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.

2. Non-local namespaces
3. Global namespace
4. Built-in namespace

For instance, consider the namespaces, scopes, and contexts for the following script:

```
x = 3
print(f"Global x: {x}")
def plus_7(y):
    x = y + 7
    print(f"Local x: {x}")
    return x
plus_7(x)
print(f"Global x: {x}")
```

This prints the following to the console:

```
Global x: 3
Local x: 10
Global x: 3
```

To interpret these results, we see that the statement `x = 3` binds the name `x` in the global namespace such that `Global x: 3` is printed. The context for this portion of code is the collection of bindings for the names in the global and built-in namespaces and the search priority (1) global namespace and (2) built-in namespace. The `plus_7()` function definition creates a new local namespace in which `x` is bound with the assignment `x = y + 7`. The context for the function code block is the set of bindings for the names in the local, global, and built-in namespaces and the search priority (1) local namespace, (2) global namespace, and (3) built-in namespace. Therefore, the use of `x` here searches the local namespace first; finding one upon the function being called, it prints (in this case) `Local x: 10`. Finally, we see that the global namespace `x` has been unchanged by the local assignment.

Example 2.1

In the previous example, if we remove the local assignment `x = y + 7`, what happens?

Because there is no binding of `x` in the local namespace of the function, `x` is not found here. Therefore, the global namespace is searched; the global namespace `x` is found and used within the function. This results in the program printing

```
Global x: 3
Local x: 3
Global x: 3
```

> Note that if `x` had not been found in the global namespace, the built-in namespace would have been searched. Because this namespace also lacks a binding for `x`, a `NameError` would be raised.

The use of global or non-local names within a function or class definition is generally discouraged. It is difficult to read and debug code that refers to names outside of its local namespace. We prefer to pass necessary objects through input arguments. Even worse than the use of global names within a function or class definition is their reassignment or their bound object's mutation. Rarely necessary and nearly always a bad idea, this can be achieved with the use of the `global` and `nonlocal` keywords. Without these keywords, global and non-local names are read-only. With their use, global and non-local names can be reassigned and bound objects mutated, as in the following example:

```
x = 3
print(f"Global x: {x}")
def plus_7(y):
    global x
    x = y + 7
    print(f"Local x: {x}")
    return x
plus_7(x)
print(f"Global x: {x}")
```

This prints the following to the console:

```
Global x: 3
Local x: 10
Global x: 10
```

So we have altered `x` in the global namespace. Again, it is inadvisable to use this unless absolutely necessary.

2.5 Defining Classes

Defining a custom class is an extremely useful way to use Python. A class object has two kinds of **class attributes**: **data attributes** that store data and **methods**, which, as we have already seen, are functions that belong to and often operate on instances of an object.

A custom class is a convenient way to represent many kinds of objects in engineering. Here are some examples with potential data attributes and methods included:

- A time-varying signal class with data attributes `periodic`, `period`, `amplitude`, and `frequency` and methods `rms()`, `abs()`, and `plot()`
- An experiment simulation class with data attributes `time`, `executed`, `input`, and `output` and methods `execute()`, `plot()`, and `save()`
- A truss class with data attributes `members`, `connections`, `pin_angles`, `member_forces`, and `reactions` and methods `analyze()`, `max_compression()`, `max_tension()`, and `max_reaction()`

The basic syntax for a class definition is as follows:

```
class ClassName:
    """Docsting description"""
    <Statement 1>
    <Statement 2>
    <etc.>
```

Data attributes can be defined via the usual variable assignment syntax and generally follow the docstring. Method definitions follow data attributes. Consider the following class definition to represent a screwdriver tool (perhaps in the context of a robot's inventory of available tools):

```
class Screwdriver:
    """Represents a screwdriver tool"""
    operates_on = "Screw"  # Class data attributes
    operated_by = "Hand"

    def drive(self, screw, angle):  # Method definition
        """Returns a screw object turned by the given angle"""
        return screw.turn(angle)
```

Any object that is an instance of the class `Screwdriver` will have the class attributes defined above. To create an instance (i.e., instantiate), call the class name as if it were a function with no arguments, as follows: