KIT103/KMA155 Practical 10: Permutations

This week you'll predict, count and generate permutations, both with and without replacement.

Preparation

Start Spyder and save the starting .py file under a new name in which to store some of the reusable functions you produce today. Add `from itertools import product` to your file so that the `product()` function is available.

1 Permutations with replacement

Permutations of reusable symbols are simpler, or at least less constrained, than permutations of 'physical' items. Let's generate some strings from a set of characters. In each example, work out how many permutations will be produced (at least as an expression, if not as an actual number).

Task: How many two-letter words are possible in English? We'll ignore whether the words exist or could be spoken.

1. First, what is *n*? What is *k*, the length of the permutation?
2. What is the expected number of permutations? Work it out on paper or in your head then enter the expression into Spyder to calculate the value.
3. To make things easier later use the following to get a string of lower case letters.

```
from string import ascii_lowercase
alphabet = ascii_lowercase[:26]
```

4. Using the lecture notes as a guide, write a set comprehension to generate the set of potential two-letter words. Try a 'manual' approach using constructs like `for a in alphabet`. Check the sets size with `len()`.
5. Now try generating the same set comprehension using the `product` function from the `itertools` module.

   Tips: You can pass a string (like `alphabet`) as the first argument to `product` and it will create the Cartesian product of its characters; use the named argument `repeat` to tell `product` how long to make the tuples (permutations) it produces.

1.1 Strange creatures

DNA is a sequence of nucleic acid base pairs, which are often given the abbreviated names A, T, G and C (short for adenine, thymine, guanine and cytosine). From an information perspective, an organism's DNA can be described by a sequence of these letters. Let's generate chromosomes for extremely simple creatures (so simple that they couldn't possibly exist, but one has to start one's meddling with Nature somewhere).

Task: Enumerate all the different possible DNA sequences of length nine (for your interest, there is a reason that the length is a multiple of three).

1. What is *n*? What is *k*?
2. How many sequences do you expect?
3. Generate them using a set comprehension and the `product()` function; assign the result to a variable and use `len()` to see if it contains the number you expected.

   Tip: *Do not* actually display the entire set.

2 Permutations without replacement

Permutations of things with 'identity' (people, books, rocks, anything physical or limited in number) are a little more complex than permutations of symbols. Because each item can be used only once the number of permutations is reduced. But by how much?

Task: Implement the `p_count()` function from the lecture and use it in the remainder of today's class to answer the question of how many *r*-permutations are possible.

2.1 Two-letter words with distinct letters

Before proceeding, modify the import statement for product to add `permutations` to the list of things imported from `itertools`, as in:

```
from itertools import product, permutations
```

1. When generating two-letter words without reusing letters, what is *n* and what is *r*?
2. How many such words are possible? What's the mathematical expression? What's its value? After you've worked it out on paper, or in your head, use the `p_count` function to check your answer. **Is it smaller or larger than the number of all possible two-letter words?**
3. Using the lecture notes as a guide, write a set comprehension that uses the `permutations` function to generate this set. **How big is the generated set?**

Note that the result of adapting the example in the lecture slides will be a set of tuples (which is perfectly valid), but if you actually want to generate a string for each one then you'll need to use the trick of `''.join(p)`, where `''` is an empty string (no characters between two single quotes) and `p` is the tuple whose elements you want to join into one string.

2.2 Who's in first?

Imagine you are a keen Dungeons & Dragons player and you are currently playing a campaign with three other friends. In the game your party of four explorers has reached a narrow passage between two sheer cliffs. While you are exploring the passage the people at the front and back are most vulnerable to attack. In what order should you send people in? (We will only partially answer this question.)

Task: Imagine that your party's members are Alice, Bob, Charlie and Daphne. How many possible permutations of these names are there?
Write code to generate them.

Task: Since only the people at the front and back are vulnerable, do you actually need to generate all possible permutations? Could you generate fewer permutations that indicate who's at the front and back (with the order in the middle left undecided, or to chance)?
Write a set comprehension to do this.

3 (optional) Challenge exercise: Recursively generating permutations

If you'd like a challenge, try implementing the general recursive approach to generating permutations given in the lecture. What change would you need to make for it to produce *r*-permutations (i.e., permutations of shorter length than the number of items?

Note that if you try to implement this then you should use `result.add( tuple(p) )` to take a copy of the permutation in the list `p` and add it to the `result` set (which can contain tuples but not lists).

Notes

permutations()

The Python Standard Library includes a flexible function for generating *r*-length permutations of elements, called, unsurprisingly, `permutations`:
https://docs.python.org/3/library/itertools.html#itertools.permutations It is also part of the `itertools` module, like `combinations`, which can be used to generate combinations (next week).

In an exam setting you'll likely be asked to write a set comprehension to generate permutations (to demonstrate that you understand how they are constructed), but in the future when writing your own code that needs to generate permutations, use the `permutations` function.