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.

2.3.2 Packages

In addition to the standard library modules, a vast collection of Python **packages** can be installed and `import`ed. A package is a collection of modules. Packages are created to organize code into reusable units and distribute them to others.

The official source for Python packages is the Python Package Index (PyPI) (Python Community 2024b). The `pip` program distributed with Python is the most popular tool for installing and managing packages. Packages can be installed in Anaconda environments with `pip`, but the use of its own package manager called `conda` is preferred. The `base` Anaconda environment comes with many preinstalled packages useful for engineering computing. The installation process for installing a package includes adding the package to the Python path so that it is available to all your Python programs.

Once a package is installed, it can be `import`ed in a script. Most packages import by default one or more modules; this allows us to `import` the package in our script without individually importing each module. For instance, if we would like to use a function `do_something()` in the `foo.py` module of the `pkg` package, we could write the following:

```
import pkg                # Import the entire package
pkg.foo.do_something()    # Call a function in a module loaded by default
```

If the module is not loaded by default, or if we would only like to load a specific module, we can manually `import` the module in the usual way:

```
import pkg.foo            # Import the module
pkg.foo.do_something()    # Call a function in the module
```

Often, packages will import some important functions into its top-level namespace such that they can be called with a shorter name. In the example above, the package could elevate `do_something()` to its top-level namespace such that it can be called via `pkg.do_something()`.

Packages can contain packages, called **subpackages**. Simple packages do not require this nesting feature, but large and complex packages may.

You may one day create a package of your own. All that is required is to place your modules into a directory. If you place a special file named `__init__.py` in the directory `my_pkg`, it will be executed whenever the package is loaded.[1] Often, we want to load certain (or all) modules in this file such that they are imported by default when the package is loaded.

Your package can be distributed via PyPI or another means.

1. For earlier versions of Python, the `__init__.py` file was obligatory for a package. Now it is optional but advisable.

Box 2.1 Further Reading

- Python Community (2024a; § The Python Standard Library)
- Python Community (2024a; § The Python Tutorial, 10. Brief Tour of the Standard Library)
- Python Community (2024b), to browse PyPI packages
- Python Community (2024c), for information about creating and distributing packages

2.4 Namespaces, Scopes, and Contexts

A **namespace** is a binding of (i.e., a map from) names (identifiers) to objects. Each name is unique within a namespace. For instance, there can be only one variable `x`. In Python, as in many programming languages, namespaces are created and destroyed throughout the execution of a program. When a main script is run, the Python interpreter creates (and never destroys) the **built-in namespace** that includes mappings for several built-in objects such as the functions `print()`, `len()`, and `abs()` and the constants `True`, `False`, and `None`.

As we saw in section 2.2, the names in a namespace for an imported module `a_module` begin with the name of the module, as in `a_module.do_something()`. Or, if the module was imported with an alias, as in `import a_module as am`, the names in its namespace begin with `am`.

The main script or a module has a top-level namespace called the **global namespace**. Names defined in the script or module and outside of any function or class definition go into this top-level namespace. The execution of a function or class creates a new namespace for it. This is true for nested function and class definitions, as well. Therefore, a hierarchy of namespaces is created with nested function and class definitions. At the bottom of this hierarchy is an innermost **local namespace**. Levels below the global namespace and above a local namespace are called **non-local namespaces** (i.e., enclosing namespaces).

The **scope** of a name binding (to an object) is the portion of the code of a program in which the name is bound (i.e., where it can be used).[2] The scope of `x = 3` is the part of the code in which the use of `x` will return that `3`. The **context** for a given portion of a program is the collection of all bound names and the ordering of namespaces searched when a name is used. The context of a scope of names in a local namespace has the following search priority:

1. Local namespace

2. Sometimes the term "scope" is used to mean what we call a "context of a scope." We will try to avoid this usage, but it is quite common.