Worksheet 00

Name: Daniyal Ahmed UID: U11469883

Topics

- course overview
- python review

Course Overview

a) Why are you taking this course?

I am taking this course because I am very interested in Data Science

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

To Succeed in this class and all my other class. My professional goal is to get an internship

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

I have very little Data Science experience, my experience in Data Science revolves around natural language processing and training LLM Models

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 believe calculas is the one I am least familiar with (mostly because I took calculas freshman year and now I am a junior), Though I do have a lot of trouble knowing when to apply 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:

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 [2]:
```
(lambda x,y,z: x<y<z)(2,3,4)

#This is pretty self explanatory, we simply have three varaibles and are check
ing if the boolean condition x<y<z is true or false.
```

Out[2]: 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 [ ]:
```
(lambda n: lambda y:(n*y))(2)(3)

#The way this works is by having a lambda n which we use in our second lambda
along with y to multiply the two numbers
```

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 [ ]:
```
c = [-2, -1, 0, 1, 2]
gt_zero = map(lambda x: x>0, c)
'''The way this works if by adding the boolean value of x>0 into gt_zero , whe
re x takes on every value in c '''


list(gt_zero)
```

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

In [1]:
```
d = [1, 3, 6, 11, 2]
mul_of3 = map(lambda x: x%3==0, d)
'''The way this works if by adding the boolean value of x%3 into mul_of3 , whe
re x takes on every value in d '''


list(mul_of3)
```

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

'''The way this works is by checking the boolean condition to see if the numbe
r is even, if it is true we add that number to the list'''


list(evens)
```

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 [4]:
```
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 [ ]:
```
factorial = lambda x :  reduce(lambda acc,y: acc*y, [i for i in range(1,x)] )
'''The way this works is by creating two values one is acc and the other is y
what we are doing is creating a list of values ranging from one to x and y tak
es on each value in the list and
multiplies it by accumalator'''


factorial(10)
```

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

In [7]:
```
sieve = lambda x: filter(lambda y:(reduce(lambda acc,z : acc and y % z != 0,
[i for i in range(2,y)], True)), [j for j in range(2,x)])

'''The way this function works is that we have three parameters for filter, th
e first one being the function of w and a reduce function inside the filter fu
nction
The reduce function uses the acculmuater but by setting the accu to boolean va
lue instead of a number, we can do something very smart, any time in the funct
ion that
the boolean value of accc turns false it will stay false for the entirety of t
he iteration of the function (due to the and operation). Thus if there exists
a number in the range of 2 to y-1
w being the number to check prime, we will always know. finally since we requi
re a list of numbers for filter I have provided a list in filter '''
print(list(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 [10]:
```
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 reason why the program is saying overdrawn is because you are missing the parenthesis in is overdrawn hence, we need the parenthesis to tell python this is a function. What it is doing the version about is evaluating by memory address and in python any number above zero results to true and since the memory address is above 0, it will always result to True.

See corrected version below:

In [ ]:
```
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")
```

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

```
0
1
2
3
```

The problem with this program is that you cannot reassign the value of i when using it in a for loop since the range function will reassign it to automatically, See corrected version below

In [14]:
```
for i in range(4):
    x=10
    print(x)
```

```
10
10
10
10
```

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

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

This is because each item in board points to the same address of row, so thus when you change the value of one point at of the board, you will subsequently change the value in all rows since all rows point to the same address

See Corrected Version Below

In [ ]:
```
row = [""] * 3 # row i['', '', '']
row2 = [""] *3
row3=[""] *3
print(row)
board = [row,row2,row3]
print(board) # [['', '', ''], ['', '', ''], ['', '', '']]
board[0][0] = "X"
print(board)
```

In [ ]:
```
#Alternatively : you could use a list comprehension

board = [ ["","",""]  for i in range (3)]
board[0][0] = "X"
print(board)
```

In [14]:
```
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]
```

Because of the way that python executes code once we execute this line funcs_results = [func() for func in funcs] what python does is go back look at last value that x assumed. Since the for loop stopped, it is like how if you were to run a for loop, such as for i in range(3), and print i after the for loop is done interating you will get 2, since the for loop stopped, all you can set is the last element.

In [15]:
```
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
```

This is because when we are using with open we are using "w+" although you can use the w+ notation to read and write a file in python. However, there is a down side to as w+ clears the file everytime it opens it.

Corrected Version:

In [ ]:
```
f = open("./data.txt", "w+")
f.write("1,2,3,4,5")
f.close()

nums = []
with open("./data.txt", "r") as f:
  lines = f.readlines()
  for line in lines:
    nums += [int(x) for x in line.split(",")]

print(sum(nums))
```