# 2.4 Methods: Functions Belonging to a Data Type

The built-in functions we've studied so far all have one interesting property in common: they can all be given arguments of at least two different data types. For example, `abs` works with both `int` and `float`, `len` and `sorted` work with `set` and `list` (and others), and `type` and `help` work with values of absolutely any data type. In fact, this is true for almost all built-in functions in Python, as part of the design of the language itself.

However, Python's data types also support operations that are specific to that particular data type: for example, there are many operations we can perform on strings that are specific to textual data, and that wouldn't make sense for other data types.

Python comes with many functions that perform these operations, but handles them a bit differently than the functions we've seen so far. A function that is defined as part of a data type is called a **method**.[1] *All methods are functions, but not all functions are methods.* For example, the built-in functions we looked at above are all not methods. We refer to functions that are not methods as **top-level functions**. So far we've only seen how to define top-level functions, but we'll learn later how to define methods too. But for now, let's look at a few examples of built-in methods of various Python data types.

[1] The terms *function* and *method* are sometimes blurred in other programming languages, but for us in Python these terms have precise and distinct meanings!

## *String method examples: `str.lower` and `str.split`*

One `str` method in Python is called `lower`, and has the effect of taking a string like `'David'` and returning a new string with all uppercase letters turned into lowercase: `'david'`. To call this method, we refer to it by first specifying the name of the data type it belongs to (`str`), followed by a period (`.`) and then the name of the method.[2]

[2] There is another syntax for calling methods in Python that is actually more common, without explicitly referring to the data type of the method. We'll introduce this alternate syntax later on in the course, but for now we use this more explicit notation to help you keep track of which data type you're working with.

```
>>> str.lower('David')
'david'
>>> str.lower('MARIO')
'mario'
>>> help(str.lower)
Help on method_descriptor:

lower(self, /)
    Return a copy of the string converted to lowercase.
```

And here is an example of a second, *very useful* string method that splits a string into words (based on where spaces appear in the string):

```
>>> str.split('David is cool')
['David', 'is', 'cool']
```

## *Set method examples: `set.union` and `set.intersection`*

Next we'll look at two Python set methods that perform fundamental set operations. First, let's define these operations on mathematical sets. Let $A$ and $B$ be sets.

- We define the **union** of $A$ and $B$, denoted $A \cup B$, to be the set consisting of all elements that occur in $A$ or in $B$ (or in both).
- We define the **intersection** of $A$ and $B$, denoted $A \cap B$, to be the set consisting of all elements that occur in both $A$ and $B$.

Python's `set` data type defines two methods, `set.union` and `set.intersection`, that implement these two operations:

```
>>> set1 = {1, 2, 3}
>>> set2 = {2, 10, 20}
>>> set.union(set1, set2)
{1, 2, 3, 20, 10}
>>> set.intersection(set1, set2)
{2}
```

## *List method examples: `list.count`*

And finally, here is an example of a `list` method that takes a list and a value, and returns the number of times that value appears in the list.[3]

[3] Remember that unlike sets, lists can contain duplicates! So you can view `list.count` as a generalization of the `in` operator for lists.

```
>>> list.count([10, 20, 30, 10, 20, 40, 20], 20)
3
```

## *References*

- Appendix A.2 Python Built-In Data Types Reference