Python built-in functions

There are numerous functions in Python that perform a task or calculate a result on certain objects without being methods on the class. Their purpose is to abstract common calculations that apply to many types of classes. This is applied duck typing; these functions accept objects with certain attributes or methods that satisfy a given interface, and are able to perform generic tasks on the object.

Len

The simplest example is the `len()` function. This function counts the number of items in some kind of container object such as a dictionary or list. For example:

```
>>> len([1,2,3,4])
4
```

Why don't these objects have a length property instead of having to call a function on them? Technically, they do. Most objects that `len()` will apply to have a method called `__len__()` that returns the same value. So `len(myobj)` seems to call `myobj.__len__()`.

Why should we use the function instead of the method? Obviously the method is a special method with double-underscores suggesting that we shouldn't call it directly. There must be an explanation for this. The Python developers don't make such design decisions lightly.

The main reason is efficiency. When we call `__len__` on an object, the object has to look the method up in its namespace, and, if the special `__getattribute__` method (which is called every time an attribute or method on an object is accessed) is defined on that object, it has to be called as well. Further `__getattribute__` for that particular method may have been written to do something nasty like refusing to give us access to special methods such as `__len__`! The len function doesn't encounter any of this. It actually calls the `__len__` function on the underlying class, so `len(myobj)` maps to `MyObj.__len__(myobj)`.

Another reason is maintainability. In the future, the Python developers may want to change `len()` so that it can calculate the length of objects that don't have a `__len__`, for example by counting the number of items returned in an iterator. They'll only have to change one function instead of countless `__len__` methods across the board.

Reversed

The `reversed()` function takes any sequence as input, and returns a copy of that sequence in reverse order. It is normally used in for loops when we want to loop over items from back to front.

Similar to `len`, `reversed` calls the `__reversed__()` function on the class for the parameter. If that method does not exist, reversed builds the reversed sequence itself using calls to `__len__` and `__getitem__`. We only need to override `__reversed__` if we want to somehow customize or optimize the process:

```
normal_list=[1,2,3,4,5]
class CustomSequence():
```

```
    def __len__(self):
    return 5
    def __getitem__(self, index):
    return "x{0}".format(index)
    class FunkyBackwards(CustomSequence):
    def __reversed__(self):
    return "BACKWARDS!"
    for seq in normal_list, CustomSequence(), FunkyBackwards():
    print("\n{}: ".format(seq.__class__.__name__), end="")
    for item in reversed(seq):
    print(item, end=", ")
```

The `for` loops at the end print the reversed versions of a normal list, and instances of the two custom sequences. The output shows that `reversed` works on all three of them, but has very different results when we define `__reversed__` ourselves:

```
    list: 5, 4, 3, 2, 1,
    CustomSequence: x4, x3, x2, x1, x0,
    FunkyBackwards: B, A, C, K, W, A, R, D, S, !,
```

Enumerate

Sometimes when we're looping over an iterable object in a for loop, we want access to the index (the current position in the list) of the current item being processed. The for loop doesn't provide us with indexes, but the enumerate function gives us something better: it creates a list of tuples, where the first object in each tuple is the index and the second is the original item.

This is useful if we want to use index numbers directly. Consider some simple code that outputs all the lines in a file with line numbers:

```
    import sys
    filename = sys.argv[1]
    with open(filename) as file:
    for index, line in enumerate(file):
    print("{0}: {1}".format(index+1, line), end='')
```

Running this code on itself as the input file shows how it works:

```
    1: import sys
    2: filename = sys.argv[1]
    3:
    4: with open(filename) as file:
    5: for index, line in enumerate(file):
    6: print("{0}: {1}".format(index+1, line), end='')
```

The enumerate function returns a list of tuples, our for loop splits each tuple into two values, and the print statement formats them together. It adds one to the index for each line number, since enumerate, like all sequences is zero based.

Zip

The `zip` function is one of the least object-oriented functions in Python's collection. It takes two or more sequences and creates a new sequence of tuples. Each tuple contains one element from each list.
This is easily explained by an example; let's look at parsing a text file. Text data is often stored in tab-delimited format, with a "header" row as the first line in the file, and each line below it describing data for a unique record. A simple contact list in tab-delimited format might look like this:

```
first      last       email
john       smith      jsmith@example.com
jane       doan       janed@example.com
david      neilson    dn@example.com
```

A simple parser for this file can use zip to create lists of tuples that map headers to values. These lists can be used to create a dictionary, a much easier object to work with in Python than a file!

```
import sys
filename = sys.argv[1]

contacts = []
with open(filename) as file:
     header = file.readline().strip().split('\t')
     for line in file:
          line = line.strip().split('\t')
          contact_map = zip(header, line)
          contacts.append(dict(contact_map))

for contact in contacts:
     print("email: {email} -- {last}, {first}".format(
          **contact))
```

What's actually happening here? First we open the file, whose name is provided on the command line, and read the first line. We strip the trailing newline, and split what's left into a list of three elements. We pass '\t' into the strip method to indicate that the string should be split at tab characters. The resulting header list looks like `["first", "last", "email"]`.

Next, we loop over the remaining lines in the file (after the header). We split each line into three elements. Then, we use `zip` to create a sequence of tuples for each line. The first sequence would look like `[("first", "john"), ("last", "smith"), ("email", "jsmith@example.com")]`.

Pay attention to what zip is doing. The first list contains headers; the second contains values. The zip function created a tuple of header/value pairs for each matchup.
The dict constructor takes the list of tuples, and maps the first element to a key and the second to a value to create a dictionary. The result is added to a list.
At this point, we are free to use dictionaries to do all sorts of contact-related activities. For testing, we simply loop over the contacts and output them in a different format. The format line, as usual, takes variable arguments and keyword arguments. The use of `**contact` automatically converts the dictionary to a bunch of keyword arguments (we'll understand this syntax before the end of the chapter) Here's the output:

```
email: jsmith@example.com -- smith, john
email: janed@example.com -- doan, jane
email: dn@example.com -- neilson, david
```

If we provide `zip` with lists of different lengths, it will stop at the end of the shortest list. There aren't many useful applications of this feature, but `zip` will not raise an exception if that is the case. We can always check the list lengths and add empty values to the shorter list, if necessary.

The `zip` function is actually the inverse of itself. It can take multiple sequences and combine them into a single sequence of tuples. Because tuples are also sequences, we can "unzip" a zipped list of tuples by zipping it again. Huh? Have a look at this example:

```
>>> list_one = ['a', 'b', 'c']
>>> list_two = [1, 2, 3]
>>> zipped = zip(list_one, list_two)
>>> zipped = list(zipped)
>>> zipped
[('a', 1), ('b', 2), ('c', 3)]
>>> unzipped = zip(*zipped)
>>> list(unzipped)
[('a', 'b', 'c'), (1, 2, 3)]
```

First we zip the two lists and convert the result into a list of tuples. We can then use parameter unpacking to pass these individual sequences as arguments to the `zip` function. `zip` matches the first value in each tuple into one sequence and the second value into a second sequence; the result is the same two sequences we started with!

Other functions

Another key function is `sorted()`, which takes an iterable as input, and returns a list of the items in sorted order. It is very similar to the `sort()` method on lists, the difference being that it works on all iterables, not just lists.

Like `list.sort`, `sorted` accepts a key argument that allows us to provide a function to return a sort value for each input. It can also accept a `reverse` argument.

Three more functions that operate on sequences are `min`, `max`, and `sum`. These each take a sequence as input, and return the minimum or maximum value, or the sum of all values in the sequence. Naturally, sum only works if all values in the sequence are numbers. The `max` and `min` functions use the same kind of comparison mechanism as `sorted` and `list.sort`, and allow us to define a similar `key` function. For example, the following code uses `enumerate`, `max`, and `min` to return the indices of the values in a list with the maximum and minimum value:

```
def min_max_indexes(seq):
    minimum = min(enumerate(seq), key=lambda s: s[1])
    maximum = max(enumerate(seq), key=lambda s: s[1])
    return minimum[0], maximum[0]
```

The `enumerate` call converts the sequence into `(index, item)` tuples. The `lambda` function passed in as a key tells the function to search the second item in each tuple (the

original item). The `minimum` and `maximum` variables are then set to the appropriate tuples returned by `enumerate`. The `return` statement takes the first value (the index from `enumerate`) of each tuple and returns the pair. The following interactive session shows how the returned values are, indeed, the indices of the minimum and maximum values:

```
>>> alist = [5,0,1,4,6,3]
>>> min_max_indexes(alist)
(1, 4)
>>> alist[1], alist[4]
(0, 6)
```

We've only touched on a few of the more important Python built-in functions. There are numerous others in the standard library, including:

- `all` and `any`, which accept an iterable and returns `True` if all, or any, of the items evaluate to true (that is a non-empty string or list, a non-zero number, an object that is not `None`, or the literal `True`).
- `eval`, `exec`, and `compile`, which execute string as code inside the interpreter.
- `hasattr`, `getattr`, `setattr`, and `delattr`, which allow attributes on an object to be manipulated as string names.
- And many more! See the interpreter help documentation for each of the functions listed in `dir(__builtins__)`.
- The `type` function either returns the type of the object or returns a new type object based on the arguments passed. The type() function has two different forms:

  `type(object)` or `type(name, bases, dict)`