Lists II

The lists are so useful that you have to know how to handle them in a simple and efficient way. That's the purpose of this chapter!

Lesson 1 (Manipulate lists efficiently).

- **Slicing lists.**
  - You already know `mylist[a:b]` that returns the sublist of elements from the rank a to the rank $b-1$.
  - `mylist[a:]` returns the list of elements from rank a until the end.
  - `mylist[:b]` returns the list of elements from the beginning to the rank $b-1$.
  - `mylist[-1]` returns the last element, `mylist[-2]` returns the penultimate element, ...
  - **Exercise.**

	7	2	4	5	3	10	9	8	3
rank :	0	1	2	3	4	5	6	7	8

    With `mylist = [7,2,4,5,3,10,9,8,3]`, what do the following instructions return?
    - `mylist[3:5]`
    - `mylist[4:]`
    - `mylist[:6]`
    - `mylist[-1]`
- **Find the rank of an element.**
  - `mylist.index(element)` returns the first position at which the item was found. Example: with `mylist = [12, 30, 5, 9, 5, 21]`, `mylist.index(5)` returns 2.
  - If you just want to know if an item belongs to a list, then the statement:

    `element in mylist`

    returns `True` or `False`. Example: with `mylist = [12, 30, 5, 9, 5, 21]`, "`9 in mylist`" is true, while "`8 in mylist`" is false.
- **List comprehension.**
  A set can be defined by listing all its elements, for example $E = \{0, 2, 4, 6, 8, 10\}$. Another way is to say that the elements of the set must verify a certain property. For example, the same set E can be defined by:
  $$E = \{x \in \mathbb{N} \mid x \leqslant 10 \text{ and } x \text{ is even}\}.$$
  With `Python` there is a way to define lists this way. It is an extremely powerful and efficient syntax. Let's look at some examples:
  - Let's start from a list, for example `mylist = [1,2,3,4,5,6,7,6,5,4,3,2,1]`.

  - The command `mylist_doubles = [ 2*x for x in mylist ]` returns a list that contains the double of each item of the `mylist` list. So this is the list `[2,4,6,8,...]`.
  - The command `mylist_squares = [ x**2 for x in mylist ]` returns the list of squares of the items in the initial list. So this is the list `[1,4,9,16,...]`.
  - The command `mylist_partial = [x for x in mylist if x > 2]` extracts the list composed only of elements greater than 2. So this is the list `[3,4,5,6,7,6,5,4,3]`.
- **List of lists.**
  A list can contain other lists, for example:
  ```
  mylist = [ ["Harry", "Hermione", "Ron"], [101,103] ]
  ```
  contains two lists. We will be interested in lists that contain lists of integers, which we will call *arrays*. For example:
  ```
  array = [ [2,14,5], [3,5,7], [15,19,4], [8,6,5] ]
  ```
  Then `array[i]` returns the sublist of index i, while `array[i][j]` returns the integer located at the index j in the sublist of index i. For example:
  - `array[0]` returns the list `[2,14,5]`,
  - `array[1]` returns the list `[3,5,7]`,
  - `array[0][0]` returns the integer 2,
  - `array[0][1]` returns the integer 14,
  - `array[2][1]` returns the integer 19.

Activity 1 (Lists comprehension).

Goal: practice list comprehension. In this activity the lists are lists of integers.

1. Program a `multiplication(mylist,k)` function that multiplies each item in the list by k. For example, `multiplication([1,2,3,4,5],2)` returns `[2,4,6,8,10]`.
2. Program a `power(mylist,k) function` that raises each element of the list to the power k. For example, `power([1,2,3,4,5],3)` returns `[1,8,27,64,125]`.
3. Program an `addition(mylist1,mylist2)` function that adds together the elements of two lists of the same length. For example, `addition([1,2,3],[4,5,6])` returns `[5,7,9]`.
   Hint. This is an example of a task where list comprehension is not used!
4. Program a `non_zero(mylist)` function that returns a list of all non-zero elements. For example, `non_zero([1,0,2,3,0,4,5,0])` returns `[1,2,3,4,5]`.
5. Program an `even(mylist)` function that returns a list of all even elements. For example, `even([1,0,2,3,0,4,5,0])` returns `[0,2,0,4,0]`.

Activity 2 (Reach a fixed amount).

Goal: try to reach the total of 100 *in a list of numbers.*

We consider a list of n integers between 1 and 99 (included). For example, this list of 25 integers:
```
[16,2,85,27,9,45,98,73,12,26,46,25,26,49,18,99,10,86,7,42]
```
which was obtained at random by the command:
```
mylist_20 = [randint(1,99) for i in range(20)]
```
We are looking for different ways to find numbers from the list whose sum is exactly 100.

1. Program a `sum_twoinarow_100(mylist)` function that tests if there are two consecutive elements in the list whose sum is exactly 100. The function returns `True` or `False` (but it can also

display numbers and their position for verification). For the example given, this function returns `False`.

2. Program a `sum_two_100(mylist)` function that tests if there are two items in the list, located at different positions, whose sum is equal to 100. For the example given, this function returns `True` and can display the integers 2 and 98 (at ranks 1 and 6 of the list).
3. Program a `sum_seq_100(mylist)` function that tests if there are consecutive elements in the list whose sum is equal to 100. For the example given, this function returns `True` and can display the sequence of integers 25, 26, 49 (at ranks 11, 12 and 13).
4. *(Optional.)* The larger the size of the list, the more likely it is to have values in the list whose sum is 100. For each of the three previous situations, determine the size n of the list such that the probability of obtaining a sum of 100 is greater than 1/2.
   Hints. For each case, you can get an estimate of this integer n, by writing a `proba(n,N)` function that performs a large number N of random draws of lists having n items ($N = 10\,000$ for example). The probability is approximated by the number of successful cases (where the function returns `True`) divided by the total number of cases (here N).

Activity 3 (Arrays).

Goal: working with lists of lists.

In this activity we work with arrays of size $n \times n$ containing integers. The object `array` is therefore a list of n lists, each having n elements.
For example ($n = 3$):

```
array = [ [1,2,3], [4,5,6], [7,8,9] ]
```

represents the array:

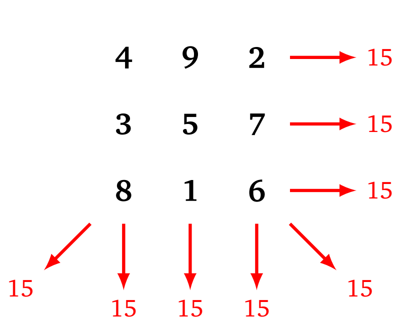
$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

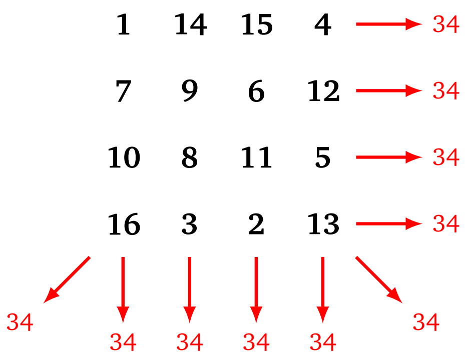
1. Write a `sum_diagonal(array)` function that calculates the sum of the elements located on the main diagonal of an array. The main diagonal of the example given is 1, 5, 9, so the sum is 15.
2. Write a `sum_antidiagonal(array)` function that calculates the sum of the elements located on the other diagonal. The anti-diagonal of the example given is composed of 3, 5, 7, the sum is still 15.
3. Write a `sum_all(array)` function that calculates the total sum of all elements. In this example the total sum is 45.
4. Write a `print_array(array)` function that displays an array properly on the screen. You can use the command:
   ```
   print('{:>3d}'.format(array[i][j]), end="")
   ```
   Explanations.
   - The command `print(string,end="")` allows you to display a string of characters without going to the next line.
   - The command `'{:>3d}'.format(k)` displays the integer k on three characters (even if there is only one digit to display).

Activity 4 (Magic Squares).

Goal: build magic squares as big as you want! You must first have done the previous activity.

A *magic square* is a square array of size $n \times n$ that contains all the integers from 1 to n^2 and satisfies that: the sum of each row, the sum of each column, the sum of the main diagonal and the sum of the anti-diagonal all have the same value.
Here is an example of a magic square with a size of 3×3 and one of size 4×4.





For a magic square of size $n \times n$, the value of the sum is:

$$S_n = \frac{n(n^2+1)}{2}.$$

1. **Examples.** Define an array for each of the 3×3 and 4×4 examples above and display them on the screen (use the previous activity).
2. **To be or not to be.** Define an `is_magic_square(square)` function that tests whether a given array is (or isn't) a magic square (use the previous activity for diagonals).
3. **Random squares.** *(Optional.)* Randomly generate squares containing integers from 1 to n^2 using a `random_square(n)` function. Experimentally verify that it is rare to obtain a magic square in this way!
   Hints. For a list `mylist`, the command `shuffle(mylist)` (from the `random` module) randomly mixes the list (the list is modified in place).

   The purpose of the remaining questions is to create large magic squares.
4. **Addition.** Define an `addition_square(square,k)` function which adds an integer k to all the elements of the array. With the example of the 3×3 square, the command `addition_square(square,-1)` subtracts 1 from all the elements and returns an array that would look like this:

   ```
   3  8  1
   2  4  6
   7  0  5
   ```

   Hints. To define a new square, start by filling it with 0's:

   ```
   new_square = [[0 for j in range(n)] for i in range(n)]
   ```

   then fill it with the correct values using commands of the type:

   ```
   new_square[i][j] = ...
   ```
5. **Multiplication.** Define a `multiplication_square(square,k)` function which multiplies all the elements of the array by k. With the example of the 3×3 square, the `multiplication_square(square,2)` command multiplies all the elements by 2 and thus returns an array that would be displayed as follows:

   ```
    8  18   4
    6  10  14
   16   2  12
   ```

6. **Homothety.** Define a `homothety_square(square,k)` function which enlarges the array by a factor of k as shown in the examples below. Here is an example of the 3×3 square with a homothety ratio of $k = 3$.

4	9	2
3	5	7
8	1	6

$\longrightarrow$

4	4	4	9	9	9	2	2	2
4	4	4	9	9	9	2	2	2
4	4	4	9	9	9	2	2	2
3	3	3	5	5	5	7	7	7
3	3	3	5	5	5	7	7	7
3	3	3	5	5	5	7	7	7
8	8	8	1	1	1	6	6	6
8	8	8	1	1	1	6	6	6
8	8	8	1	1	1	6	6	6

Here is an example of a 4×4 square with a homothety ratio of $k = 2$.

1	14	15	4
7	9	6	12
10	8	11	5
16	3	2	13

$\longrightarrow$

1	1	14	14	15	15	4	4
1	1	14	14	15	15	4	4
7	7	9	9	6	6	12	12
7	7	9	9	6	6	12	12
10	10	8	8	11	11	5	5
10	10	8	8	11	11	5	5
16	16	3	3	2	2	13	13
16	16	3	3	2	2	13	13

7. **Block addition.** Define a `block_addition_square(big_square,small_square)` function that adds a small array of size $n \times n$ to each block of the larger $nm \times nm$ sized array as shown in the example below with $n = 2$ and $m = 3$ (hence $nm = 6$). The small 2×2 square on the left is added to the large square in the center to give the result on the right. For this addition the large square is divided into 9 blocks, there is a total of 36 additions.

1	2
3	4

4	4	9	9	2	2
4	4	9	9	2	2
3	3	5	5	7	7
3	3	5	5	7	7
8	8	1	1	6	6
8	8	1	1	6	6

$\longrightarrow$

5	6	10	11	3	4
7	8	12	13	5	6
4	5	6	7	8	9
6	7	8	9	10	11
9	10	2	3	7	8
11	12	4	5	9	10

8. **Products of magic squares.** Define a `product_squares(square1,square2)` function which from two magic squares, calculates a larger magic square called the product of the two squares. The algorithm is as follows:

Algorithm.

- - Inputs: a magic square C_1 of size $n \times n$ and a magic square C_2 of size $m \times m$.
  - Output: a magic square C of size $(nm) \times (nm)$.
- Create square C_{3a} by subtracting 1 from all elements of C_2. (Use the `addition_square(square2,-1)` command.)
- Define square C_{3b} as the homothety of square C_{3a} of ratio n. (Use the `homothety(square3a,n) command`.)
- Define square C_{3c} by multiplying all the terms of square C_{3b} by n^2. (Use the `multiplication_square(square3b,n**2)` command.)
- Define square C_{3d} by adding square C_1 to square C_{3c} block by block. (Use the `block_addition_square(square3c,square1)` command.)
- Return the square C_{3d}.

- Implement this algorithm.
- Test it on examples, checking that the square obtained is indeed a magic square.
- Build a magic square of size 36×36.
- Also confirm that the order of the product is important: $C_1 \times C_2$ is not the same square as $C_2 \times C_1$.