Chapter 12

MODULES AND LIBRARIES

This chapter further explores modules—pre-packaged Python resources that add functionality to the core language and simplify many programming tasks. Modules are reused from program to program, providing a framework for consolidating, recycling, and sharing programming code. We present a few more built-in modules (supplementing those you have already seen in previous chapters), and introduce some widely used third-party modules for data processing and analysis.

Modules are packages of computer code that can be imported into your programs as you need them. Modules contain a variety of data types and functions that extend the Python language to enable and simplify a wide range of tasks, from general-purpose mathematical calculations to drawing graphical objects on the screen. In the previous chapters you were briefly introduced to several built-in Python modules, including tools for regular expressions (the `re` module) and system operations (the `sys` and `os` modules).

In order to use a module, it must be both installed on your computer and imported into your program. There are many built-in modules that come with Python, including the modules you used in the previous chapters. These built-in modules are distributed as the Python standard library (see `http://docs.python.org/library/`), and all computers with Python will have them installed and ready to import. In addition to `re`, `sys`, and `os`, useful built-in modules include tools for `time` (for obtaining or converting dates and times) and `random` (for functions involving randomness).

In addition to the ubiquitous modules in the Python standard library, there are also many third-party modules. Anybody can write a third-party module, and this can then be provided to others to install and import into their own Python programs. These third-party modules typically provide specialized capabilities, including tools for graphics, scientific analyses, and the creation of user-interface elements. Even though they aren't part of the official Python standard library, a

number of the more popular third-party modules are often installed with Python by default. You will need to check on a case-by-case basis to see if you already have a particular third-party module installed. You may get lucky and find that the module you want to use is on your computer and ready to go; if not, you will need to install it. Options for installing modules vary from module to module and computer to computer, and will be discussed later in this chapter.

Importing modules

Modules are actually collections of objects, including functions and custom variables. When you import a module, the code for these objects is read from the corresponding files and the objects become available for use within your program. It may seem like an undue burden to have to import a module when it is already installed on your computer, but there are good reasons for this added step. Specifying modules only as needed avoids the overhead that would be incurred if every Python program had to load every installed module. It also avoids cluttering your program with names of variables and functions you don't need. Assuming they have been installed on your computer, modules can be imported into your program in several ways. The simplest is to import the entire module with the `import` statement. This is how you imported modules in previous chapters. For example:

```
import re
import sys
```

When modules are imported in this way, all of the objects in the module are available for use. This manner of importing has another effect as well: it requires you to refer to the objects within the module using dot notation. You are already familiar with dot notation from Chapter 7, and from using it when calling object methods, such as `.count()` and `.strip()` for strings. In this case, because modules are objects that contain objects, the dot specifies which objects are being called within the module object. For instance, when you used the `search()` function from the regular expressions `re` module in Chapter 10, you called `re.search()` rather than just `search()`. The dot notation specified that you were calling the `search()` function of the `re` module, instead of some other function with the same name. Likewise, in later programs you have called the `stderr.write()` function from the `sys` module with `sys.stderr.write()`, and accessed the `argv` variable from the `sys` module with `sys.argv`.

It is also possible to import individual module objects without importing the module as a whole. This makes for more efficient programs if the modules are large. This alternative notation uses the `from` statement. For example:

```
from sys import argv
```

Note that there is still an `import` statement, but instead of importing an entire module, the statement imports only specified objects (`argv`) from within a specified module (`sys`). Because the object within the module is being imported directly, you don't name the module when using it. That is, no dot notation is required.

To see this in play, recall this simple program from the last chapter. It prints the arguments that are passed to it:

```
#!/usr/bin/env python
import sys
for MyArg in sys.argv:
   print MyArg
```

In this original formulation, the entire `sys` module is imported and the `argv` variable is then referred to with dot notation.

The following program has the exact same behavior as the original, but note the modifications to the `import` statement, and the omission of `sys.` before `argv`:

```
#!/usr/bin/env python
from sys import argv
for MyArg in argv:
   print MyArg
```

There is no `sys` object in this program, so a call to `sys.argv` would result in an error.

Finally, there is also a way to import all the objects from within a module without importing the module object itself. Simply use the `*` wildcard in conjunction with `from` instead of a particular object name. In this case, `*` acts like the `bash` `*` wild card rather than the regular expression quantifier:

```
#!/usr/bin/env python
from sys import *
for MyArg in argv:
   print MyArg
```

This is convenient if you would like access to all the contents of a module but don't want to use dot notation each time you use those objects. It can also be dangerous, however, as the names of some of the objects within the modules may be the same as the names of objects imported from other modules—or potentially, even some of the names you have created in your own code. Since you don't have the dot notation to wall off the module names from each other, different objects with the same name might end up overwriting each other. In practice this rarely happens—but when it does, it can lead to some very strange program behavior.

Which strategy you take to importing modules and objects from modules will depend upon the particular program and your personal preference, but you will routinely see both types of strategies in publicly available Python programs.

More built-in modules from the standard library

The Python standard library is large and diverse, and includes modules that will meet many of your primary needs. Drawing upon the Python standard library rather than upon third-party modules has the great advantage that you can be relatively certain these modules will be available on any system with Python installed.

The `urllib` *module*

The `urllib` module provides tools with similar functionality to the `curl` command you already used at the command line. It lets you download resources directly from the Internet for use within your Python program. For example:

```
#!/usr/bin/env python
# coding: utf-8
import urllib
NewUrl = "http://practicalcomputing.org/aminoacid.html"
WebContent = urllib.urlopen(NewUrl)

for Line in WebContent:
   print Line.strip()
```

This module is useful for making "smart" `curl` commands to download, parse, extract, and save data from a Web page.

The `os` *module*

To perform shell operations from within a Python script, you can use the `os` (operating system) module (Table 12.1).

Especially useful is the ability to run non-Python command-line programs from within Python. This is achieved with the `os.popen()` command, with the `'r'` option indicating you want to read the output:

```
DirOutput = os.popen('ls -F| grep \/','r').read()
```

Here, the shell `ls -F` command is sent through a `grep` command. The result is read, and the text is assigned to the variable `DirOutput`. Any shell program name can be substituted for `ls`, including pipes to other commands. This is a powerful way to build a data analysis tool that requires several programs, as described in Chapter 16. You will want to be careful with these commands, however, because

TABLE 12.1 Some useful functions of the os module, along with their shell equivalents

Module command	Operation	Shell equivalent
`os.chdir('/Users/lucy/pcfb')`	Change directory	`cd ~/pcfb`
`os.getcwd()`	Get current dir	`pwd`
`os.listdir('.')`	List directory	`ls`
`glob.glob("*.txt")`	Wildcard search for files (requires `import glob`)	`ls *.txt`
`os.path.isfile('data.txt')`	Does file exist?	
`os.path.exists('/Users/lucy')`	Does folder exist?	
`os.rename('test.txt','test2.txt')`	Rename file	`mv test.txt test2.txt`
`os.popen('pwd','r').read()`	Run a shell command and load the output into a variable	Any command; in this case pwd is shown

they can potentially wreak the same kinds of havoc incurred by unrestricted shell operations.

The `math` *module*

A number of basic mathematical functions are available upon importing Python's built-in math module. These tools and resources include `sin()`, `log()`, `sqrt()` and the constant `pi`:

```
>>> from math import *
>>> pi
3.1415926535897931
>>> log(8,2)
3.0
>>> sqrt(64)
8.0
```

Though the math module only has a limited number of functions, they do meet many common needs.

The `random` *module*

This isn't a section about a randomly selected module—it is about `random`, a module that can generate random numbers. We wrote the following script, using this module's `randint` function, to determine the order in which our names would be listed on the cover of this book.

```
#!/usr/bin/env python

import random

Casey = 418
Steve = 682
Num = -1
Possible = 1000
NumList = []

while (Num != Casey) and (Num != Steve):
    Num = random.randint(0,Possible)
    NumList.append(Num)

print NumList
print ['Casey','Steve'][Num == Steve] + ' wins!!'
```

Here, the `random.randint()` function returns an integer between 0 and the value of `Possible`. The `while` loop keeps going, generating random numbers and storing them in a list, until one of the random values matches a value defined at the start of the program.

Note that the last `print` statement is not cheating in favor of Steve. It uses the fact that the logical values `True` and `False` correspond to the numerical values `1` and `0`. This numerical interpretation is used in square brackets to index the list of names, returning `'Casey'`, the zeroth element, when the logical comparison is `False`, and `'Steve'` when it's `True`.

Other useful `random` functions are listed in Table 12.2. If you import the entire module, these are used in dot notation with the prefix `random`, as in our example script.

TABLE 12.2 Some useful functions of the `random` module

Function	Result
`randint(5,50)`	Return a random integer, in this case between 5 and 50, inclusive
`random.random()`	Return a random fraction between 0 and 1
`choice(['A',0,'B'])`	Return a randomly chosen item from the list passed as a parameter
`sample(MyList,10)`	Return a sample of 10 items from `MyList`, without replacement
`shuffle(MyList)`	Randomly rearrange the items in `MyList`, in place

Randomization analysis often requires "bootstrapping" a data set—that is, creating many data sets by resampling an original dataset with replacement. The `random` module offers sampling without replacement using `random.sample()`, but not sampling with replacement. To generate bootstrapped samples of an indicated size, you can use list comprehension (described in Chapter 9) along with the `random.choice` function:

```
#!/usr/bin/env python
"""demo of bootstrapping via resample with replacement"""
import random

NumSamples = 100  # Number of resamplings to conduct
Bootstraps=[] # List to store the random lists

# for the demo, create a list of numbers 0 to 19
# normally these would be the original data
DataList = range(20)

# loop to perform repeated operations:
# Values of X and Y below are not used -- just counters
for X in range(NumSamples):
   # list comprehension in [] builds a list via sampling
   Resample = [random.choice(DataList) for Y in DataList]
   Bootstraps.append(Resample)

for Z in Bootstraps:
   print Z
```

Remember that list comprehension is like a one-line `for` loop. The program here uses it to repeat an operation once per value in the original data list. The actual value assigned to `Y` during the list comprehension doesn't have special meaning; it is equivalent to using `len(DataList)` to know how many times to choose a random sample from the original data set.

The `time` module

It is often necessary to keep track of time, convert times and dates from one format to another, and perform calculations with time. All of these needs can be addressed with the `time` module. A printable string version of the current date and time is produced by `time.asctime()`. (Remember ASCII refers to text characters.) You can also keep time in milliseconds using your computer's clock and the function `time.time()`, or get a list with the current time split into elements (year, month, day, etc.) with `time.localtime()`. This module also contains functions for converting between times and doing time math. As with all the modules, try the `help(time)` command or `dir(time)` for more options.

Third-party modules

The modules we have discussed up to now have all been part of the Python standard library. Third-party modules, on the other hand, aren't necessarily distributed with the core Python components. In many cases, you will have to download and install these modules on your computer before you can import them into a program. The specifics of the installation process can vary greatly. Some external modules consist of just a few text files containing Python code, while others include dozens of files and require additional libraries and programs be installed.

There are often multiple ways to install a particular module on a particular computer. These fall into two broad categories:

- First, you can download the module from its Web site. In addition to the code for the module itself, such downloads usually include detailed installation instructions (often in a file called `README` or `INSTALL`) and automated installer scripts. Even so, it can take some skill to get all the various components installed and operational, and to troubleshoot any problems that arise.
- Second, and usually easier, you can use a package manager to handle installation. Package managers are programs that provide semi-automated installation for a wide variety of software collected into ready-to-go bundles. There are several package managers for OS X; one of the most popular is MacPorts, which can be found at `www.macports.org`. Most versions of Linux come with built-in package managers, such as the graphical Synaptic Package Manager in Ubuntu and the `apt-get` command-line package manager developed for Debian. Package managers have the great advantage of also installing all the other programs or libraries a module requires. (You may hear these additional programs and libraries referred to as dependencies.) However, such automation can sometimes lead to unexpected consequences, such as installing an entirely new version of Python in addition to the one you already have. Be prepared for a package manager to take anywhere from a few seconds to several hours to install a module, depending on the module's size and dependencies.

This high degree of variation in installation approaches can get confusing, especially when there are additional choices to be made for installing a particular module. In general, there are a few steps to take when you would like to use a third-party module.

First, check to see if it is already installed. You may get lucky. As Python becomes more popular, more and more third-party modules are being included by default, just to make Python easier to use. And even if the module wasn't installed by default, it might have been installed alongside another piece of software that depends upon it.

Second, visit the module's Web site. In addition to downloads, there are often specific recommendations for each operating system. Some sites may even suggest that you use a package manager to facilitate the installation, despite the module being available as a direct download.

Third, investigate the package managers for your system to see if the module is available there. If installing the module directly from the Web site looks particularly difficult, or if you've encountered problems during installation that you can't resolve, a package manager may be the best option.

Software installation is discussed in greater detail in Chapter 21 and Appendix 1. Many of the issues explored there are relevant to installing modules, so flip ahead if you encounter problems with the third-party modules described here, or if you have any other installation-related problems. As discussed in Chapter 21, many installations require that the `gcc` compiler program be installed on your computer. This tool is not installed by default on OS X; rather, it is part of the optional Xcode developer tools package.[1] These tools are provided on the DVD for installing or upgrading OS X, which came with your computer. You will need these tools sooner or later, so it is a good idea to install them before you get in a pinch without them.

NumPy

To anyone familiar with MATLAB, Mathematica, or R, Python's [in]ability to handle numerical matrices with core objects and functions will seem rather limiting. For example, as you saw with list comprehension in Chapter 9, there is no direct way to access the second column of a two-dimensional list of lists. Fortunately, there are a variety of modules that add such functionality. The most widely used is the third-party numerical module, **NumPy**.[2]

NumPy is installed by default on OS X 10.6 (Snow Leopard), but if you have an earlier version of OS X you will need to install it yourself. General installers and documentation for NumPy can be found at `numpy.scipy.org`. Make sure you have `gcc` installed (test by typing `which gcc` at the command line, and then install the OS X Developer Tools if you don't), then download the Unix version (e.g., numpy-1.3.0.zip) from `new.scipy.org/download.html`. This archive will uncompress into a folder. Open a terminal window. Change into that directory with the `cd` command and then run the installer script:

```
host:~ lucy$ python setup.py build
```

---

[1] Package managers like MacPorts can also install the `gcc` compiler, as well as compilers for Fortran and other languages.

[2] Although there are many uses for Python in processing your data files, we nonetheless recommend that you learn to use R or MATLAB for more intensive numerical analyses. Because of these more suitable tools, we don't show you here how to carry out numerical analyses or generate graphics in Python.

This will print out several pages of feedback. Once it's done, install as follows:

```
host:~ lucy$ sudo python setup.py install
```

If these installation steps do not work, or if they did work and you just want to understand more about what was done, consult Chapter 21 for further information on installing software.

Once NumPy is installed on your computer, you can import it to your programs or at the interactive prompt with:

```
>>> from numpy import *
```

This style of importing the module will make the objects from the NumPy module available to you without prefixing them with `numpy`. Be aware of potential naming conflicts. For example, `e` (the mathematical constant) and `float` are both built-in NumPy objects. If you import the module components this way, you shouldn't use these names for anything else. If in doubt, use `dir()` to see all the names in use.

The first NumPy feature you will probably want to use is its support for creating and accessing array objects. The syntax for NumPy arrays is similar to that for regular Python lists:

```
>>> MyArray = array([[1,2,3],[4,5,6],[7,8,9]])
>>> MyArray
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

The array object in the above example is a 2-D table of numbers. Note that the function `array()` and `array` objects are not part of the Python language itself, but were imported from `numpy`.

Now the fun begins. You can easily access rows in these 2-D structures, just as you can in Python lists. You can also access a column of numbers from this array, something not directly possible in Python without using loops or list comprehension. The syntax for extracting various subsets of arrays is very similar to that of MATLAB and R:

```
>>> MyRow = MyArray[1,:]    ← Second row, all columns
>>> MyRow
array([4, 5, 6])
>>> MyColumn = MyArray[:,1] ← Second column, all rows
>>> MyColumn
array([2, 5, 8])
>>> MyArray[:2,:2]
array([[1, 2],
       [4, 5]])
```

As illustrated above, NumPy can extract square subsets of the arrays in addition to rows and columns.

NumPy has many other mathematical tools, and chances are that if you do any data analysis in Python, you will become a heavy user of this module. These additional tools include some pretty basic functions that aren't available in the core Python tools, including resources for statistics:

```
>>> V = [1,1,3,5,6]
>>> median(V)
3
>>> mean(V)
3.200000
>>> max(V)
6
>>> min(V)
1
```

Biopython

For simple molecular sequence processing, you can create your own scripts based on the `seqread.py` program presented earlier. For more complex tasks, though, the Biopython module contains a variety of tools for the analysis of molecular sequence data, including the ability to read and convert sequence files and to retrieve BLAST results. You can read about and download Biopython at `biopython.org`.[3] After uncompressing the source archive, it can be installed with a few commands:

```
host:~ lucy$ cd biopython-1.53
host:biopython-1.53 lucy$ python setup.py build
host:biopython-1.53 lucy$ sudo python setup.py install
```

The directory name in the first line here will depend upon the version available at the time of download. For more details on installing software, see Chapter 21.

Extensive tutorials for using Biopython are available at the Web site `tinyurl.com/pcfb-biopy` and elsewhere, so we won't go into detail here. Briefly, a special type of sequence variable is created when importing data files or when creating sequences from scratch with the `Seq()` function. These kinds of variables have many properties and built-in functions, which are revealed by using the `dir()` function:

---

[3] If you install Biopython using a package manager, it will attempt to install NumPy also, so if you would like both, try installing Biopython first.

```
>>> from Bio.Seq import Seq
>>> MySeq = Seq("ACGGCAACGTTTTGTTATGGAAACAGATGCTTT")
>>> dir(MySeq)
 [... 'complement', 'count', 'data', 'endswith', 'find', 'lower',
 'lstrip', 'reverse_complement', 'rfind', 'rsplit', 'rstrip', 'split',
 'startswith', 'strip', 'tomutable', 'tostring', 'transcribe',
 'translate', 'ungap', 'upper']
>>> MySeq.translate()
 Seq('TATFCYGNRCF', ExtendedIUPACProtein())
>>> help(MySeq)
```

Standard applications of Biopython include loading sequences from FASTA files, searching them against sequence databases with BLAST, then parsing the BLAST results and organizing the sequence files according to their similarities to known sequences.

Other third-party modules

The third-party modules described here are only a small fraction of those that are available. You will encounter many others as you use and modify existing Python programs and write your own more specialized software.

There are many modules that add graphical capabilities to your programs. The `matplotlib` package, available at `http://matplotlib.sourceforge.net/`, makes it possible to generate data plots with MATLAB-like commands from within a Python script. This module requires many other software packages, and it is best installed with a package manager (e.g., MacPorts; see Chapter 21) that can automatically chase these down. An example script that uses `matplotlib`, called `matplotCTD.py`, is available in the scripts folder. This script was used to generate the plots in examples Figure 17.9C–F. If you have `matplotlib` installed, you can try running the script from within the `examples/ctd` folder using `o_*.txt` for the file list.

Modules also make it possible to interact with hardware in new ways. The `pyserial` module, for example, can read and write data from a serial port. You could use this to write custom control and logging software for a wide variety of instruments. The installation of `pyserial` is explained in Chapter 21 and more about designing and interfacing with hardware is discussed in Chapter 22.

If you are working with very large data sets, or data with complex relationships, you will probably want to look into using a relational database such as MySQL. The module `MySQLdb` enables direct connections to MySQL databases from within your Python programs. Chapter 15 is devoted to explaining databases and using Python in this context.

You will sometimes encounter `.csv` data files containing comma-separated values. These are not as simple to parse as tab-delimited files because the fields

can contain commas within quotation marks, and a simple `.split()` function will not properly subdivide them. For this reason, we recommend using tabs as delimiters whenever possible. To help you parse comma-delimited files, however, Python has a built-in `csv` module, which can wrap around your file-opening statements to pull out the values as a list. This functionality is illustrated in the short snippet below:

```
#! /usr/bin/env python
import csv
AllRows = csv.reader(open('shaver_etal.csv','rU'))
# AllRows can be stepped through like a file object
# the Line variable will contain a list of parsed values
for Line in AllRows:
    print Line
```

Finally, there is a built-in module that is used to parse and generate XML files. The `xml` library contains a couple of alternatives for working with XML. The simpler one is probably the `xml.dom.minidom` module. The use of this module is more involved than we can cover, even in later chapters, but if you are interested in working more with Python and XML files, there are many tutorials online which walk through `minidom` usage.

Making your own modules

There is nothing magical about the libraries and modules that you have been importing in this last section. In essence, they just contain more Python code which you are adding to your script. You can easily make your own modules, containing functions and features which you have defined yourself.

For example, in Chapter 10, you created a `decimalat()` function by placing a block of code at the beginning of your program. If you want to use this code in other programs that you write, you can simply copy the `def:` block into a file, and save it in your `scripts` folder as `mymodules.py`. Be sure to include in this file whatever `import` commands are necessary for the module to run. (In this case, it would need the `re` module.)

Now, in another script, you can import everything from your `mymodules` file using:

```
from mymodules import *
```

Note that you don't need to include the `.py` file extension at the end of `mymodules`. The `decimalat()` function should now be available for use, just as if you had typed its function definition into the top of your current script.

Remember that functions shouldn't rely upon variables in the outside world—only the ones that you send in via the function parentheses—and the outside

world can't use the function's internal variables either, except those sent back with the `return` command. This means that when you define a function, you must also define any variables you wish to be sent to it. You can even set default values if it makes sense to do so, for those cases where the function is called by the user without specifying variables. For example:

```
def bootstrap(DataList,Samples=10):
```

This creates a function that can be passed two parameters in a script, for example using the line `bootstrap(MyData,100)` to generate 100 bootstrap datasets. If called just as `bootstrap(MyData)`, it will default to generating 10 bootstraps, as indicated in the original definition.

A PRACTICAL NOTE When studying examples of Python programs found on the Web, you may begin to notice some strange-looking `if __main__` lines near the beginnings of these programs. These lines are for more advanced applications of a program, where it has to know whether it is being run as a stand-alone program, or being imported as a module by a master program. In this book, we recommend a simpler approach, in which your program is always the master program or simple module—so for now, you can safely ignore such esoteric considerations. To use the example code that has such lines, you can either remove them and rearrange the program, or leave it in and work around them.

Going further with Python

In the last several chapters, we have only been able to skim the surface of the capabilities of Python, in hopes that you will become comfortable writing and adapting short programs. A quick reference of common Python commands is presented in Appendix 4, and debugging and program troubleshooting tips are discussed in the next chapter. Chapter 14 provides an overview to general categories of data analysis problems, and makes recommendations of which tools to use, drawing on the techniques we have discussed in Chapters 1–12.

In general, our goal is to empower you to branch out on your own. Think of the question you want to tackle ("How do I …?") and you should be able to find the beginnings of a solution, whether in the previous example files or online. When faced with a challenge, don't hesitate to take advantage of any relevant example code. You can also consult with colleagues at `practicalcomputing.org`.

To reinforce and expand your skills, you will want to refer to any of the excellent Python resources online or in print. If you search the Web for "python tutorial," you'll retrieve something like 1.5 million hits. With that in mind, here are some of the best references, both in print and online:

- Lutz, Mark. *Learning Python*. Farnham: O'Reilly, 2007.
  This book provides a good walk-through of starting out in Python, and includes more advanced topics as well.
- Pilgrim, Mark. *Dive into Python*. Berkeley, CA: Apress, 2004.
  This is both a book and a Web site, `http://diveintopython.org`. It operates at a fairly high level, but has clear explanations of string manipulations, lists, and other essential Python components.
- Python Quick Reference, `http://rgruet.free.fr/PQR26/PQR2.6.html`.
  An excellent reference table for Python syntax, with a very clear presentation of what has changed between versions.

Python's next revision, Python 3, is available for installation but hasn't been widely adopted yet. You may begin to come across systems where it is in use. Most programs can be converted to Python 3 with a few modifications, or by using the `2to3` utility. One difference that you will notice immediately, though, is that the `print` statement has been changed to become a function, `print()`. In practical terms, this means it is used more like the `.write()` method you have used to write to files and to the `sys.stderr.write()` function. You can read about other differences at `docs.python.org/3.1/whatsnew/3.0.html`.

SUMMARY

In this chapter you have learned how to:

- Add specialized capabilities to Python programs using external modules, including `urllib`, `os`, `math`, `random`, `time`, `numpy`, Biopython, and `matplotlib`
- Create your own modules using function definitions

Moving forward

- Recreate the results of the `curl` example from the end of Chapter 5, but in Python with the `urllib` module.
- Modify `seqread.py` to print out sequence sizes, sorted from the smallest to the longest sequences.
- Modify some of the example programs so they use function definitions stored in a custom module.