List Comprehensions

We have seen how Python can easily work with all of the elements in a list using recursive techniques. Today we will look at another way for processing lists in Python called list comprehensions. List comprehensions create a list by evaluating an expression for each item in a list. The basic syntax looks like:

```
[ <expression> for <item> in <list> ]
```

Simple List Comprehensions

Download the file `lab4pr1.py` from Moodle. It contains the functions shown in the following examples, the first two of which do the same thing, but take two slightly different approaches.

Recall that you cannot open this file by clicking on it but must open it through the File menu in Visual Studio Code.

```
def triple_threat(lst):
    """ triple_threat is a function that takes a list as
        input and returns a list of elements each
        three times as large as the original
    """
    tripled = [ 3 * x for x in lst ]
    return tripled
```

Try the `triple_threat()` function with a few lists of different lengths, as in the examples below, to get an understanding of what it does. The input to the function must be a list, so something like `triple_threat(1, 2, 3)` will not work, (note the missing brackets.)

```
In [1]: triple_threat([1, 2, 3])
Out[1]: [3, 6, 9]

In [2]: triple_threat([18, 4, 21])
Out[2]: [54, 12, 63]
```

Index-based List Comprehensions

It is also possible to access the elements of the original list by referring to the index of each element. In `triple_threat_by_index()`, it is now the location of each element, here named `index`, that is changing for each item in evaluating the list comprehension.

```
def triple_threat_by_index(lst):
    """ triple_threat_by_index has the same input and output
        behaviour as triple_threat, but it uses the index of
        each list element, instead of the element itself.
    """
    n = len(lst)
    tripled = [ 3 * lst[index] for index in list(range(n)) ]
    return tripled
```

Add several printed tests to your file, (as in Lab 3), to show that these two functions produce the same result when given the same input list.

Conditional List Comprehensions

We can also limit which items in the original list to evaluate for the resulting list by using an if statement as the last term in the list comprehension to select only items which satisfy a particular condition. Now the syntax looks like:

```
[ <expression> for <item> in <list> if <condition> ]
```

```
In [1]: days = ['Sun','Mon','Tue','Wed','Thu','Fri','Sat']
In [2]: wknd = [day for day in days if day[0] == 'S']
In [3]: print(wknd)
['Sun', 'Sat']
```

And sometimes, as in the `len()` example from the lectures, all we want to do for an item in the list is keep track of the fact that it appeared in the list.

```
def length(lst):
    """ returns the length of the provided list
    """
    lc = [ 1 for item in lst ]
    return sum(lc)
```

Using the examples above as a guide, write a new function which returns the total obtained by adding up only the positive numbers in a list of numbers.

Examples:

list	value returned
`[ 1, 2, 3 ]`	6
`[  6, -3, 7, 4, -9 ]`	17
`[ -7, -5, -11 ]`	0

It is up to you decide what its signature will look like, i.e. the name of the function, and what input parameters it needs are and what they will be named.

Add several printed tests to your program to check show that your function is working correctly.

Submitting your work

Once you have completed testing the `triple_threat()` functions and implementing and testing a solution for totaling the positive numbers in a list, you are ready to submit your work.

Check that your file is named correctly, `lab4pr1.py`, and that both you and your partner's names are included in the header comment.

Upload your file to the Lab 4 page on Moodle. Make sure that both partners will have access to the work you did today for when you each want to review the lab material.

Have one of the TAs in the lab come over and briefly evaluate your work.