**MSCF Python Programming Basics**

**Homework 4**

***Due At 11:59 pm US Eastern Daylight Time,***

***July 17, 2022***

1. **(40 points) Function Definitions and Recursion**
2. Create a new program, **fun\_defs.py**, that displays the line:

hello, fun\_defs.py

Add comments for the file name, authors, and date at the top of your code file. Save and run the program to test.

1. The **fact\_rec(***n***)** (factorial, recursive) function from the lecture materials crashes (on my Windows 10 system, anyway) for an argument of 1000.

Add the definition of **fact\_rec(***n***)** to your program, test whether the function crashes on your system for an argument value of 1000. If not, try 2000, or higher. (The limit on a Mac may be very high.)

1. The **sys** module provides facilities for interacting with your system. Add

**import sys**

near the top of your program. Then, find out what your system’s default recursion limit is by displaying the value returned by **sys.getrecursionlimit()**.

1. Try doubling your recursion limit by calling **sys.setrecursionlimit(***n***)**, with an appropriate value for *n*. Test whether you can call **fact\_rec(***n***)** with nearly double the maximum argument value as before. If you set the recursion limit too high for your system, it is possible that your Python shell or IDE will crash, rather than catching the high recursion level and displaying an error message.
2. Define a *recursive* version of the Fibonacci function, **fibonacci\_rec(***n***)**. Display test results for argument values 0, 1, 3, 10, and 40 (and others, if you wish). You may—or may not—feel that the recursive implementation is easier to understand than the iterative implementation you defined previously.
3. The **rev\_str\_rec(***s***)** function that we defined in the lecture materials may be easy to understand, but it is quite inefficient: each slice of the argument string passed to the recursive function call is a *copy* of all but one character of the argument string. We would like to have a recursive function that does *not* make multiple copies of data.

One way to do this is to pass a *reference* to the original string to each recursive call, and the *length* of the not-yet-reversed part of the string, something like this:

**def rev\_str\_rec2(s, length):**

**if length == 0:**

**return ''**

**return s[length-1] + rev\_str\_rec2(s, length – 1)**

Now,

**print(rev\_str\_rec2('', 0))** # displays nothing

**print(rev\_str\_rec2('a', 1))** # displays a

**print(rev\_str\_rec2(**

**'hello, world', 12))** # displays dlrow ,olleh

This is much more efficient, but unpleasant for the caller of the function: why should the caller have to compute and pass the length of the string?

What we want in a case like this is a function that is convenient for the user to call, that in turn calls a *helper function* that implements the recursive algorithm, like this:

**def rev\_str\_rec3(s):**

**return rev\_str\_rec3\_helper(s, len(s))**

**def rev\_str\_rec3\_helper(s, length):**

**if length == 0:**

**return ''**

**return s[length-1] + rev\_str\_rec3\_helper(**

**s, length - 1)**

This split between a function convenient for a programmer to call and a helper function that does the “messy” work is common. Think of it as the programming equivalent of a mathematical theorem that uses a lemma in its proof.

Add and test this code in your program, with some additional test strings.

1. Define the function **first\_n\_primes(***n***)** that returns a list of the first *n* prime numbers (so, for example, **first\_n\_primes(3)** would return **[2, 3, 5]**).

Add these tests to your code:

**print(first\_n\_primes(0))** # an empty list

**print(first\_n\_primes(3))**

**print(first\_n\_primes(10))**

**print(first\_n\_primes(100))**

1. Define the function **nth\_prime(***n***)** that returns the *n*th prime number (so, for example, **nth\_prime(3)** would return **5**).

Add these tests to your code:

**print(nth\_prime(1))**

**print(nth\_prime(3))**

**print(nth\_prime(10))**

**print(nth\_prime(100))**

1. Define the function **is\_prime(***j***)** that returns **True** if *j* is prime, otherwise **False**.

Add these tests to your code:

**print(is\_prime(1))**

**print(is\_prime(3))**

**print(is\_prime(11))**

**print(is\_prime(51))**

1. **(40 points) NumPy**
2. Create a new program, **numpy\_tests.py**, that displays the line:

hello, numpy\_tests.py

Add comments for the file name, authors, and date at the top of your code file. Save and run the program to test.

1. Import the NumPy module using the abbreviation **np**. Then, create a 5 by 5 **ndarray** of random values drawn from a standard normal distribution. Use the variable name **an1** as the reference to this **ndarray**. Round the values to 3 decimal places, and **print()** the **ndarray**.
2. Create another 5 by 5 **ndarray** of random values drawn from a standard normal distribution, named **an2**, with values rounded to 3 decimal places. Display **an2** and confirm that **an2** is (almost surely) not the same as **an1**.
3. You can set a seed for the NumPy random number generation module with:

**np.random.seed(***seed***)**

Set the seed to **0**, create a 5 by 5 **ndarray** of values from standard normal, named

**an3**, with values rounded to 3 decimal places. Then, set the seed to **0** again, and

create a 5 by 5 **ndarray** of values from standard normal, named **an4**, with values

rounded to 3 decimal places. Display both **an3** and **an4** and confirm that they contain

the same values. Also, confirm that **an3** and **an4** are ***not*** references to the same

object.

1. Using *array slice* notation, add **3.3** to each value in the middle row of **an3**. Display the modified **an3**.
2. Using *array slice* notation, change the values in the first column of **an3** to the negatives of the original values. Display the modified **an3**.
3. Using *array slice* notation, change the central 9 elements of **an3** to have the values of a 3 by 3 identify matrix. Display the modified **an3**.
4. Using *array slice* notation, make **an5** refer to a *copy* of the 2 by 2 array in the lower right corner of **an3**. Display **an5**. Multiply the elements of **an5** by **2.2**. Display **an3** to confirm that **an3** *has not* changed, and display **an5** to confirm that **an5** *has* changed.
5. Display the *shape* of **an3**, the *number of dimensions* of **an3**, the *data type* of **an3** as a whole, and the *type* of the upper left element of **an3**.
6. Use *Boolean indexing* to subtract **1.1** from the elements of the *first*, *middle*, and *last* rows of **an3**. Display the modified **an3**.
7. Use *Boolean indexing* to replace all values in **an3** that are greater than **1.0** *or* less than **-1.0** with **0.333**. Display the modified **an3**.
8. Use *integer* (or *“fancy”*) *indexing* to reverse the rows of **an3**. Display the modified **an3**.
9. Use a combination of *array slice* notation and *integer indexing* to swap the second and fourth columns of **an3**. Display the modified **an3**.
10. Compute and display the *minimum*, *maximum*, *sum*, *mean*, *variance*, and *sample* *standard deviation* of the elements of **an3**. (You should compare the results of your own **mean\_val(***v***)**, **stdev\_of\_vals(***v***)**, and **min\_max\_vals(***v***)** function definitions from a prior homework with the results from NumPy’s analogous functions. Copy the code for your own functions into your **numpy\_tests.py** file. Since your functions operate on **list** arguments rather than 2-D **ndarray** arguments, write a **list** comprehension to create a **list** **v** containing the same values as **an3**, and pass **v** as the argument to your functions. For NumPy’s **std()** function, set the *degrees of freedom* keyword parameter to 1 to get the sample standard deviation rather than the population standard deviation.)
11. Set the variable **an3trans** to refer to the *transpose* of **an3**. Display **an3trans**.
12. Set the variable **an3sqr** to refer to the “square” of **an3** (that is, to **an3** times **an3**). Display **an3sqr**.
13. Compute and display the *determinants* of **an3**, **an3trans**, and **an3sqr**, and confirm that the values make sense, relative to each other.
14. Set the variable **an3inv** to the *inverse* of **an3**. Display **an3inv**.
15. Compute and display the *product* of **an3** and **an3inv**, rounded to **10** decimal places. Confirm that the result makes sense.
16. Set the variable **b** to refer to a 1-D **ndarray** made from the list **[1, 1, 1, 1, 1]**.
17. Set the variable **x** to refer to the solution of **an3** times **x** equals **b**. Display **x**.
18. Set the variable **x2** to refer to **an3inv** times **b**. Display **x2** and confirm that it is equal to **x** from part 2.u.
19. **(20 points) Wes McKinney, Pandas BDFL (Benevolent Dictator For Life)**

Wes McKinney is the originator of the Pandas module for Python, and is officially the

Pandas BDFL. His book, **Python for Data Analysis: Data Wrangling with Pandas,**

**NumPy, and IPython, 2nd Edition** is an outstanding reference for Pandas, NumPy, and

several other data analysis modules in the Python “ecosystem.” It is available for free

online through the Carnegie Mellon University Library:

1. Browse to **library.cmu.edu**
2. Enter **Python for Data Analysis** in the Search box, and click Search
3. Scroll down to find the version dated 2018, and click **Available Online**
4. In the View Online section, click **O’Reilly Online Learning: Academic/Public Library Edition**
5. If you are asked for your academic institution, select Not listed? Click here
6. Enter your CMU email address, [*andrewid***@andrew.cmu.edu**](mailto:andrewid@andrew.cmu.edu)
7. You should gain access to the book at this point, and can locate the Table of Contents

We will use NumPy and Pandas (and other) facilities in homework assignments for the

rest of this course, and you will use them in future MSCF courses, as well. But we won’t

spend more lecture time on them, for the most part.

For this part of the homework, first you should read Chapter 2 of McKinney’s book and

become familiar with Jupyter notebooks. (You can also read Chapter 1 and Chapter 3,

but it is not required.)

1. Using a Jupyter notebook named **HW4\_part\_3a**, read **Chapter 4, NumPy Basics: Arrays and Vectorized Computation** and try all of the examples. About 70% of this material will be review, but McKinney goes into more depth on several topics. Save your Jupyter notebook for submission as part of Homework 3. (The name of the Jupyter notebook file will be **HW4\_part\_3a.ipynb**.)
2. Using another Jupyter notebook named **HW4\_part\_3b**, read **Chapter 5, Getting Started with pandas** and try all of the examples. Again, much of this material will be review, but McKinney goes into more depth on several topics. Save your Jupyter notebook for submission as part of Homework 3. (The name of the Jupyter notebook file will be **HW4\_part\_3b.ipynb**.)

***And Finally***

Create a zip archive named **HW4\_Team\_***N***.zip**, where *N* is your team number, containing your code files for this assignment: **fun\_defs.py**, **numpy\_tests.py**, **HW4\_part\_3a.ipynb**, and **HW4\_part\_3a.ipynb**. One team member should upload this zip archive to Canvas.