Lambda Operators in Python

For most of our work with ***NumPy*** arrays and ***Pandas*** data-frames, we try to avoid to use loops over the data structures:

- loops are executed at ***Python*** level:
  - -> slow interpreter and slow memory access
- most built in ***NumPy*** and ***Pandas*** functionality come from highly (hardware) optimized pre-build libraries
  - offering fast special purpose alternatives for loops
  - and generic operators from functional programming

In [16]:
```
#example speed comparison
import numpy as np
A = np.random.random((10000,10000))
```

In [16]:
```
#example speed comparison
import numpy as np
A = np.random.random((10000,10000))
```

In [23]:
```
%%time
for y in range(10000):
    for x in range(10000):
        A[y,x]=A[y,x]*2
```

```
CPU times: user 1min, sys: 80.3 ms, total: 1min
Wall time: 1min
```

In [24]:
```
%%time
(lambda x:x*2)(A)
```

```
CPU times: user 89.8 ms, sys: 716 ms, total: 806 ms
Wall time: 823 ms
```

Out[24]:
```
array([[9.43139925e-03, 1.90388052e+00, 3.12051914e+00, ...,
        1.66411615e+00, 1.82784303e+00, 2.21361073e+00],
       [2.14608853e-01, 1.11299215e+00, 5.35783758e-01, ...,
        1.23733658e+00, 1.82811428e+00, 2.36926606e-01],
       [1.45982605e+00, 2.58752508e+00, 2.20967064e+00, ...,
        1.93691748e+00, 7.48728399e-02, 5.18407694e-01],
       ...,
       [3.88968789e+00, 3.62891944e+00, 2.82408261e+00, ...,
        5.36651836e-02, 2.14925542e+00, 1.95236934e+00],
       [3.17778253e-01, 3.44470473e-01, 3.17208003e-01, ...,
        2.74015158e+00, 7.88515264e-01, 3.96493430e+00],
       [2.12882054e-03, 1.22874730e+00, 3.17103968e+00, ...,
        7.94879229e-01, 7.28504573e-01, 7.47132727e-01]])
```

Lambda Functions

Lambda functions (or more general *Lambda Calculus*) is a concept from ***functional programming***:

- each program is a nested sequence of math like function calls
- *Lambda Calculus* is Turing complete

Lambda functions in Python

are implemented as *anonymous* functions. Here a basic example:

Lambda functions in Python

are implemented as *anonymous* functions. Here a basic example:

In [ ]:
```
#standard paython function (needs def and name)
def identity(x):
    return x
```

Lambda functions in Python

are implemented as *anonymous* functions. Here a basic example:

In [ ]:
```
#standard paython function (needs def and name)
def identity(x):
    return x
```

In [ ]:
```
#function call
identity(2)
```

In [ ]:
```
#lambda function: needs no name, directly executed
(lambda x : x)(2)
```

Slightly more complicated example:

Slightly more complicated example:

In [ ]:
```
#stadard function:
def add5(x):
    return x+5
```

Slightly more complicated example:

In [ ]:
```
#stadard function:
def add5(x):
    return x+5
```

In [ ]:
```
#lambda version - direct evaluation of argument (here 2)
(lambda x: x+5)(2)
```

```
In [ ]: #lambda functions as callable object
        add5 = (lambda x: x+5)
        add5(3)
```

Recall: in Python functions can be arguments, just like scalar values Main advantage: we can use anonymous functions as arguments in other calls, without the need to define it before hand.

Recall: in Python functions can be arguments, just like scalar values Main advantage: we can use anonymous functions as arguments in other calls, without the need to define it before hand.

In [ ]:
```
#example target funktion - applies some function to some list
def listOp(aList, aFunction):
    for i in range(len(aList)):
        aList[i]=aFunction(aList[i])
    return aList
```

Recall: in Python functions can be arguments, just like scalar values Main advantage: we can use anonymous functions as arguments in other calls, without the need to define it before hand.

In [ ]:
```
#example target funktion - applies some function to some list
def listOp(aList, aFunction):
    for i in range(len(aList)):
        aList[i]=aFunction(aList[i])
    return aList
```

In [ ]:
```
def plusOne(x):
    return x+1
```

Recall: in Python functions can be arguments, just like scalar values Main advantage: we can use anonymous functions as arguments in other calls, without the need to define it before hand.

In [ ]:
```
#example target funktion - applies some function to some list
def listOp(aList, aFunction):
    for i in range(len(aList)):
        aList[i]=aFunction(aList[i])
    return aList
```

In [ ]:
```
def plusOne(x):
    return x+1
```

In [ ]:
```
A=[1,2,3,4]
```

Recall: in Python functions can be arguments, just like scalar values Main advantage: we can use anonymous functions as arguments in other calls, without the need to define it before hand.

In [ ]:
```
#example target funktion - applies some function to some list
def listOp(aList, aFunction):
    for i in range(len(aList)):
        aList[i]=aFunction(aList[i])
    return aList
```

In [ ]:
```
def plusOne(x):
    return x+1
```

In [ ]:
```
A=[1,2,3,4]
```

In [ ]:
```
listOp(A,plusOne)
```

In [ ]:
```
#now with a lambda function
listOp(A,(lambda x:x+1))
```

Lambda functions with more than one argument

Lambda functions with more than one argument

In [ ]:
```
myFunc = (lambda x,y,z: x*x+y+z)
myFunc(2,2,2)
```

if-else statements in lambda expressions

if-else statements in lambda expressions

In [ ]:
```
listOp(A, (lambda x: True if x > 2 else False) )
```

if-else statements in lambda expressions

In [ ]:
```
listOp(A, (lambda x: True if x > 2 else False) )
```

In [ ]:
```
A=[1,2,3,4]
listOp(A, (lambda x: 0 if x > 2 else x+1) )
```

Combining *lambda functions* with *Map*

The ***map*** call allows us to directly apply functions **element wise** to container objects (like lists).

Combining *lambda functions* with *Map*

The ***map*** call allows us to directly apply functions **element wise** to container objects (like lists).

```
In [ ]: A=[1,2,3,4]
        list(map(lambda x:x+1,A))
```

Hochschule Offenburg
offenburg.university

Combining *lambda functions* with *Map*

The ***map*** call allows us to directly apply functions **element wise** to container objects (like lists).

In [ ]:
```
A=[1,2,3,4]
list(map(lambda x:x+1,A))
```

In [ ]:
```
#even works for multiple inputs:
A=[2,2,2,2]
B=[1,1,1,1]
C=[1,2,3,4]
list(map(lambda x,y,z : x+y-z, A,B,C))
```

Lambda Operators in *NumPy*

```
In [15]: #we can directly apply lambda function on arrays!
         import numpy as np
         A=np.ones((10,10))
         (lambda x:x+1)(A)

Out[15]: array([[2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
                [2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
                [2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
                [2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
                [2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
                [2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
                [2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
                [2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
                [2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
                [2., 2., 2., 2., 2., 2., 2., 2., 2., 2.]])
```

In [ ]:
```
#use lambdafunctions in slicing
A[3:6,3:6]=5 #set some pos to 5
A[(lambda x:x==5)(A)]
```

In [ ]:
```
#but this is not really needed - numpy supports this directly
A[A==5]
```

In [ ]:
```
# applying lambda functions on array slices
A[3,:]=(lambda x: x*x)(A[3,:])
```

In [ ]:
```
# applying lambda functions on array slices
A[3,:]=(lambda x: x*x)(A[3,:])
```

In [ ]:
```
A
```

Lambda Operators in *Pandas*

Pandas provides the *apply* method, which allows to use lambda functions directly with data-frames.

In [ ]:
```
import pandas as pd
```

In [ ]:
```
import pandas as pd
```

In [ ]:
```
#Reading CSV file
d=pd.read_csv(path+'/DATA/weather.csv')
```

```
In [ ]: import pandas as pd
```

```
In [ ]: #Reading CSV file
        d=pd.read_csv(path+'/DATA/weather.csv')
```

```
In [ ]: d.head()
```

In [ ]:
```
#simple pandas selection of all rows where the humidity is higher than 0.9
d[d['Humidity']>0.9]
```

```
In [ ]: #same with lambda expression
        d['Humidity'].apply(lambda x: x +1)
```

In [ ]:
```
#example if-else
d['Humidity'].apply(lambda x: 0 if x < 0.5 else 1)
```

In [ ]:
```
#multiple rows in one expression
d.apply(lambda x: x['Humidity']+x['Temperature (C)'], axis=1)
```

In [ ]:
```
#more complex example
d['myNewRow']=d.apply(lambda x: x['Humidity']+x['Temperature (C)'] if x['Humidity']>0.5 else 0, axis=1)
```

In [ ]:
```
d.head()
```

In [ ]:
```

```