-	-	-
Exercise 1 (column comparison)	Exercise 2 (first half second half)	Exercise 3 (most frequent first)
Exercise 4 (matrix power)	Exercise 5 (correlation)	Exercise 6 (meeting lines)
Exercise 7 (meeting planes)	Exercise 8 (almost meeting lines)	

NumPy (continues)

In [ ]:
```
import numpy as np
```

Comparisons and masking

Just like NumPy allows element-wise arithmetic operations between arrays, for example addition of two arrays, it is also possible to compare two arrays element-wise. For example

In [ ]:
```
a=np.array([1,3,4])
b=np.array([2,2,7])
c = a < b
print(c)
```

```
[ True False  True]
```

Now we can query whether all comparisons resulted `True`, or whether some comparison resulted `True`:

In [ ]:
```
print(c.all())   # were all True
print(c.any())   # was some comparison True
```

```
False
True
```

We can also count the number of comparisons that were `True`. This solution relies on the interpretation that `True` corresponds to 1 and `False` corresponds to 0:

In [ ]:
```
print(np.sum(c))
```

```
2
```

Because the broadcasting rules apply also to comparison, we can write

In [ ]:
```
print(a > 0)
```

```
[ True  True  True]
```

To try these operations on real data, we download some weather statistics from Helsinki (Kumpula) weather station. More precisely, we will get the daily average temperatures from year 2017. We use the Pandas library, which we will cover later during this course, to load the data.

In [ ]:
```
import pandas as pd
a=pd.read_csv("https://raw.githubusercontent.com/csmastersUH/data_analysis_with_python_2020/master/kumpula-weather-2017.csv")['Air
```

In [ ]:
```
a
```

Out[ ]:
```
array([  0.6,  -3.9,  -6.5, -12.8, -17.8, -17.8,  -3.8,  -0.5,   0.5,
         1.7,  -1.6,  -2.8,   1.1,   0.8,  -2.8,  -4.2,  -3.5,   1.1,
         1.6,  -0.6,  -1.8,   1. ,   0.1,  -2.2,  -3.8,   1.9,   1.6,
         0.8,   0.6,   1. ,   0.2,  -0.6,  -0.8,  -0.2,   0.4,  -2.5,
        -7.3, -12.1,  -8.8, -10.1,  -8.3,  -5.4,  -2.7,   1.5,   4.4,
         0. ,   0.5,   1.5,   1.9,   2.2,   0.4,  -2.5,  -4.6,  -0.7,
        -5.3,  -5.6,  -2. ,  -2.3,   2.1,   1.5,   1.5,   0.9,  -1.8,
        -4.2,  -4.2,  -2.9,   0.2,   1.2,   1.4,   2.2,  -0.9,   0.6,
         0.6,   3.8,   3.4,   2. ,   1.6,   0.6,   0.2,   1.6,   2.3,
         2.5,   0.5,   1.9,   4.8,   6.6,   2.4,   0.4,  -0.4,   0.2,
         1.5,   4.3,   3. ,   3.8,   3.3,   4.7,   4.1,   3.3,   3.7,
         6. ,   4.3,   1.8,   1.2,  -0.5,  -1.4,  -1.7,   0.6,   0.3,
         1.2,   2.5,   5.7,   3.1,   2.9,   3.6,   2.8,   3.2,   4.4,
         4.1,   2.3,   2.2,   7.6,   9.4,   9.2,   6.7,  10.1,  11.1,
         6.1,   4.4,   1.5,   1.5,   2.9,   5.1,   7.5,   9.8,   9. ,
         7.3,   9.4,  11.2,  16.1,  16.9,  14.7,  14.6,  13.3,  12.4,
        14.1,  14.1,   9.6,  13.2,  14.1,   9.6,  10.8,   6.5,   6.2,
        10. ,   9.9,  10.4,  14.5,  16.4,  12.3,  14. ,  16.3,  16.8,
        12.6,  12.6,  14.8,  15.1,  18.4,  19.3,  16.9,  17.8,  12.7,
        12.4,  12.2,  10.8,  12.9,  14. ,  13.5,  13.7,  15.3,  16.3,
        17.2,  14.1,  16.4,  14.5,  14.1,  14. ,  11.5,  14.7,  16.1,
        17.4,  18. ,  16.8,  16.8,  14.7,  15.3,  16.7,  18. ,  15.4,
        15. ,  13.7,  14.6,  15.8,  15.6,  16.9,  15.7,  16.1,  17.2,
        19. ,  19.1,  17.8,  18.3,  17.8,  18.9,  17.5,  18.2,  16.6,
        17.1,  16. ,  15.2,  17.2,  18.3,  18.7,  18.1,  19.6,  17.4,
        16.1,  16.4,  17.6,  19. ,  18. ,  18.2,  17.4,  16.2,  14.9,
        13.5,  12.9,  11.2,  10.4,  10.7,  12. ,  14.4,  16.8,  17.2,
        15.1,  12.9,  12.9,  11.9,  11.4,  10.4,  10.3,  10.6,  13.1,
        14.6,  14.6,  14.9,  14.4,  12.9,  12.2,  12.1,   9.9,   8.7,
         9. ,   9.2,  12.4,  11.5,  12.8,  12.5,  12.6,  12. ,  12.1,
        10.1,   8.9,   8.8,   9.1,   9.2,   8.3,  11.2,   8.8,   7.7,
         8.1,   9.3,   8.6,   8.1,   6.9,   6. ,   7.5,   7.2,   8.3,
        10.7,   8.5,   8.3,   4.6,   2. ,   0.2,   0.1,   1.3,   0.8,
         2.1,   0.3,  -0.3,   3.3,   2.1,   1.2,  -0.4,   0.1,   0.5,
         3.2,   8. ,   8.4,   7.5,   3.9,   5.9,   6.7,   7.2,   5.9,
         3.1,   2.4,   1.8,   3.1,   3.5,   5.8,   5.9,   4. ,   1.2,
         0.9,   1. ,   1.5,   6.1,   4.2,   2.3,   4.6,   3.9,   2.8,
         3.1,   0.9,   1.4,   5. ,   1.3,   0. ,  -1.2,  -0.8,   5.2,
         4.2,   2. ,   1.4,   1.6,   1.6,   1.6,   1.7,   2.4,   0.1,
```

```
       2. ,   1. ,   2.6,   2.5,  -0.1,   1.2,  -0.3,   0.3,   1.9,
       3.8,   2.8,   3.8,   2.5,   1.6])
```

In [ ]:
```
print("Number of days with the temperature below zero", np.sum(a < 0))
```

```
Number of days with the temperature below zero 49
```

In core Python we can combine truth values using the `and`, `or`, and `not` keywords. For boolean array however we have to use the elementwise operators `&`, `|`, and `~`, respectively. An example of these:

In [ ]:
```
np.sum((0 < a) & (a < 10))     # Temperature is greater than 0 and less that 10
```

Out[ ]:
```
185
```

Note that we had to use parentheses around the comparisons, because the precedence of the `&` is higher than that of `<`, and so it would have been performed before the comparisons.

Another use of boolean arrays is that they can be used to select a subset of elements. For example

In [ ]:
```
c = a > 0
print(c[:10])    # print only the first ten elements
print(a[c])      # Select only the positive temperatures
```

```
[ True False False False False False False False  True  True]
[  0.6   0.5   1.7   1.1   0.8   1.1   1.6   1.    0.1   1.9   1.6   0.8
   0.6   1.    0.2   0.4   1.5   4.4   0.5   1.5   1.9   2.2   0.4   2.1
   1.5   1.5   0.9   0.2   1.2   1.4   2.2   0.6   0.6   3.8   3.4   2.
   1.6   0.6   0.2   1.6   2.3   2.5   0.5   1.9   4.8   6.6   2.4   0.4
   0.2   1.5   4.3   3.    3.8   3.3   4.7   4.1   3.3   3.7   6.    4.3
   1.8   1.2   0.6   0.3   1.2   2.5   5.7   3.1   2.9   3.6   2.8   3.2
   4.4   4.1   2.3   2.2   7.6   9.4   9.2   6.7  10.1  11.1   6.1   4.4
   1.5   1.5   2.9   5.1   7.5   9.8   9.    7.3   9.4  11.2  16.1  16.9
  14.7  14.6  13.3  12.4  14.1  14.1   9.6  13.2  14.1   9.6  10.8   6.5
   6.2  10.    9.9  10.4  14.5  16.4  12.3  14.   16.3  16.8  12.6  12.6
  14.8  15.1  18.4  19.3  16.9  17.8  12.7  12.4  12.2  10.8  12.9  14.
  13.5  13.7  15.3  16.3  17.2  14.1  16.4  14.5  14.1  14.   11.5  14.7
  16.1  17.4  18.   16.8  16.8  14.7  15.3  16.7  18.   15.4  15.   13.7
  14.6  15.8  15.6  16.9  15.7  16.1  17.2  19.   19.1  17.8  18.3  17.8
  18.9  17.5  18.2  16.6  17.1  16.   15.2  17.2  18.3  18.7  18.1  19.6
  17.4  16.1  16.4  17.6  19.   18.   18.2  17.4  16.2  14.9  13.5  12.9
  11.2  10.4  10.7  12.   14.4  16.8  17.2  15.1  12.9  12.9  11.9  11.4
  10.4  10.3  10.6  13.1  14.6  14.6  14.9  14.4  12.9  12.2  12.1   9.9
   8.7   9.    9.2  12.4  11.5  12.8  12.5  12.6  12.   12.1  10.1   8.9
   8.8   9.1   9.2   8.3  11.2   8.8   7.7   8.1   9.3   8.6   8.1   6.9
   6.    7.5   7.2   8.3  10.7   8.5   8.3   4.6   2.    0.2   0.1   1.3
   0.8   2.1   0.3   3.3   2.1   1.2   0.1   0.5   3.2   8.    8.4   7.5
   3.9   5.9   6.7   7.2   5.9   3.1   2.4   1.8   3.1   3.5   5.8   5.9
   4.    1.2   0.9   1.    1.5   6.1   4.2   2.3   4.6   3.9   2.8   3.1
   0.9   1.4   5.    1.3   5.2   4.2   2.    1.4   1.6   1.6   1.6   1.7
   2.4   0.1   2.    1.    2.6   2.5   1.2   0.3   1.9   3.8   2.8   3.8
   2.5   1.6]
```

This operation is called *masking*. It can also be used to assign a new value. For example the following zeroes out the negative temperatures:

In [ ]:
```
a[~c] = 0
print(a)
```

```
[  0.6   0.    0.    0.    0.    0.    0.    0.    0.5   1.7   0.    0.
   1.1   0.8   0.    0.    0.    1.1   1.6   0.    0.    1.    0.1   0.
   0.    1.9   1.6   0.8   0.6   1.    0.2   0.    0.    0.    0.4   0.
   0.    0.    0.    0.    0.    0.    0.    1.5   4.4   0.    0.5   1.5
   1.9   2.2   0.4   0.    0.    0.    0.    0.    0.    0.    2.1   1.5
   1.5   0.9   0.    0.    0.    0.    0.2   1.2   1.4   2.2   0.    0.6
   0.6   3.8   3.4   2.    1.6   0.6   0.2   1.6   2.3   2.5   0.5   1.9
   4.8   6.6   2.4   0.4   0.    0.2   1.5   4.3   3.    3.8   3.3   4.7
   4.1   3.3   3.7   6.    4.3   1.8   1.2   0.    0.    0.    0.6   0.3
   1.2   2.5   5.7   3.1   2.9   3.6   2.8   3.2   4.4   4.1   2.3   2.2
   7.6   9.4   9.2   6.7  10.1  11.1   6.1   4.4   1.5   1.5   2.9   5.1
   7.5   9.8   9.    7.3   9.4  11.2  16.1  16.9  14.7  14.6  13.3  12.4
  14.1  14.1   9.6  13.2  14.1   9.6  10.8   6.5   6.2  10.    9.9  10.4
  14.5  16.4  12.3  14.   16.3  16.8  12.6  12.6  14.8  15.1  18.4  19.3
  16.9  17.8  12.7  12.4  12.2  10.8  12.9  14.   13.5  13.7  15.3  16.3
  17.2  14.1  16.4  14.5  14.1  14.   11.5  14.7  16.1  17.4  18.   16.8
  16.8  14.7  15.3  16.7  18.   15.4  15.   13.7  14.6  15.8  15.6  16.9
  15.7  16.1  17.2  19.   19.1  17.8  18.3  17.8  18.9  17.5  18.2  16.6
  17.1  16.   15.2  17.2  18.3  18.7  18.1  19.6  17.4  16.1  16.4  17.6
  19.   18.   18.2  17.4  16.2  14.9  13.5  12.9  11.2  10.4  10.7  12.
  14.4  16.8  17.2  15.1  12.9  12.9  11.9  11.4  10.4  10.3  10.6  13.1
  14.6  14.6  14.9  14.4  12.9  12.2  12.1   9.9   8.7   9.    9.2  12.4
  11.5  12.8  12.5  12.6  12.   12.1  10.1   8.9   8.8   9.1   9.2   8.3
  11.2   8.8   7.7   8.1   9.3   8.6   8.1   6.9   6.    7.5   7.2   8.3
  10.7   8.5   8.3   4.6   2.    0.2   0.1   1.3   0.8   2.1   0.3   0.
   3.3   2.1   1.2   0.    0.1   0.5   3.2   8.    8.4   7.5   3.9   5.9
   6.7   7.2   5.9   3.1   2.4   1.8   3.1   3.5   5.8   5.9   4.    1.2
   0.9   1.    1.5   6.1   4.2   2.3   4.6   3.9   2.8   3.1   0.9   1.4
   5.    1.3   0.    0.    0.    5.2   4.2   2.    1.4   1.6   1.6   1.6
   1.7   2.4   0.1   2.    1.    2.6   2.5   0.    1.2   0.    0.3   1.9
   3.8   2.8   3.8   2.5   1.6]
```

Compare this result with the original array `a` to make sure you understand what's going on!

Exercise 1 (column comparison)

Write function `column_comparison` that gets a two dimensional array as parameter. The function should return a new array containing those rows from the input that have the value in the second column larger than in the second last column. You may assume that the input contains at least two columns. Don't use loops, but instead vectorized operations. Try out your function in the main function.

For array

```
[[8 9 3 8 8]
[0 5 3 9 9]
[5 7 6 0 4]
[7 8 1 6 2]
[2 1 3 5 8]]
```

the result would be

```
[[8 9 3 8 8]
[5 7 6 0 4]
[7 8 1 6 2]]
```

Exercise 2 (first half second half)

Write function `first_half_second_half` that gets a two dimensional array of shape `(n,2*m)` as a parameter. The input array has `2*m` columns. The output from the function should be a matrix with those rows from the input that have the sum of the first `m` elements larger than the sum of the last `m` elements on the row. Your solution should call the `np.sum` function or the corresponding method exactly twice.

Example of usage:

```
a = np.array([[1, 3, 4, 2],
              [2, 2, 1, 2]])
first_half_second_half(a)
array([[2, 2, 1, 2]])
```

Fancy indexing

Using indexing we can get a single elements from an array. If we wanted multiple (not necessarily contiguous) elements, we would have to index several times:

In [ ]:
```
np.random.seed(0)
a=np.random.randint(0, 20,20)
a2=np.array([a[2], a[5], a[7]])
print(a)
print(a2)
```

```
[12 15  0  3  3  7  9 19 18  4  6 12  1  6  7 14 17  5 13  8]
[ 0  7 19]
```

That's quite verbose. *Fancy indexing* provides a concise syntax for accessing multiple elements:

In [ ]:
```
idx=[2,5,7]           # List of indices
print(a[idx])         # In fancy indexing in place of a single index, we can provide a list of indices
print(a[[2,5,7]])     # Or directly
```

```
[ 0  7 19]
[ 0  7 19]
```

We can also assign to multiple elements through fancy indexing:

In [ ]:
```
a[idx] = -1
print(a)
```

```
[12 15 -1  3  3 -1  9 -1 18  4  6 12  1  6  7 14 17  5 13  8]
```

Fancy indexing works also for higher dimensional arrays:

In [ ]:
```
b=np.arange(16).reshape(4,4)
print(b)
row=np.array([0,2])
col=np.array([1,3])
print(b[row, col])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
[ 1 11]
```

Note that the result array was one dimensional! The shape of the result array is defined by the shape of the index arrays, not by the shape of the original array. The next example will demonstrate this.

In [ ]:
```
row2=np.array([[0, 0], [2,2]])
col2=np.array([[1,3], [1,3]])
print(row2)
print(col2)
print(b[row2, col2])
```

```
[[0 0]
 [2 2]]
[[1 3]
 [1 3]]
[[ 1  3]
 [ 9 11]]
```

Both index arrays, row and col, had shape (2,2), so also the result was of shape (2,2). However, this form has some repetition in the indices. But we can rely that the broadcasting rules will enable us to avoid the repetition:

In [ ]:
```
row2=row.reshape(2,1)   # has shape (2, 1)
col2=col.reshape(1,2)   # has shape (1, 2)
print(row2)
print(col2)
print(b[row2, col2])    # the index arrays will be broadcasted here to shape (2,2)
```

```
[[0]
 [2]]
[[1 3]]
[[ 1  3]
 [ 9 11]]
```

One can also combine normal indexing, slicing and fancy indexing:

In [ ]:
```
b[:,[0,2]]
```

Out[ ]:
```
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10],
       [12, 14]])
```

Sorting arrays

In [ ]:
```
a=np.array([2,1,4,3,5])
print(np.sort(a))          # Does not modify the argument
print(a)
```

```
[1 2 3 4 5]
[2 1 4 3 5]
```

In [ ]:
```
a.sort()            # Modifies the argument
print(a)
```

```
[1 2 3 4 5]
```

In [ ]:
```
b=np.random.randint(0,10, (4,4))
print(b)
```

```
[[9 4 3 0]
 [3 5 0 2]
 [3 8 1 3]
 [3 3 7 0]]
```

In [ ]:
```
np.sort(b, axis=0)           # sort each column
```

Out[ ]:
```
array([[3, 3, 0, 0],
       [3, 4, 1, 0],
       [3, 5, 3, 2],
       [9, 8, 7, 3]])
```

In [ ]:
```
np.sort(b, axis=1)           # Sort each row
```

Out[ ]:
```
array([[0, 3, 4, 9],
       [0, 2, 3, 5],
       [1, 3, 3, 8],
       [0, 3, 3, 7]])
```

Note that each row or column is sorted independently.

A related operation is the `argsort` function. Which doesn't sort the elements, but returns the indices of the sorted elements. An example will demonstrate this:

In [ ]:
```
a=np.array([23,12,47,35,59])
print("Array a:", a)
idx = np.argsort(a)
print("Indices:", idx)
```

```
Array a: [23 12 47 35 59]
Indices: [1 0 3 2 4]
```

These indices say that the smallest element of the array is in position 1 of `a`, second smallest elements is in position 0 of `a`, third smallest is in position 3, and so on. We can verify that these indices will indeed order the elements using fancy indexing:

In [ ]:
```
print(a[idx])
```

```
[12 23 35 47 59]
```

Exercise 3 (most frequent first)

Note: This exercise is fairly difficult. Feel free to skip if you get stuck.

Write function `most_frequent_first` that gets a two dimensional array and an index `c` of a column as parameters. The function should then return the array whose rows are sorted based on column `c`, in the following way. Rows are ordered so that those rows with the most frequent element in column `c` come first, then come the rows with the second most frequent element in column `c`, and so on. Therefore, the values outside column `c` don't affect the ordering in any way.

Example of usage:

```
a:
 [[5 0 3 3 7 9 3 5 2 4]
 [7 6 8 8 1 6 7 7 8 1]
 [5 9 8 9 4 3 0 3 5 0]
 [2 3 8 1 3 3 3 7 0 1]
 [9 9 0 4 7 3 2 7 2 0]
 [0 4 5 5 6 8 4 1 4 9]
 [8 1 1 7 9 9 3 6 7 2]
 [0 3 5 9 4 4 6 4 4 3]
 [4 4 8 4 3 7 5 5 0 1]
 [5 9 3 0 5 0 1 2 4 2]]
print(most_frequent_first(a, -1))
 [[4 4 8 4 3 7 5 5 0 1]
 [2 3 8 1 3 3 3 7 0 1]
 [7 6 8 8 1 6 7 7 8 1]
 [5 9 3 0 5 0 1 2 4 2]
 [8 1 1 7 9 9 3 6 7 2]
 [9 9 0 4 7 3 2 7 2 0]
 [5 9 8 9 4 3 0 3 5 0]
 [0 3 5 9 4 4 6 4 4 3]
 [0 4 5 5 6 8 4 1 4 9]
 [5 0 3 3 7 9 3 5 2 4]]
```

If we look at the last column, we see that the number 1 appears three times, then both numbers 2 and 0 appear twice, and lastly numbers 3, 9, and 4 appear only once. Note that, for example, among those rows that contain in column `c` a number that appear twice in column `c` the order can be arbitrary.

Hint: the function `np.unique` may be useful.

Matrix operations

The `*` operator in NumPy corresponds to the element-wise product of two arrays. However, very often we would like to use the *matrix multiplication*. As a reminder, if a and b are two matrices with shapes $n \times k$ and $k \times m$ respectively, then the matrix product of a and b is the matrix c with shape $n \times m$, where $c_{ij} = \sum_{h=1}^{k} a_{ih} b_{hj}$ for $1 \leq i \leq n$ and $1 \leq j \leq m$. In NumPy we can compute the matrix product using the `np.matmul` function or the `@` operator. An example of this:

In [ ]:
```
np.random.seed(0)
a=np.random.randint(0,10, (3,2))
b=np.random.randint(0,10, (2,4))
print(a)
print(b)
```

```
[[5 0]
 [3 3]
 [7 9]]
[[3 5 2 4]
 [7 6 8 8]]
```

In [ ]:
```
def info(name, a):
    print(f"{name} has dim {a.ndim}, shape {a.shape}, size {a.size}, and dtype {a.dtype}:")
    print(a)
```

In [ ]:
```
c=np.matmul(a,b)
info("c", c)
```

```
c has dim 2, shape (3, 4), size 12, and dtype int64:
[[ 15  25  10  20]
 [ 30  33  30  36]
 [ 84  89  86 100]]
```

In [ ]:
```
print(a@b)      # This can be more convenient when complex expressions are used
```

```
[[ 15  25  10  20]
 [ 30  33  30  36]
 [ 84  89  86 100]]
```

So, matrices are just two dimensional arrays with the product defined differently from the element-wise product. Note that for the matrix product to be defined, the two matrices have to have compatible shapes. So the following would produce an error:

In [ ]:
```
# a@a   # Try uncommenting and running to see the error message
```

Let's test some basic identities that should hold for an invertible matrix:

In [ ]:
```
s=np.array([[1, 6, 7],
 [7, 8, 1],
 [5, 9, 8]])
print("Original matrix:", s, sep="\n")
s_inv = np.linalg.inv(s)
print("Inverted matrix:", s_inv, sep="\n")
print("Matrix product:", s @ s_inv, sep="\n")                    # This should be pretty close to the identity matrix np.e
print("Matrix product the other way:", s_inv @ s, sep="\n")                    # Same here
```

```
Original matrix:
[[1 6 7]
 [7 8 1]
 [5 9 8]]
Inverted matrix:
```

```
[[-0.61111111 -0.16666667  0.55555556]
 [ 0.56666667  0.3        -0.53333333]
 [-0.25555556 -0.23333333  0.37777778]]
Matrix product:
[[  1.00000000e+00   0.00000000e+00   4.44089210e-16]
 [  7.21644966e-16   1.00000000e+00   1.11022302e-16]
 [ -4.44089210e-16  -2.22044605e-16   1.00000000e+00]]
Matrix product the other way:
[[  1.00000000e+00  -8.88178420e-16   0.00000000e+00]
 [  4.44089210e-16   1.00000000e+00   8.88178420e-16]
 [  0.00000000e+00  -4.44089210e-16   1.00000000e+00]]
```

Let's try mapping a vector with the matrix and its inverse:

In [ ]:
```
x=np.array([1,4,2])
y = s @ x                  # Transforms x to y
print(y)
print(s_inv @ y)           # Transforms y back to x
```

```
[39 41 57]
[ 1.  4.  2.]
```

Exercise 4 (matrix power)

Repeat the functionality of the NumPy function `numpy.linalg.matrix_power`, which raises a square matrix of shape (m,m) to the `n` th power. Here the multiplication is the matrix multiplication. Square matrix `a` raised to power `n` is therefore `a @ a @ ... @ a` where `a` is repeated `n` times. When n is zero then a^0 is equal to `np.eye(m)`.

Write function `matrix_power` that gets as first argument a square matrix `a` and as second argument a non-negative integer `n`. The function should return the matrix `a` multiplied by itself `n-1` times. Use Python's `reduce` function and a generator expression.

Extend the `matrix_power` function. For negative powers, we define a^{-1} to be equal to the multiplicative inverse of `a`. You can use NumPy's function `numpy.linalg.inv` for this. And you may assume that the input matrix is invertible.

Exercise 5 (correlation)

This exercise can give two points at maximum!

Load the iris dataset using the provided function `load()` in the stub solution. The four columns of the returned array correspond to

- sepal length (cm)
- sepal width (cm)
- petal length (cm)
- petal width (cm)

Part 1. What is the Pearson correlation between the variables `sepal length` and `petal length`. Compute this using the `scipy.stats.pearsonr` function. It returns a tuple whose first element is the correlation. Write a function `lengths` that loads the data and returns the correlation.

Part 2. What are the correlations between all the variables. Write a function `correlations` that returns an array of shape (4,4) containing the correlations. Use the function `np.corrcoef`. Which pair of variables is the most highly correlated?

Note the input formats of both functions `pearsonr` and `corrcoef`.

Solving system of linear equations

The NumPy library has function np.linalg.solve to solve systems of linear equations. The `solve` function gets as parameters the coefficient matrix and the vector containing the constant term. Below is an example of the general system of linear equations

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots a_{1m}x_m &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots a_{2m}x_m &= b_2 \\
&\cdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots a_{nm}x_m &= b_n
\end{aligned}
$$

and the equivalent matrix form

$$A \cdot x = b$$

where A has shape `(n,m)`, the unknown x has shape `(m,1)`, and b has shape `(n,1)`.

Exercise 6 (meeting lines)

Write function `meeting_lines` that gets the coefficients of two lines as parameters and returns the point where the two lines meet. The equations for the lines are $y = a_1x + b_1$ and $y = a_2x + b_2$. Use the `np.linalg.solve` function. Create a main function to test your solution.

Example of usage:

```
x, y = meeting_lines(a1, b1, a2, b2)
```

Exercise 7 (meeting planes)

Write function `meeting_planes` that gets the coefficients of three planes as parameters and returns the point where the planes meet. The equations for the planes are: $z = a_1y + b_1x + c_1$, $z = a_2y + b_2x + c_2$, and $z = a_3y + b_3x + c_3$.

Example of usage:

```
x, y, z = meeting_planes(a1, b1, c1,  a2, b2, c2,  a3, b3, c3)
```

Exercise 8 (almost meeting lines)

In the earlier "meeting lines" exercise there is a problem if the lines don't meet at all. Extend that solution so that it tells the meeting point if it exists, and otherwise finds the point that is closest to the both lines. You can use the `numpy.linalg.lstsq` for this.

Example of usage:

```
(x, y), exact = almost_meeting_lines(1, 2, -1, 0)
print(x, y, exact)
-1.000000 1.000000 True
```
