

Worksheet 00

Name: Zihan Li UID: U83682995

Topics

- course overview
- python review

Course Overview

a) Why are you taking this course?

for interest

b) What are your academic and professional goals for this semester?

Learn things

c) Do you have previous Data Science experience? If so, please expand.

NO

d) Data Science is a combination of programming, math (linear algebra and calculus), and statistics. Which of these three do you struggle with the most (you may pick more than one)?

linear algebra

Python review

Lambda functions

Python supports the creation of anonymous functions (i.e. functions that are not bound to a name) at runtime, using a construct called `lambda`. Instead of writing a named function as such:

In [1]:

```
def f(x):  
    return x**2  
f(8)
```

Out[1]:

64

One can write an anonymous function as such:

In [2]:

```
(lambda x: x**2)(8)
```

Out[2]:

64

A `lambda` function can take multiple arguments:

In [3]:

```
In [3]:
```

```
(lambda x, y : x + y) (2, 3)
```

```
Out[3]:
```

```
5
```

The arguments can be `lambda` functions themselves:

```
In [4]:
```

```
(lambda x : x(3)) (lambda y: 2 + y)
```

```
Out[4]:
```

```
5
```

a) write a `lambda` function that takes three arguments `x, y, z` and returns `True` only if `x < y < z`.

```
In [3]:
```

```
(lambda x, y, z: (x<y) and (y<z)) (3, 4, 5)
```

```
Out[3]:
```

```
True
```

b) write a `lambda` function that takes a parameter `n` and returns a lambda function that will multiply any input it receives by `n`. For example, if we called this function `g`, then `g(n) (2) = 2n`

```
In [4]:
```

```
(lambda n: (lambda x: n*x)) (2) (4)
```

```
Out[4]:
```

```
8
```

Map

```
map(func, s)
```

`func` is a function and `s` is a sequence (e.g., a list).

`map()` returns an object that will apply function `func` to each of the elements of `s`.

For example if you want to multiply every element in a list by 2 you can write the following:

```
In [5]:
```

```
mylist = [1, 2, 3, 4, 5]  
mylist_mul_by_2 = map(lambda x : 2 * x, mylist)  
print(list(mylist_mul_by_2))
```

```
[2, 4, 6, 8, 10]
```

`map` can also be applied to more than one list as long as they are the same size:

```
In [9]:
```

```
a = [1, 2, 3, 4, 5]  
b = [5, 4, 3, 2, 1]  
  
a_plus_b = map(lambda x, y: x + y, a, b)  
list(a_plus_b)
```

```
Out[9]:
```

```
[6, 6, 6, 6, 6]
```

c) write a map that checks if elements are greater than zero

In [5]:

```
c = [-2, -1, 0, 1, 2]
gt_zero = map(lambda x:x>0,c)
list(gt_zero)
```

Out[5]:

```
[False, False, False, True, True]
```

d) write a map that checks if elements are multiples of 3

In [6]:

```
d = [1, 3, 6, 11, 2]
mul_of3 = map(lambda x:x%3==0,d)
list(mul_of3)
```

Out[6]:

```
[False, True, True, False, False]
```

Filter

`filter(function, list)` returns a new list containing all the elements of `list` for which `function()` evaluates to `True`.

e) write a filter that will only return even numbers in the list

In [7]:

```
e = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
evens = filter(lambda x:x%2==0, e)
list(evens)
```

Out[7]:

```
[2, 4, 6, 8, 10]
```

Reduce

`reduce(function, sequence[, initial])` returns the result of sequentially applying the function to the sequence (starting at an initial state). You can think of reduce as consuming the sequence via the function.

For example, let's say we want to add all elements in a list. We could write the following:

In [13]:

```
from functools import reduce

nums = [1, 2, 3, 4, 5]
sum_nums = reduce(lambda acc, x : acc + x, nums, 0)
print(sum_nums)
```

15

Let's walk through the steps of `reduce` above:

1) the value of `acc` is set to 0 (our initial value) 2) Apply the lambda function on `acc` and the first element of the list: `acc = acc + 1 = 1 3) acc = acc + 2 = 3 4) acc = acc + 3 = 6 5) acc = acc + 4 = 10 6) acc = acc + 5 = 15 7) return acc`

`acc` is short for `accumulator`.

f) *challenging Using `reduce` write a function that returns the factorial of a number. (recall: $N! (N \text{ factorial}) = N (N - 1) (N - 2) \dots 2 * 1$)

In [12]:

```
from functools import reduce
factorial = lambda x : reduce(lambda fac,x:fac*x,[i for i in range(2,x+1)],1)
factorial(10)
```

Out[12]:

3628800

g) *challenging Using `reduce` and `filter`, write a function that returns all the primes below a certain number

In [13]:

```
sieve = lambda x: reduce(
    lambda fac, num: fac + [num] if not reduce(lambda fg,n:fg or num%n==0, fac, False) else fac,
    range(2, x + 1),
    []
)
print(sieve(100))
```

[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]

What is going on?

For each of the following code snippets, explain why the result may be unexpected and why the output is what it is:

In [1]:

```
class Bank:
    def __init__(self, balance):
        self.balance = balance

    def is_overdrawn(self):
        return self.balance < 0

myBank = Bank(100)
if myBank.is_overdrawn :
    print("OVERDRAWN")
else:
    print("ALL GOOD")
```

OVERDRAWN

Because there are no "()" after the function `is_overdrawn`, so the function is not called, so it always return true.

In [2]:

```
for i in range(4):
    print(i)
    i = 10
```

0
1
2
3

Because `i = 10` is after print line, so 10 will not be printed.

In [4]:

```
row = [""] * 3 # row i['', '', '']
board = [row] * 3
print(board) # [['', '', ''], ['', '', ''], ['', '', '']]
board[0][0] = "X"
print(board)

[['', '', ''], ['', '', ''], ['', '', '']]
[['X', '', ''], ['X', '', ''], ['X', '', '']]
```

Initializing the board with a deepcopy of the rows will rectify the issue. Because each of the three rows on the board refers to the same list, altering this will impact all the cells in the first column.

In [5]:

```
funcs = []
results = []
for x in range(3):
    def some_func():
        return x
    funcs.append(some_func)
    results.append(some_func()) # note the function call here

funcs_results = [func() for func in funcs]
print(results) # [0,1,2]
print(funcs_results)

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

The list 'results' holds the outcomes of executing the function directly within the loop, so it is [0, 1, 2]. The 'funcs_results' list includes outcomes from calling the same function at the loop's conclusion, resulting in [2, 2, 2].

In [15]:

```
f = open("./data.txt", "w+")
f.write("1,2,3,4,5")
f.close()

nums = []
with open("./data.txt", "w+") as f:
    lines = f.readlines()
    for line in lines:
        nums += [int(x) for x in line.split(",")]

print(sum(nums))
```

0

f is not closed after open it, so it can not read anything.