2.1 Python's Built-In Functions

In the previous chapter, we began our study of programming in Python by studying three main ingredients: literals, operators, and variables. We can express complex computations using just these forms of Python code, but as the tasks we want to perform grow more complex, so too does the code we need to write. In this chapter, we'll learn about using *functions* in Python to organize our code into useful sections that can be written and updated independently and reused again and again across our programs.

Review: Functions in mathematics

Before looking at functions in Python, we'll first review some of the mathematical definitions related to functions from the First-Year CS Summer Prep.

Let A and B be sets. A **function** $f : A \to B$ is a mapping from elements in A to elements in B. A is called the **domain** of the function, and B is called the **codomain** of the function.

Functions can have more than one input. For sets $A_1, A_2, \ldots, A_k$ and B, a k-ary function $f : A_1 \times A_2 \times \cdots \times A_k \to B$ is a function that takes k arguments, where for each i between 1 and k, the i-th argument of f must be an element of A_i, and where f returns an element of B. We have common English terms for small values of k: *unary*, *binary*, and *ternary* functions take one, two, and three inputs, respectively. For example, the function $f_1 : \mathbb{Z} \to \mathbb{Z}$ defined as $f_1(x) = x^2 - 10$ is a *unary* function, and the function $f_2 : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ defined as $f_2(x, y, z) = \frac{x}{y} + z$ is a *ternary* function.

We can *call* a mathematical function on a particular input (element of its domain), and calculate the corresponding output value by substituting the input value into the function definition and evaluating the expression. For example, using the functions f_1 and f_2 defined earlier,

$$f_1(3) = (3)^2 - 10 = -1$$

and

$$f_2(0.5, -1, 100) = \frac{0.5}{(-1)^2} + 100 = 100.5$$

Python's built-in functions

We've seen that Python has many operators like `+` and `in` that can be used on various data types. These operators represent mathematical functions using special symbols (e.g., addition through the `+` symbol). But because these operators are written between two expressions, they are restricted to representing *binary* functions. So of course Python must have a way of representing functions beyond the operators we've studied so far.

Now, we'll see some of Python's **built-in functions**, which are functions that are made automatically available anywhere in a Python program. For example, Python has a built-in function named `abs` that takes a single numeric input and returns its absolute value. But just knowing this function exists isn't enough—how do we actually use it?

A Python expression that uses a function to operate on a given input is called a **function call**, and has the same syntax as in mathematics:

```
<function>(<argument>, <argument>, ...)
```

Here are two examples of function call expressions that use `abs`:

```
>>> abs(-10)  # Returns the absolute value of -10.
10
>>> abs(100)
100
```

Function calls are central to programming, and come with some new terminology that we'll introduce now and use throughout the next year.

- In a function call expression, the input expressions are called **arguments** to the function call. For example, in the expression `abs(-10)`, we say that "`-10` is the *argument* of the function call".
- When we evaluate a function call, we say that the arguments are **passed** to the function. For example, in the expression `abs(-10)`, we say that "`-10` *is passed to* `abs`".
- When the function call produces its output value, we say that the function call **returns** the value, and refer to this value as the **return value** of the function call expression. For example, we say that "the return value of `abs(-10)` is `10`".

Rounding numbers

Here is a second example of a numeric function, `round`. This one is a bit more complex than `abs`, and can be used in two different ways:

1. Given a single argument number `x`, `round(x)` returns the `int` that equals `x` rounded to the nearest integer.

   ```
   >>> round(3.3)
   3
   >>> round(-1.678)
   -2
   ```

2. Given an argument number `x` and a non-negative `int` `d`, `round(x, d)` returns the `float` value of `x` rounded to `d` decimal places.

   ```
   >>> round(3.456, 2)
   3.46
   >>> round(3.456, 0)  # This still returns a float
   3.0
   ```

More than numbers!

In your mathematical studies so far, you've mainly studied *unary numeric* functions, i.e., functions that take in just one numeric argument and return another number.[1] In programming, however, it is very common to work with functions that operate on a wide variety of data types, and a wide number of arguments. Here are a few examples of built-in Python functions that go beyond taking a single numeric argument:

- The `len` function takes a string or collection data type (e.g., `set`, `list`) and returns the size of its input. For a string, its size is the number of characters it contains; for a set or list, its size is its number of elements; and for a dictionary, its size is its number of key-value pairs.

  ```
  >>> len({10, 20, 30})
  3
  >>> len('')
  0
  >>> len(['a', 'b', 'c', 'd', 'e'])
  5
  >>> len({'David': 100, 'Mario': 0})
  2
  ```

- The `sum` function takes a collection of numbers (e.g., a `set` or `list` whose elements are all numbers) and returns the sum of the numbers.

  ```
  >>> sum({10, 20, 30})
  60
  >>> sum([-4.5, -10, 2, 0])
  -12.5
  >>> sum([])  # The sum of an empty collection is 0
  0
  ```

- The `sorted` function takes a collection and returns a `list` that contains the same elements as the input collection, sorted in ascending order.

  ```
  >>> sorted({10, 3, 20, -4})
  [-4, 3, 10, 20]
  >>> sorted({10, 3, 20, -4})  # Works with sets, too!
  [-4, 3, 10, 20]
  ```

- The `max` function is a bit special, because there are two ways it can be used. When it is called with two or more inputs, those inputs must be numeric, and in this case `max` returns the largest one.

  ```
  >>> max(2, 3)
  3
  >>> max(3, -2, 10, 0, 1, 7)
  10
  ```

  But `max` can also be called with just a single argument, a non-empty collection of numbers. In this case, `max` returns the largest number in the collection.

  ```
  >>> max({2, 3})
  3
  >>> max([3, -2, 10, 0, 1, 7])
  10
  ```

- The `min` function is similar to the `max` function, except it returns the smallest of its inputs.

  ```
  >>> min(2, 3)
  2
  >>> max([3, -2, 10, 0, 1, 7])
  -2
  ```

The `type` function

One additional useful built-in function is `type`, which takes *any* Python value and returns its data type. Let's check it out:[2]

```
>>> type(3)
<class 'int'>
>>> type(3.0)
<class 'float'>
>>> type(True)
<class 'bool'>
>>> type('David')
<class 'str'>
>>> type({1, 2, 3})
<class 'set'>
>>> type([1, 2, 3])
<class 'list'>
>>> type({'a': 1, 'b': 2})
<class 'dict'>
```

If you're ever unsure about the data type of a particular value, you can always call `type` on it to check!

The `help` function:

The last special built-in Python function we'll cover in this section is `help`, which takes a single argument and displays help documentation for that argument. `help` is most commonly used for finding out more about other functions: if we call `help` on a function, the Python interpreter will display information about how to use the function.[3]

```
>>> help(abs)
Help on built-in function abs in module builtins:

abs(x, /)
    Return the absolute value of the argument.
```

It is also possible to call `help` on individual values like `3` or `{1, 2, 3}`; doing so will display the documentation for the data type of that value, like `int` or `set`. We don't recommend doing this for beginners, however, as the amount of documentation shown can be a bit overwhelming! Instead, we recommend using `help` for specific functions, at least when first starting out.

A note about nesting function calls

Just like other Python expressions, you can write function calls within each other, or mix them with other kinds of expressions like arithmetic expressions.

```
>>> max(abs(-100), 15, 3 * 20)
100
>>> sorted({10, 2, 3}) + sorted([-1, -2, -3])
[7, 7, 10, -3, -2, -1]
```

However, just as we saw with deeply nested arithmetic expressions earlier, too much nesting can make Python expressions difficult to read and understand. So, it is a good practice to break down a complex series of function calls into intermediate steps using variables:

```
>>> value1 = abs(-100)
>>> value2 = 15
>>> value3 = 3 * 20
>>> max(value1, value2, value3)
100
```

References

- CSC108 videos: Functions (Part 1, Part 2, Part 3)
- Appendix A.1 Python Built-In Function Reference

[1] Examples include the sin and log functions.

[2] The term `class` that you see returned here is the word Python uses to mean "data type". More on "classes" in later chapters.

[3] You'll find that the documentation for Python's functions and data types contain terminology or concepts that you aren't familiar with yet. That's totally normal! Being able to read and make sense of programming language documentation is an essential skill for a computer scientist, and one that you will gain experience with throughout the year. If you ever encounter something in the Python `help` documentation that you don't quite understand, please ask us about it!