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 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.

2.3 The Python Standard Library and Packages

This section introduces the importing of modules from the Python standard library and the importing of external code libraries (packages).

2.3.1 The Standard Library

Like many programming languages, Python has an extensive **standard library** with many built-in data types, constants, functions, and modules included. We have encountered some of these already, and this section gives a very short introduction to some additional aspects of the library.

Some of the standard library is available in the built-in namespace, such as the constants `True`, `False`, and `None` and the functions `print()`, `len()`, and `type()`. However, much of the standard library requires the `import`ing of modules. A list of modules of particular interest to the engineer is given in table 2.1.

Just as with our own modules, we can `import` a module from the standard library with

```
import math
```

and the related variations of `import`. The standard library modules are always in the Python **search path**, which is a list of directories in which Python searches for modules. The search path begins locally, so if you create a module `math.py`, the search path will find it before the standard library version.

Table 2.1:

Module	Description
`math`	Math constants and functions for integers and real numbers.
`cmath`	Math constants and functions for integers, real numbers, and complex numbers.
`random`	Functions for generating pseudorandom numbers.
`os`	Functions for interacting with the computer's operating system and file system.
`pathlib`	Classes for representing file paths in an operating-system independent way.
`json`	Functions for importing and exporting data in the universal JSON format.
`pickle`	Functions for saving and loading objects to files in serialized (compact) form.