Lesson 5 - Thomas Deneuville

October 11, 2017

1 Advanced Functions

Let's take a look at some advanced functions that Python has to offer. These are:

- `map`
- `reduce`
- `filter`
- `any` and `all`

We will also look at **anonymous functions** and how they're useful.

2 `map`

`map` in Python takes in an iterable (i.e. a list, a set, etc.) and applies some function or some operation to each element in this iterable. The output is another iterable of the same type where each element is the output of this function.

Good example is let's say we have a list of temperatures in degrees Celsius. We want to convert these elements into their corresponding temperatures in Fahrenheit.

With `map` we need two things:

1. An iterable to iterate over (list, set, etc.)
2. Need to have some function or some operation you wish to apply to each element

2.1 Syntax

```
out = map(function, in)
```

`function` is the function or operation you wish to apply to each element and `in` is the expected input (list, set etc.). The output is a `map` object, which you can convert into whatever iterable was fed into the input (list, set, etc.).

```
In [2]: # Input is c which is the temperature in degrees C
        def c_to_f(c):
            return (9 / 5)*c + 32 # Formula to convert from C to F

        # Define some input list
        A = [0.0, 5, 10, -1.8, 32, 65]
```

```
        # Convert from degrees C to degrees F
        B = list(map(c_to_f, A))

        print(A)
        print(B)
```

```
[0.0, 5, 10, -1.8, 32, 65]
[32.0, 41.0, 50.0, 28.759999999999998, 89.6, 149.0]
```

You may notice that `c_to_f` can be a bit wasteful. We only declare this and use it once and never again. If you want, we have ways to save memory and space by declaring the function temporarily through **anonymous functions**.

To declare an anonymous function, you use the keyword `lambda`. `lambda` denotes that we have an anonymous function and you declare inputs and outputs the same way as any other function. However, anonymous functions can only run **one** line of code.

We can, if you wish, store anonymous functions in a variable and you can use it like any other function you wish.

```
In [ ]: # Original Function
        def c_to_f(c):
            return (9/5)*c + 32

        # Declares an anonymous function
        f = lambda c: (9/5)*c + 32 # Notice we don't need a return statement

        print(f(32))
```

This is fine, but what we can do is specify an anonymous function into `map`:‘

```
In [4]: #           --- anonymous function --- --input--
        C = list(map(lambda c: (9/5) * c + 32, A))
        print(C)
```

```
[32.0, 41.0, 50.0, 28.759999999999998, 89.6, 149.0]
```

```
In [8]: # I can also operate on a list of lists
        # So the input into map is the expected type
        # that your anonymous function is supposed to handle
        B = [ [0, 1, 2], [3, 4], [5, 6, 7] ]
        C = list(map(lambda x: x[0], B)) # Extracting the first element of every list
        print(C)
```

```
[0, 3, 5]
```

3 filter

You supply an iterable (list, set, etc.) and a function which evaluates to `True` or `False`. The output is another iterable that returns only the elements that evaluate to `True`. Same syntax as `map`, but the anonymous function you write should return `True` or `False`

```
In [12]: A = [ [0, 1], [0, 2], [1, 3], [2, 4], [0, 5], [1, 6] ]

         # Declared a 2D list above where each element is a 2 element list.
         # Objective is to return every list where the first element is equal to 0

         # lambda x : x[0] == 0 -- Checks the first element of the list
         # to see if it's equal to 0.  Return True if it is, and False
         # otherwise.  The goal is to return another list which
         # only contains elements where the first element in the list
         # is equal to 0
         B = list(filter(lambda x: x[0] == 0, A))
         print(B)
```

```
[[0, 1], [0, 2], [0, 5]]
```

4 reduce

Assume we have some iterable that contains elements. For example it's a list:

`s = [s1, s2, ..., sN]`

What `reduce` does is that it evaluates pairs of elements in the list with some function, then takes the result and evaluates the function again with the output and the next element, until we get to one value.

Given a function `f` that takes in two elements, the process looks like this:

4.1 First Step

`[f(s1, s2), s3, s4 ,... , sN]`

Let `a = f(s1, s2)`

4.2 Second Step

`[f(a, s3), ..., sN]`

Let `b = f(a, s3)`

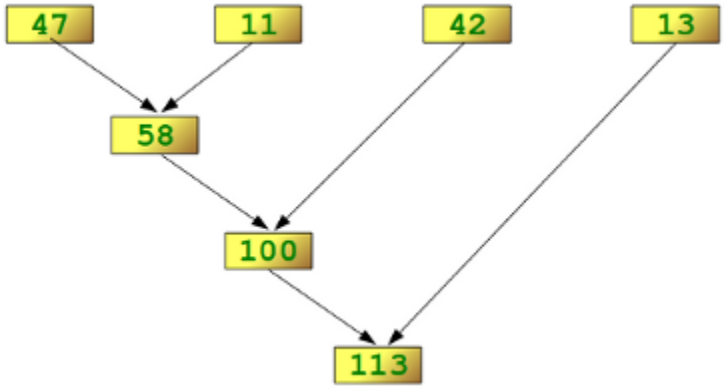
4.3 Third Step

`[f(b, s4), ..., sN]`

Keep going until the end. A good example to illustrate this mechanism is let's try summing elements in a list.

```
In [13]: from IPython.display import Image
         Image('http://www.python-course.eu/images/reduce_diagram.png')
```

Out[13]:



Reduce in this case we have 4 elements in this list. We first start off with adding the first two numbers... so 47 + 58. We take this result, apply the function to add this result and 42. Take this output, apply the function to add this with 13, and we're done.

Syntax is the same as `map` and `filter`, but your function must provide **two** inputs. This functionality is part of the `functools` package. So we must import it from there. In Python 2, it's part of the native language so there's no need to import.

```
In [18]: from functools import reduce
         A = [47, 11, 42, 13]
         B = reduce(lambda x,y: x+y, A)
         print(B)
```

```
113
```

Let's also try finding the largest value in a list. This is also implemented as the `max()` function but let's do it here as an exercise.

```
In [20]: A  = [47, 11, 42, 13]

         # To find the maximum, first compare two neighbouring numbers
         # and return whichever is the largest.  Keep going until
         # we hit the end.
         B = reduce(lambda x,y: x if x > y else y, A) # This is already implemented.
         # Called the max() function
         print(B)
```

```
47
```

5 `any` and `all`

`any` and `all` accept an iterable of `True` and `False`. `any` returns `True` if ANY values are `True`, `False` otherwise. `all` returns `True` if ALL values are `True`.

```
In [25]: C = [True, True, False, True]
         print(all(C))
         print(any(C))
         D = [True, True, True, True]
         print(all(D))
         E = [True, False, False, False, False, False]
         print(any(E))
```

```
False
True
True
True
```