

2 The Structure, Style, and Design of Programs



With the development environment and basic elements of a Python program described in chapter 1, we can write a great many interesting programs. In this chapter, we consider how these programs should be structured and styled.

2.1 Python Interpreters and Interactive Sessions



When we execute (i.e., run) a Python program, an **interpreter** translates the program code into an efficient intermediate representation and carries out the corresponding instructions, which are ultimately represented in the lowest-level computer language called **machine code**. Instructions in machine code can be given directly to the processor and thereby executed.

An interpreter is a program that translates another program line-by-line. There is another way of translating a program (written in a programming language) to machine code—compiling. A compiler takes the entire program at once and translates it into a highly optimized machine code program, ready for execution. An interpreter cannot optimize a program as much as can a compiler, but there are advantages to using an interpreter, including that programs can be run interactively.

The official Python interpreter CPython was installed to our development environment in section 1.4. The basic way to run a Python program `hello.py` is in a terminal window with the command


```
| python hello.py
```

Here the interpreter program `python` is called to interpret `hello.py` and the results are printed to the terminal. Programs written in a file like `hello.py` are called **scripts**.

Another way to run a Python script is within an **interactive session** (i.e., interactive shell, read-evaluate-print loop (REPL), or kernel) that runs lines of code as they are entered by the programmer. There are multiple programs that provide interactive Python sessions, including the standard one provided by the CPython distribution and invoked in a terminal window with the command `python` (without

a script given). This command causes the terminal to start an interactive session with a prompt that appears similar to the following:

```
$ python
Python 3.<specific version number> | <additional information>
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```



Python statements can be entered here and executed by pressing the key . For instance, a list can be created and sorted:

```
>>> l = [1.3, 0.8, 6.1, 3.9]
>>> l.sort()
>>> l
[0.8, 1.3, 3.9, 6.1]
```

Note that no explicit `print()` function call is necessary for objects to be printed in the interactive session. When a statement returns a value of probable interest, the session automatically prints it.

A more richly featured program for interactive Python sessions is **IPython**. The Spyder IDE in our development environment provides an IPython console (like a terminal) in the lower-right corner in the default layout.

```
In[1]: x = 4.29
In[2]: y = -38.1
In[3]: x**2 + x*y + y**2
Out[3]: 1306.5651
```

Within an IPython interactive session, a script can be run with the `runfile()` function. This is precisely what Spyder does when a script in the editor is run via the  **Run** menu item, the  button, or the **F5** key. An advantage to this technique is that the variables defined in the script now enter the existing IPython interactive session, which allows us to play with them for further analytic exploration or for debugging purposes. Note however that the reverse is not the case: variables from the interactive session are not available to the script run within that session. This allows us to have confidence that a script developed with the use of `runfile()` in an interactive session can be run outside that session. Therefore, in this book, we need not be concerned about which method—direct calling of the interpreter via `python` or the use of `runfile()` within an interactive session—was used to run a Python script.

2.2 Scripts, Modules, and Imports



As we have seen, the main code for a Python program is written in a file called a script. There are a few reasons we often want to use multiple files for a single program:

1. For large programs, a single file becomes unwieldy or difficult to navigate.
2. Portions of our program, like function and class definitions, are self-contained and potentially useful for other programs.
3. A code library (with its own files) is available to perform certain tasks without writing that code ourselves (see section 2.3).

In these cases, files containing definitions and statements (usually constants, functions, and classes) called **modules** can be **imported** to a main script. We often write our own modules for the first two cases above; that is, when our script gets long or certain definitions may be useful for other programs. For example, if we have defined a function `do_something()` in a module file `a_module.py` placed in the same directory as our main script, we can import the module and use that function in the main script with the following statements:

```
| import a_module          # Import the module
| a_module.do_something() # Call the imported function
```

Note that the function is available as an attribute of `a_module` (sans `.py`); that is, to access the function `do_something()`, we must call `a_module.do_something()`. This keeps us from accidentally overwriting names in the main script or from other modules. Occasionally, we may want a specific definition from a module to be directly available in the script. This can be achieved with the following statements:

```
| from a_module import do_something # Import function from the module
| do_something()                   # Call the imported function
```

Occasionally, the name of a module is longer than is convenient to use within a script. In this case, we can give the module a nickname, as in

```
| import a_module as am    # Import the module
| am.do_something()        # Call the imported function
```

For a project with several modules, it is best to move modules into subdirectories with names that clearly indicate their functionality. For instance, with a module in one subdirectory `blue_things/cyanotype.py` and another module in another subdirectory `red_things/redscale.py`, importing these modules requires the following statements:

```
| import blue_things.cyanotype
| import red_things.redscale
```

Note that the dot “.” indicates that a directory contains the module that follows.