**Lambda Operator**

1. The lambda operator or lambda function is a way to create small anonymous functions, i.e. functions without a name.
2. they are just needed where they have been created.
3. Lambda functions are mainly used in combination with the functions filter(), map() and reduce().
4. he general syntax of a lambda function is quite simple:  
   lambda argument\_list: expression
5. >>> f = lambda x, y : x + y

>>> f(1,1)

2

### The map() Function

1. *map()* applies the function *func* to all the elements of the sequence *seq*. It returns a new list with the elements changed by *func*
2. r = map(func, seq)
3. The first argument *func* is the name of a function and the second a sequence (e.g. a list) *seq*.
4. Ex:

>>> Celsius = [39.2, 36.5, 37.3, 37.8]

>>> Fahrenheit = map(lambda x: (float(9)/5)\*x + 32, Celsius)

>>> print Fahrenheit

[102.56, 97.700000000000003, 99.140000000000001, 100.03999999999999]

>>> C = map(lambda x: (float(5)/9)\*(x-32), Fahrenheit)

>>> print C

[39.200000000000003, 36.5, 37.300000000000004, 37.799999999999997]

5. map() can be applied to more than one list. The lists have to have the same length.

6. >>> a = [1,2,3,4]

>>> b = [17,12,11,10]

>>> c = [-1,-4,5,9]

>>> map(lambda x,y:x+y, a,b)

[18, 14, 14, 14]

>>> map(lambda x,y,z:x+y+z, a,b,c)

[17, 10, 19, 23]

>>> map(lambda x,y,z:x+y-z, a,b,c)

[19, 18, 9, 5]

### Filtering

1. The function filter(function, list) offers an elegant way to filter out all the elements of a list, for which the function *function*returns True.
2. The function filter(f,l) needs a function f as its first argument. f returns a Boolean value, i.e. either True or False. This function will be applied to every element of the list *l*. Only if f returns True will the element of the list be included in the result list.
3. >>> fib = [0,1,1,2,3,5,8,13,21,34,55]

>>> result = filter(lambda x: x % 2, fib)

>>> print result

[1, 1, 3, 5, 13, 21, 55]

>>> result = filter(lambda x: x % 2 == 0, fib)

>>> print result

[0, 2, 8, 34]

### Reducing a List

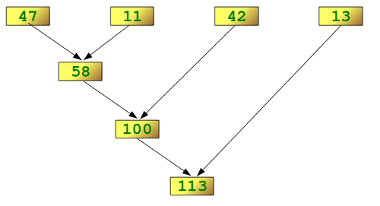
The function reduce(func, seq) continually applies the function func() to the sequence seq. It returns a single value.   
  
If seq = [ s1, s2, s3, ... , sn ], calling reduce(func, seq) works like this:

* At first the first two elements of seq will be applied to func, i.e. func(s1,s2) The list on which reduce() works looks now like this: [ func(s1, s2), s3, ... , sn ]
* In the next step func will be applied on the previous result and the third element of the list, i.e. func(func(s1, s2),s3)  
  The list looks like this now: [ func(func(s1, s2),s3), ... , sn ]
* Continue like this until just one element is left and return this element as the result of reduce()

We illustrate this process in the following example:

>>> reduce(lambda x,y: x+y, [47,11,42,13])

113

The following diagram shows the intermediate steps of the calculation:   


List comprehension: http://www.python-course.eu/list\_comprehension.phpex:

1.List comprehension is a complete substitute for the lambda function as well as the functions map(), filter() and reduce().

2. In the chapter on lambda and map() we had designed a map() function to convert Celsius values into Fahrenheit and vice versa. It looks like this with list comprehension:

>>> Celsius = [39.2, 36.5, 37.3, 37.8]

>>> Fahrenheit = [ ((float(9)/5)\*x + 32) for x in Celsius ]

>>> print Fahrenheit

[102.56, 97.700000000000003, 99.140000000000001, 100.03999999999999]

>>>

3.ex: Cross product of two sets:

>>> colours = [ "red", "green", "yellow", "blue" ]

>>> things = [ "house", "car", "tree" ]

>>> coloured\_things = [ (x,y) for x in colours for y in things ]

>>> print coloured\_things

[('red', 'house'), ('red', 'car'), ('red', 'tree'), ('green', 'house'), ('green', 'car'), ('green', 'tree'), ('yellow', 'house'), ('yellow', 'car'), ('yellow', 'tree'), ('blue', 'house'), ('blue', 'car'), ('blue', 'tree')]

>>>

### Generator Comprehension

**Set Comprehension**

A set comprehension is similar to a list comprehension, but returns a set and not a list. Syntactically, we use curly brackets instead of square brackets to create a set. Set comprehension is the right functionality to solve our problem from the previous subsection. We are able to create the set of non primes without doublets:

>>> from math import sqrt

>>> n = 100

>>> sqrt\_n = int(sqrt(n))

>>> no\_primes = {j for i in range(2,sqrt\_n) for j in range(i\*2, n, i)}

>>> no\_primes

{4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 22, 24, 25, 26, 27, 28, 30, 32, 33, 34, 35, 36, 38, 39, 40, 42, 44, 45, 46, 48, 49, 50, 51, 52, 54, 55, 56, 57, 58, 60, 62, 63, 64, 65, 66, 68, 69, 70, 72, 74, 75, 76, 77, 78, 80, 81, 82, 84, 85, 86, 87, 88, 90, 91, 92, 93, 94, 95, 96, 98, 99}

>>> primes = {i for i in range(n) if i not in no\_primes}

>>> print(primes)

{0, 1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97}

>>>

### Recursive Function