Worksheet 00

Name: Jeong Yong Yang UID: U95912941

Topics

- course overview
- python review

Course Overview

a) Why are you taking this course?

I am taking this course because I am interested in data science as I also applied for Master's degree. Through taking this course, I want to learn about various data science methods as discussed in the first lecture today.

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

My academic goal is to keep up my good grades and learn more about data science. I learned about some topics that were introduced by professor today, such as the SVD, regression, etc. but want to study more about them in depth. My professional goal is to either get into a graduate school which I applied for Data science or be accepted a job in the United States.

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

Yes, I have Data Science experience at Boston University. I took machine learning and NLP courses in BU. Outside of school, however, I do not have a professional experience.

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)?

I struggle the most in statistics.

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:

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

```
64
```

One can write an anonymous function as such:

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

```
64
```

A `lambda` function can take multiple arguments:

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

```
5
```

The arguments can be `lambda` functions themselves:

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

```
5
```

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

```
1 (lambda x, y, z: x < y and y < z) (1,2,3)
```

```
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`

```
1 (lambda n: (lambda inputReceived: inputReceived * n))(3)(2)
```

```
6
```

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:

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

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

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

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

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

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

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

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

```
[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

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

```
[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:

```
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 factorial) = N * (N - 1) * (N - 2) * ... * 2 * 1)

```
factorial = lambda x : reduce(lambda y, z: y * z, range(1, x + 1), 1)
factorial(10)
```

```
3628800
```

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

```
sieve = lambda x : reduce(lambda y, z: y + [z] if all(z % n != 0 for n in y) else y, filter(lambda a: a >= 2, range(x)), [])
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:

```
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
```

Here, the result may be unexpected because if we check the is_overdrawn function in class Bank, the input 100 is greater than 0, so people might think that it should have printed "ALL GOOD". However, the function is not being called here since there is no parenthesis. It should have been "is_overdrawn()", but it wrote "is_overdrawn" without the parenthesis which always gives True.

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

```
0
1
2
3
```

Here, the result may be unexpected because i changes to the value 10 after being printed. However, the for loop reinitiates the value of i to each 0, 1, 2, 3 when the loop restarts so 10 never gets printed.

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

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

Here, the result may be unexpected since if the [0][0] becomes "X", it might allow people to think that the first element is only "X" giving [['X', '', ''], ['', '', ''], ['', '', '']]. However, this does not occur because all the sublists ['', '', ''] are the same object and therefore changing one changes everything if they are in the same memory, giving [['X', '', ''], ['X', '', ''], ['X', '', '']].

```
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]
```

Here, the result may be unexpected since if we append x for values 0, 1, 2, funcs_results should be [0,1,2]. However, this does not occur because funcs appends some_func instead of some_func() without the parenthesis, meaning that it does not call the function some_func() but gets the 'x' variable itself. After the for loop ends, x is set the value of 2, giving [x,x,x] to [2,2,2].

```
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
```

Here, the result may be unexpected because if the user wrote 1,2,3,4,5, giving the summed value to be 15. However, this does not occur because when the code reopens the file, it is in write form with the code "w+", giving an empty file without any contents, giving the sum to be 0 since the file is empty.