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

A common paraphrase of this leaves off the qualifier "foolish," suggesting that any consistency should be dispatched, which would be … inconsistent … with so much wisdom that embraces the value of consistency. However, a *foolish* consistency is, indeed, a hobgoblin; a style guide is most effective when its wielder knows when and how to break from it.

Rather than presenting the PEP 8 style guide in detail, we will learn it through experience. Most of the code in this book uses the PEP 8 style, with some variation for succinct presentation. Furthermore, the reader should turn on autoformatting in the Spyder IDE by opening preferences with Ctrl + , navigating to the tab Completion and linting, and, under the "Code formatting" section, choose the code formatting provider black. Check the box "Autoformat files on save" and click OK. Now, whenever you save a file, it will be autoformatted in conformance with the PEP 8 style guide.

The Black code formatter (Langa and contributors to Black 2024) necessarily goes beyond the PEP 8 guide, which still has some flexibility, to enforce a strict style. It is used extensively in the software development community, so beginning with this as a baseline should help you develop well in your own style.

There are some important aspects of style that are not enforced by PEP 8 or Black, including some aspects of docstrings and type hints. For these, we will follow another popular style guide from Google (2024), as described in the following sections.

2.6.1 Docstrings

A **docstring** is a string literal that is the first statement of a function definition, class definition, or module (i.e., a `.py` file). By convention, it is surrounded with three double-quotation marks, like

```
"""Here is a docstring."""
```

For simple functions, classes, and modules, this can be a single line of 88 characters or less. For instance,

```
def foo(x):
    """Return a fun string that ends with x."""
    return f"This string is fun {x}"
```

For complex functions, a multiline docstring is necessary and should be formatted as follows:

```
"""A succinct description or imperative.

A longer description or imperative.

Args:
    arg1: A description of the first argument.
    arg2: A description of the second argument. This one is going to be
        longer to show the hanging indent.

Returns:
    A description of the return value(s).

Raises:
    IOError: An error occurred doing X.
    ValueError: An error occurred doing Y.
"""
```

For complex classes, a multiline docstring should be formatted as follows:

```
"""A succinct description or imperative.

A longer description or imperative.

Attributes:
    attr1: A description of the first attribute.
    attr2: A description of the second attribute.
"""
```

For complex modules, a multiline docstring should be formatted as follows:

```
"""A succinct description or imperative.

A longer description or imperative.

Typical usage example:

    x = FooClass()
    y = x.BarFunction()
"""
```

The “typical usage example” idiom can also be added to function and class docstrings.

2.6.2 Type Hints

Unlike programming languages like C, Python is dynamically typed, meaning we can replace an object a given name refers to with an object of another type. For instance, the following is fine (but inadvisable):

```
x = 4  # An int type
x = "foo"  # A str type
```

In a statically typed language like C, a name's type is explicitly declared with a statement like `int x`. This makes it clear to the compiler, interpreter, or (human) programmer the type of objects to which it can refer.

In Python, type declarations are not required; however, **type hints** have been introduced to the language to serve a similar purpose. A type hint is an annotation of a name (variable or return value of a function) that indicates the type (i.e., class) of objects that should be stored in it. For instance, we can indicate that variable `x` should be of type `int` with the statement

```
x: int  # A type hint for variable x, stating x should be an int
x = 3  # Actually assign x
```

Similarly, a type hint can be included in an assignment statement, as in

```
x: int = 3 # An assignment of variable x with a type hint
```

The Python interpreter does not check these hints. However, a separate **type-checker** like mypy or pytype can be applied.[4] We will not use a type checker, but we will still find value in type hints as hints to programmers, ourselves most of all. Before the introduction of these hints to Python, it was common to annotate types via comments. Now we can reserve comments for more semantic descriptions.

For function definitions, it is very useful to use type checking, as follows:

```
def foo(x: int, y: complex, z: str) -> str:
    """An operation that returns the score."""
    return str(x + y) + z
```

As we can see, each argument can be annotated, as can the return value via the syntax `->`. Note that for numbers, a type annotation of `complex` indicates that the value can have type `complex`, `float`, or `int` (Rossum, Lehtosalo, and Langa 2024). Similarly, a type annotation of `float` indicates that the value can have type `float` or `int`. This is an interpretation of Python's "numeric tower" (Yasskin 2024).

4. At this point, unlike some other IDEs, Spyder doesn't have type-checking integration.

Box 2.2 Further Reading

- Rossum, Warsaw, and Coghlan (2024), the general Python style guide
- Rossum, Lehtosalo, and Langa (2024), the original type hint style guide
- Gonzalez et al. (2024), the variable annotation style guide
- Google (2024), the Google Python style guide

2.7 The Design of Programs

A program should be designed. Engineers are designers, so we should make excellent programmers. Unfortunately, many engineers fail to carefully consider the design of their programs, at least when it comes to those programs used for analysis and design. This often leads to programs that function poorly and are difficult to maintain. The costs associated with this are usually much greater than those accrued by a systematic design process.[5]

How should a program be designed? Software design methods abound; however, they are similar to the (many) design methods used for more conventional engineering products. Our familiarity with these, as engineers, may allow a cursory introduction to suffice.

A design typically begins with a **design problem**: a product (i.e., solution) is desired, one that does something (i.e., produces an output) for someone (i.e., a customer). For engineering computing programs, we engineers are often the customers, and the output is usually information for an analysis or design problem. The product or solution is the program itself. The design problem is often rather ill-defined at first, and we must question the customer about their goals throughout the design process. Often, the problem we thought we were solving at the beginning changes throughout the design process. Similarly, the program (solution) undergoes several iterations.

Two of the most important ways to think about program design are introduced in the rest of this section: (1) the functional analysis design method and (2) the pseudocode algorithm representation.

5. It should be noted, however, that for back-of-the-envolope calculations, we need not spend the time on a systematic design process. A paradigm I like to use is to create an exploratory `play.py` file in a project or simply use an interactive session. In this type of environment, we can explore without concern for structure, style, and careful design processes.