CMPUT 274/5 - Tangible Computing **Fall 2019**
Weekly Exercise #2

Unfair Dice

In this assignment, you will create a program to simulate an *unfair or biased die*. First, using the `random` module, you will create a function called `biased_rolls` to simulate rolls of an m-sided biased or unfair die. Second, you will write a function called `draw_histogram` that will visualize the results of your unfair dice roller.

Pseudo-Random Number Generation and Seed Values

In lab 3 of the Python Intro Labs, you were introduced to the `random` module in Python to create a simple dice roller with outcomes using the `randint` function. However, the `randint` function does not produce truly random values. The values are only *pseudo-random*.

Truly random numbers are difficult to generate efficiently. A *pseudo-random number generator (PRNG)* is an algorithm which produces a stream of numbers that are deterministic and periodic. This means the sequence values produced by successive calls to `randint` is predictable. Eventually, the numbers will repeat themselves. Typically, the *period*[1] of a number generator will be very large (something like $2^{1900} - 1$).

The generator uses a *seed value*, which is used to initialize the sequence. If you run the same pseudo-random number generator twice using the same seed, you will get the same "random" number sequence twice! This feature of pseudo-random number generation can be very useful for demonstration or testing purposes. In this exercise, you will use a seed value so that a grading TA can predictably grade your solution (by comparing your output to the solution's output when using the same seed).

According to the documentation, the Python module `random` uses a pseudo-random number generator called the Mersenne Twister. Although this generator is good enough for our assignment, it has a period of $2^{19937} - 1$, so it is not truly random! Since it is predictable, this method of generating random values is unsuitable (for example) for purposes such as security or cryptography.

An Introduction to Biased Dice:

In the original dice roller in the Python Intro Labs, your goal was to implement a fair die. On a fair die, every number has an equal chance of being rolled (or a 1 out of 6 chance on a cubic 6-sided die). On a *biased die*, some numbers are more likely to be rolled than others. A real die may become biased because of its shape (shaving some edges, putting weights inside the die) or some other method of manipulating the probabilities.

For example, a 6-sided biased die might have the following probabilities:

[1] Number of steps until the sequence repeats itself

```
Side |  Probability
---- | -------------
 1   | 1/4   =  3/12
 2   | 1/6   =  2/12
 3   | 1/12  =  1/12
 4   | 1/12  =  1/12
 5   | 1/4   =  3/12
 6   | 1/6   =  2/12
               -----
               12/12
```

The probability of rolling a 1 on this die is, for example, 1/4 or 25%, rather than the 1/6 expected for a fair die. Notice that the sum of all the probabilities is still equal to 1.

Your Task #1: `biased_rolls`

In a module called `unfairDice.py`, write a function called `biased_rolls` which takes 3 parameters. Here is an example declaration of the function:

```
def biased_rolls(prob_list, s, n):
    """ Simulate n rolls of a biased m-sided die and return
    a list containing the results.

    Arguments:
        prob_list: a list of the probabilities of rolling the
                   number on each side of the m-sided die. The list
                   will always have the length m (m >= 2), where m is
                   the number of sides numbered 1 to m. Therefore,
                   for example, the probability stored at index 0 in
                   the list is the probability of rolling a 1 on
                   the m-sided die.
        s: the seed to use when initializing the PRNG
        n: the number of rolls to return

    Return:
        rolls: a list (of length n) containing each of the n rolls of the
               biased die, in the order they were generated.
    """
    pass
```

Input Guarantees: You may assume the following are true of the input to the function. In other words, we will only test your function using inputs that have the following properties. You do not have to check for them in the input.

1. The biased die will always have at least 2 sides ($m \geq 2$), so `prob_list` will always have at least two elements.

2. The argument `prob_list` will always have m elements, and the sum of all probabilities in the list will always equal 1.

3. The probabilities in `prob_list` will all be passed in as floating point values between 0 and 1.

4. The sides of each biased m-sided die passed in will always be labelled with the numbers $1, 2, ..., m$ where the side labelled 1 has its probability stored at index 0 of `prob_list`, the side 2 has probability at index 1, and so on until the side m has probability at index $m - 1$.

Using the example die above, a call to this function to generate a list with 200 rolls and a seed value of 42 would look like:

```
biased_rolls([1/4, 1/6, 1/12, 1/12, 1/4, 1/6], 42, 200)
```

A Simple Mapping Example:

In order to generate numbers with fixed probabilities, we cannot just use the `random.randint` function anymore. Instead, we will use `random.random`, which generates a pseudo-random floating point number in the range `[0.0, 1.0)`, where 1.0 is not included. It will be your job to *map* this range to the appropriate numbers on the faces of the die.

Here is a simple example of how you could do this. Imagine you had a 2-sided coin that flips heads 75% of the time (probability: 0.75) and tails 25% of the time (probability: 0.25). If you use the random.random function (a number between 0.0 and 1.0), you could map the number like this:

Any number greater than or equal to 0.0 but less than 0.75 (75%) = HEADS
Any number greater than or equal to 0.75 but less than 1.0 (25%): TAILS

Here, we have *mapped* the larger range 0.0 to 1.0 into a smaller range (heads or tails) so that 75% of the range $[0, 1)$ is mapped to heads. The `biased_rolls` function you write must be able to do this for any number of sides m, ($m \geq 2$), that may be passed in.

Within the `biased_rolls` function, implement the following steps:

1. Initialize the pseudo-random number generator using the `random.seed` function. Before you make any calls to the random module, you must initialize it using this function (documentation of random.seed here).

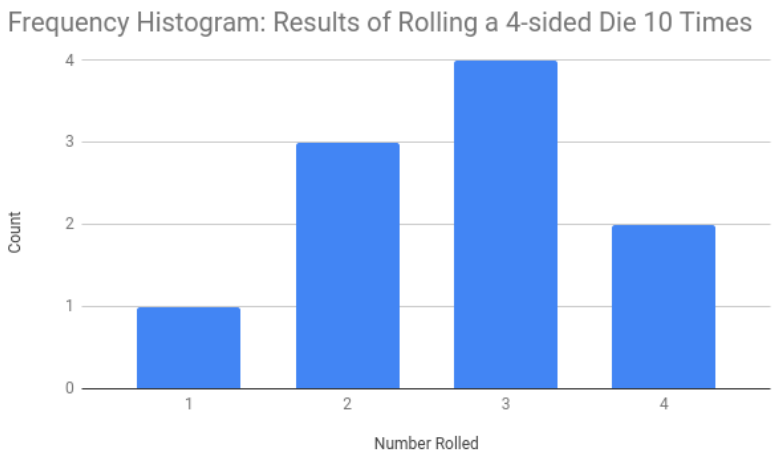
2. Generate n rolls of the die using your mapping, where each roll should generate an integer from 1 to m inclusive. This should be done using a **single** call to `random.random` for each of the n rolls. So, in total, there should be exactly n calls to `random.random`.

3. Return a single list containing the value of all n rolls in the order they were rolled.

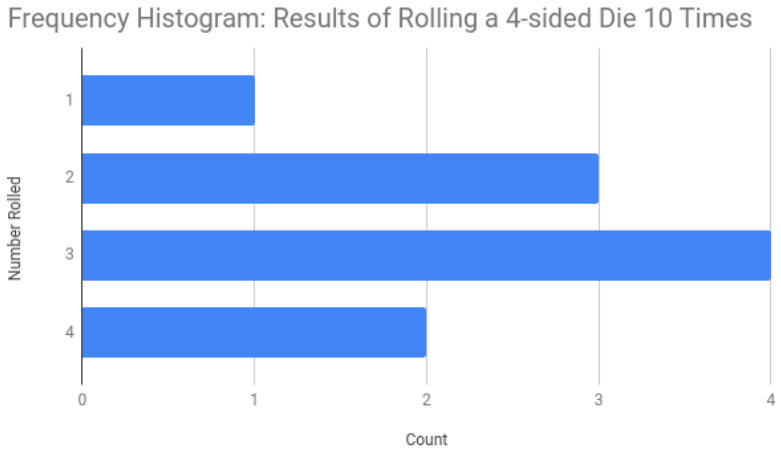
Your Task #2: `draw_histogram`

A *frequency histogram* is a type of graph that uses vertical or horizontal bars to show *frequencies* (number of times each score occurs). For example, if we rolled a 4-sided die

10 times and got the numbers `3, 2, 1, 2, 2, 3, 4, 4, 3, 3` we could plot a histogram like this:



Notice that we can display the same information horizontally by swapping the axes:



Within `unfairDice.py`, write a function called `draw_histogram` which takes 3 parameters and prints a horizontal frequency histogram of the results of rolling a biased or unbiased die. Here is an example declaration of the function:

```
def draw_histogram(m, rolls, width):
    """ Draws a frequency histogram of the rolls of an m-sided die
    mapped to a fixed width.

    Arguments:
        m (int): the number of sides on the die
        rolls (list): the list of rolls generated by the biased die
        width (int): the fixed width of the histogram, in characters
                     (this is the length of the longest bar in the
                     histogram, to maximize space in the chart)

    Returns:
```

```
        None (but prints the histogram to standard output)
    """
    pass
```

Input Guarantees:

1. All rolls in `rolls` will be integers in the range of $1, 2, ..., m$. No roll will be less than 1 or greater than m.
2. The `width` parameter should be the length of the longest bar in the chart. All other bars should be scaled accordingly.

This function should produce a *horizontal frequency histogram* that looks like this, when invoked using `draw_histogram(4, [3, 2, 1, 2, 2, 3, 4, 4, 3, 3], 4)`:

```
Frequency Histogram: 4-sided Die
1:*...
2:***.
3:****
4:**..
```

Each asterisk (*) represents the area of a bar of the histogram and each dot (.) represents empty space. The title should always be printed first and have the format: **"Frequency Histogram: M-sided Die"**, where M is the number of sides.

Notice that the width parameter here is 4 and the number of asterisks in the longest column is also 4. Because we will be testing your code with hundreds or thousands of rolls, we cannot use raw histograms. Printing that many dots and asterisks is impractical. Instead, we need to scale the histograms smaller to make them easier to read. To do this, a *max character length*, called `width` will be passed in as a parameter to the function (as shown above). This is the number of asterisks long the longest histogram bar should be. All other bars must be scaled accordingly to maintain the shape of the graph. **You will need to round the scaled length** using the `round` function, since you may not always get an exact integer. For example, here is another call: `draw_histogram(4, [3, 2, 1, 2, 2, 3, 4, 4, 3, 3], 25)`:

```
Frequency Histogram: 4-sided Die
1:******...................
2:******************......
3:************************
4:************............
```

Notice that only the scale has changed: The shape of the graph remains the same, and only the length of the histogram is different. It now has a fixed width of 25 characters.

Here are a few sample test cases:

You can test these by adding the lines below into unfairDice.py under 'if __name__ = "__main__":'. Alternately, in another file or the Python3 interpreter (in the same directory as unfairDice.py!), you can use the following line to **import** the functions from your solution file:

```
from unfairDice import *
```

and then run the following code snippets. You should be able to see the output directly as standard output in the terminal.

Code Snippet #1: A six-sided, biased die, rolled 200 times with a seed of 2^{32-1}.

```
rolls = biased_rolls([1/12, 1/4, 1/3, 1/12, 1/12, 1/6], 2**32-1, 20)
print(rolls)
draw_histogram(6, rolls, 50)
```

Output #1:

```
[3, 2, 3, 2, 3, 3, 4, 6, 2, 6, 3, 2, 6, 2, 5, 5, 3, 4, 4, 2]
Frequency Histogram: 6-sided Die
1:..................................................
2:**************************************************
3:**************************************************
4:************************..........................
5:*****************.................................
6:************************..........................
```

Code Snippet #2: A six-sided, biased die, rolled 200 times with a seed of 42.

```
rolls = biased_rolls([1/4, 1/6, 1/12, 1/12, 1/4, 1/6], 42, 200)
draw_histogram(6, rolls, 10)
```

Output #2:

```
Frequency Histogram: 6-sided Die
1:**********
2:*****.....
3:***.......
4:***.......
5:********..
6:******....
```

Code Snippet #3: A three-sided, balanced die, rolled 1000 times with a seed of 2^{32-1}.

```
rolls = biased_rolls([1/3, 1/3, 1/3], 2**32-1, 1000)
draw_histogram(3, rolls, 10)
```

Output #3:

```
Frequency Histogram: 3-sided Die
1:**********
2:**********
3:*********.
```

Explanation: Just because this die is balanced does not mean it will roll exactly the same number of 1s, 2s, and 3s! In this case, pseudo-random generation results in slightly fewer 3s.

Submission Guidelines:

Submit all of the following files as a compressed archive called `unfair_dice.tar.gz` or `unfair_dice.zip`:

- unfairDice.py, containing your implementation of both functions, `biased_rolls` and `draw_histogram`
- your `README`, following the Code Submission Guidelines.

Note that your files and functions must be named **exactly** as specified above. As with Weekly Exercise #1, you may put any amount of input or tests you want under an `if __name__ == "__main__":` section of the code. The graders will ignore this part.

Getting Started: A Few Hints

If you are struggling with this exercise, the following hints may help you:

- You may create extra functions to help if you want.
- It will be helpful to develop the part of `biased_rolls` that maps a value in the range $[0, 1)$ to a roll of the biased die. Test this functionality out extensively so you are convinced it works. Then use it to generate random rolls.
- For `draw_histogram`, you can solve a simpler version of the problem first. To begin with, ignore the scale value and implement simple printing using the "real" number of asterisks. Then implement scaling when you know that this works.
- Although some aspects of this assignment may seem complicated, you can solve the entire exercise using only lists, strings, and simple variables. You do not need dictionaries or complex data structures, although you may use them if you wish.