Name: Chance Bowman
Collaborators:

Problem Set 5: Functional & NumPy

ECE 2140
November 22, 2023

Instructions

- This assignment must be turned in electronically, via Canvas, by the deadline. Once the Canvas grace period ends, there will be no opportunity to submit the assignment.
- **Submit your answers to each question in a file called ps5.pdf.** If you complete the **bonus** question at the end, submit this in a file called **bonus.py**
- You must produce all answers for this assignment by yourself. You may freely consult with TAs or the instructor. You **may not copy code from the internet** without the instructor's consent, and if you do you must cite it appropriately.
- You may consult with other students in the course about this assignment (except for the bonus question). If you do, *you must list all of their names on the first page.* When discussing the assignment with others, *keep all discussions about specific problems verbal. Do not write anything or make any permanent notes that are specific to the assignment problems.*

Functional Programming

For all of the functional programming questions, *you are not allowed to use for loops, while loops or if statements.*

1. [*5 points*] Fill in the following function, called `compose`, that takes 2, 1-argument functions as arguments, `f1` and `f2`. Compose returns *a new function* which takes a single argument and, when called, returns the result of calling `f2` on the result of calling `f1` on its argument.

```
def compose(f1, f2):
    #TODO fill in this function body
    return lambda arg: f2(f1(arg))
```

2. [*5 points*] Fill in the following function, called `best_answer` which takes one number as an argument and then an arbitrary number of 1-argument functions as its next argument. `best_answer` returns the *largest* value that results from calling any of the 1-argument functions on its first argument.

   For example: `best_answer(-1, square, add_one, cube)` returns 1, which is the result of `square(-1)` and is larger than the results of `cube(-1)` and `add_one(-1)`.

```
from functools import reduce
def best_answer(x, *fxns):
    #TODO fill in this function body
    def is_greater(greater_num, current_v):
        return max(greater_num, current_v)

    results = map(lambda fxn: fxn(x), fxns)
    get_real_num = filter(None, results)
    return reduce(is_greater, get_real_num, float('-inf'))
```

3. [*10 points*] Write a function, `s_contains` that takes in a set, `s`, and returns a new function. The return value is a function that takes an arbitrary number of arguments and returns `True` when *all* of the arguments are in the set `s` and `False` otherwise.

```
def s_contains(s):
    #TODO fill in this function body
    def check_if_in(*args):
        booleans = map(lambda arg: arg in s, args)
        return False not in booleans
    return check_if_in
```

Reading Functional Programs

4. [*10 points*] Describe, in your own words, what these expressions each evaluate to:

- Assume x is a string.

```
functools.reduce(lambda d, k: d[k] = d.get(k, 0) + 1,
        filter(lambda s: s.isdigit(), x), {})
```

It starts in the filter function, filtering out all values of the string 'x' that are digits, and uses them moving forward. For the next part, the program moves to the first argument of the reduce function. It checks whether each number (key) is in the dictionary or not, and creates a count (value) for each. Afterward, the entire reduce function is ran, creating a single set from all of the key-value pairs.

- Assume x is a dictionary.

```
list(map(lambda t: (t[1], t[0]),
        sorted(x.items(), key=lambda t: t[1])))
```

It starts at the second argument of the map function. It takes the items of the 'x' dictionary as a list of key-value pairs, which are then sorted by their second elements (of keys, that is--not values). Next, the map function takes this sorted list and swaps the first and second elements (thus, switching it to (value, key)). Afterward, the function takes all of those tuples and makes a list from them.

- Assume x is a list of lists.

```
functools.reduce(lambda l1, l2: l1 + l2, x, [])
```

This, simply, concatenates the lists in the list of lists ('x') into a single list. So, to get rid of any confusion, it does this: [[1,2,3],[4,5,6],[7,8,9]] --> [1,2,3,4,5,6,7,8,9]. l1=[], l2=x.

- Assume x is a list, and n and m are ints.

```
list(map(lambda t: t[1],
    filter(lambda x: x[0] >= n and x[0] < m, enumerate(x))))
```

It starts with the enumeration of x, creating tuples of the type (index, element). Afterward, it filters through those tuples, on the basis of the first index value. If the index value of a tuple is greater than or equal to n, but less than m, it continues forward. If else, it is filtered out. Next, the map function takes the second index element of each of the filtered tuples and creates a list from them.

- Assume x is a list and n is an int.

```
map(lambda t: t[1],
    sorted(map(lambda t: ((t[0] + n % len(x)), t[1]), x),
        key=lambda t: t[0]))
```

It starts at the first argument of the nested map function. It takes the list, 'x', and creates tuples from it, assigning each of the current list values to the second index of each tuple. The first tuple is created through a modification--each list value is added to 'n', then divided by the length of the list, then the remainder is taken of each, and each remainder is assigned the first index in the tuple. From there, the tuples are sorted, based on the value of the first index. Afterward, the value of the second index is taken from each tuple, and a list is created from them.

Numerical Python

You may use loops and if statements for this section, yay! you *must also use the numpy library* to answer these questions.

5. [*5 points*] Write a function `mk_matrix` that takes two arguments, n and m. It returns a 2D ndarray with the values from 0 to $n * m - 1$. The first row should contain the values 0 to $m - 1$, the second row should contain the values m to $2m - 1$, etc.

```
def mk_matrix(n, m):
    #TODO fill in this function body
    return np.arrange(0, n*m).reshape(n,m)
```

6. [*5 points*] Write a function that solves the following linear equation for x: $Ax = b$ (where x and b are 1 dimensional ndarrays and A is a square matrix represented as a 2D ndarray).

   Hint: Use the Numpy Linalg Inverse function to find the inverse of a matrix, if it exists.

```
def solv_lin(A, b):
    #TODO fill in this function body to return x
    determinant_A = np.linalg.det(A)
    A_inverse - np.linalg.inv(A)
    x = np.dot(A_inverse, b)
    return x

    A = np.array([[t,u], [v,w]])
    b = np.array([y,z])
```

This would work for both, right?

7. [*10 points*] Using the function defined in the previous part, write a program that solves the following system of equations:

$$x_1 + 3x_2 + x_3 = 2$$
$$3x_1 - 4x_2 + x_3 = 0$$
$$2x_2 - x_3 = 1 = 1$$
$$2x_1 + x_4 = 12$$

Typo?

Program:

```
import numpy as np

def solv_lin(A.b):
    determinant_A = np.linalg.det(A)
    A_inverse - np.linalg.inv(A)
    x = np.dot(A_inverse, b)
    return x

A = np.array([[1,3,1,0], [3,-4,1,0], [0,2,-1,0], [2,0,0,1]])
b = np.array([2,0,1,12])

answer = solv_lin(A,b)
print(answer)
```

The code begins with an interactive application, asking the user for the number of rows and columns in matrix A. This initializes the size of matrix A, as well as vector b. Next, each value in that matrix and vector is initialized to a value of zero, as a matter of standard convention. It then asks the user for the values to each point in the matrix (although, tedious to enter). Afterward, it asks the user for the values to each point in the vector. Next are the calculations. The solve_lin function takes the determinant of matrix A, and later takes the inverse of that value. Mathematically, that would allow the user to find the x-value, by multiplying that inverse by the vector b. And, that's exactly what's done (I hope). The x-values are then returned to the user for whatever use necessary.

Solution: $x_1 =$ 0.64705882 $x_2 =$ 0.47058824 $x_3 =$ -0.05882353 $x_4 =$ 10.70588235

8. [*25 points*] **BONUS QUESTION:** Implement the function in Question 6 *without using any of they numpy.linalg functions – you may use other numpy features.* Submit this in a file called **bonus.py**. **This must be completed without the assistance of other students. You must also explain your code as a final answer in your ps5.pdf submission.**

   Partial credit will not be awarded for this question.

The code begins with an interactive application, asking the user for, essentially, the side lengths of matrix A. That allows for the creation of an nxn A matrix, and additionally, for the creation of a ndarray b vector (as there are dimension restrictions for the multiplication). There are certain cases, however, that needed different methods for calculation. Notably, those were matrices of 0x0 and 1x1. In the case of zero, no matrix exists--so, no calculations can be done on the non-existent value. As for the case of one, the matrix consists of a single value--so, no determinant is required to be calculated. The program then skips calculations in the next function. Next, all values inside both the matrix and vector are initialized to zero, as a matter of standard mathematical convention. Afterward-- and although tedious--the program asks the user to input values for the individual indexes of both the matrix and vector. From there, a determinant is calculated, based on the size of the matrix (nxn). To go a little more in depth, the larger matrix is divided into smaller, submatrices, in which the smaller units of the determinant are calculated. It's simply just finding the cross product, with one smaller unit being the result of cross multiplication and the subtraction of those parts, which is then multiplied by that column's header. Those smaller units are then added together (taking my 'sign' variable into account), summing to the value of the determinant. If the value of the determinant is non-zero, the program moves to the next function. Again, taking into account the size of the matrix (nxn), the inverse of matrix A is calculated. To keep it short, certain indexes are switched with others, while others are negated (it gets a little difficult to explain). In the final function, the x-values are calculated. That's done by multiplying the inverse matrix of A with the normal vector of b. At the end, those x-values are printed as a list of values. If, in the first function, the determinant was found to be zero, the print statements, rather say that no value of x exists.