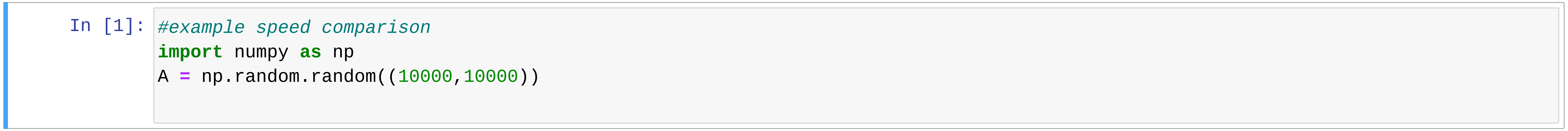
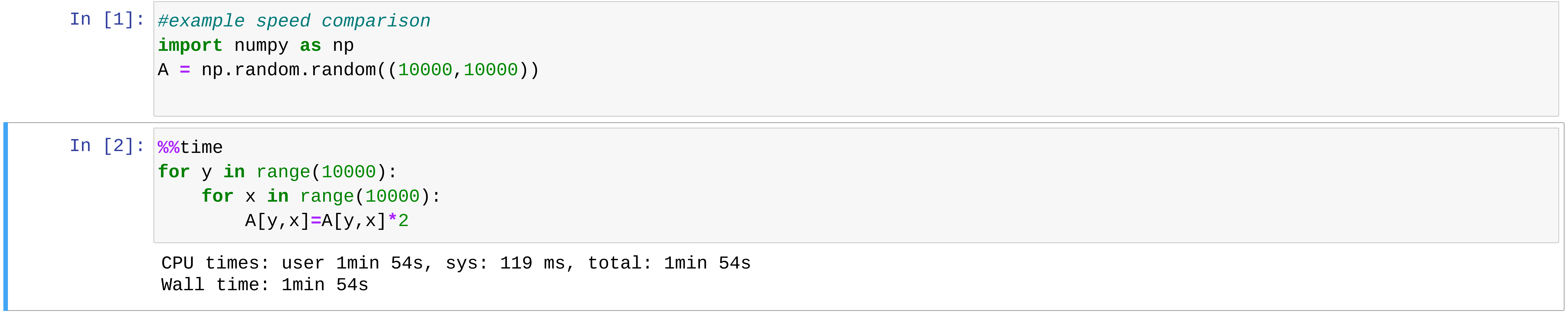
Lambda Operators in Python

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



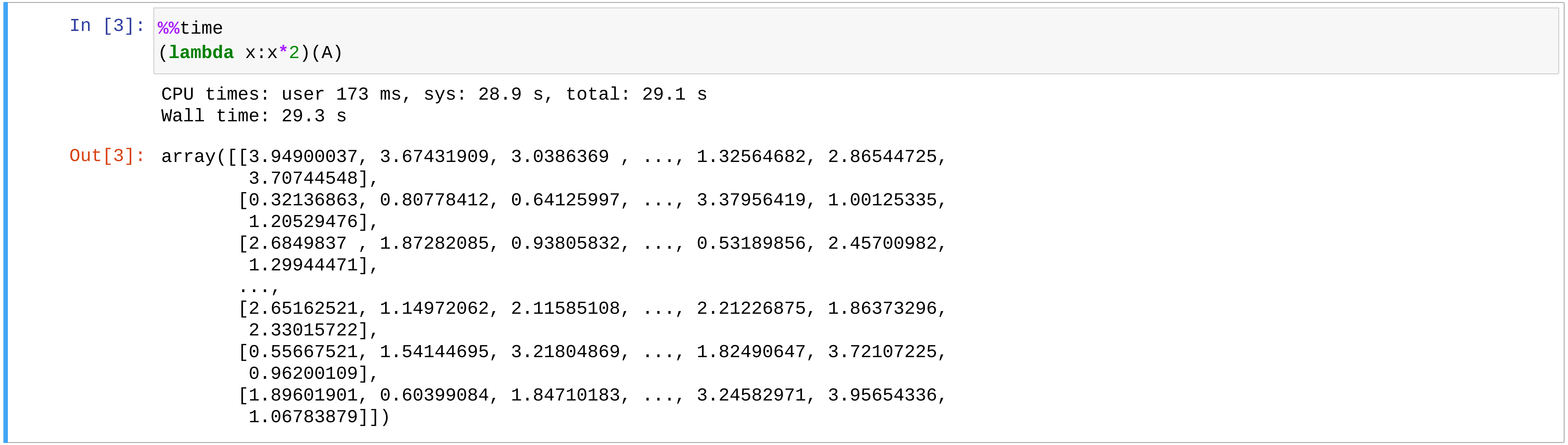
In [1]:

```
#example speed comparison
import numpy as np
A = np.random.random((10000,10000))
```

In [2]:

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

```
CPU times: user 1min 54s, sys: 119 ms, total: 1min 54s
Wall time: 1min 54s
```



```
In [3]: %%time
        (lambda x:x*2)(A)

        CPU times: user 173 ms, sys: 28.9 s, total: 29.1 s
        Wall time: 29.3 s

Out[3]: array([[3.94900037, 3.67431909, 3.0386369 , ..., 1.32564682, 2.86544725,
                3.70744548],
               [0.32136863, 0.80778412, 0.64125997, ..., 3.37956419, 1.00125335,
                1.20529476],
               [2.6849837 , 1.87282085, 0.93805832, ..., 0.53189856, 2.45700982,
                1.29944471],
               ...,
               [2.65162521, 1.14972062, 2.11585108, ..., 2.21226875, 1.86373296,
                2.33015722],
               [0.55667521, 1.54144695, 3.21804869, ..., 1.82490647, 3.72107225,
                0.96200109],
               [1.89601901, 0.60399084, 1.84710183, ..., 3.24582971, 3.95654336,
                1.06783879]])
```

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

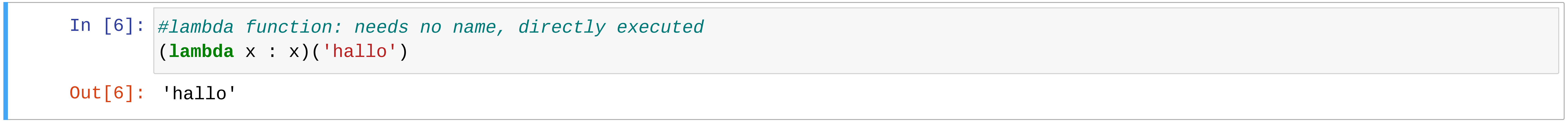
Lambda functions in Python

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

In [4]:
```
#standard paython function (needs def and name)
def identity(x):
    return x
```

In [5]:
```
#function call
identity(2)
```

Out[5]:
```
2
```

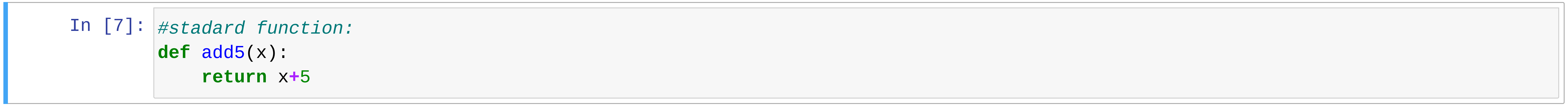


In [6]:
```
#lambda function: needs no name, directly executed
(lambda x : x)('hallo')
```

Out[6]:
```
'hallo'
```

Slightly more complicated example:

Slightly more complicated example:

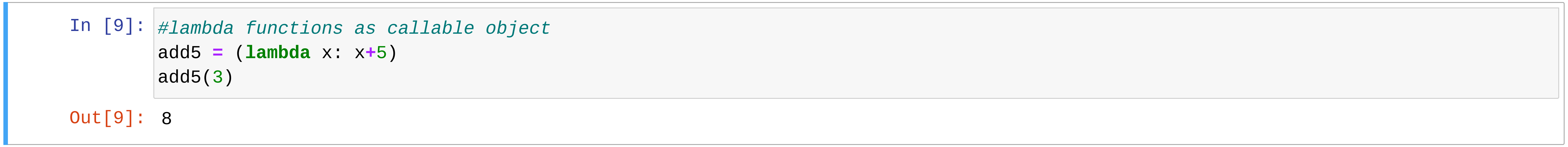


In [7]:
```
#stadard function:
def add5(x):
    return x+5
```

Slightly more complicated example:

```
In [7]: #stadard function:
        def add5(x):
            return x+5
```

```
In [8]: #lambda version - direct evaluation of argument (here 2)
        (lambda x: x+5)(7)

Out[8]: 12
```



```
In [9]: #lambda functions as callable object
        add5 = (lambda x: x+5)
        add5(3)

Out[9]: 8
```

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 [10]:

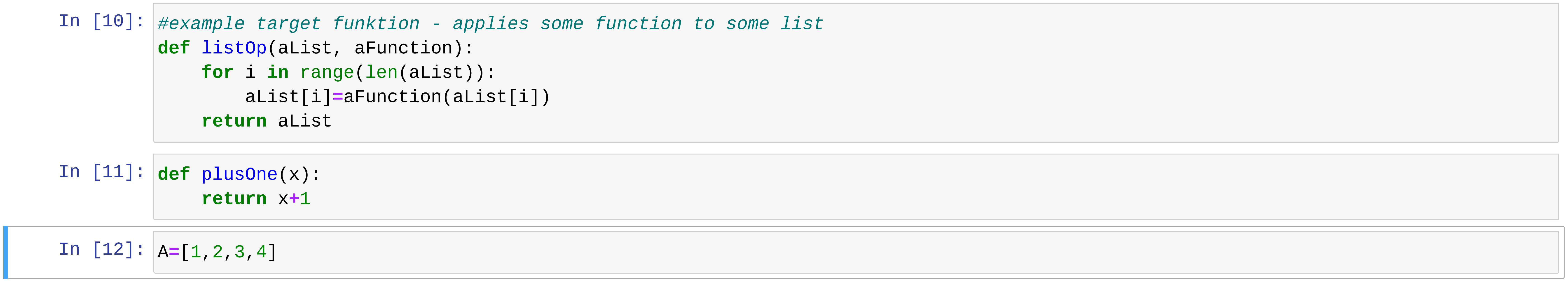
```
#example target funktion - applies some function to some list
def listOp(aList, aFunction):
    for i in range(len(aList)):
        aList[i]=aFunction(aList[i])
    return aList
```

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 [10]:
```
#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 [11]:
```
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 [10]:

```
#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
```

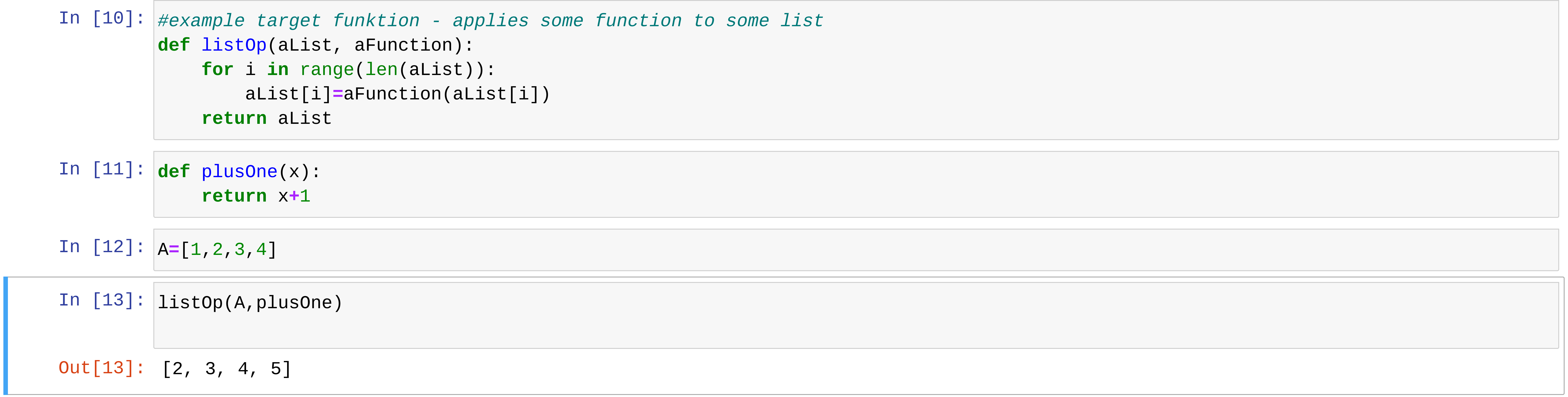
In [11]:

```
def plusOne(x):
    return x+1
```

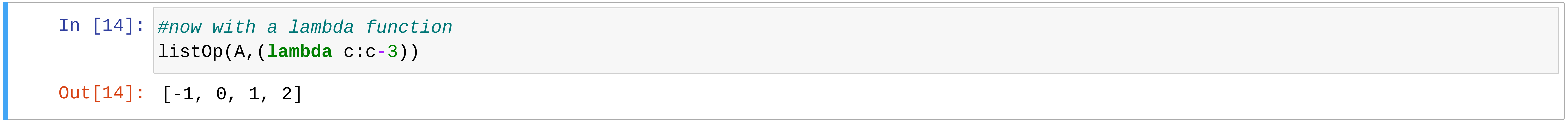
In [12]:

```
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 [10]: #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 [11]: def plusOne(x):
             return x+1

In [12]: A=[1,2,3,4]

In [13]: listOp(A,plusOne)

Out[13]: [2, 3, 4, 5]
```



In [14]:

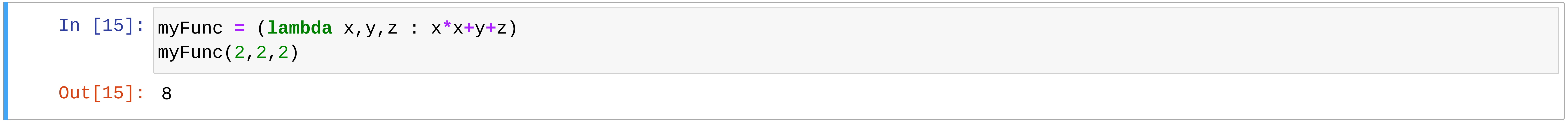
```
#now with a lambda function
listOp(A,(lambda c:c-3))
```

Out[14]:

```
[-1, 0, 1, 2]
```

Lambda functions with more than one argument

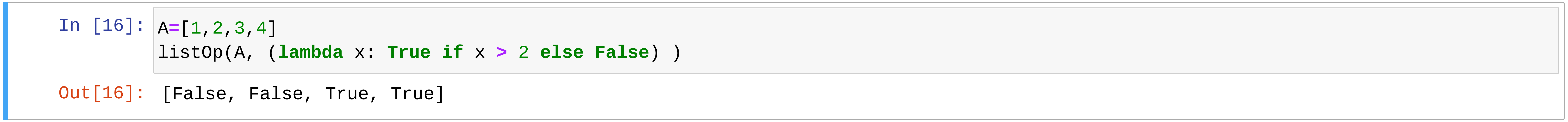
Lambda functions with more than one argument



```
In [15]: myFunc = (lambda x,y,z : x*x+y+z)
         myFunc(2,2,2)

Out[15]: 8
```

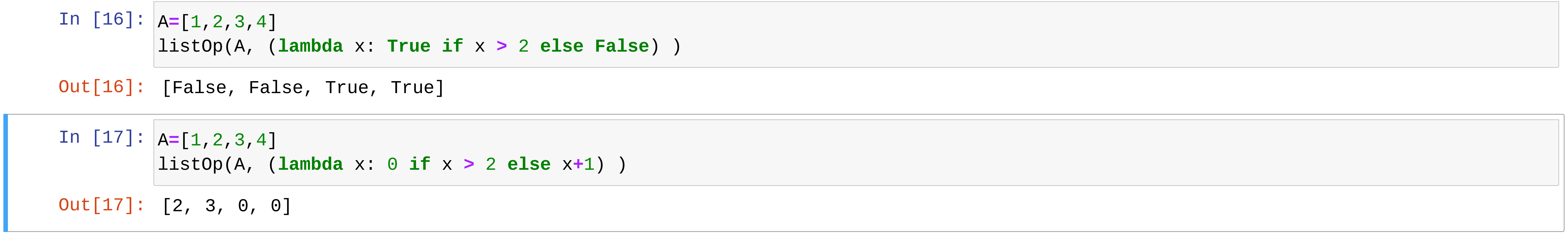
if-else statements in lambda expressions

if-else statements in lambda expressions



```
In [16]: A=[1,2,3,4]
         listOp(A, (lambda x: True if x > 2 else False) )

Out[16]: [False, False, True, True]
```

if-else statements in lambda expressions



```
In [16]: A=[1,2,3,4]
         listOp(A, (lambda x: True if x > 2 else False) )

Out[16]: [False, False, True, True]
```

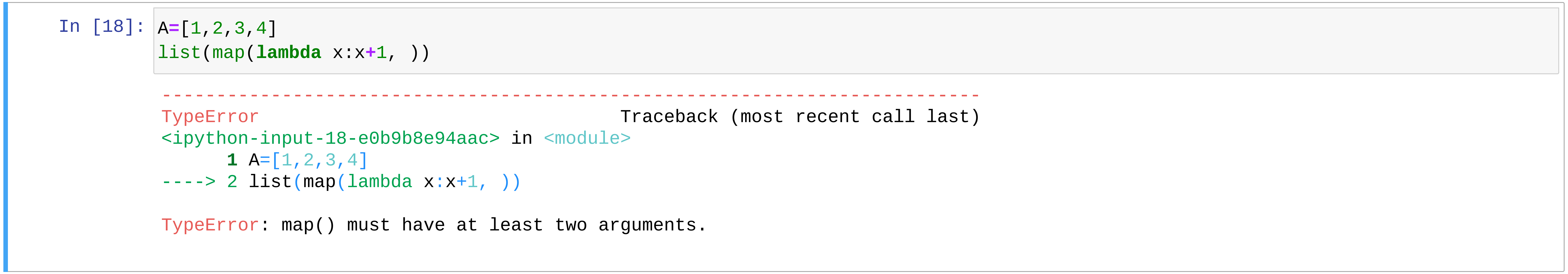
```
In [17]: A=[1,2,3,4]
         listOp(A, (lambda x: 0 if x > 2 else x+1) )

Out[17]: [2, 3, 0, 0]
```

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).

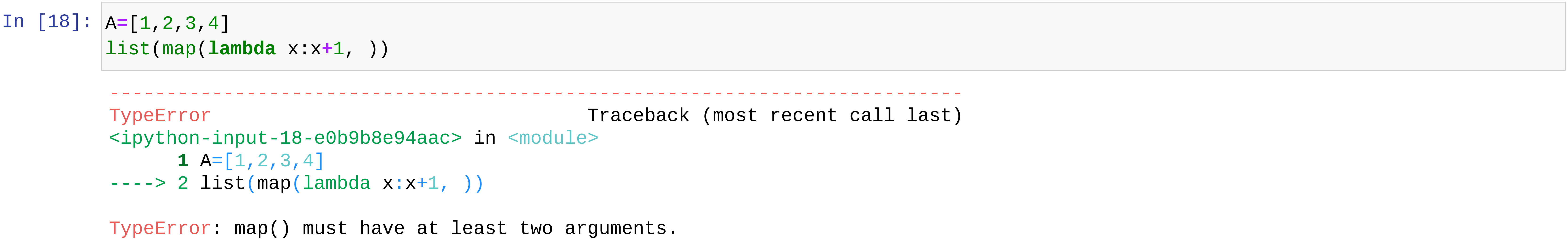


```
In [18]: A=[1,2,3,4]
         list(map(lambda x:x+1, ))
```

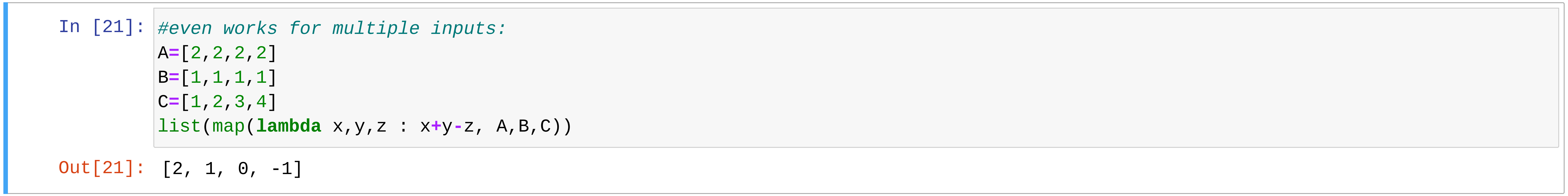
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-e0b9b8e94aac> in <module>
      1 A=[1,2,3,4]
----> 2 list(map(lambda x:x+1, ))

TypeError: map() must have at least two arguments.
```

Combining *lambda functions* with *Map*

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

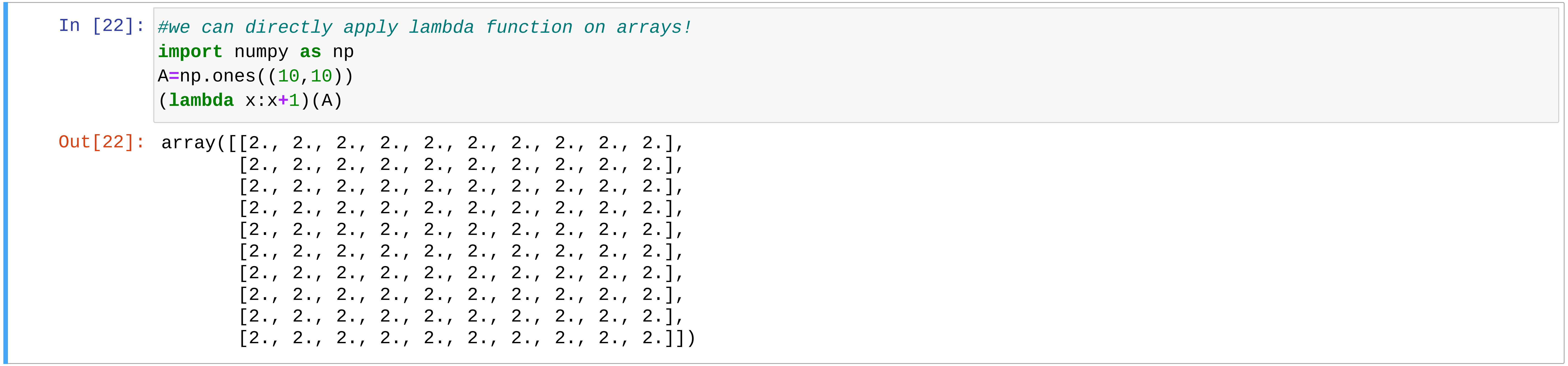


```
In [18]: A=[1,2,3,4]
         list(map(lambda x:x+1, ))

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-e0b9b8e94aac> in <module>
      1 A=[1,2,3,4]
----> 2 list(map(lambda x:x+1, ))

TypeError: map() must have at least two arguments.
```



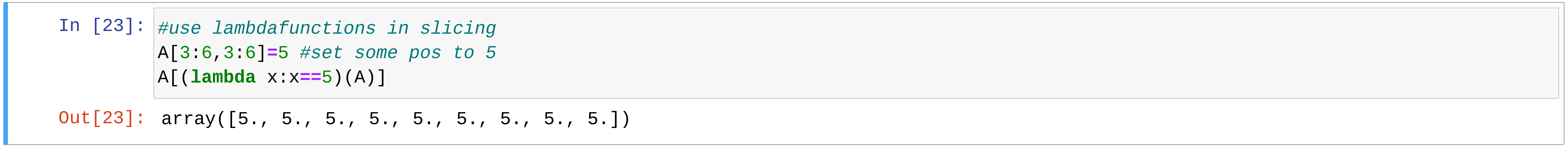
```
In [21]: #even works for multiple inputs:
         A=[2,2,2,2]
         B=[1,1,1,1]
         C=[1,2,3,4]
         list(map(lambda x,y,z : x+y-z, A,B,C))

Out[21]: [2, 1, 0, -1]
```

Lambda Operators in *NumPy*



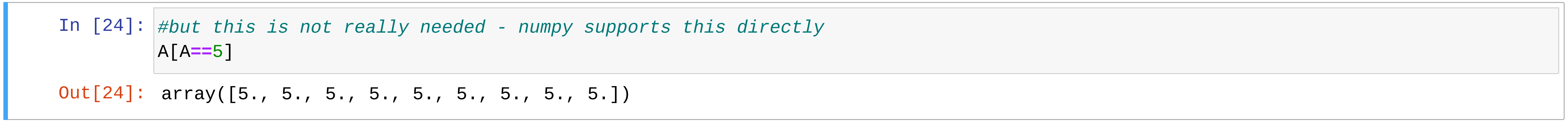
```
In [22]: #we can directly apply lambda function on arrays!
         import numpy as np
         A=np.ones((10,10))
         (lambda x:x+1)(A)
```

```
Out[22]: 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.]])
```



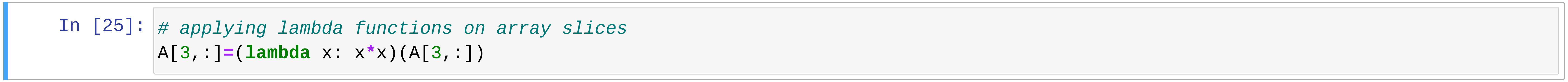
In [23]:
```
#use lambdafunctions in slicing
A[3:6,3:6]=5 #set some pos to 5
A[(lambda x:x==5)(A)]
```

Out[23]:
```
array([5., 5., 5., 5., 5., 5., 5., 5., 5.])
```



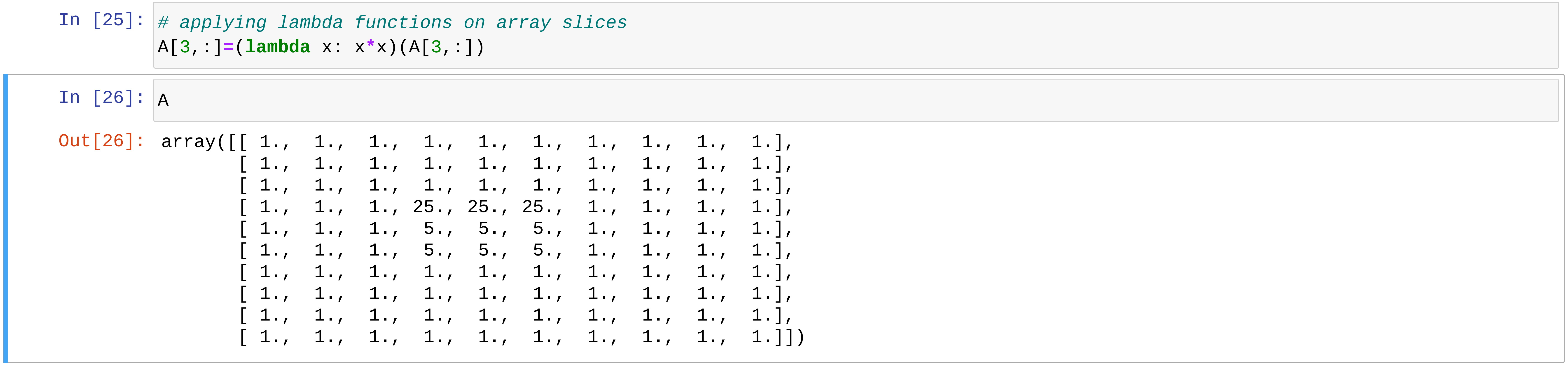
```
In [24]: #but this is not really needed - numpy supports this directly
         A[A==5]
```

```
Out[24]: array([5., 5., 5., 5., 5., 5., 5., 5., 5.])
```



In [25]:

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



```
In [25]: # applying lambda functions on array slices
         A[3,:]=(lambda x: x*x)(A[3,:])
```

```
In [26]: A
```

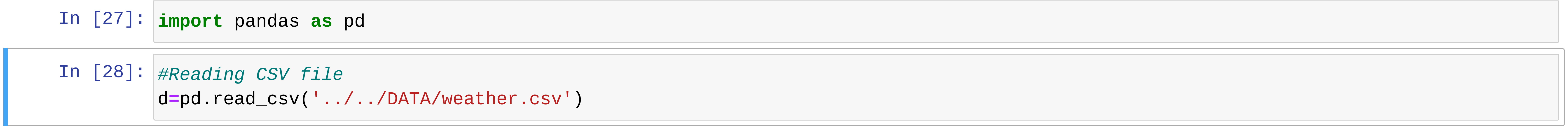
```
Out[26]: array([[ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1., 25., 25., 25.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  5.,  5.,  5.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  5.,  5.,  5.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.]])
```

Lambda Operators in *Pandas*

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

In [27]:

```
import pandas as pd
```



In [27]:
```python
import pandas as pd
```

In [28]:
```python
#Reading CSV file
d=pd.read_csv('../../DATA/weather.csv')
```

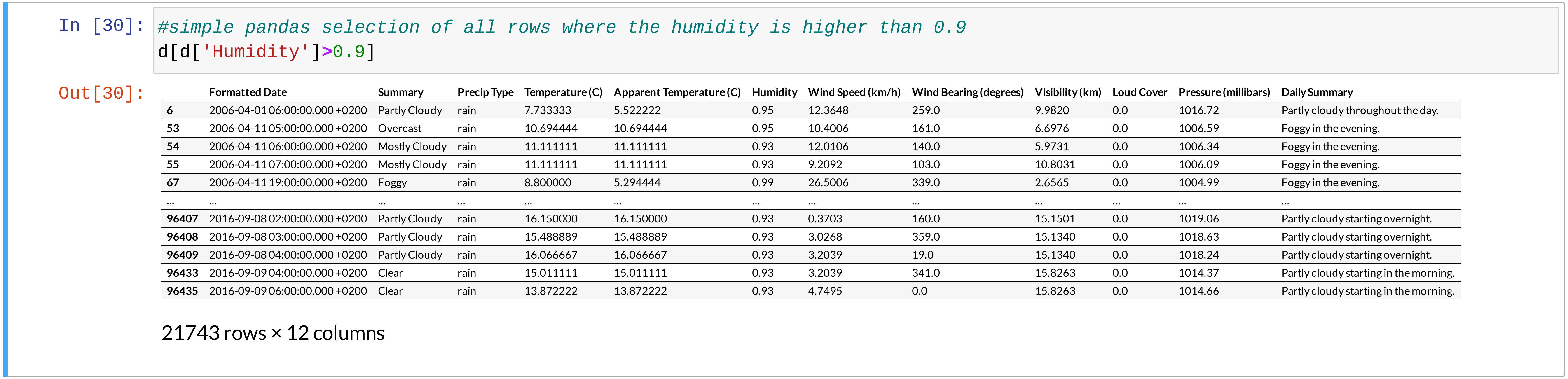
```
In [27]: import pandas as pd
```

```
In [28]: #Reading CSV file
         d=pd.read_csv('../../DATA/weather.csv')
```

```
In [29]: d.head()
```

Out[29]:

	Formatted Date	Summary	Precip Type	Temperature (C)	Apparent Temperature (C)	Humidity	Wind Speed (km/h)	Wind Bearing (degrees)	Visibility (km)	Loud Cover	Pressure (millibars)	Daily Summary
0	2006-04-01 00:00:00.000 +0200	Partly Cloudy	rain	9.472222	7.388889	0.89	14.1197	251.0	15.8263	0.0	1015.13	Partly cloudy throughout the day.
1	2006-04-01 01:00:00.000 +0200	Partly Cloudy	rain	9.355556	7.227778	0.86	14.2646	259.0	15.8263	0.0	1015.63	Partly cloudy throughout the day.
2	2006-04-01 02:00:00.000 +0200	Mostly Cloudy	rain	9.377778	9.377778	0.89	3.9284	204.0	14.9569	0.0	1015.94	Partly cloudy throughout the day.
3	2006-04-01 03:00:00.000 +0200	Partly Cloudy	rain	8.288889	5.944444	0.83	14.1036	269.0	15.8263	0.0	1016.41	Partly cloudy throughout the day.
4	2006-04-01 04:00:00.000 +0200	Mostly Cloudy	rain	8.755556	6.977778	0.83	11.0446	259.0	15.8263	0.0	1016.51	Partly cloudy throughout the day.



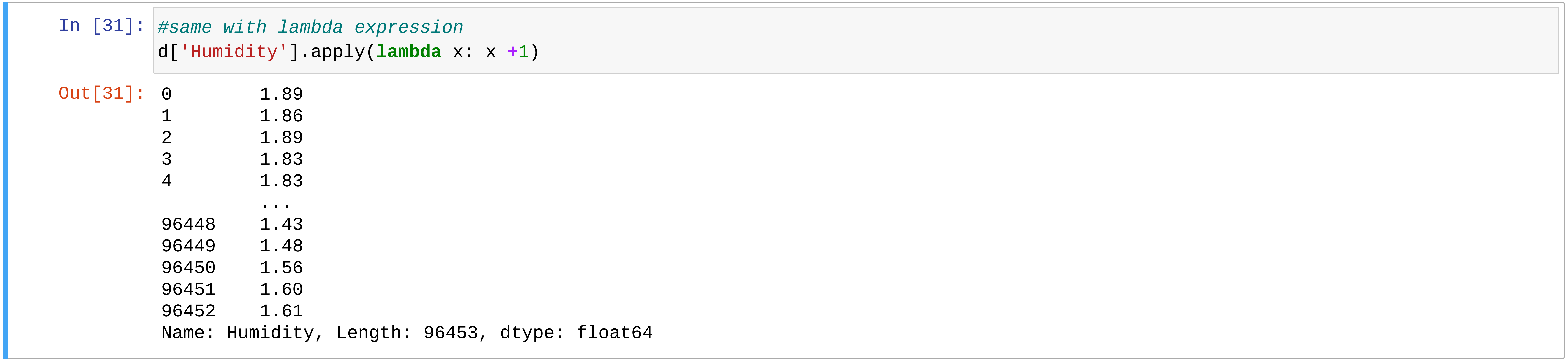
In [30]:

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

Out[30]:

	Formatted Date	Summary	Precip Type	Temperature (C)	Apparent Temperature (C)	Humidity	Wind Speed (km/h)	Wind Bearing (degrees)	Visibility (km)	Loud Cover	Pressure (millibars)	Daily Summary
6	2006-04-01 06:00:00.000 +0200	Partly Cloudy	rain	7.733333	5.522222	0.95	12.3648	259.0	9.9820	0.0	1016.72	Partly cloudy throughout the day.
53	2006-04-11 05:00:00.000 +0200	Overcast	rain	10.694444	10.694444	0.95	10.4006	161.0	6.6976	0.0	1006.59	Foggy in the evening.
54	2006-04-11 06:00:00.000 +0200	Mostly Cloudy	rain	11.111111	11.111111	0.93	12.0106	140.0	5.9731	0.0	1006.34	Foggy in the evening.
55	2006-04-11 07:00:00.000 +0200	Mostly Cloudy	rain	11.111111	11.111111	0.93	9.2092	103.0	10.8031	0.0	1006.09	Foggy in the evening.
67	2006-04-11 19:00:00.000 +0200	Foggy	rain	8.800000	5.294444	0.99	26.5006	339.0	2.6565	0.0	1004.99	Foggy in the evening.
...	...	...	...	...	...	...	...	...	...	...	...	...
96407	2016-09-08 02:00:00.000 +0200	Partly Cloudy	rain	16.150000	16.150000	0.93	0.3703	160.0	15.1501	0.0	1019.06	Partly cloudy starting overnight.
96408	2016-09-08 03:00:00.000 +0200	Partly Cloudy	rain	15.488889	15.488889	0.93	3.0268	359.0	15.1340	0.0	1018.63	Partly cloudy starting overnight.
96409	2016-09-08 04:00:00.000 +0200	Partly Cloudy	rain	16.066667	16.066667	0.93	3.2039	19.0	15.1340	0.0	1018.24	Partly cloudy starting overnight.
96433	2016-09-09 04:00:00.000 +0200	Clear	rain	15.011111	15.011111	0.93	3.2039	341.0	15.8263	0.0	1014.37	Partly cloudy starting in the morning.
96435	2016-09-09 06:00:00.000 +0200	Clear	rain	13.872222	13.872222	0.93	4.7495	0.0	15.8263	0.0	1014.66	Partly cloudy starting in the morning.

21743 rows × 12 columns

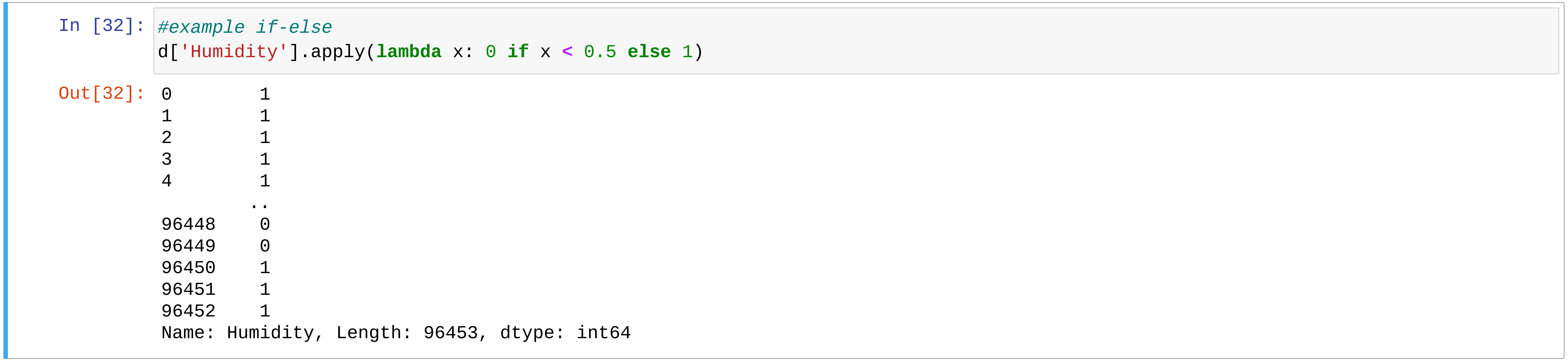


In [31]:

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

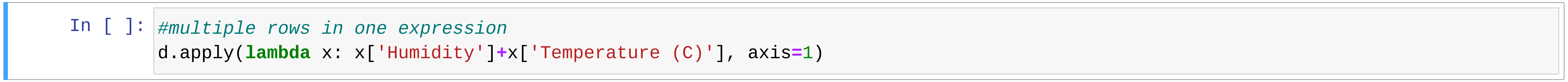
Out[31]:

```
0        1.89
1        1.86
2        1.89
3        1.83
4        1.83
         ... 
96448    1.43
96449    1.48
96450    1.56
96451    1.60
96452    1.61
Name: Humidity, Length: 96453, dtype: float64
```

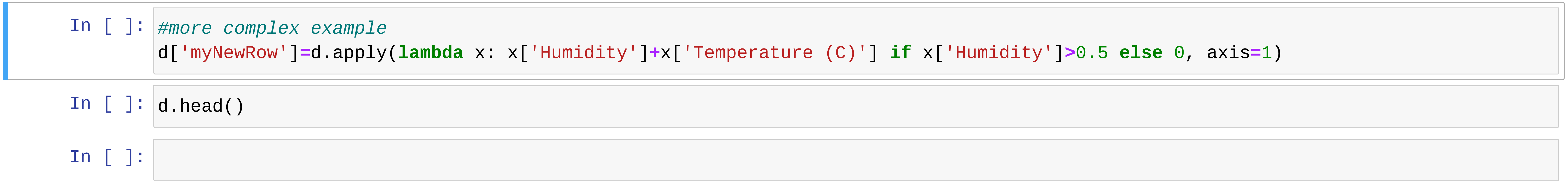


```
In [32]: #example if-else
         d['Humidity'].apply(lambda x: 0 if x < 0.5 else 1)
```

```
Out[32]: 0        1
         1        1
         2        1
         3        1
         4        1
                 ..
         96448    0
         96449    0
         96450    1
         96451    1
         96452    1
         Name: Humidity, Length: 96453, dtype: int64
```



```
#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 [ ]:
```