Homework02

February 22, 2021

Victor Cruz Ramos
Section 101

1 Homework 2: Control Structures

** Submit this notebook to bCourses to receive credit for this assignment. **

Please complete this homework assignment in code cells in the iPython notebook. Include comments in your code when necessary. Enter your name in the cell at the top of the notebook, and rename the notebook [email_name]_HW02.ipynb, where [email_name] is the part of your UCB email address that precedes "@berkeley.edu". Please also save the notebook once you have executed it as a PDF and upload that to bcourses as well (note, that when saving as PDF you don't want to use the option with latex because it crashes, but rather the one to save it directly as a PDF).

1.1 Problem 1: Binomial Coefficients

[Adapted from Newman, Exercise 2.11] The binomial coefficient $\binom{n}{k}$ is an integer equal to

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n \times (n-1) \times (n-2) \times \cdots \times (n-k+1)}{1 \times 2 \times \cdots \times k}$$

when $k \geq 1$, or $\binom{n}{0} = 1$ when $k = 0$. (The special case $k = 0$ can be included in the general definition by using the conventional definition $0! \equiv 1$.)

1. Write a function `factorial(n)` that takes an integer n and returns $n!$ as an integer. It should yield 1 when $n = 0$. You may assume that the argument will also be an integer greater than or equal to 0.

2. Using the form of the binomial coefficient given above, write a function `binomial(n,k)` that calculates the binomial coefficient for given n and k. Make sure your function returns the answer in the form of an integer (not a float) and gives the correct value of 1 for the case where $k = 0$. (Hint: Use your `factorial` function from Part 1.)

3. Using your `binomial` function, write a function `pascals_triangle(N)` to print out the first N lines of "Pascal's triangle" (starting with the 0th line). The nth line of Pascal's triangle contains $n + 1$ numbers, which are the coefficients $\binom{n}{0}$, $\binom{n}{1}$, and so on up to $\binom{n}{n}$. Thus the first few lines are 1 1 1 1 2 1 1 3 3 1 1 4 6 4 1
   This would be the result of `pascals_triangle(5)`. Print the first 10 rows of Pascal's triangle.

4. The probability that an ubiased coin, tossed n times, will come up heads k times is $\binom{n}{k}/2^n$. (Or instead of coins, perhaps you'd prefer to think of spins measured in a Stern-Gerlach experiment.)

    - Write a function `heads_exactly(n,k)` to calculate the probability that a coin tossed n times comes up heads exactly k times.
    - Write a function `heads_atleast(n,k)` to calculate the probability that a coin tossed n times comes up heads k or more times.
    - Print the probabilities (to four decimal places) that a coin tossed 100 times comes up heads exactly 70 times, and at least 70 times. You should print corresponding statements with the numbers so it is clear what they each mean.

5. Along with the printed statements from Part 4, have your code generate and display two labelled plots for `heads_exactly(n,k)` and `heads_atleast(n,k)` with $n = 100$. You should have values of k on the x-axis, and probabilities on the y-axis. (Note that k only takes integer values from 0 to n, inclusive. Your plots can be connected curves or have discrete markers for each point; either is fine.)

Output To summarize, your program should output the following things:

1. The first 10 rows of Pascal's triangle
2. The probabilities (to three decimal places) that a coin tossed 100 times comes up heads exactly 70 times, and at least 70 times, with corresponding statements so it is clear what each number signifies.
3. Two labeled plots for `heads_exactly(n,k)` and `heads_atleast(n,k)` with $n = 100$, representing probability distributions for 100 coin flips.

Reminder Remember to write informative doc strings, comment your code, and use descriptive function and variable names so others (and future you) can understand what you're doing!

[2]:
```
'''The numpy library has a lot of useful functions
and we always use matplotlib for plotting, so it's
generally a good idea to import them at the beginning.'''
import numpy as np
import matplotlib.pyplot as plt

def factorial(n):
    factorial = 1
    if (n) >= 0:
        for i in range(1,(n)+1):
            factorial = factorial * i
    return factorial
factorial(5)
```

[2]: 120

[3]:
```
def binomial(n, k):
    """Returns the binomial coefficient n choose k"""
    value_1 = factorial(n)
```

```
    if k == 0:
        value_2 = factorial(1)
    else:
        value_2 = factorial(k)
    b = (n-k)
    value_3 = factorial(b)

    return (value_1/(value_2*value_3))
binomial(3,4)
```

[3]: 0.25

[36]:
```
def pascals_triangle(N):
    """Prints out N rows of pascal's triangle"""


    for row in range(0, N + 1): #This is the outer loop; each pass through the␣
 ↪loop corresponds to one row of the triangle
        for k in range(0, row + 1):   #This is is the inner loop; each pass␣
 ↪through the loop corresponds to a number on the row
            print(int(binomial(row, k)), "", end = "")
        print()
            #Code here is part of each inner loop iteration (i.e. print a␣
 ↪binomial coefficient)
        #Code here is part of the outer loop

    #This function doesn't need to return anything
    #https://www.geeksforgeeks.org/pascal-triangle/ assisted me thru this code
n = 4
pascals_triangle(n)
```

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

[51]:
```
def heads_exactly(n,k):
    """Returns the probability of getting k heads if you flip a coin n times"""
    heads = (binomial(n, k))/(2**n)
    #print("The probablility of {:.1f} heads given {:.2f} tosses is {:.8f}".
 ↪format(k, n, heads))
    return heads
heads_exactly(100, 70)
```

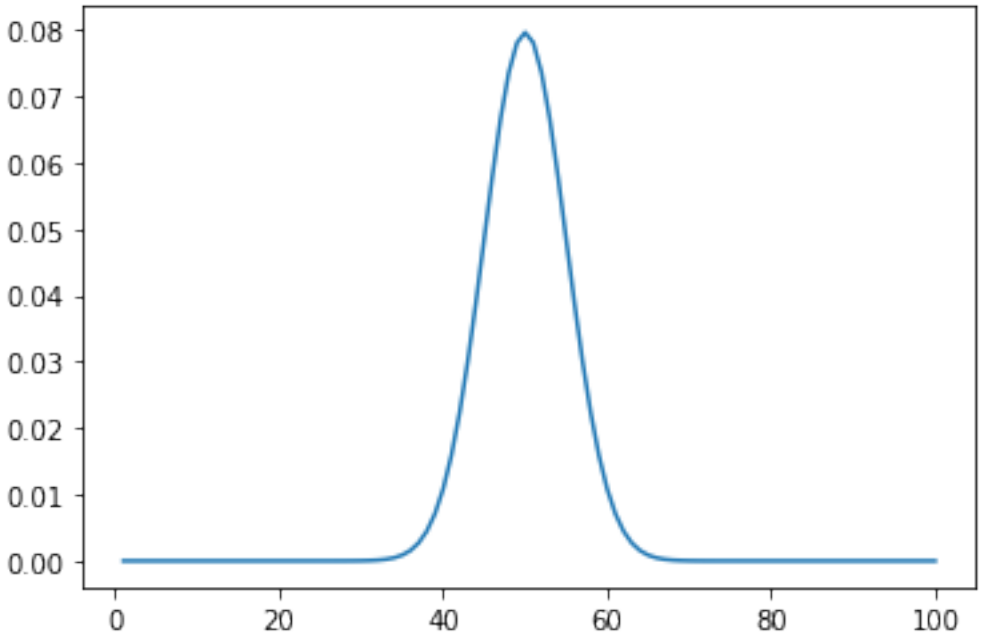
[51]: 2.3170690580135184e-05

[52]:
```
def heads_atleast(n,k):
    """Returns the probability of getting at least k heads if you flip a coin n
 ↪times"""
    total_prob = 0

    for i in range(k, n):
        heads = (binomial(n, k))/(2**i)
    new = total_prob + heads
    #Use a for loop and heads_exactly(n,k) to update total_prob
    #print("The probablility of {:.1f} heads given {:.2f} tosses is {:.8f}".
 ↪format(k, n, new))
    return new
heads_atleast(100, 70)
```

[52]: 4.634138116027037e-05

[56]:
```
#Now use your defined functions to produce the desired outputs
#For the plots, the np.arange() function is useful for creating a numpy array
 ↪of integers

#integers from 1 to 100 (lower bound is inclusive; upper bound is exclusive)

k_values = np.arange(1,101)
array = []

for k in (k_values):
    array.append(heads_exactly(100, k))
#for i in (k_values):
    #other_array.append(heads_atleast(100, i ))


plt.plot(k_values, array)
#plt.plot(k_values, other_array) # I tried to plot the other probability but it
 ↪have me an error claiming:
"""local variable 'heads referenced before assignment'"""# which I did not
 ↪understand why
plt.show
```

[56]: <function matplotlib.pyplot.show(close=None, block=None)>



1.2 Problem 2: Semi-Empirical Mass Formula

[Adapted from Newman, Exercise 2.10] In nuclear physics, the semi-empirical mass formula is a formula for calculating the approximte nuclear binding energy B of an atomic nucleus with atomic number Z and mass number A:

$$B = a_V A - a_S A^{2/3} - a_C \frac{Z^2}{A^{1/3}} - a_S \frac{(A-2Z)^2}{A} + \delta \frac{a_P}{A^{1/2}},$$

where, in units of millions of electron volts (MeV), the constants are $a_V = 14.64$, $a_S = 14.08$, $a_C = 0.64$, $a_S = 21.07$, $a_P = 11.54$, and

$$\delta = \begin{cases} 0 & \text{if } A \text{ is odd,} \\ +1 & \text{if } A \text{ and } Z \text{ are both even,} \\ -1 & \text{if } A \text{ is even and } Z \text{ is odd.} \end{cases}$$

The values above are taken from D. Benzaid et al., NUCL SCI TECH 31, 9 (2020); https://doi.org/10.1007/s41365-019-0718-8

1. Write a function `binding_energy(A, Z)` that takes as its input the values of A and Z, and returns the binding energy for the corresponding atom. Check your function by computing the binding energy of an atom with $A = 58$ and $Z = 28$. (Hint: The correct answer is around 490 MeV.)

2. Write a function `binding_energy_per_nucleon(A, Z)` which returns not the total binding energy B, but the binding energy per nucleon, which is B/A.

3. Write a function `max_binding_energy_per_nucleon(Z)` which takes as input just a single value of the atomic number Z and then goes through all values of A from $A = Z$ to $A = 3Z$, to find the one that has the largest binding energy per nucleon. This is the most stable nucleus with the given atomic number. Have your function return the value of A for this most stable nucleus and the value of the binding energy per nucleon.

4. Finally, use the functions you've written to write a program which runs through all values of Z from 1 to 100 and prints out the most stable value of A for each one. At what value of Z does the maxium binding energy per nucleon occur? (The true answer, in real life, is $Z = 28$, which is nickel. You should find that the semi-empirical mass formula gets the answer roughly right, but not exactly.)

Output Your final output should look like

```
Z =   1 : most stable A is 2
Z =   2 : most stable A is 4
      .
      .
      .
Z =  10 : most stable A is 20
Z =  11 : most stable A is 23
      .
      .
      .
Z = 100 : most stable A is 210
The most stable Z is ____
with binding energy per nucleon ____
```

With the …'s and _____'s replaced with your results. The binding energy per nucleon in the last line should have three decimal places.

For maximum readability, you should include the extra whitespace around the Z = numbers so everything lines up, as shown. (To remember the `print` formatting syntax to do this, see Table 1.1 in the Ayars text.)

Reminder Remember to write informative doc strings, comment your code, and use descriptive function and variable names so others (and future you) can understand what you're doing!

[115]:
```
import numpy as np

def binding_energy(A, Z):
    """Returns the nuclear binding energy in MeV of an atomic nucleus with␣
 ↪atomic number Z and mass number A"""
    aV = 14.64
    aS = 14.08
    aC = 0.64
    aA = 21.07
    aP = 11.54
```

```
    delta = 0

    if A%2 != 0:
        delta = 1
    if A%2 != 0 and Z%2 !=0:
        delta = 1
    if A%2 == 0 and Z%2 != 0:
        delta = -1

    a = aV * A
    b = aS * A**(2/3)
    c = aC * Z**2/(A**(1/3))
    d = aA * (A - 2*Z)**2 / A
    e = delta * aP / (A**.5)

    return a - b - c - d + e

binding_energy(58, 28)
```

[115]: 507.0722141579769

[116]:
```
def binding_energy_per_nucleon(A, Z):
    """Returns the nuclear binding energy per nucleon in MeV of an atomic␣
↪nucleus with atomic number Z and mass number A"""
    return binding_energy(A, Z) / A
```

[118]:
```
def max_binding_energy_per_nucleon(Z):
    """For atomic nucleus with atomic number Z, returns that mass number A that␣
↪yields that maximum binding energy
        per nucleon, as well as that resultant maximum binding energy per␣
↪nucleon in MeV"""

    #We can make our default return value A = Z and the corresponding binding␣
↪energy
    max_A = Z
    max_binding_energy_per_nucleon = binding_energy_per_nucleon(Z, Z)

    #Use a for loop to go from A = Z to A = 3*Z, and update the return␣
↪variables if a new maximum is found
    #A conditional statement within the loop is useful for comparing␣
↪max_binding_energy_per_nucleon to a potential new maximum

    for i in range (z+1, 3*z+1):
        maxima = binding_energy_per_nucleon(A, Z)
        if maxima > max_binding_energy_per_nulceon:
            max_binding_energy_per_nucleus = maxima
```

```
            Max_A = A
    #I was able to fix my own code based off of what I saw in Kenny's Office␣
 ↪Hours

    return max_A, max_binding_energy_per_nucleon
```

[123]:
```
#Now use a for loop and the function max_binding_energy_per_nucleon(Z) to print␣
 ↪the final output
global_max_binding_energy = 0
max_A, max_binding_energy_nucleon = 0, 0
for Z in range(1, 101):
    A, max_binding_energy_nucleon =  max_binding_energy_nucleon(Z)
    if global_max_binding_energy > max_binding_energy_nucleon:
        max_binding_energy_nucleon = global_max_binding_energy
    print('Z = {:.1f} : most stable A is {:.2f}'.format(Z, A))
print('The most stable Z is {:.1f} with binding energy per nucleon {:.2f}'.
 ↪format(100,max_binding_energy_nucleon ))

# I was unsure of how to produce this, I tried to follow what we did in office␣
 ↪hours but I was still confused
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-123-4567d277c787> in <module>
      3 max_A, max_binding_energy_nucleon = 0, 0
      4 for Z in range(1, 101):
----> 5     A, max_binding_energy_nucleon =  max_binding_energy_nucleon(Z)
      6     if global_max_binding_energy > max_binding_energy_nucleon:
      7         max_binding_energy_nucleon = global_max_binding_energy

TypeError: 'int' object is not callable
```

1.3 Problem 3: Particle in a Box

[Adapted from Ayars, Problem 3-1] The energy levels for a quantum particle in a three-dimensional rectangular box of dimensions $\{L_1, L_2,$ and $L_3\}$ are given by

$$E_{n_1,n_2,n_3} = \frac{\hbar^2\pi^2}{2m}\left[\frac{n_1^2}{L_1^2} + \frac{n_2^2}{L_2^2} + \frac{n_3^2}{L_3^2}\right]$$

where the n's are integers greater than or equal to one. Your goal is to write a program that will calculate, and list in order of increasing energy, the values of the n's for the 10 lowest *different* energy levels, given a box for which $L_2 = 2L_1$ and $L_3 = 4L_1$.

Your program should include two user-defined functions that you may find helpful in accomplishing your goal:

1. A function `energy(n1, n2, n3)` that takes integer values n_1, n_2, and n_3, and computes the corresponding energy level in units of $\hbar^2\pi^2/2mL_1^2$.
2. A function `lowest_unique_K(K, List)` which takes a positive integer K and a list of real numbers `List`, and returns an ordered (ascending) list of the lowest K unique numbers in the list `List`. For instance, `lowest_unique_K(3, [-0.5, 3, 3, 2, 6, 7, 7])` would return `[-0.5, 2, 3]`. The function should not modify the original list `List`.
    - As with most programming puzzles, there are several ways to write this function. Depending on how you do it, you may or may not find it helpful to Google how to "sort" lists, or how to "del" or "pop" items out of lists.

You may also wish to make other user-defined functions depending on how you go about solving the problem. In fact, if you find some clever way to solve the problem that doesn't use `lowest_unique_K`, that is fine too! (You still need to write `lowest_unique_K`, though.) But whatever you do, be sure to comment your code clearly!

Output Your final output should look like this (though with different numbers, and not necessarily the same number of lines):

```
energy, n1, n2, n3
(0.4375, 1, 1, 1)
(0.625, 1, 2, 1)
(0.8125, 2, 1, 1)
(0.9375, 1, 3, 1)
(1.0, 2, 2, 1)
(1.1875, 1, 1, 2)
(1.3125, 2, 3, 1)
(1.375, 1, 2, 2)
(1.375, 1, 4, 1)
(1.4375, 3, 1, 1)
(1.5625, 2, 1, 2)
```

Notice how there are only 10 unique energies listed, but more than 10 lines. Each line could also have brackets instead of parentheses if you prefer, like this: `[0.4375, 1, 1, 1]`.

Reminder Remember to write informative doc strings, comment your code, and use descriptive function and variable names so others (and future you) can understand what you're doing!

Just for fun If you'd like, write a function `print_table(list_of_lists)` that takes a list of lists (or a list of tuples) and prints them in a nicely aligned table. Feel free to Google to get ideas on how to do this. Try to get your function to produce something like

```
energy  n1 n2 n3
0.4375  1  1  1
0.625   1  2  1
0.8125  2  1  1
0.9375  1  3  1
1.0     2  2  1
1.1875  1  1  2
```

```
1.3125  2  3  1
1.375   1  2  2
1.375   1  4  1
1.4375  3  1  1
1.5625  2  1  2
```

[ ]:
```
## I was unable to finish this portion of the homework since I had a lot of␣
 ↪difficulty understanding and also
# midterms have me really busy :/

def energy(n1, n2, n3):
    """Returns the n-dependent coefficient of the particle-in-a-3D-box energy␣
 ↪level for quantum numbers n1, n2, and n3.
    The box's lengths along dimensions 1, 2, and 3 go as L, 2*L, 4*L"""
    return #Use the formula given above
```

[ ]:
```
import copy

def lowest_unique_K(K, List):
    """Takes a positive integer K and a list of real numbers List, and returns␣
 ↪an ordered (ascending) list of the lowest K unique numbers in the list␣
 ↪List"""
    lowest_unique_K_list = [0] * K  #This is a list of zeros (K of them)
    #Or you may want to start with lowest_unique_K_list = [], an empty list

    copied_list = copy.copy(List)  #This gives us a copy of List; any changes␣
 ↪you make to copied_list will not affect List

    #There's a lot of different ways to approach writing this function
    #Try breaking it up into smaller steps and figure out what you'd like to do␣
 ↪before writing any code
    #If you have trouble turning logical steps into actual code, feel free to␣
 ↪ask for help

    return lowest_unique_K_list
```

[ ]:
```
#Now create a list of energies for different values of n1, n2, n3 (taking each␣
 ↪from 1 to 10 should be sufficient)
#Remember to keep track of the corresponding n1, n2, n3 values for each energy,␣
 ↪since we need to print them
#Then use lowest_unique_K(10, List) on this list of energies to find the first␣
 ↪10
#Finally, print these 10 energies and their corresponding n1, n2, n3 values
#You may find a dictionary helpful for keeping track of the association between␣
 ↪energy values and n values
```