Python Programming

Homework 5

Instructor: Dr. Ionut Cardei

The following problems are from Chapters 15 (Recursion), 16 (Fun Stuff with Python), and from the Numpy module. The problems involve iterators, generators, generator expressions, techniques of functional programming described in the Ch16 lecture notes on the Module 6 Canvas page, and a little bit of numpy array manipulation for graphics.

Additional reading required:

- "Functional Programming HOWTO", at https://docs.python.org/3/howto/functional.htm
- "Python Closures", at https://www.programiz.com/python-programming/closure
- "NumPy Quickstart Tutorial" at https://docs.scipy.org/doc/numpy-dev/user/quickstart.html and
- Module 7 lecture notes, that have relevant examples.

To get full credit the programs must have all the required features and run as expected. Pay attention to details. If in doubt, check the Homework 5 Q&A Discussion Forum or send an email to the instructor.

Write your answers into a new Word document called **h5.doc**.
Write your full name at the top of the file and add a heading before each problem solution.

For each problem:

- write a **subtitle** with the problem number,
- insert the code from the Python file,
- insert any additional items, such as screenshots.

Convert the doc file to PDF.
For this homework you need to upload on Canvas the PDF file and the Python .py source files (p1.py, p2.py,…).

Don't forget to read the **Important Notes** at the end of this file.

Problem 1. Recursive Functions

a) Binary Tree.
Write a function in file **p1.py** with signature *binary_tree(depth, length)* that draws a binary tree figure like that from Figure 1 using the turtle module. The figure produced is similar to the example from the textbook with a significant difference that you will notice immediately: the left branches align on a right(60) angle from the vertical and the right branches all align on a left(60) angle from the vertical.

The binary tree drawing must look like that from Figure 1, except the dots are NOT required:

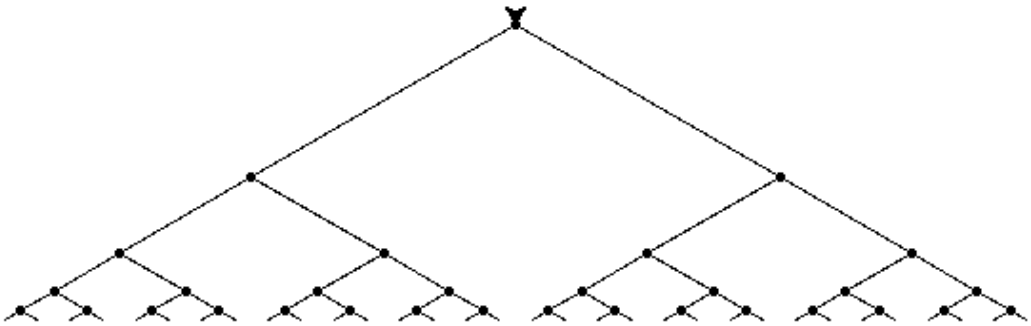

Figure 1. Binary tree produced by calling binary_tree(160, **5**), *with dots shown at starting points for clarification.*

Include a screenshot of the figure drawn by the call **binary_tree(6, 160)**.

To get credit for part a):

- do not change the function signature,
- **do NOT draw any black dots**,
- the screenshot must be of the figure drawn by the call **binary_tree(6, 160)**.

b) Efficient Power.
Consider the *power*(*x*,*n*) function, with *x* any number, and *n* a non-negative integer, that computes x^n. Since we love recursion we may consider this trivial implementation of *power*:

```
def power_linear(x, n):
    if n == 0:
        return 1
    return x * power_linear(x, n-1)
```

The problem with this function is that it has a runtime proportional to *n*; we say its runtime complexity is linear in *n* and we write that as *O*(*n*). E.g. if n == 31, this function does *31* multiplications, in *31* recursive calls.

We quickly realize that we can make *power* work a lot faster by using a smarter divide-and-conquer approach. For example, if n == 28, we can compute power(x, 28) as follows:

power(x, 28) = power(x, 14)^2
power(x, 14) = power(x, 7)^2
power(x, 7) = x * power(x, 3)^2
power(x, 3) = x * power(x, 1)^2
power(x, 1) = x * power(x, 0)
power(x, 0) = 1

To do for part b):

- Write a new version of the *power* function that uses this second approach (in file **p1.py)**. Write the contract for the function in its docstring: what it does and the preconditions.
- Write a function called *test_power*() that uses the *testif* module/function to test the power() function for the base case and some other cases. Compare with ** the operator.

Part c) Memoized Slice Sums.
For this problem, **first** you need to write a recursive function ***slice_sum*(*lst, begin, end*)** that computes recursively the sum *lst*[*begin*] + *lst*[*begin*+1] + *lst*[*begin*+2] + ...+*lst*[*end*-1]. Parameter lst is a list object or a tuple. This function returns exactly the same value as expression *sum*(*lst*[*begin:end*]).

The base case deals with an empty slice and the recursive case adds lst[begin] to the ***slice_sum***() on the remainder of the list.

Your implementation of the *slice_sum*() function must **NOT** use slices (e.g. lst[i:j]) in order to get any credit.
Write the function contract (including preconditions related to valid parameter range) in its docstring. The function must throw *IndexError* if the index parameters are invalid.

Write a unit test function ***test_slice_sum***() that uses *testif*() to test function *slice_sum*() for interesting cases, including the base case.

Second, after the recursive version of ***slice_sum***() function works correctly, write a second version of this function that uses memoization to optimize it. Call the memoized function ***slice_sum_m***. This function caches function values returned for various values of parameters *begin* and *end*..

We can make the **simplifying assumptions** that over the execution time of the program the function ***slice_sum_m*** is called always with the same object referenced by parameter *lst* and that object *never* changes. (Otherwise it would make no sense to cache results!)

Write a unit test function ***test_slice_m***() that uses *testif*() to test function *slice_sum_m*() for interesting cases, including the base case.

So, for part c) write in file p1.py functions ***slice_sum***(), ***test_slice_sum***()**,** ***slice_sum_m***(), and ***test_slice_sum_m***()**.**

Problem 2. Prime Number Generator

Part a). Iterator Class
Write a class called ***PrimeSeq*** that generates the sequence of the first *n* prime numbers, starting with 2, 3, 5,.... The class must comply with the iterator interface and must support the *for* statement.
The *__next__()* method must return the next prime number from the sequence or throw *StopIteration* if *n* primes have already been generated.

This class should work like this:

```
# prints in order the first 100 prime numbers: 2,3,5,...
primeseq = PrimeSeq(100)
```

```
for p in primeseq:
    print(p)

# Create a list with the first 100 prime numbers:
primes_lst = [p for p in PrimeSeq(100)]    # uses the new object to get an iterator

print(primes_lst)
```

To get full credit a *PrimeSeq* object **must use** an instance attribute list *self.__primes* that accumulates all prime numbers computed until the most recent call to *__next__*(). This method must use that list to speed up the computation of the next prime number from the sequence by checking divisibility for a number *p* only with **prime divisors** <= *sqrt*(*p*).

Part b). Generator

Write a **Python generator** called ***prime_gen***(*n*) that takes an integer *n* >= 0 and produces the sequence of the first *n* prime numbers. This generator is defined as a function that uses the ***yield*** keyword to output a value, as seen on the Chapter 16 lecture PDF file, on slides 60-70. This generator produces the same number sequence as the *PrimeSeq* class from part a). Here is how it can be used in a *for* loop to print the first 10 prime numbers:

```
for p in prime_gen(10):
    print(p)
```

Write the code from parts a) and b) in a file p2.py.

Add in this file a function called *main*() that demonstrates both the *PrimeSeq* class and the *prime_gen*() generator by creating and printing lists with the first 100 prime numbers.

Take a screenshot with the output of the *main*() function and insert in the h5.doc file.

Problem 3. Functional Code with Random Number Sequences

Write a generator *gen_rndtup*(*n*) that creates an infinite sequence of tuples (*a*, *b*) where *a* and *b* are random integers, with 0 < *a*,*b* < *n*. If *n* == 7, then *a* and *b* could be the numbers on a pair of dice. Use the *random* module.

a) Write code in file p3.py that uses lambda expressions, the `itertools.islice` function (https://docs.python.org/3/library/itertools.html#itertools.islice), and the *filter* function to display **the first 10** generated tuples (*a*, *b*) from *gen_rndtup*(*7*) that have *a* + *b* >= n // 2.

Example: with n==7 the output could be: (4,1), (2,6), (6,6),(3,5),...

b) Write code in a *main*() function using *generator expressions* and one *for* loop that displays **the first 10** random integer tuples (*a*, *b*), with 0<*a*,*b*<*n*, where *a* + *b* >= *n* // 2 and *n* being a positive integer local

variable initialized with value 7. Do not use the *gen_rndtup*(*n*) generator from part a). You may use other functions.

Place all the code in file p3.py and paste that in h5.doc.

Take a screenshot with the output of the *main*() function and insert in the h5.doc file.

Extra credit part, for 3 points:

c) Use lambda expressions, *map*(), the *itertools.islice*, *functools.reduce*(), and the *filter* function to display the **sum of first 10 generated tuples** (*a, b*) that have sum *a* + *b* >= n // 2.

The sum of tuples is done component-wise for each tuple element. E.g. if the sequence filtered is (4,1), (2,6), (6,6),(3,5), then the sum of these tuples that is displayed is (4+2+6+3, 1+6+6+5) = (15, 18).

Problem 4. Fade to Black and White

The **animation-transition.py** program discussed in class (Module 7) and posted on the Module 7 Canvas page animates a smooth transition from one image to another and back.

Modify that program so that it displays an animation of a transition from the original color image to its black-and-white (B/W) version and back to original. An animation video of that transition is posted on the homework 5 page to give you the idea of how it should look like.

CAUTION: To get any credit do not corners, such as calling np.imread with a grayscale option, or generating a B/W version of the color image and then using that as image 2 with no changes to the code. It's silly, but students do this sort of thing and hope to get a good grade. Follow the requirements **exactly** or expect a low grade.

Requirements:
a) Write a function **convert_bw(**img**)** that takes an image ndarray with shape (height,width,3) and dtype **np.uint8** and returns the B/W (actually, grayscale) version of that image, also with shape (height,width,3) and the same dtype.
The color image pixel is represented by a shape (3,) array of np.uint8 integers: [red, green, blue], where these numbers are in the range 0, 1, 2,…, 255. A color with equal values for red, green, and blue is a grayscale color.
Use this simple conversion of a pixel from [r,g,b] to B/W: the B/W pixel color is the average of the r, g, b values. So, for each pixel [r,g,b] in the original color image we compute the B/W image pixel [a, a, a] where *a* is the **np.uint8 average** of the red, green, and blue values.

Understand that we represent a B/W image with pixels that have all values for its colors (i.e. red, green, and blue) equal to the average of red, green, and blue. It's worth repeating that to avoid confusion. Hence, the B/W image will have the same shape (height, width, 3) as the original color image.

The **convert_bw** function **MUST use broadcasting** and **numpy's functions** to compute the B/W image. Do **NOT** use nested for loops to compute the average for each individual pixel. If any loop is used in this function, no credit is given for part a).

HINTS: the tricky part is converting from a (h,w) B/W array to a (h,w,3) array, where in the latter all elements along the 3rd axis are repeated 3 times. One can use the **np.dstack**() function to stack arrays on the 3rd dimension, or the np.newaxis object, as in arr[:,:, **np.newaxis**], to create an array with a new axis (dimension), or reshape. Look these up.

b) Modify the **image_gen**() function so that it takes just one image file name and the steps as parameters and then it uses the B/W version of the original image to yield ndarray objects that are intermediate images between the color version and the B/W version, just like in the original py file.

c) Modify the remainder of the program accordingly in order to animate image transition from color to B/W.

Name your program p4.py and insert it in the h5.doc file.

Take a screenshot with the output of the *main*() function and insert in the h5.doc file.

Submission Instructions

Convert the **h5.doc** file to PDF format (file **h5.pdf**) and upload the following files on Canvas by clicking on the Homework 6 link:

1. h5.pdf
2. p1.py
3. p2.py
4. p3.py
5. p5.py

Grading (100 points max + extra credit):

Problem 1: 25%

- code correctness: 15%
- coding style and following standards: 6%
- screenshot(s): 4%

Problem 2: 25%

- code correctness: 16%
- coding style and following standards: 5%
- screenshot(s): 4%

Problem 3: 25% + 3 extra credit points

- code correctness: 16%
- coding style and following standards: 5%
- screenshot(s): 4%

Problem 4: 25%

- code correctness: 16%
- coding style and following standards: 5%
- screenshot(s): 4%

IMPORTANT NOTES:

- A submission that does not follow the instructions 100% (i.e. perfectly) will not get full credit.
- Upload the PDF, **and .py source files** on Canvas.
- Only submissions uploaded before the deadline will be graded.
- You have unlimited attempts to upload this assignment, but only the last one uploaded before the deadline will be graded.

APPENDIX A

The *testif* function used for writing unit tests:

```
def testif(b, testname, msgOK="", msgFailed=""):
    """Function used for testing.
    param b: boolean, normally a tested condition: true if test passed, false otherwise
    param testname: the test name
    param msgOK: string to be printed if param b==True  ( test condition true )
    param msgFailed: string to be printed if param b==False   ( test condition false )
    returns b
    """
    if b:
        print("Success: "+ testname + "; " + msgOK)
    else:
        print("Failed: "+ testname + "; " + msgFailed)
    return b
```