, r):
    Price = (div*(1+g)/(r-g)) #❶
    return Price #❷
print(StockPrice(2, 0.05, 0.1)) #❸
```

Listing 2-1: Python code to define a function with three arguments

We have just defined the constant dividend growth model in finance, but it's easy to follow even if finance is not your area of expertise. In order to calculate the stock price, using current dividend per share *div*, dividend growth rate *g*, and discount rate *r*, as the three inputs. At ❶, we calculate the price of the stock based on the three pieces of information according to the dividend

growth model. At ❷, we tell the program what the output of the function is by using the `return` command. As a result, when the function `StockPrice()` is called, you get the price of the stock.

We then call the function `StockPrice()` at ❸. We'll get an output of 42 from the program: the stock price is $42 if the current annual dividend is $2 per share, the expected dividend growth rate is 5% per year, and the annual interest rate is 10%.

A Function with A List as the Argument

In many situations, the number of inputs is unknown. This makes the definition of functions difficult. For example, you want to define a function to calculate the total sales from a group of salespeople, while different groups have different numbers of salespeople in them. If the size of group ranges from 3 to 10, you need to define eight different functions: one with 3 inputs, one with 4 inputs, …, and one with 10 inputs.

It turns out that you can define one single function for the above purpose, regardless of the size of the group. The argument in a function can be a collection of values, such as a list. We'll formally introduce Python lists in Chapter 3, but for the moment, simply treat it as a collection of values.

The following program, *TotalSales.py*, accomplishes the job:

```
def TotalSales(nums):
    total = 0  #❶
    for i in range(len(nums)): #❷
        total = total + nums[i]   #❸
    return total
--snip--
```

Listing 2-2: First part of the program TotalSales.py

In Listing 2-2, we first initiate the definition of the function `TotalSales()`, which takes one argument: *nums*. At ❶, we first set the value of the variable *total* to 0. At ❷, we loop through each element in the collection variable *nums*.

For example, if a group has three salespeople with sales of $500, $670, and $800, respectively, we'll set *nums = [500, 670, 800]*. You can refer to Chapter 3 on how lists work. The Python built-in function `len()` returns the length of argument. In this case, `len(nums)` is 3 because the list *nums* has three elements.

At ❸, for each element in the list *nums*, we add it to the variable *total*. Once we go through all elements in *nums*, we have the total sales of the group, and use it as the output of the function.

After defining the function `TotalSales()`, if you run the rest of the program *TotalSales.py* in Listing 2-3 below:

```
--snip--
print(TotalSales([200,100,100,100]))
print(TotalSales([800,500,400]))
```

Listing 2-3: Second half of the program TotalSales.py

you'll get the following output:

```
500
1700
```

As you can see, the function takes one argument, the list *nums*, as the input. You can put as many elements in the list *nums* as you want.

How Modules Work and How to Create Your Own Modules

We have learned three different ways of importing functions from modules in Section 2.3. You may wonder: How do modules work? How to create a module? How to install modules that are created by others?

We'll answer these questions below.

Earlier in this chapter, we have defined the function `StockPrice()` in the program *Price.py* and then call the function to calculate the stock price. If you need to calculate the stock price in many different programs, you must define the function `StockPrice()` in each program and then call it to do the calculation. You may wonder, is there a way to define the function `StockPrice()` just once so that you can use it again and again in different programs?

The answer is yes, and the solution is through modules.

Let's first create a program called *CreateLocalModule.py* as shown in Listing 2-4:

```
def StockPrice(div, g, r):
    P = (div*(1+g)/(r-g))
    return P
```

Listing 2-4: Code for the local module

This program just defines the function `StockPrice()` but do not call it. Let's create a new program *ImportLocalModule.py* as in Listing 2-5:

```
# Make sure you put CreateLocalModule.py in the same folder as this program
from CreateLocalModule import StockPrice
print(StockPrice(2, 0.05, 0.1))
print(StockPrice(1, 0.08, 0.1))
print(StockPrice(2, 0.02, 0.08))
```

Listing 2-5: Code to import the local module

Make sure you put the new program *ImportLocalModule.py* in the same folder as the program *CreateLocalModule.py* we just created. This way, the program will know where to look for the module *CreateLocalModule*.

If you run the program, you'll have the following results:

```
42.0
53.99999999999999
34.0
```

As you can see, the function `StockPrice()` correctly calculates the prices for three different stocks.

Even though the function `StockPrice()` is not defined in the program, we can import it from the program *CreateLocalModule.py*, which acts as a local module in this case. It works the same as modules in the Python standard library. One small difference is this: for non-self-made modules, the relevant files are placed in specific, and difficult to find folders on your computer (with good reason: in case you accidentally open it and change the code, the module won't work any more).

For example, the *tkinter* module is in the Python standard library, and we'll use it later in this book. The files for the *tkinter* module are placed under this path if you are using the Windows operating system:

```
C:\Users\ME\Anaconda3\envs\MYEV\Lib\tkinter
```

where `ME` is your username on your computer and `MYEV` is the name of your virtual environment (the meaning of which we'll discuss soon).

If you are using Mac, files for the *tkinter* module are placed under this path:

```
/Macintosh HD/opt/anaconda3/envs/MYEV/lib/python3.8/tkinter
```

If you are using Linux, files for the *tkinter* module are here:

```
/home/anaconda3/envs/MYEV/lib/python3.8/tkinter
```

Again, `MYEV` is the name of your virtual environment.

BEGIN BOX

TRY IT YOURSELF

2-1. Based on the program *TotalSales.py* above, create a local module to define the function `TotalSales()`. After that, import the module and use the function in the module to calculate total sales.

END BOX

Install Modules That Are Not in the Python Standard Library

One of the main advantages of Python is the fact that programmers can share modules with each other for free. Many of these modules are not in the Python standard library. For example, in this book, we rely on the text to speech and the speech recognition modules to make Python talk and listen. However, none of them is in the Python standard library. Therefore, knowing how to install modules on your computer is important, especially when you start to build unique projects (as we do in this book).

We'll discuss below how to install these modules.

Check If A Module Needs Installation

A module needs no installation if it's in the Python standard library. All modules in the Python standard library are automatically installed on your machine when you install Python. Other modules may be installed on your machine if it's needed for another module that you are using. For example, when you install the module *pandas* on your computer, about 23 other supporting modules (*scipy*, *tensorboard*, and so on; which we call *dependencies*) are installed as well because *pandas* depends on those 23 modules to work.

There are two ways to check if a module is installed on your computer for the program you are about to run. The first way is to run this line of code in your Spyder editor:

```
help("modules")
```

and Python will provide you with the list of all modules installed on your computer. You can check if the module you are about to import is in the list. If not, you'll need to install them.

The drawback with this method is that it can take a long time for Python to list all the modules and for you to check them. The Python standard library alone includes several hundred modules, and this does not include the modules you install yourself.

NOTE: For a list of all modules in the Python standard library, go to https://docs.python.org/3/library/. The list is constantly changing because more and more modules are added to the library over time.

The second way to check if a module is installed on your computer is to run this line of code in your Spyder editor:

```
import ModuleName
```

For example, if you want to check if the module *pandas* is installed on your computer, run `import pandas` in Spyder. If there is no error message in the IPython console, the module is already installed on your computer and no installation is needed.

On the other hand, if the output shows "ModuleNotFoundError," you need to install the module before you import it. We'll discuss how to install modules below.

How to Install A Module

There are two possible ways you can install a module in Anaconda, if the module is available for installation to begin with.

Pip Install Modules

Most Python modules are installed using this method. For example, if you want to install the *gtts* module, you can pip install it as follows.

First, open the Anaconda prompt (in Windows) or a terminal (in Mac or Linux), and type in

```
pip install gtts
```

in the command line and press the ENTER key on your keyboard. Follow the on-screen instructions all the way through, and the *gtts* module will be installed.

Conda Install Modules

Conda install is similar to pip install. In case you can't pip install it, you can try conda install as follows.

Suppose you want to install the *yt* module. First, open the Anaconda prompt (in Windows) or a terminal (in Mac or Linux), and type in

```
conda install yt
```

and execute.

You may wonder the difference between pip install and conda install. Pip is the Python packaging authority's recommended tool for installing packages from the Python packaging index. You can only install Python software using pip install.

In contrast, conda is a cross-platform package and environment manager that installs not only Python software, but also packages in C or C++ libraries, R packages, or other software.

Create A Virtual Environment on Your Computer

As you have more and more projects in Python, you'll install many different modules. Some modules may interfere with other modules. Also, different projects may use different versions of the same module.

To avoid the above problems, you should consider having a separate virtual environment for each project. A virtual environment is a way to isolate different projects in case the required modules for different projects interfere with each other.

Below we'll discuss how to create and activate a virtual environment in different operating systems.

Create A Virtual Environment

To create a virtual environment, first open the Anaconda Prompt (in Windows) or a terminal (in Mac or Linux).

Let's name the virtual environment we want to create for this book *chatting*, and you can use other names of your choice. Enter the following command in the Anaconda prompt or the terminal:

```
conda create -n chatting
```

After pressing the ENTER key on your keyboard, follow the instructions on the screen and press **y** when the prompt asks you `y/n`.

Activate the virtual environment

Once you have created the virtual environment on your machine, you need to activate it.
In the Anaconda prompt (in Windows) or a terminal (in Mac or Linux), type in

```
conda ctivate chatting
```

Again, *chatting* is the name of the virtual environment. If you have used a different name, replace *chatting* with that name. If you are using Windows, you'll see the following on your Anaconda prompt:

```
(chatting) C:\>
```

Note that you can see `(chatting)` in front of `C:\>`, which indicates that the command line in the prompt is now in the virtual environment *chatting* that you have just created.

If you are using Mac, you should see something similar to the following in the terminal (the username will be different):

```
(chatting) Macs-MacBook-Pro:~ macuser$
```

If you are using Linux, you should see something similar to below on your terminal (the username will be different):

```
(chatting) mark@mark-OptiPlex-9020:~$
```

NOTE: If you are using Ubuntu 18 instead of Ubuntu 19 in Linux, use `activate chatting` instead of `conda activate chatting` to activate the virtual environment. You won't be able to see the `(chatting)` part on your terminal even if the virtual environment is activated.

Set Up Spyder in the Virtual Environment

Now we need to set up the Spyder editor in the newly created virtual environment on your computer.

To install Spyder in the new virtual environment, activate the virtual environment as described above in the Anaconda prompt (in Windows) or a terminal (in Mac or Linux). After that, run the command

```
conda install spyder
```

Once done, if you want to launch Spyder, execute the following command in the same terminal with the virtual environment activated:

```
spyder
```

Summary

In this chapter, you learned how functions work in Python. You learned three different ways of importing a module into a program and the pros and cons of each method.

You also learned how to create your own functions. You learned how to create a local module and import it to a program to make the program clean and concise. Finally, you learned how to create and activate a virtual environment in order to separate packages in different projects.

In Chapter 3, you'll learn different ways of handling collections of elements in Python, so that you can use these tools for more complicated tasks.

End of Chapter Exercises

1: What is the output from each of the following lines of command? First write down the answer and then run the command line in Spyder and verify.

```
for i in range(5): print(i)
for i in range(10,15): print(i)
for i in range(10,15,2): print(i)
```

2: Suppose you define a function *printcellpos(cols)* as follows:

```
def printcellpos(rows):
    for col in ("A","B"):
        for row in range(1, rows+1):
            print("{0}{1}".format(col, row))
```

What is the output from *printcellpos(6)?*

3: What is the value of StockPrice(5, 0.08, 0.1) according to the defined function in this chapter?

4: Change the module-import method in the program *ImportLocalModule.py* from the "from module import function" method to the "import module" method. Name the new program *ImportLocalModule1.py* and make sure it produces the same output.

5: Change the module-import method in the program *ImportLocalModule.py* from the "from module import function" method to the "import *" method. Name the new program *ImportLocalModule2.py* and make sure it produces the same output.

6: Based on the program *TotalSales.py* above, create a local module to define the function `TotalSales()`. After that, create a new program called *ImportTotalSales.py*, import the module by using the "from module import function" method. Then calculate total sales in *ImportTotalSales.py* if the individual sales numbers in the group are 500, 600, and 900, respectively.