Higher Order Functions

In this document we cover some so-called *higher order functions* (https://en.wikipedia.org/wiki/Higher-order_function), discussing what they are useful for and when they can be used. As you will notice one can typically perform the same task in several different ways, and usually no single one is more correct than another.

In the Mandatory Assignment 4 (MA4) you should use as many of the following concepts and functions as possible. Note that one often can use them on their own or combined together to perform the same task.

Higher order functions either take functions as input arguments or return functions. The concepts originate from so-called *functional programming languages* (https://en.wikipedia.org/wiki/Functional_programming). Some examples are Haskell, Erlang, OCaml and Scheme. Most modern languages have this type of functionality. The curious reader can read more here:

https://www.guru99.com/functional-programming-tutorial.html.

We shall in this document cover some important concepts which you can use in Python.

1. List comprehensions (https://en.wikipedia.org/wiki/List_comprehension)
2. Lambda-functions (https://en.wikipedia.org/wiki/Anonymous_function)
3. `map()` (https://en.wikipedia.org/wiki/Map_(higher-order_function))
4. `functools.reduce()` (https://en.wikipedia.org/wiki/Fold_(higher-order_function))
5. `filter()` (https://en.wikipedia.org/wiki/Filter_(higher-order_function))
6. `zip()` (https://en.wikipedia.org/wiki/Convolution_(computer_science))

1 List comprehensions

When constructing more involved and complicated lists in Python, it can be easier and more readable to use *list comprehensions* instead of `for`-loops etc. List comprehensions consist of an expression with `for`-loops and `if-else` expressions written inside square brackets `[]`.

Example: Create a list with the squares of numbers 1 through 4. This can be created with:

```
>>> lst = [ii**2 for ii in range(1,5)]
>>> lst
[1, 4, 9, 16]
```

This is essentially just a shorthand way of writing:

```
lst = []
for ii in range(1,5):
    lst.append(ii*2)
```

Had we instead wanted a list taking $\Gamma(n)$ for $n = 2, 4, 6, 8$, one can write:

```
>>> import math
>>> [math.gamma(ii) for ii in range(1,10) if ii % 2 == 0]
[1.0, 6.0, 120.0, 5040.0]
```

What happens is that the `for`-loop sets `ii` to $1, 2, \ldots, 9$, and for each value checks if the remainder of `ii % 2` equals zero. If so, `math.gamma(ii)` is executed and added as an element to the list. The `for`-loop equivalent would be:

```
import math
lst = []
for ii in range(1,10):
    if ii % 2 == 0:
        lst.append(math.gamma(ii))
```

Another example is the following:

```
>>> [[0 for ii in range(0,2)] for jj in range(0,3)]
[[0, 0], [0, 0], [0, 0]]
```

This example hosts a nested list comprehension. The inner list comprehension creates a list of length 2 with zeros, and the outer a list of length 3 with these lists as elements.

A good in-depth explanation, with examples, of list comprehensions can for instance be found here:

https://www.datacamp.com/community/tutorials/python-list-comprehension

2 Lambda-functions

Lambda-functions, also called anonymous functions, exist in most languages. E.g. in MATLAB they are written as `@(x)x^2`, and if one would like to integrate the function one can simply write `integral(@(x)x^2,0,1)` rather than writing an entire "proper" function.

The Python-equivalent is that a function is simple (can be written in one line) and that one then does not have to define a function using `def functionName:`. Lambda-functions should mainly be used for functions only being used "for a short time" (e.g. is only used in one place in the code) and their purpose should be self-explanatory.

Here is an example using Lambda-functions:

```
>>> f = lambda x : x*2
>>> f(5)
10
```

Their syntax is `lambda <argument> : <expression>`. In the example above `x` is argument to the functions (here named `f`) and the expression is `x*2`, i.e. multiply `x` by 2.

An example with two input arguments being added:

```
>>> f = lambda x,y : x+y
>>> f(2,3)
5
```

The benefit of Lambda-functions mainly shows in conjunction with other functions (in particular functions like `map`, `reduce`, `filter` and `zip` which are discussed below).

One can even use functions to create other functions. Consider for example:

```
>>> def multiply(n):
        return lambda a : a * n
>>> double = multiply(2)
>>> triple = multiply(3)
>>> double(10)
20
>>> triple(10)
30
```

Thus one can create the functions `double` and `triple` very easily using the lambda-function in `multiply`.

Further reading on lambda-functions can be found here:

https://realpython.com/python-lambda/

3 `map()`

In Python, `map` returns a map-object containing the results of applying a function to each element in an interable (list, tuple, etc.). The syntax is `map(<function>, <iterable>)`. If you for instance have `map(f,[1,2,3])` a map-object with `(f(1),f(2),f(3))` will be returned. Note that the map-object needs to be converted from type `map` using the function `list()` to get a list, i.e.: `list(map(f,[1,2,3]))`.

Example:

```
>>> import math
>>> list(map(math.gamma,range(1,5)))
[1.0, 1.0, 2.0, 6.0]
```

Returns $\Gamma(n)$ for $n = 1, 2, 3, 4$.

If you want to combine `map` with a lambda-function, which in this case sqares numbers $1, 2, 3, 4$, you could write:

```
>>> list(map(lambda x : x**2,range(1,5)))
[1, 4, 9, 16]
```

One can also have multiple arguments for `map`, e.g.:

```
>>> list1=[1,2,3,4]
>>> list2=[5,6,7,8]
>>> list(map(lambda x,y : x+y, list1, list2))
[6, 8, 10, 12]
```

An example with strings:

```
>>> list_of_strings=['A','short','sentence']
>>> list(map(list, list_of_strings))
[['A'], ['s', 'h', 'o', 'r', 't'], ['s', 'e', 'n', 't', 'e', 'n', 'c', 'e']]
```

and lastly a conversion of a tuple of tuples to a tuple of lists:

```
>>> a_tuple_of_tuples=((0,1),(1,2,3),(1,0))
>>> tuple(map(list, a_tuple_of_tuples))
([0, 1], [1, 2, 3], [1, 0])
```

More information on `map` can be found here:

https://realpython.com/python-map-function/

4 `functools.reduce()`

The reduce.function `functools.reduce()` is often used together with `map()`; the combination is often called MapReduce, https://en.wikipedia.org/wiki/MapReduce and is an important concept in Data Science and Big Data.

The purpose of `functools.reduce()` is to reduce an iterable (list, tuple, etc) down to a value. In the following examples we want to sum all values in a list, and the function to do this is a lambda-function defining addition.

```
>>> import functools
>>> lst=[1,-2,3,4]
>>> functools.reduce(lambda x,y : x+y, lst)
6
```

If one instead wants to apply the Taxi/Manhattan-norm $\|x\|_1 = \sum_i^n |x_i|$ for a vector x of length n, this can be written as follows (where the vector is called `lst`)

```
>>> import functools
>>> lst=[1,-2,3,4]
>>> functools.reduce(lambda x,y : x+y, map(abs,lst))
10
```

First, `abs` is applied to every element in `lst` through `map()`. Then its result is reduced through addition with `functools.reduce()` and a lambda-function with addition. Note that the map-object needs not be reduced to a list before applying `functools.reduce`.

Find further reading on `functools.reduce()` here:

https://realpython.com/python-reduce-function/

5 `filter()`

The function `filter()` can be used when you can formulate a yes/no (true/false) question for each element in an iterable. For example, to find all elements in a list greater than 2, one can "ask each element" *Is your value greater than 2?* This can be formulated as a function returning `True` for *Yes* and `False` for *No*.

```
def greaterThanTwo(num):
        if(num > 2):
                return True
        else:
                return False
```

The filter-function can then be called on some list `a` as:

```
>>> a=[7,1,-3,4]
>>> list(filter(greaterThanTwo, a))
[7, 4]
```

Note again that also the output of `filter` needs to be converted to a `list` (from a `filter`-object).

As you may have already figured out, a lambda function is a suitable alternative for the function `greaterThanTwo` here:

```
>>> a=[7,1,-3,4]
>>> list(filter(lambda x: x>2, a))
[7, 4]
```

The lambda function `lambda x: x>2` will return `True` if `x>2`, else `False`.

A third method to achieve the same thing would be using a list comprehension:

```
>>> a=[7,1,-3,4]
>>> [e for e in a if e>2]   # e is shorthand for "element"
[7, 4]
```

It can be debated which method is "better": list comprehension or `lambda` + `filter`, and it is up to personal taste which way you find easier to understand/more elegant. The list comprehension might be slightly faster since it does not introduce a function for the `x>2`-check.

If one wants to find all vowels in a list one can use a function

```
def filter_vowels(letter):
        vowels = ['a', 'o', 'u', 'å', 'e', 'i', 'y', 'ä', 'ö']
```

```
        return True if letter in vowels else False
        # note: 'return letter in vowels' would work too.
```

and call

```
>>> word = 'xylofon'
>>> list(filter(filter_vowels,word))
['y', 'o', 'o']
```

A few more examples on how one can use `filter` can e.g. be found here:

https://www.digitalocean.com/community/tutorials/how-to-use-the-python-filter-function

6 `zip()`

The function `zip` is used to create an iterable of tuples from one or more iterables. We start with a simple example, with a list `djur` (Swedish for *animals*) with three elements. If we do the following:

```
>>> djur = ['hund', 'katt', 'orm']
>>> list(zip(range(3),djur))
[(0, 'hund'), (1, 'katt'), (2, 'orm')]
```

a list of three tuples with two elements from each `range(3)` and `djur` is created.
One can for example use `zip` when one wants to find the index of elements with a certain property, e.g. *What are the indices for all elements in a list greater than zero?*

```
>>> tal = [2, -1, 7, 9]
>>> zip_tal=list(zip(range(len(tal)),tal))
>>> [ii[0] for ii in zip_tal if ii[1]>0]
[0, 2, 3]
```

Here, `zip_tal` becomes the list `[(0, 2), (1, -1), (2, 7), (3, 9)]`, on which a list comprehension is then used to create a list of the first element in each tuple (`ii[0]`) if the second element is greater than zero (`ii[1]>0`). The result is that the elements at indices `[0, 2, 3]` in the list `tal` are greater than zero.

Also this could perhaps more easily be achieved just using list comprehension though:

```
>>> tal = [2, -1, 7, 9]
>>> [i for i in range(len(tal)) if tal[i]>0]
[0, 2, 3]

# alternatively:
>>> tal = [2, -1, 7, 9]
>>> [tal.index(e) for e in tal if e>0]
[0, 2, 3]
```

One can also use `zip` on more than two arguments, e.g.:

```
>>> x = [0.1, 0.6, 0.5]
>>> y = [0.2, -0.2, 0.9]
>>> z = [0.4, 0.4, -0.1]
>>> list(zip(x,y,z))
[(0.1, 0.2, 0.4), (0.6, -0.2, 0.4), (0.5, 0.9, -0.1)]
```

Further examples on how `zip` can be used can for instance be found here:

https://realpython.com/python-zip-function/