Worksheet 00

Name: Rithvik Nakirikanti UID: U57718462

Topics

- course overview
- python review

Course Overview

a) Why are you taking this course?

I'm taking this course, so I can get an introduction to what Data Science even is. I've never taken a Data Science course ever, so I'm looking to get introduced to the concept.

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

Academically, I'm looking to get a good grade in this course. But besides that, I do really want to learn all the concepts that this course is aiming to teach. Getting a good grade is one thing, but I also deeply care about learning something and applying it to the real world and other projects outside of the course.

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

I don't have any prior experience, this is my first Data Science class.

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 would say statistics since I haven't really taken very many statistics classes yet. I'm very interested to learn though, so I have a very open mind.

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 [13]:
```
(lambda x : x(3))(lambda y: 2 + y)
```

Out[13]:
```
5
```

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

In [14]:
```
(lambda x, y, z : x < y < z)(2,5,7)
```

Out[14]:
```
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 [16]:
```
(lambda n : lambda y : y * n)(2)(5)
```

Out[16]:
```
10
```

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 [3]:
```
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[3]:
```
[6, 6, 6, 6, 6]
```

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

In [17]:
```
c = [-2, -1, 0, 1, 2]
gt_zero = map(lambda x : x > 0, c)
```

```
list(gt_zero)
```

Out[17]:
```
[False, False, False, True, True]
```

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

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

Out[2]:
```
[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 [18]:
```
e = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
evens = filter(lambda x : x % 2 == 0, e)
list(evens)
```

Out[18]:
```
[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 [19]:
```
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 [24]:
```
factorial = lambda x : reduce(lambda a, b: a * b, range(1, x + 1), 1)
factorial(3)
```

Out[24]:
```
6
```

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

certain number

In [25]:
```
sieve = lambda x : reduce(lambda acc, cur: acc + [cur] if all(cur % prime != 0 for prime
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 [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
```

When the method is being invoked, you need to make sure to add parantheses (open and close) after the method name. So, it should be myBank.is_overdrawn(). Once you add the parantheses, it'll call the method properly and use its results to print overdrawn or ALL GOOD correctly.

In [12]:
```
for i in range(4):
    print(i)
    i = 10
```

```
0
1
2
3
```

With this for loop, when the first line is called and the range() function is called, the python documentation says that this returns a sequence of numbers which starts by 0 and increments by 1 by default. So it can be thought of as of a list containing [0,1,2,3]. After every iteration of this loop, the next value is assigned to i based on this list. Therefore, even though i = 10 is set each time the for loop is run, it gets overwritten when i is changed to the next number in the list.

In [4]:
```
row = [""] * 3 # row i['', '', '']
board = [row] * 3
print(board) # [['', '', ''], ['', '', ''], ['', '', '']]
board[0][0] = "X"
print(board)
```

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

The issue is with the way board is initialized. Since the board is created using [row * 3] what you're effectively doing is creating a list but with three references to the original row list that was made in the first line of code. Therefore, even though we only need to change (0,0) which is only the leftmost space, the 'X' gets updated three times because they are still referencing the same row list.

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]
```

With this code, the expected behavior would be that func_results and results would print the same list [0,1,2] ideally. But unfortunately this is not the case since inside the for loop, funcs.append(some_func) is not storing the return value of the function itself but rather the function itself is appended to the list with a reference to the variable x. The return value is not stored inside funcs initially because the method was not called (by using () after). For the results.append(...) statement though, the function is being called properly and the values [0,1,2]. After the for loop, then each function that was appended is being run and being stored in funcs_results. However, since these functions are using a reference to the variable x which is now equal to 2, the whole list ends up storing [2,2,2].

In [20]:
```
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 main issue with this is that the txt file was opened using w+. w+ gives you read-write privileges but it also does file truncation. So, at the end of the code when we went to read the file, it first basically truncated all of the data so it was just basically reading nothing. That's why it printed out zero instead of 15.