Worksheet 00

Name: Mohit Sai Gutha UID: U48519832

Topics

- course overview
- python review

Course Overview

a) Why are you taking this course?

To understand how to learn from data and also to represent data in a way as to make it easier for others to easily understand and learn from it.

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

Academic Goals : Get only A's A-'s in all the courses I have registered for. Professional Goals : Build skill set in some new technologies (ML, Data Engineering and Analysis). Practice competitive coding and hopefully land a summer internship.

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

A small component of computer networking project required me to make a presentation of the project data with some cool visualisations.

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 with programming.

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]:

```
(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 [1]:

```
(lambda x,y,z: x<y<z)(3,8,5)
```

Out[1]:

```
False
```

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 [2]:

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

Out[2]:

```
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 [3]:

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

Out[3]:

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

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

In [4]:

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

Out[4]:

```
[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 [5]:

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

Out[5]:

```
[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 [6]:

```
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)

In [7]:

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

Out[7]:

```
3628800
```

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

In [8]:

```
sieve = lambda x : reduce(lambda y, z: y - set(range(z*z, x, z)) if z in y else y, range
(2, int(x**0.5) + 1), set(range(2, 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, 8
9, 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 [9]:

```
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
```

The result is unexpected because our expetation for the code snippet is to return "ALL GOOD" considering the balance is greater than 0. This is not the case and the output is "OVERDRAWN" because the line "if myBank.is_overdrawn" is missing parenthesis after the function call and is therefore only calling a reference to the function and not the function itself.

In [10]:

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

```
0
1
2
3
```

Though i is assigned to 10 inside the loop, the for loop only follows the sequence generated by the range function at the start of the loop and that statement doesn't affect the loop variable.

In [11]:

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

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

board[0][0] = "X" initializes first elements of all the three rows to 'X' because all three rows of the board are references to the same list 'row'.

In [12]:

```
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]
```

At the end of the for loop, the loop variable 'x' retains the value '2'. Since all func in funcs reference the same variable 'x', they print it's value as all 2's when we print the func_results.

In [13]:

```
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
```

The 'w+' mode truncates the file and it's empty. Therefore, the sum is just 0.