Worksheet 00

Name: Ryan Chung
UID: U33101958

Topics

- course overview
- python review

Course Overview

a) Why are you taking this course?

I want to learn cool data visualization techniques through this course

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

My goals are to constantly ask questions, help others, and learn to the best of my ability

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

I have no experience previously

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)?

Probably 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:

```
(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`.

```
(lambda x, y, z: x < 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`

```
(lambda n: (lambda x: n*x))(2)(3)
```

```
    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:

```
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:

```
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)
```

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

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

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

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

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

```
d = [1, 3, 6, 11, 2]
mul_of3 = map(lambda x: x % 3 == 0, d)
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 acc, y: acc * y, 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 acc, num: acc + [num] if all(num % p != 0 for p in acc) else acc, 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, 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
```

It is missing the parantheses to call the function is_overdrawn. It should be if myBank.is_overdrawn(): in order to call the method for the conditional

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

```
0
1
2
3
```

The reassignment of i in the for-loop is out of the scope and does not affect the initial for i in range(4)

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

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

[['first', '', ''], ['second', '', ''], ['third', '', '']] The pointers when creating the 2D matrix point to the first list. Therefore, when editing board[0][0] it is reflected to the duplicated rows that have "second" and "third"

```
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]
```

When defining some_func without any parameters, it references x from the outside scope. Therefore, by calling some_func() and appending the result to results, it uses the current value of x, which is 2 at the end of the loop

```
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 second open line ("with open("./data.txt", "w+") as f:") should read not write. By opening in write, it erases the previous content in the data.txt file