Assignment 6

Arrays and Recursion

Date Due: Monday, March 21, 2022, 11:59pm **Total Marks: 41**

General Instructions

- **This assignment is individual work.** You may discuss questions and problems with anyone, but the work you hand in for this assignment must be your own work.
- Each question indicates what to hand in. You must give your document the name we prescribe for each question, usually in the form **aNqM**, meaning Assignment N, Question M. Put your name and student number at the top of every document you hand in. These conventions assist the markers in their work. Failure to follow these conventions will result in needless effort by the markers, and a deduction of grades for you. Do not submit folders, zip documents, even if you think it will help.
- Programs must be written in **Python 3**, and the file format must be text-only, with the file extension `.py`.
- Documents submitted for discussion questions should make use of common file formats, such as plain text (`.txt`), Rich Text (`.rtf`), and PDF (`.pdf`). We permit only these formats to ensure that our markers can open your files conveniently.
- **Assignments must be submitted electronically to Canvas.** There is a link on the course webpage that shows you how to do this.
- **Canvas will not let you submit work after the assignment deadline.** It is advisable to hand in each answer that you are happy with as you go. You can always revise and resubmit as many times as you like before the deadline; only your most recent submission will be graded.
- Questions are annotated using descriptors like "easy" "moderate" and "tricky". All students should be able to obtain perfect grades on "easy" problems. Most students should obtain perfect grades on "moderate" problems. The problems marked "tricky" may require significantly more time, and only the top students should expect to get perfect grades on these. We use these annotations to help students be aware of the differences, and also to help students allocate their time wisely. Partial credit will be given as appropriate, so hand in anything you've done, even if it's not perfect.

Question 1 (12 points):

Purpose: To practice simple file I/O and numpy array operations

Degree of Difficulty: Moderate.

In Canada, the federal government is made up of a number of seats. The seats are divided among the provinces; for example, Saskatchewan has 14 seats, while Ontario has 121.

For this question, you will write a program using population data from the Canadian census to determine which provinces are over- or under- reprented in the federal government based on their populations.

Using Numpy Arrays

The main purpose of this question is to practice data manipulation using numpy arrays. Therefore, for full marks, you must perform all of the significant calculations by using **numpy array operations**.

You will need all three of: array arithmetic, array relations, and logical array indexing. The **only place where you will use loops** in your program is at the very beginning (for reading from file) and at the very end (for printing the results).

Task Breakdown

The steps below will help you break down this process into manageable pieces.

Before you start anything, make sure you have imported the `numpy` module. This module is NOT standard, so if you are working on your own machine, you may need to install it first.

Load the Data into Arrays

Download the file `provincial_seats.txt` from the course Moodle. The file looks like this:

```
Nunavut,35000,1
Alberta,4067000,34
...
```

Each line consists of a province or territory's name, its population (rounded to the nearest thousand) and its number of seats in the government, all separated by commas.

Write code to open this file and read this data into **three separate lists** (one for the names, one for the populations, and one for the seats). Then, convert each list to a **numpy array** using the `.array()` method from `numpy`. Make sure to maintain the order so that the lists line up by position (i.e. `Nunavut` is first in the array of names, and its matching population and seats are the first items in their respective arrays).

Test these arrays before you move on! You will use them to perform all the following array calculations.

Predicted Seats

Using array operations on the loaded data, create an array that contains, for each province, the expected number of seats that each province WOULD have if its seats were exactly proportional to its population. Round these values to the nearest integer (you can use numpy's `.around()` method).

Hint: As part of this calculation, you'll probably need Canada's total population, and the total number of seats. Think about how you can get these using the data you already have using numpy's `.sum()` method.

Finding Over- and Under-represented provinces

If a province has more actual seats than the number of seats you calculated based on its population above, we'll call that province **over-represented**. If it has fewer actual seats than predicted seats, then it is **under-represented**.

Use **array relational operators** and **logical indexing** to create three arrays for:

- the names of the over-represented provinces
- the expected seats of those provinces
- the actual seats of those provinces

Then do the same thing for the under-represented provinces.

Finally, print your results to the console. You can use loops for this final task.

Sample Run

The output of your program might look something like this:

```
Based on population, the following provinces are over-represented:
Nunavut   Expected: 0   Actual: 1
Saskatchewan   Expected: 11   Actual: 14
Yukon   Expected: 0   Actual: 1
Manitoba   Expected: 12   Actual: 14
Prince Edward Island   Expected: 1   Actual: 4
Newfoundland   Expected: 5   Actual: 7
Northwest Territories   Expected: 0   Actual: 1
Nova Scotia   Expected: 9   Actual: 11
New Brunswick   Expected: 7   Actual: 10

The following provinces are under-represented:
Alberta   Expected: 39   Actual: 34
British Columbia   Expected: 45   Actual: 42
Ontario   Expected: 129   Actual: 121
Quebec   Expected: 79   Actual: 78
```

What to Hand In

(a) A document entitled `aq1.py` containing your finished program, as described above.

Evaluation

- 3 marks for loading the data into numpy arrays
- 3 marks for creating the array with the predicted number of seats. (-2 if a loop is used)
- 4 marks for creating the arrays for over- and under- represented provinces. (-2 if a loop is used)
- 2 marks for correct console output
- -1 mark if the student did not include their name, NSID, student number and instructor's name at the top of the submitted file.

Question 2 (7 points):

Purpose: To practice slicing with arrays.

Degree of Difficulty: Moderate.

Background

The evil Ice King has sent his army of penguins (all named Gunther) to attack the Candy Kingdom.[1] The Candy Kingdom's wise ruler, Princess Bubblegum, has devised a defence to ward off the invading penguin army. Princess Bubblegum's "Fish-a-pult" will lob barrels of fish onto the battlefield. This will cover an area of the battlefield with fish within which the penguins will stop to feed, and then, too full to fight, will retreat for an afternoon nap.

Suppose we have a 2D integer array representing the battlefield. Each array value represents the number of penguins currently in a 1 square meter area of the battlefield. Thus, a 100 by 100 array would represent a 100m by 100m battlefield, and each value in the array is the number of penguins in the corresponding one square meter.

The Fish-a-pult can fire barrels of fish of different weights. The weight of the fish barrel determines the size of the **square** area[2] that will be covered by fish when the barrel explodes on impact. If a barrel weighs n pounds, then it covers a square area $2n + 1$ meters on each side. Thus, a 1-pound barrel of fish will cover a 3m square area (corresponding to a 3 by 3 slice of the array!).

Your Task

Your job is to write a part of the targeting software for the Fish-a-pult.

Write a function called `find_target_location` which has the following parameters:

- `battlefield`: a 2D integer array representing the current state of the battlefield (containing the number of penguins at each location, as above)
- `fish_weight`: an integer representing the weight of the barrel of fish to be fired. The weight also determines the size of the area covered by fish, as above.

Your function should return the optimal target location (as described below) as a tuple, i.e, the pair of array offsets corresponding to the location. If there is more than one location tied for the most number of penguins, it does not matter which one you return.

Your task is to determine the optimal target location for a given quantity of fish. The optimal location is the battlefield target location which, if hit with a barrel weighing `fish_weight` pounds, will distract the most penguins. In other words, if the `fish_weight` is n pounds, then find the center of the $2n+1$ by $2n+1$ square on the battlefield containing the most penguins. This can be done fairly easily using loops and slicing. You will also find the `numpy.sum()` function useful.

You need not consider locations where the fish would land outside of the battlefield – these cannot be optimal locations. A target location where the entire detonation area lies within the battlefield grid will always be as good as or better than a target location where the detonation area falls off the grid because we know there are zero penguins outside the grid.

If `fish_weight` equals or exceeds either half of the battlefield width (not rounded), or half of the battlefield height (not rounded), then this means that the detonation area is larger than the entire battlefield. Thus, there are no target locations where the detonation area falls entirely within the battlefield grid. In such a case, return `None`. Firing more fish than needed to cover the entire battlefield is a waste of tasty fish!

[1] This question was inspired by Adventure Time
[2] The perfectly square area of detonation is a miracle of Candy technology.

How do I know if it's working?

Two data files are provided, as well as starter code. The starter code reads the data files which contain the battlefield data and stores the data in 2D arrays. Use these arrays and the information below to determine if your code is working.

Battlefield	Fish Barrel Weight	Expected Result
field1.csv	1	(2,2)
field1.csv	2	(2,2)
field1.csv	3	None
field2.csv	1	(1,2)
field2.csv	2	(2,3) or (4,3)
field2.csv	3	(3,4)
field2.csv	4	(4,4)
field2.csv	5	None

Files Provided

- `aq2-starter.py`: Starter file for this assignment. This includes a function that can read CSV files containing battlefield data.
- `field1.csv` and `field2.csv`: these are files that can be read by the starter code and provide data for your program.

What to Hand In

(a) A document entitled `aq2.py` containing your finished program, as described above.

Evaluation

- 1 mark: Correctly returns None if `fish_weight` exceeds half of the battlefield width or height.
- 2 marks: Checks each battlefield location that would not cause the detonation area to fall outside the battlefield.
- 1 mark: Does **not** check battlefield locations that would cause the detonation area to fall outside the battlefield.
- 2 marks: Correctly calculates the number of penguins affected for the detonation area at battlefield locations.
- 1 mark: Correctly returns the center of the detonation area that will cover the most penguins.
- -1 mark if the student did not include their name, NSID, student number and instructor's name at the top of the submitted file.

Question 3 (7 points):

Purpose: To practice recursion with a wrapper function

Degree of Difficulty: Easy.

In the lecture slides, you may have attempted an exercise to print out all the characters in a string in reverse using recursion. For this question, you'll tackle a similar task: printing out all of the WORDS in a string in reverse. You can assume that separate words in a string are always separated by at least one space.

Sample Run

Assuming our original sentence was `DO I CHOOSE YOU PIKACHU`, your function should produce the following output [3]:

```
PIKACHU YOU CHOOSE I DO
```

All of the words should be printed on the same line. It is okay if you end up having a trailing space after the last word.

Program Design

When you write programs to solve problems using loops, very often the code doesn't jump right into a loop first thing. Often, there's a bit of set-up that happens first. The same can be true of recursion.

To solve this problem, you should write TWO functions. The **first** function should be called something like `reverse_phrase()` and **must** have a SINGLE parameter: the string that represents the sentence to be reversed. This function should not itself be recursive. It simply does any necessary set-up before calling your second (recursive) function, which is where the real work will be done.

Your **second** function should be called something like `reverse_phrase_recursive()`. It can have any number of parameters that you think you need, and those parameters can be of any data type that you think will be easiest to work with. This function must be recursive and is not allowed to use loops in any way.

To test your program, the "main" part of your program should simply call `reverse_phrase()` with the string you want to reverse as an argument.

What to Hand In

- A document called `aq3.py` containing your finished program, as described above.

Be sure to include your name, NSID, student number, course number and lecture section and laboratory section at the top of all files.

Evaluation

- 1 mark: The wrapper function `reverse_phrase()` performs some reasonable setup that makes the recursive function easier to write.
- 1 mark: The base case of the recursive function is correct.
- 1 mark: The recursive function has reasonable parameters for solving the problem.
- 2 marks: The recursive case of the recursive function has the correct behaviour
- 2 marks: The doc-string for BOTH functions is sufficiently descriptive, indicating its purpose and requirements.
- -2 marks: Global variables and/or loops are used in the recursive function.

[3] If Jedi master Yoda played Pokemon, he would need a tool like your program.

Question 4 (6 points):

Purpose: To practice recursion on a classic problem

Degree of Difficulty: Tricky.

Nim is a 2-player game where the players take turns picking up sticks from a pile. On their turn, a player can pick up either 1, 2, or 3 sticks. The player who picks up the last stick loses.

Your job is to write a **recursive function** that determines, for a given number of sticks, the winner of the game assuming that both players play optimally. Your function should take **exactly 2** parameters: an integer indicating the current number of sticks, and a boolean indicating which player is currently to move. It should return the boolean value `True` if player 1 (the player that took the very first turn) will win and `False` otherwise.

For example, if there are 4 sticks left and player 1 is to move, then player 1 can win by taking 3 sticks. This will leave only 1 stick left, which player 2 will have to pick up, thus losing the game.

This might sound a little difficult, especially the assumption that players will play optimally, but with the power of recursion, this is not a complicated problem. Your general approach should be to use recursion to simply **try all possible moves** (i.e. make 1 recursive call for each amount of sticks that can be removed) in each position.

Your function should be **no longer than 12 lines of code** (not counting comments) and possibly less (ours is 9). If you find your solution is getting any longer than that, you are overthinking it!

Sample Run

The output for your program might look something like this:

```
Enter the number of sticks for nim: 9
Will player1 win?
False
```

Extra Notes

Nim is a well-known problem, and it is of course possible to come up with a mathematical rule that will tell you which player will win. You won't get any marks for doing this though, as the point of the assignment is to practice using recursion.

What to Hand In

- A document entitled `aq4.py` containing your finished program, as described above.

Evaluation

- **-1 mark** for lack of name, nsid, student number and instructor in the solution
- 1 mark: The recursive function is called correctly using appropriate arguments
- 1 mark: The base case(s) are correct
- 3 marks: The recursive case is correct
- -2 marks: The function uses global variables, or otherwise tries to sidestep a nicely recursive solution.
- 1 mark: The recursive function has an adequate docstring

Question 5 (9 points):

Purpose: To practice recursion on a problem that can be split into parts

Degree of Difficulty: Moderate

Aspiring pokemon trainer Ash Ketchum has caught a lot of pokemon. To keep them healthy, he regularly feeds them vitamins. However, the vitamins are bigger than what a pokemon can swallow in one bite, and so when Ash's pokemon team is given a vitamin, they will break it apart into smaller pieces.

Sometimes, though, the vitamin is SO big, that the pokemon have to break it apart more than once! Here is how the process works:

- If the vitamin's weight is less than or equal to 0.1 grams, it is small enough to eat, and doesn't need to be broken. Thus, there is 1 edible piece for 1 pokemon.
- Otherwise, the pokemon will team up to smash the vitamin into pieces of equal weight. The number of new pieces seems to be random (the pokemon can be a bit excitable), either 2, 3, or 4, but the weight of each new piece is the same (one-half, one-third or one-quarter of the previous piece). The pokemon will then break each of these pieces again until they are small enough to swallow.

For example, suppose the initial vitamin weighed 0.6 grams. The pokemon break it once, and we randomly determine that the vitamin breaks into 2 parts (each part weighs 0.3 grams). The pokemon break the first 0.3 gram piece, which breaks into 3 pieces. Since these pieces will each weigh 0.1 grams, they are now edible, so we have 3 edible pieces so far. The pokemon will then break the second 0.3 gram piece; this time, it breaks into 2 parts, each of which weighs 0.15 grams, which are still too large to eat. The pokemon break each of these parts again; the first breaks into 3 parts, each weighing 0.05 grams, so we have 3 more pieces. The second breaks into 4 parts, each weighing 0.0375 grams, so that's another 4 pieces. There are no longer any pieces larger than 0.1 grams, so the total number of edible pieces is 3 + 3 + 4 = 10 pieces. So in this case, 10 different pokemon are able to get their vitamin dosage from a single, original 0.6 gram vitamin.

Tracing through the problem like in the paragraph above is exhausting! But if you bring yourself to trust in the power of recursion, solving this problem is not hard at all.

For this question, your task is to write a recursive program that will calculate how many edible vitamin pieces are made whenever a pokemon team is given a vitamin that weighs W grams.

Program Design

(a) Write a recursive function that simulates the breaking of a single vitamin. The weight of the vitamin (as a float) should be a parameter to your function. The function should return an integer, indicating the number of edible pieces produced from breaking the vitamin.
To write this function, you will need to use random numbers for when the vitamin is broken into parts. If you first `import random as rand`, then the expression `rand.randint(a,b)` will give you a random number in the range from a to b (including a and b). **For this question, you ARE allowed to use a loop in your recursive function if you like; however, recursion should still do the "real work"**. For instance, you might want to use a loop to iterate over the number of vitamin pieces created from a single smash.

(b) In the 'main' part of your program, write a few lines of code that asks the user for the size of a vitamin, and uses a loop that will **call your piece-counting recursive function 1000 times**. Use the results of those 1000 simulations to report the average number of edible pieces produced from a vitamin.

An example of your program running might look like this:

```
How big is the vitamin, in grams? 1.0
On average, a pokemon team can get 18.774 bite-sized pieces from a 1 gram vitamin!
```

Note that because of the randomness involved you might never get this exact result with an input of 1.0 grams but it should be pretty close.

Run your program using vitamins of weight 5, 10 and 100 grams. Copy/paste your output for all of the examples above into a document called `aq5_output.txt` for submission.

What to Hand In

- A document entitled `aq5.py` containing your finished program, as described above.
- A document called `aq5_output.txt` (rtf and pdf formats are also allowable) with the output of your program copy/pasted into it, to demonstrate that your program works.

Be sure to include your name, NSID, student number, course number and lecture section and laboratory section at the top of all files.

Evaluation

- **-1 mark** for lack of name, nsid, student number and instructor in the solution
- 1 mark: The function has a parameter for the size of the vitamin.
- 1 mark: The base case is correct.
- 3 marks: The recursive case is correct.
- -2 marks: The function uses global variables, or otherwise tries to sidestep a nicely recursive solution.
- 1 mark: The recursive function has an adequate docstring
- 2 marks: You program correctly takes an average count of edible pieces, using 1000 trials.
- 1 mark: You submitted a document showing the output of your program on all of the examples given above.