1.7 Functions

In Python, **functions** are reusable blocks of code that accept input arguments and return one or more values. As we have seen, a method is a special type of function that is contained within an object. We typically do not refer to methods as "functions," instead reserving the term for functions that are not methods. A function that computes the square root of the sum of the squares of two arguments can be defined as:

```
def root_sum_squared(arg1, arg2):
    sum_squared = arg1**2 + arg2**2
    return sum_squared**(1/2)
```

The syntax requires the block of code following the `def` line to be indented. A block ends where the indent ends. The indent should, by convention, be 4 space characters. The function ends with a **return statement**, which begins with the keyword `return` followed by an expression, the value of which is returned to the caller code. The variable `sum_squared` is created inside the function, so it is local to the function and cannot be accessed from outside. **Calling** (using) this function could look like

```
root_sum_squared(3, 4)
```

This call returns the value `5.0`.

The arguments `arg1` and `arg2` in the previous example are called **positional arguments** because they are identified in the function call by their position; that is, `3` is identified as `arg1` and `4` is identified as `arg2` based on their positions in the argument list. There is another type of argument, called a **keyword argument** (sometimes called a "named" argument), that can follow positional arguments and have the syntax `<key>=<value>`. For instance, we could augment the previous function as follows:

```
def root_sum_squared(arg1, arg2, pre="RSS ="):
    sum_squared = arg1**2 + arg2**2
    rss = sum_squared**(1/2)
    print(pre, rss)
    return rss
```

The `pre` positional argument is given a default value of `"RSS ="`, and the function now prints the root sum square with `pre` prepended. Calling this function with

```
sum_squared(4, 6)
```

prints the following to the console:

```
RSS = 7.211102550927978
```

Alternatively, we could pass a value to `pre` with the call

```
sum_squared(4, 6, pre="Root sum square =")
```

which prints

```
Root sum square = 7.211102550927978
```

1.8 Branching

There are special statements in all programming languages that allow the programmer to control which portions are to be executed next (or at all); that is, the **control flow**. The primary forms of control flow statements are **branching** and **looping**, and we introduce branching in this section and looping in section 1.9.

1.8.1 Branching with `if`/`elif`/`else` Statements

Branching control flow statements are based on logical conditions that are tested by the statement. The primary branching statements in Python are the `if`/`elif`/`else` statements. For instance, consider the following statements:

```
if x < 0:
    print("negative")
elif x == 0:
    print("zero")
else:
    print("positive")
```

If `x` is less than 0, it will print `negative`; if `x` is equal to 0, it will print `zero`, and otherwise (when `x` is positive) it will print `positive`. Note that the blocks of code that follow the branching statements must be indented. The `elif` (i.e., else if) and `else` statements are optional, and there can be multiple `elif` statements. Once a condition is met and the corresponding block executed, the rest of the control statements in the block are skipped.

The conditional expression is evaluated to a `bool` type (class). A `bool`ean object can have one of two possible values, `True` and `False`. If the conditional expression of a branching statement evaluates to `True`, its corresponding block of code is executed. Note that Python will evaluate non-`bool`ean conditional expression value with the built-in `bool()` function. For instance, if the conditional expression evaluates to a string `"foo"`, it will be evaluated as `bool("foo")`, which, like all nonempty strings, evaluates to `True`. However, an empty string `""` evaluates to `False`.