CMPS 5P
Introduction to Programming in Python
Programming Assignment 6

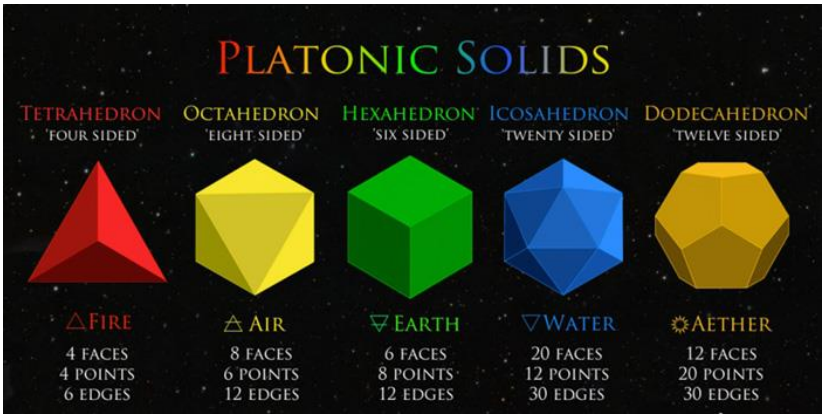
In this project you will write a Python program that simulates a dice game. The number of sides on each die, the number of dice, and the number of simulations to perform will all be taken from user input. After each simulation, your program will calculate the sum of the numbers on the dice. Then, after the specified number of simulations, your program will produce an estimate of the probability of each possible sum. This is a simple version of a well known computational technique known as *Monte Carlo*. Begin by carefully studying the example `DiceProbabilities.py` posted on the class webpage. Your program will be a direct generalization of that example, and will be called `Probability.py`.

The Monte Carlo method was invented by scientists working on the atomic bomb in the 1940s. They named their technique for the city in Monaco famed for its casinos. The core idea is to use randomly chosen inputs to explore the behavior of a complex dynamical system. These scientists faced difficult problems of mathematical physics, such as neutron diffusion, that were too complex for a direct analytical solution, and must therefore be evaluated numerically. They had access to one of the earliest computers (ENIAC), but their models involved so many dimensions that exhaustive numerical evaluation was prohibitively slow. Monte Carlo simulation proved to be surprisingly effective at finding solutions to these problems. Since that time, Monte Carlo methods have been applied to an incredibly diverse range of problems in science, engineering, and finance. In our case, a pure analytical solution is possible for the probabilities that we seek, but since this is not a class in probability theory, we will take the computational/experimental approach. You can find a very interesting history of early computing machines, the Monte Carlo Method, and the development of the atomic bomb in the book *Turing's Cathedral* by George Dyson. Follow the link

https://en.wikipedia.org/wiki/Monte_Carlo_method

for an article on Monte Carlo methods.

A normal six-sided die is a symmetrical cube that, when thrown, is equally likely to land with any of it's six faces up (provided its mass distribution is uniform.) By labeling its faces with the numbers 1-6, we have a physical device capable generating random numbers in the set {1, 2, 3, 4, 5, 6}. It is possible to make perfectly symetrical dice in the shape of any of the so-called Platonic Solids, whose number of sides are 4 (Tetrahedron), 6 (Cube), 8 (Octahedron), 12 (dodecahedron), and 20 (Icosahedron).



See https://www.mathsisfun.com/geometry/platonic-solids-why-five.html for a nice explanation as to why these are the only perfectly symmetrical shapes possible for dice. For purposes of this project however, we shall assume it is possible to make dice with any number of faces in such a way that each face is equally likely to land in the up position. To simulate a throw of an *k*-sided die in Python, use the `randrange()` function belonging to the `random` module, which was discussed in class and illustrated in the example `DiceProbability.py`.

Your program will include a function called `throwDice()` with heading

```
def throwDice(m, k):
```

that simulates a throw of *m* independent and symmetrical *k*-sided dice, and returns the result in a *m*-tuple. The main section of your program will prompt for, and read three quantities: the number of dice, the number of sides on each die, and the number of simulations (or throws) to perform. These prompts will be robust, in that, if the user enters an integer less than 1 for the number of dice, or an integer less than 2 for the number of sides on each die, or an integer less than 1 for the number of simulations, then your program will continue to prompt until adequate values are entered. Your program is not required to handle non-integer input like floats or general strings.

Once these values have been entered by the user, your program will perform the specified number of simulations, recording the frequency of each possible sum as it goes. To do this you must first calculate the range of possible sums, and create a list of appropriate length. If you call this list `frequency[]`, for instance, then by the time the simulations are complete, `frequency[i]` will be the number of simulations in which the sum of the dice was `i`. Again, emulate the example `DiceProbability.py` to accomplish this. Calculate the relative frequency for each possible sum (the number of simulations resulting in that sum, divided by the total number of simulations). Also calculate the experimental probability for each sum (the relative frequency expressed as a percent.) Print out these quantities in a table formatted as in the sample runs below.

```
$ python Probability.py

Enter the number of dice: 3

Enter the number of sides on each die: 6

Enter the number of trials to perform: 10000

 Sum     Frequency     Relative Frequency     Experimental Probability
-------------------------------------------------------------------------
   3         45           0.00450                 0.45 %
   4        126           0.01260                 1.26 %
   5        281           0.02810                 2.81 %
   6        494           0.04940                 4.94 %
   7        677           0.06770                 6.77 %
   8        968           0.09680                 9.68 %
   9       1191           0.11910                11.91 %
  10       1257           0.12570                12.57 %
  11       1257           0.12570                12.57 %
  12       1164           0.11640                11.64 %
  13        932           0.09320                 9.32 %
  14        683           0.06830                 6.83 %
  15        469           0.04690                 4.69 %
  16        282           0.02820                 2.82 %
  17        122           0.01220                 1.22 %
  18         52           0.00520                 0.52 %

$
```

The `$` here represents the Unix (or other) command line prompt. Note the blank lines before, after and within program output. The following sample run shows what happens when the user enters invalid parameters.

```
$ python Probability.py

Enter the number of dice: -1
The number of dice must be at least 1
Please enter the number of dice: 4

Enter the number of sides on each die: 1
The number of sides on each die must be at least 2
Please enter the number of sides on each die: 7

Enter the number of trials to perform: -1
The number of trials must be at least 1
Please enter the number of trials to perform: 10000

 Sum     Frequency     Relative Frequency     Experimental Probability
-------------------------------------------------------------------------
   4           6           0.00060                 0.06 %
   5          18           0.00180                 0.18 %
   6          52           0.00520                 0.52 %
   7          83           0.00830                 0.83 %
   8         166           0.01660                 1.66 %
   9         273           0.02730                 2.73 %
  10         346           0.03460                 3.46 %
  11         469           0.04690                 4.69 %
  12         630           0.06300                 6.30 %
  13         738           0.07380                 7.38 %
  14         836           0.08360                 8.36 %
  15         930           0.09300                 9.30 %
  16         930           0.09300                 9.30 %
  17         985           0.09850                 9.85 %
  18         844           0.08440                 8.44 %
  19         737           0.07370                 7.37 %
  20         589           0.05890                 5.89 %
  21         526           0.05260                 5.26 %
  22         326           0.03260                 3.26 %
  23         238           0.02380                 2.38 %
  24         124           0.01240                 1.24 %
  25          86           0.00860                 0.86 %
  26          49           0.00490                 0.49 %
  27          13           0.00130                 0.13 %
  28           6           0.00060                 0.06 %

$
```

To get full credit, your output must be formatted exactly as above. See the example `FormatNumbers.py` on the class webpage to see how this might be accomplished. If you seed your random number generator with the integer 237, then your numbers should match mine exactly. You should experiment with other seeds, and with no seed, but when you submit your program, use the seed 237. This will facilitate automated grading of the project.

What to turn in

Submit the file `Probability.py` to the assignment name pa6 in the usual way. As always, start early and ask questions if anything is not clear.