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:

```
sd1 = Screwdriver()  # Create an instance of the Screwdriver class
sd2 = Screwdriver()  # Another instance
sd1.operates_on  # Access class attributes
sd1.operated_by
```

```
'Screw'
'Hand'
```

In many cases, we will define a special **constructor method** named `__init__()`, which will be called at instatiation and passed any arguments provided as follows (we remove docstrings for brevity):

```
class Screwdriver:
    operates_on = "Screw"  # Class data attributes
    operated_by = "Hand"

    def __init__(self, head, length):
        self.head = head  # Instance data attributes
        self.length = length

    def drive(self, screw, angle):  # Method definition
        return screw.turn(angle)
```

The attributes assigned to `self` in the `__init__()` method are called **instance data attributes**. The arguments `head` and `length` are required positional arguments that are assigned to the instance data attributes `head` and `length`.

Consider the following instances:

```
sd1 = Screwdriver(head="Phillips", length=7)
sd2 = Screwdriver(head="Flat", length=8)
print(f"sd1 is a {sd1.head}head operated by {sd1.operated_by}")
print(f"sd2 is a {sd2.head}head operated by {sd2.operated_by}")
```

```
sd1 is a Phillipshead operated by Hand
sd2 is a Flathead operated by Hand
```

So we see that instances can have different instance data attributes but they share the same class data attributes.

Note that every method has as its first argument `self`, which is the conventional name given to the first argument, which is always the instance object that includes the method. When calling a method of an instance, we do not provide the `self` argument because it is provided automatically. Before we can call the `Screwdriver` method `drive()`, we should define a `Screw` class as follows:

```
class Screw:
    """Represents a screw fastener"""
    def __init__(self, head, angle=0, handed="Right"):
        self.head = head
        self.angle = angle
        self.handed = handed

    def turn(self, angle):
        """Mutates angle attribute by adding angle"""
        self.angle += angle
```

Instances of the `Screw` class have 3 instance attributes, `head`, `angle`, and `handed`. Let's instantiate a screw and give it a turn as follows:

```
s1 = Screw(head="Phillips")
print(f"Initial angle: {s1.angle}")
sd1.drive(screw=s1, angle=3)  # Turn the screw 3 units
print(f"Mutated angle: {s1.angle}")
sd1.drive(screw=s1, angle=6)  # Turn the screw 6 units
print(f"Mutated angle: {s1.angle}")
```

```
Initial angle: 0
Mutated angle: 3
Mutated angle: 9
```

As we have seen in this example, instance data attributes can represent the **state** of an object and methods can mutate or **transition** that state. This opens up a vast number of possibilities for the engineer, for we often need to keep track of the states and state transitions of objects in engineering systems.

2.5.1 Derived Classes

A **derived class** (also called a subclass) is a class that uses another class as its basis. A class that is a derived class's basis is called a **base class** for the derived class. A derived class **inherits** all of the class data attributes and methods of its base class, and it typically has additional class data attributes or methods of its own.

Continuing the screw and screwdriver example from above, let's define a derived class for representing set screws[3] as follows:

```
class SetScrew(Screw):
    """Represents a set screw fastener"""
    def __init__(self, head, tip, angle=0, handed="Right"):
        self.tip = tip  # Add instance attribute
        super().__init__(head, angle, handed)  # Call base constructor
```

3. A set screw is a screw that holds an object in place via a force applied by its tip.

The base class was specified by placing it in parentheses in `SetScrew(Screw)`. It is not necessary to define a new constructor, but to we did so to add the instance attribute `tip`. Rather than duplicating the rest of the base class's constructor, the base constructor was called via `super().__init__()`, to which its relevant arguments were passed straight through. Trying out the subclass,

```
sd3 = Screwdriver(head="Hex", length=5)
ss1 = SetScrew(head="Hex", tip="Nylon")
sd3.drive(ss1, 2)  # Drive the set screw
print(f"Set screw angle: {ss1.angle}")
```

```
Set screw angle: 2
```

Note what has occurred: the `Screwdriver` instance `sd3` used its `drive()` method to call the `turn()` method of `SetScrew` instance `ss1`. This `turn()` method was inherited from the `Screw` class, so we didn't have to repeat the definition of `turn()` in the subclass definition of `SetScrew`.

2.6 Style Conventions

As we have seen, the syntax and semantics of Python leave open many semantically equivalent choices to be made for a given program. For instance, a list can be defined with

```
l = [
    "foo",
    "bar",
    "baz"
]
```

or with

```
l = ["foo", "bar", "baz"]
```

Semantically, these are equivalent. Which is better? This is the question of **style**: what is a good way to make these decisions?

A **style guide** is a collection of rules to be applied consistently to a program, a software suite, or even all programs in a given language. Consistency is crucial; without it, a program will be harder to read, maintain, and improve. The Python standard library style guide by Rossum, Warsaw, and Coghlan (2024) (often called simply "PEP 8") has become the de facto official Python style guide. Most professionally written programs will follow this guide with more or less variation (e.g., there might be a "house" style for certain cases). However, as Emerson tells us and PEP 8 reminds us,

> A foolish consistency is the hobgoblin of little minds