index

January 9, 2022

1 Permutations and Factorials - Lab

1.1 Introduction

Before, we saw how the creation of a sample space is crucial in finding probabilities. The issue, however is that, when the sample space grows bigger, it is not straightforward to manually compute the size of sample sets anymore.

Luckily, probability theory provides us with several formulas that can help us out. One set of formulas is known as **permutations**. This lab will help you get a better understanding of permutations, and provide practice!

1.2 Objectives

You will be able to:

- Mathematically derive how many permutations there are for large sets
- Calculate permutations of a subset
- Calculate permutations with repetition and replacement

1.3 A Note on Factorials

In the last lesson, we talked about permutations in the context of a coverband creating a setlist. We wanted to calculate how many ways we can order 3 songs in their setlist. We can use factorials for that. For 3 songs, this boils down to

[1]:
```
setlist = 3*2*1
```

Now, writing this out is not an issue when n is small. What if n grows though? Imagine there are 10 songs in the setlist

[2]:
```
setlist = 10*9*8*7*6*5*4*3*2*1
```

You wouldn't want to repeat this for 25 songs… Let's create a function for this!

What you'll do below is:

- create a function that takes one argument, n
- initialize `prod` as 1
- next, use `prod` in a `while` loop (what is the stopping criterion?)
- update n so it decreases with value 1 each iteration. This way you essentially calculate $n * (n-1) * (n-2) * ... * (1)$

[7]:
```
def factorial(n):
    prod = 1
    while n >= 1:
        prod *= n
        n -= 1
    return prod
```

Now, test your function with n=20

[10]:
```
factorial(20)
```

[10]:
```
2432902008176640000
```

Just so you know, Python has a built-in function `factorial` in the `math` library as well! Let's use our own function in this lab, but just use the `math` function once to check your previous answer!

[11]:
```
import math

math.factorial(20)
```

[11]:
```
2432902008176640000
```

1.4 Some Practice on Permutations

Let's go back to the appointments exercise from the last lab. A teaching assistant is holding office hours so students can make appointments. She has 6 appointments scheduled today, 3 by male students and 3 by female students. How many ways are there to order the appointments, based on gender of the students? Just to clarify, we're looking for size of the sample space that lists possible orders like this:

FMFMFM MMMFFF FMFMMF …

From what you learned in the permutations lecture, you now have a more structured way of getting to the whole sample space!

Hint: a permutation with repetition is needed here, with formula $\frac{n!}{n_1!n_2!\ldots n_k!}$. Think carefully of what needs to go in the denominator and the numerator respectively.

[12]:
```
app_num = factorial(6)
app_num
```

[12]:
```
720
```

[13]:
```
app_denom = factorial(3) * factorial(3)
app_denom
```

[13]:
```
36
```

[14]:
```
app_total = app_num / app_denom
app_total
```

[14]:
```
20.0
```

1.5 Permutations: Hack a Phone

You misplaced your iPhone and are afraid it was stolen. Luckily, your iPhone needs a 4-digit code in order to get in. Imagine that a potential thief can do five attempts at getting the code right before the phone is permanently locked, how big is the chance the thief unlocks the phone?

Think about the sample space and the event space separately. You'll use the formula $P(E) = \frac{|E|}{|S|}$ here.

So what should go in the denominator?

[15]:
```
denom_phone = 10**4
denom_phone
```

[15]:
```
10000
```

And the numerator?

[16]:
```
numer_phone = 5
numer_phone
```

[16]:
```
5
```

[17]:
```
prob_unlock = numer_phone / denom_phone
prob_unlock
```

[17]:
```
0.0005
```

Right before you lost your phone you ate a pretzel, and you are pretty sure a grease pattern was left on the four crucial digits of your screen. The four letters in your access code are 3,4,7 and 8, and you realize that this information can increase the thief's chances massively. Assuming the thief interprets the smudgemarks in an intelligent way, what are the chances that the phone will be unlocked successfully?

[18]:
```
denom_phone_smudge = factorial(4) #or math.factorial(4)
denom_phone_smudge
```

[18]:
```
24
```

[20]:
```
numer_phone_smudge = 5
numer_phone_smudge
```

[20]:
```
5
```

[21]:
```
prob_unlock_smudge = numer_phone_smudge / denom_phone_smudge
prob_unlock_smudge
```

[21]:
```
0.20833333333333334
```

Now, imagine you chose an iphone access code containing 3 different numbers, with numbers 2,7 and 8 (the code is still 4 digits). Now, the thief knows 1 number was reused (permutations with repetition!), but he doesn't know which one. what is the probability now that the phone will be unlocked successfully?

- For the denominator here, use a permutation with repetition, along with the fact that **you don't know which one is repeated**. Hint: you'll have to multiply your final "permutation with repetition"-result to account for that.
- For the numerator, use the numerator you used before: the number of trials the thief can try before the phone access is blocked.

[25]:
```
denom_phone_smudge_2 = factorial(4)/factorial(2) * 3 #or use math.factorial(4),
                                                     #2 identical numbers,
                                                     #3 possible reused numbers


denom_phone_smudge_2
```

[25]:
```
36.0
```

[26]:
```
numer_phone_smudge_2 = 5
numer_phone_smudge_2
```

[26]:
```
5
```

[27]:
```
prob_unlock_smudge_2 = numer_phone_smudge_2 / denom_phone_smudge_2
prob_unlock_smudge_2
```

[27]:
```
0.1388888888888889
```

1.6 Permutations to Find the Sample and Event Space

What are the odds of throwing a "full house" when throwing 5 dice? Recall, a full house means that you'd throw a three of a certain number along with a pair of a different number.

1.6.1 a) Sample space

First, calculate the sample space. Recall that replacement is possible here.

[29]:
```
sample_space_fh = 6**5
sample_space_fh
```

[29]:
```
7776
```

1.6.2 b) Event space

Next, calculate the event space. The best way to think of the event space here is to split it up in 2 parts: - first, try to constrain your problem to a more specific problem, let's say, how many ways can we throw a full house if we have a pair of 4s and three 6s? - next, extend your problem by asking yourself how many *different* full houses are possible. - multiply the two!

[31]:
```
ways_to_throw_given_fh = factorial(5)/(factorial(3) * factorial(2)) #␣
 ↪permutation with repetitions
ways_to_throw_given_fh
```

[31]:
```
10.0
```

[32]:
```
diff_fhses = 6 * 5
diff_fhses
```

[32]:
```
30
```

Then the event space is

[33]:
```
event_space_fh =  ways_to_throw_given_fh * diff_fhses
event_space_fh
```

[33]:
```
300.0
```

[ ]:
```
## From GitHub


# Note on this solution:
#
# There are 10 different ways of throwing a full house of a particular kind.
# This is a permutation with repetition. Recall the TENNESSEE example in the␣
 ↪lesson.
# assuming you have 2 x 4 and 3 x 6, how many ways can we throw this?
#
# 44666
# 46446
# 44646
#
# The formula used here is, again ((n!)/(n_1! n_2! ... n_k!)) = (6!)(2!3!)$
#
# 30 types of full houses exist: you're using the formula for permutation of a␣
 ↪subset:
# with 6 possible outcomes when throwing a dice, you want to make sure you only␣
 ↪pick
# 2 outcomes, hence using the formula P_{k}^{n}= n!/(n-k)! = (6!)/(6-2)!.
# Note that order matters! a full house with 3 x 6 and 2 x 5 is not the same as␣
 ↪a full
```

```
# house with 3 x 5 and 2 x 6.
#
# When you multiply these two together, you get to the event space!
```

1.6.3 c) Probability of full house

[34]:
```
prob_fh = event_space_fh / sample_space_fh

prob_fh
```

[34]:
```
0.038580246913580245
```

1.7 Level-Up: Factorials and Recursion

Let's return to factorials for a moment. In the last lesson, we talked about permutations in the context of a coverband creating a setlist. We wanted to know how many ways we could order 3 songs in their setlist. There were 3 possible ways of choosing a first song, 2 possible ways of choosing a second song, and only 1 way of choosing a third and final song, for $3*2*1=6$ different ways in which the three different songs could be played. This number, 6, is equal to the factorial of 3, 3!, the number of permutations of 3 distinct objects.

Here, $3! = (3*2*1) = 6$. Notice that this is the same as writing $3*2! = 3*(2*1)$ and $3*2*1! = 3*2*(1)$. (By definition, the factorial of 1, 1!, is equal to 1. The factorial of 0, 0! is also defined to be equal to 1.)

We can generalize this to the case of computing the factorial of an integer n, $n!$. The factorial of n, $n!$, can be written as $n*(n-1)!$, which itself can be written as $n*(n-1)*(n-2)!$. That is, we can define the factorial of n in terms of the product of n by the factorial of (n-1), and so on and so forth, as seen in the equation below:

$$n! = n*(n-1)! = n*(n-1)*(n-2)! = ... = n*(n-1)*(n-2)*...*1! = n*(n-1)*(n-2)*...*2*1$$

Earlier in this lab, we defined a Python function, `factorial(n)`, to compute the factorial of an integer n. In that case, we used a `while` loop with a stopping criterion to obtain a result.

However, there is another way we could have defined this function, using the **recursive** nature of the factorial function.

1.7.1 Recursion

When we define a function in terms of itself, in this case, the factorial of n in terms of the factorial of (n-1), we are using **recursion**. Recursive functions are functions that can call themselves in order to loop until a condition is met.

We can use recursion to define a function that will return the factorial of an integer n as follows:

```
def factorial_recursion(n):
    if n == 1:
```

```
        return 1
    else:
        return n * factorial_recursion(n-1)
```

Let's go over what happens with this function for the case n = 3:

- To start, since n is not equal to 1, we skip the `if` statement and continue to the `else` statement, where we obtain that `factorial_recursion(3) = 3 * factorial_recursion(2)`.
- To calculate `factorial_recursion(2)`, since the argument passed to the function is not equal to 1, we once again skip the `if` statement and continue to the `else` statement, where we obtain that `factorial_recursion(2) = 2 * factorial_recursion(1)`.
  - At this point, `factorial_recursion(3) = 3 * (2 * factorial_recursion(1))`
- To calculate `factorial_recursion(1)`, since the argument passed to the function *is* equal to 1, we return 1.
  - At this point, `factorial_recursion(3) = 3 * 2 * 1`, and our code returns the answer we expect, 6.

Try it out in the code cell below!

[37]:
```
# Uncomment the lines of code below:

def factorial_recursion(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

factorial_recursion(3)
```

[37]:
```
6
```

As a last step, check that the result you obtain with the `factorial_recursion` function for n = 3 is the same as the result you obtain with the `factorial` function defined earlier in this lab:

[40]:
```
assert factorial(3) == factorial_recursion(3)
```

Good job!

1.8 Summary

In this lab, you got quite some practice with permutations and factorials, and were even able to use it to calculate probability. You also had a gentle introduction to recursive functions in Python. You'll have a chance to learn much more about recursion in Python and solve some problems using recursion in the Appendix to this Module.

Now, we'll move over to another concept in combinatorics: combinations.