worksheet_00

September 7, 2023

1 Worksheet 00

Name: Alexander Miller BUID: U52161825

1.0.1 Topics

- course overview
- python review

1.0.2 Course Overview

a) Why are you taking this course?

I am taking this course because I love coding, and I heard that this class is mostly coding-based. I am excited to hone my programming skills and learn about data science.

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

My academic goal for this semester is to get straight A's. I have been close in previous semesters but never have had a 4.0. My professional goal for this semester is to get an internship for the summer. I have been applying to many internships, and I hope to get one this summer.

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

Yes, I have taken a couple data science courses at BU and have some experience with various data science topics.

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 linear algebra. I lacked a fundamental understanding when I took the class and have struggled in previous classes which have required linear algebra, so I have made an attempt to improve my understanding of linear algebra for this semester and also hope that this course is an oppotunity to do so.

The rest of this worksheet is optional. If you have prior Python experience, you are welcome to skip it HOWEVER I strongly encourage you to try out the questions marked as `challenging`.

1.0.3 Python review (Optional)

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:

[10]:
```
def f(x):
    return x**2
f(8)
```

[10]: 64

One can write an anonymous function as such:

[11]:
```
(lambda x: x**2)(8)
```

[11]: 64

A `lambda` function can take multiple arguments:

[12]:
```
(lambda x, y : x + y)(2, 3)
```

[12]: 5

The arguments can be `lambda` functions themselves:

[28]:
```
(lambda x : x(3))(lambda y: 2 + y)
```

[28]: 5

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

[2]:
```
(lambda x, y, z : (x < y and y < z))
```

[2]: `<function __main__.<lambda>(x, y, z)>`

b) write a `lambda` function that takes a parameter `n` and returns a lambda function that will multiply any input it receives by `n`.

[29]:
```
(lambda n : (lambda x : x*n))
```

[29]: `<function __main__.<lambda>(n)>`

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:

[30]:
```
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:

[31]:
```
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)
```

[31]:
```
[6, 6, 6, 6, 6]
```

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

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

[3]:
```
[False, False, False, True, True]
```

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

[5]:
```
d = [1, 3, 6, 11, 2]
mul_of3 = map(lambda x: x % 3 == 0, d)
list(mul_of3)
```

[5]:
```
[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

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

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

Reduce `reduce(function, sequence[, initial])` returns the result of sequencially 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:

[9]:
```
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)

[14]:
```
factorial = lambda x : reduce(lambda acc, y: acc*y, range(1, x+1), x)
factorial(10)
```

[14]:
```
36288000
```

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

[20]:
```
sieve = lambda x : reduce(lambda acc, y: acc + [y], filter((lambda z: all(z % i
 ↪!= 0 for i in range(2, int(z ** 0.5) + 1))), 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]
```

1.0.4 What is going on?

This whole section is `*challenging`

For each of the following code snippets, explain why the output is what it is:

[26]:
```
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 issue with this code snippet is that you forgot to put parenthsis after the call to .is_overdrawn in line 9. To use the code as it seems to be intended, you would need to write it as such: if myBank.is_overdrawn():

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

```
0
1
2
3
```

Even though i is set to 10 at the end of each iteration of the loop, at the beginning of each iteration i is then set to the next value in the seuquence created by range(4), so it will still only print the values of the sequence, and never 10.

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

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

This is because all three "rows" of the board reference the same list. Although it may seem like line 2 of this code produces 3 new lists it actually produces three references to the same list of variable name "row". Then, when the the first element of the first row in the board is changed, it actually changes all the rows because all the rows reference the same list.

[ ]:
```
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]
```

In each iteration of the for loop, the definition of some_func is changed. When some_func is stored in the funcs list, a reference to the some_func function is what is actually store. Therefore, the reason all of the funcs_restuls are 2 is because all that's stored in the funcs list is three of the same reference to the some_func function which returns 2 as was set by the last iteration of the for loop

[32]:
```
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))
```

```
15
```

The file is opened in write mode when it is opened the second time, so when the code goes to read it, there is nothing to read, the nums array remains empty, and finally the nums arrays is summed to 0.

[ ]:
```

```