-	-	-
Exercise 11 (rows and columns)	Exercise 12 (row and column vectors)	Exercise 13 (diamond)
Exercise 14 (vector lengths)	Exercise 15 (vector angles)	Exercise 16 (multiplication table revisited)

NumPy

NumPy is a Python library for handling multi-dimensional arrays. It contains both the data structures needed for the storing and accessing arrays, and operations and functions for computation using these arrays. Although the arrays are usually used for storing numbers, other type of data can be stored as well, such as strings. Unlike lists in core Python, NumPy's fundamental data structure, the array, must have the same data type for all its elements. The homogeneity of arrays allows highly optimized functions that use arrays as their inputs and outputs.

There are several uses for high-dimensional arrays in data analysis. For instance, they can be used to:

- store matrices, solve systems of linear equations, find eigenvalues/vectors, find matrix decompositions, and solve other problems familiar from linear algebra
- store multi-dimensional measurement data. For example, an element `a[i,j]` in a 2-dimensional array might store the temperature t_{ij} measured at coordinates i, j on a 2-dimension surface.
- images and videos can be represented as NumPy arrays:
    - a gray-scale image can be represented as a two dimensional array
    - a color image can be represented as a three dimensional image, the third dimension contains the color components red, green, and blue
    - a color video can be represented as a four dimensional array
- a 2-dimensional table might store a sequence of *samples*, and each sample might be divided into *features*. For example, we could measure the weather conditions once per day, and the conditions could include the temperature, direction and speed of wind, and the amount of rain. Then we would have one sample per day, and the features would be the temperature, wind, and rain. In the standard representation of this kind of tabular data, the rows corresponds to samples and the columns correspond to features. We see more of this kind of data in the chapters on Pandas and Scikit-learn.

In this chapter we will go through:

- Creation of arrays
- Array types and attributes
- Accessing arrays with indexing and slicing
- Reshaping of arrays
- Combining and splitting arrays
- Fast operations on arrays
- Aggregations of arrays
- Rules of binary array operations
- Matrix operations familiar from linear algebra

We start by importing the NumPy library, and we use the standard abbreviation `np` for it.

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

Creation of arrays

There are several ways of creating NumPy arrays. One way is to give a (nested) list as a parameter to the `array` constructor:

In [ ]:
```
np.array([1,2,3])   # one dimensional array
```

Out[ ]:
```
array([1, 2, 3])
```

Note that leaving out the brackets from the above expression, i.e. calling `np.array(1,2,3)` will result in an error.

Two dimensional array can be given by listing the rows of the array:

In [ ]:
```
np.array([[1,2,3], [4,5,6]])
```

Out[ ]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

Similarly, three dimensional array can be described as a list of lists of lists:

In [ ]:
```
np.array([[[1,2], [3,4]], [[5,6], [7,8]]])
```

Out[ ]:
```
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])
```

There are some helper functions to create common types of arrays:

In [ ]:
```
np.zeros((3,4))
```

Out[ ]:
```
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
```

```
       [ 0.,  0.,  0.,  0.]])
```

To specify that elements are `int`s instead of `float`s, use the parameter `dtype`:

In [ ]:
```
np.zeros((3,4), dtype=int)
```

Out[ ]:
```
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])
```

Similarly `ones` initializes all elements to one, `full` initializes all elements to a specified value, and `empty` leaves the elements uninitialized:

In [ ]:
```
np.ones((2,3))
```

Out[ ]:
```
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

In [ ]:
```
np.full((2,3), fill_value=7)
```

Out[ ]:
```
array([[7, 7, 7],
       [7, 7, 7]])
```

In [ ]:
```
np.empty((2,4))
```

Out[ ]:
```
array([[  2.49233826e-316,   2.35541533e-312,   2.14321575e-312,
          8.48798317e-313],
       [  1.06099790e-312,   1.08221785e-312,   8.70018274e-313,
          6.93498991e-310]])
```

The `eye` function creates the identity matrix, that is, a matrix with elements on the diagonal are set to one, and non-diagonal elements are set to zero:

In [ ]:
```
np.eye(5, dtype=int)
```

Out[ ]:
```
array([[1, 0, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 1, 0],
       [0, 0, 0, 0, 1]])
```

The `arange` function works like the `range` function, but produces an array instead of a list.

In [ ]:
```
np.arange(0,10,2)
```

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

For non-integer ranges it is better to use `linspace`:

In [ ]:
```
np.linspace(0, np.pi, 5)  # Evenly spaced range with 5 elements
```

Out[ ]:
```
array([ 0.        ,  0.78539816,  1.57079633,  2.35619449,  3.14159265])
```

With `linspace` one does not have to compute the length of the step, but instead one specifies the wanted number of elements. By default, the endpoint is included in the result, unlike with `arange`.

Arrays with random elements

To test our programs we might use real data as input. However, real data is not always available, and it may take time to gather. We could instead generate random numbers to use as substitute. They can be generated really easily with NumPy, and can be sampled from several different distributions, of which we mention below only a few. Random data can simulate real data better than, for example, ranges or constant arrays. Sometimes we also need random numbers in our programs to choose a subset of real data (sampling). NumPy can easily produce arrays of wanted shape filled with random numbers. Below are few examples.

In [ ]:
```
np.random.random((3,4))          # Elements are uniformly distributed from half-open interval [0.0,1.0)
```

Out[ ]:
```
array([[ 0.64355769,  0.65043361,  0.71772602,  0.08071449],
       [ 0.77220621,  0.61008633,  0.0084991 ,  0.01825542],
       [ 0.86390974,  0.41933486,  0.38268666,  0.66887484]])
```

In [ ]:
```
np.random.normal(0, 1, (3,4))    # Elements are normally distributed with mean 0 and standard deviation 1
```

Out[ ]:
```
array([[ 1.94873266,  0.17407498, -0.33554709, -0.76756567],
       [ 1.93884219, -0.64023313, -0.15919656,  1.36181768],
       [ 0.63809912, -0.89464785, -0.7089435 ,  0.65445769]])
```

In [ ]:
```
np.random.randint(-2, 10, (3,4))  # Elements are uniformly distributed integers from the half-open interval [-2,10)
```

Out[ ]:
```
array([[ 6,  5,  3,  2],
       [ 0,  8,  2, -2],
       [ 3,  3,  7,  7]])
```

Sometimes it is useful to be able to recreate exactly the same data in every run of our program. For example, if there is a bug in our program, which manifests itself only with certain input, then to debug our program it needs to behave deterministically. We can create random numbers deterministically, if we always start from the same starting point. This starting point is usually an integer, and we call it a *seed*. Example of use:

In [ ]:
```
np.random.seed(0)
print(np.random.randint(0, 100, 10))
print(np.random.normal(0, 1, 10))
```

```
[44 47 64 67 67  9 83 21 36 87]
[ 1.26611853 -0.50587654  2.54520078  1.08081191  0.48431215  0.57914048
 -0.18158257  1.41020463 -0.37447169  0.27519832]
```

If you run the above cell multiple times, it will always give the same numbers, unlike the earlier examples. Try rerunning them now!

The call to `np.random.seed` initializes the *global* random number generator. The calls `np.random.random`, `np.random.normal`, etc all use this global random number generator. It is however possible to create new random number generators, and use those to sample random numbers from a distribution. Example on usage:

In [ ]:
```
new_generator = np.random.RandomState(seed=123)  # RandomState is a class, so we give the seed to its constructor
new_generator.randint(0, 100, 10)
```

Out[ ]:
```
array([66, 92, 98, 17, 83, 57, 86, 97, 96, 47])
```

You will see these used later in the materials and in the exercises, just so we can agree what the random input data is. How else could we agree whether result is correct or not, if we can't agree what the input is!

Array types and attributes

An array has several attributes: `ndim` tells the number of dimensions, `shape` tells the size in each dimension, `size` tells the number of elements, and `dtype` tells the element type. Let's create a helper function to explore these attributes:

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 [ ]:
```
b=np.array([[1,2,3], [4,5,6]])
info("b", b)
```

```
b has dim 2, shape (2, 3), size 6, and dtype int64:
[[1 2 3]
 [4 5 6]]
```

In [ ]:
```
c=np.array([b, b])          # Creates a 3-dimensional array
info("c", c)
```

```
c has dim 3, shape (2, 2, 3), size 12, and dtype int64:
[[[1 2 3]
  [4 5 6]]

 [[1 2 3]
  [4 5 6]]]
```

In [ ]:
```
d=np.array([[1,2,3,4]])               # a row vector
info("d", d)
```

```
d has dim 2, shape (1, 4), size 4, and dtype int64:
[[1 2 3 4]]
```

Note above how Python printed the three dimensional array. The general rules of printing an n-dimensional array as a nested list are:

- the last dimension is printed from left to right,
- the second-to-last is printed from top to bottom,
- the rest are also printed from top to bottom, with each slice separated from the next by an empty line.

Indexing, slicing and reshaping

Indexing

One dimensional array behaves like the list in Python:

In [ ]:
```
a=np.array([1,4,2,7,9,5])
print(a[1])
print(a[-2])
```

```
4
9
```

For multi-dimensional array the index is a comma separated tuple instead of a single integer:

In [ ]:
```
b=np.array([[1,2,3], [4,5,6]])
print(b)
print(b[1,2])    # row index 1, column index 2
print(b[0,-1])   # row index 0, column index -1
```

```
[[1 2 3]
 [4 5 6]]
6
3
```

In [ ]:
```
# As with lists, modification through indexing is possible
b[0,0] = 10
print(b)
```

```
[[10  2  3]
 [ 4  5  6]]
```

Note that if you give only a single index to a multi-dimensional array, it indexes the first dimension of the array, that is the rows. For example:

In [ ]:
```
print(b[0])    # First row
print(b[1])    # Second row
```

```
[10  2  3]
[4 5 6]
```

Slicing

Slicing works similarly to lists, but now we can have slices in different dimensions:

In [ ]:
```
print(a)
print(a[1:3])
print(a[::-1])    # Reverses the array
```

```
[1 4 2 7 9 5]
[4 2]
[5 9 7 2 4 1]
```

In [ ]:
```
print(b)
print(b[:,0])
print(b[0,:])
print(b[:,1:])
```

```
[[10  2  3]
 [ 4  5  6]]
[10  4]
[10  2  3]
[[2 3]
 [5 6]]
```

We can even assign to a slice:

In [ ]:
```
b[:,1:] = 7
print(b)
```

```
[[10  7  7]
 [ 4  7  7]]
```

A common idiom is to extract rows or columns from an array:

In [ ]:
```
print(b[:,0])    # First column
print(b[1,:])    # Second row
```

```
[10  4]
[4 7 7]
```

Reshaping

When an array is reshaped, its number of elements stays the same, but they are reinterpreted to have a different shape. An example of this is to interpret a one dimensional array as two dimension array:

In [ ]:
```
a=np.arange(9)
anew=a.reshape(3,3)
info("anew", anew)
info("a", a)
```

```
anew has dim 2, shape (3, 3), size 9, and dtype int64:
[[0 1 2]
 [3 4 5]
 [6 7 8]]
a has dim 1, shape (9,), size 9, and dtype int64:
[0 1 2 3 4 5 6 7 8]
```

In [ ]:
```
d=np.arange(4)             # 1d array
dr=d.reshape(1,4)          # row vector
dc=d.reshape(4,1)          # column vector
info("d", d)
info("dr", dr)
info("dc", dc)
```

```
d has dim 1, shape (4,), size 4, and dtype int64:
[0 1 2 3]
dr has dim 2, shape (1, 4), size 4, and dtype int64:
[[0 1 2 3]]
dc has dim 2, shape (4, 1), size 4, and dtype int64:
[[0]
 [1]
 [2]
 [3]]
```

Note the 1d array and the row and column vectors, which are 2d arrays, are fundamentally different objects, even though they look similar. They behave differently when we combine or otherwise operate arrays of different shapes, as we shall see in the next section and later in this material.

An alternative syntax to create, for example, column or row vectors is through the `np.newaxis` keyword. Sometimes this is easier or more natural than with the `reshape` method:

In [ ]:
```
info("d", d)
info("drow", d[:, np.newaxis])
info("drow", d[np.newaxis, :])
info("dcol", d[:, np.newaxis])
```

```
d has dim 1, shape (4,), size 4, and dtype int64:
[0 1 2 3]
drow has dim 2, shape (4, 1), size 4, and dtype int64:
[[0]
 [1]
 [2]
 [3]]
drow has dim 2, shape (1, 4), size 4, and dtype int64:
[[0 1 2 3]]
dcol has dim 2, shape (4, 1), size 4, and dtype int64:
```

```
[[0]
 [1]
 [2]
 [3]]
```

Exercise 11 (rows and columns)

Write two functions, `get_rows` and `get_columns`, that get a two dimensional array as parameter. They should return the list of rows and columns of the array, respectively. The rows and columns should be one dimensional arrays. You may use the *transpose* operation, which flips rows to columns, in your solution. The transpose is done by the `T` method:

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

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

Test your solution in the main function. Example of usage:

```
a = np.array([[5, 0, 3, 3],
 [7, 9, 3, 5],
 [2, 4, 7, 6],
 [8, 8, 1, 6]])
get_rows(a)
[array([5, 0, 3, 3]), array([7, 9, 3, 5]), array([2, 4, 7, 6]), array([8, 8, 1, 6])]
get_columns(a)
[array([5, 7, 2, 8]), array([0, 9, 4, 8]), array([3, 3, 7, 1]), array([3, 5, 6, 6])]
```

Array concatenation, splitting and stacking

The are two ways of combining several arrays into one bigger array: `concatenate` and `stack`. `Concatenate` takes n-dimensional arrays and returns an n-dimensional array, whereas `stack` takes n-dimensional arrays and returns n+1-dimensional array. Few examples of these:

In [ ]:
```
a=np.arange(2)
b=np.arange(2,5)
print(f"a has shape {a.shape}: {a}")
print(f"b has shape {b.shape}: {b}")
np.concatenate((a,b))  # concatenating 1d arrays
```

```
a has shape (2,): [0 1]
b has shape (3,): [2 3 4]
```

Out[ ]:
```
array([0, 1, 2, 3, 4])
```

In [ ]:
```
c=np.arange(1,5).reshape(2,2)
print(f"c has shape {c.shape}:", c, sep="\n")
np.concatenate((c,c))   # concatenating 2d arrays
```

```
c has shape (2, 2):
[[1 2]
 [3 4]]
```

Out[ ]:
```
array([[1, 2],
       [3, 4],
       [1, 2],
       [3, 4]])
```

By default `concatenate` joins the arrays along axis 0. To join the arrays horizontally, add parameter `axis=1`:

In [ ]:
```
np.concatenate((c,c), axis=1)
```

Out[ ]:
```
array([[1, 2, 1, 2],
       [3, 4, 3, 4]])
```

If you want to catenate arrays with different dimensions, for example to add a new column to a 2d array, you must first `reshape` the arrays to have same number of dimensions:

In [ ]:
```
print("New row:")
print(np.concatenate((c,a.reshape(1,2))))
print("New column:")
print(np.concatenate((c,a.reshape(2,1)), axis=1))
```

```
New row:
[[1 2]
 [3 4]
 [0 1]]
New column:
[[1 2 0]
 [3 4 1]]
```

Use `stack` to create higher dimensional arrays from lower dimensional arrays:

In [ ]:
```
np.stack((b,b))
```

Out[ ]:
```
array([[2, 3, 4],
       [2, 3, 4]])
```

In [ ]:
```
np.stack((b,b), axis=1)
```

Out[ ]:
```
array([[2, 2],
       [3, 3],
       [4, 4]])
```

Inverse operation of `concatenate` is `split`. Its argument specifies either the number of equal parts the array is divided into, or it specifies explicitly the break points.

In [ ]:
```
d=np.arange(12).reshape(6,2)
print("d:")
print(d)
d1,d2 = np.split(d, 2)
print("d1:")
print(d1)
print("d2:")
print(d2)
```

```
d:
[[ 0  1]
 [ 2  3]
 [ 4  5]
 [ 6  7]
 [ 8  9]
 [10 11]]
d1:
[[0 1]
 [2 3]
 [4 5]]
d2:
[[ 6  7]
 [ 8  9]
 [10 11]]
```

In [ ]:
```
d=np.arange(12).reshape(2,6)
print("d:")
print(d)
parts=np.split(d, (2,3,5), axis=1)
for i, p in enumerate(parts):
    print("part %i:" % i)
    print(p)
```

```
d:
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]
part 0:
[[0 1]
 [6 7]]
part 1:
[[2]
 [8]]
part 2:
[[ 3  4]
 [ 9 10]]
part 3:
[[ 5]
 [11]]
```

Exercise 12 (row and column vectors)

Create function `get_row_vectors` that returns a list of rows from the input array of shape `(n,m)`, but this time the rows must have shape `(1,m)`.

Similarly, create function `get_columns_vectors` that returns a list of columns (each having shape `(n,1)`) of the input matrix .

Example: for a 2x3 input matrix

```
[[5 0 3]
 [3 7 9]]
```

the result should be

```
Row vectors:
[array([[5, 0, 3]]), array([[3, 7, 9]])]
Column vectors:
[array([[5],
        [3]]),
 array([[0],
        [7]]),
 array([[3],
        [9]])]
```

The above output is basically just the returned lists printed with `print`. Only some whitespace is adjusted to make it look nicer. Output is not tested.

Exercise 13 (diamond)

Create a function `diamond` that returns a two dimensional integer array where the `1` s form a diamond shape. Rest of the numbers are `0` . The function should get a parameter that tells the length of a side of the diamond. Do this using the `eye` and `concatenate` functions of NumPy and array slicing.

Example of usage:

```
print(diamond(3))
[[0 0 1 0 0]
 [0 1 0 1 0]
 [1 0 0 0 1]
 [0 1 0 1 0]
 [0 0 1 0 0]]
print(diamond(1))
[[1]]
```

Fast computation using universal functions

In addition to providing a way to store and access multi-dimension arrays, NumPy also provides several routines to perform computations on them. One of the reasons for the popularity of NumPy is that these computations can be very efficient, much more efficient than what Python can normally do. The biggest bottle-necks in efficiency are the loops, which can be iterated millions, billions, or even more times. The loops should be as efficient as possible. What slows down loops in Python as the fact that Python is dynamically typed language. That means that at each expression Python has find out the types of the arguments of the operations. Let's consider the following loop:

In [ ]:
```
L=[1, 5.2, "ab"]
L2=[]
for x in L:
    L2.append(x*2)
print(L2)
```

```
[2, 10.4, 'abab']
```

At each iteration of this loop Python has find out the type of the variable x, which can in this example be an int, a float or a string, and depending on this type call a different function to perform the "multiplication" by two. What makes NumPy efficient, is the requirement that each element in an array must be of the same type. This homogeneity of arrays makes it possible to create *vectorized* operation, which don't operate on single elements, but on arrays (or subarrays). The previous example using vectorized operations of NumPy is shown below.

In [ ]:
```
a=np.array([2.1, 5.0, 17.2])
a2=a*2
print(a2)
```

```
[  4.2  10.   34.4]
```

Because each iteration is using identical operations only the data differs, this can compiled into machine language, and then performed in one go, hence avoiding Python's dynamic typing. The name vector operation comes from linear algebra where the addition of two vectors $v = (v_1, v_2, \ldots, v_d)$ and $w = (w_1, w_2, \ldots, w_d)$ is defined element-wise as $v + w = (v_1 + w_1, v_2 + w_2, \ldots, v_d + w_d)$.

In addition to addition there are several mathematical functions defined in the vector form. The basic arithmetic operations are: addition `+` , subtraction `-` , negation `-` , multiplication `*` , division `/` , floor division `//` , exponentation `**` , and remainder `%` .

The can be combined into more complicated expressions. An example:

In [ ]:
```
b=np.array([-1, 3.2, 2.4])
print(-a**2 * b)
```

```
[   4.41   -80.    -710.016]
```

Several other mathematical functions are defined as well. A few examples of these can be found below.

In [ ]:
```
print(np.abs(b))
print(np.cos(b))
print(np.exp(b))
print(np.log2(np.abs(b)))
```

```
[ 1.   3.2  2.4]
[ 0.54030231 -0.99829478 -0.73739372]
[  0.36787944  24.5325302   11.02317638]
[ 0.          1.67807191  1.26303441]
```

In NumPy nomenclature these vector operations are called *ufuncs* (universal functions).

Aggregations: max, min, sum, mean, standard deviation...

Aggregations allow us to condense the information in an array into just few numbers.

In [ ]:
```
np.random.seed(0)
a=np.random.randint(-100, 100, (4,5))
print(a)
print(f"Minimum: {a.min()}, maximum: {a.max()}")
print(f"Sum: {a.sum()}")
print(f"Mean: {a.mean()}, standard deviation: {a.std()}")
```

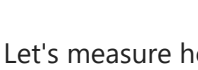
```
[[ 72 -53  17  92 -33]
 [ 95   3 -91 -79 -64]
 [-13 -30 -12  40 -42]
 [ 93 -61 -13  74 -12]]
Minimum: -91, maximum: 95
```

```
Sum: -17
Mean: -0.85, standard deviation: 58.39886557117355
```

Instead of aggregating over the whole array, we can aggregate over certain axes only as well:

In [ ]:
```
np.random.seed(9)
b=np.random.randint(0, 10, (3,4))
print(b)
print("Column sums:", b.sum(axis=0))
print("Row sums:", b.sum(axis=1))
```

```
[[5 6 8 6]
 [1 6 4 8]
 [1 8 5 1]]
Column sums: [ 7 20 17 15]
Row sums: [25 19 15]
```



Note that most of the aggregation functions in NumPy have corresponding methods. In addition, Python language has builtin functions $\sum$, $\min$, $\max$, any, and all for sequences. Make sure you don't accidentally use these for arrays, since they may have slightly different semantics, and they will be significantly slower than NumPy's functions and methods.

Python function	NumPy function	NumPy method
sum	np.sum	a.sum
-	np.prod	a.prod
-	np.mean	a.mean
-	np.std	a.std
-	np.var	a.var
min	np.min	a.min
max	np.max	a.max
-	np.argmin	a.argmin
-	np.argmax	a.argmax
-	np.median	-
-	np.percentile	-
any	np.any	a.any
all	np.all	a.all

Let's measure how much slower Python's `sum` function is compared to NumPy's equivalent when aggregating over an array:

In [ ]:
```
a=np.arange(1000)
%timeit np.sum(a)
```

```
2.37 µs ± 33.6 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

In [ ]:
```
%timeit sum(a)
```

```
66.6 µs ± 792 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

The speed of NumPy is partly due to the fact that its arrays must have same type for all the elements. This requirement allows some efficient optimizations.

Exercise 14 (vector lengths)

Write function `vector_lengths` that gets a two dimensional array of shape (n,m) as a parameter. Each row in this array corresponds to a vector. The function should return an array of shape (n,), that has the length of each vector in the input. The length is defined by the usual Euclidean norm. Don't use loops at all in your solution. Instead use vectorized operations. You must use at least the `np.sum` and the `np.sqrt` functions. You can't use the function `scipy.linalg.norm` . Test your function in the main function.

Exercise 15 (vector angles)

Let x and y be m-dimensional vectors. The angle α between two vectors is defined by the equation $\cos_{xy}(\alpha) := \frac{\langle x,y \rangle}{||x||||y||}$, where the angle brackets denote the inner product, and $||x|| := \sqrt{\langle x,x \rangle}$.

Write function `vector_angles` that gets two arrays X and Y with same shape (n,m) as parameters. Each row in the arrays corresponds to a vector. The function should return vector of shape (n,) with the corresponding angles between vectors of `X` and `Y` in degrees, not in radians. Again, don't use loops, but use vectorized operations.

Note: function `np.arccos` is only defined on the domain [-1.0,1.0]. If you try to compute `np.arccos(1.000000001)` , it will fail. These kind of errors can occur due to use of finite precision in numerical computations. Force the argument to be in the correct range (`clip` method).

Test your solution in the main program.

Broadcasting

We have seen that NumPy allows array operations that are performed element-wise. But NumPy also allows binary operations that don't require the two arrays to have the same shape. For example, we can add 4 to all elements of an array with the following expression:

In [ ]:
```
np.arange(3) + np.array([4])
```

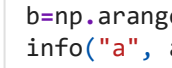
Out[ ]:
```
array([4, 5, 6])
```

In fact, because an array with only one element, say 4, can be thought of as a scalar 4, NumPy allows the following expression, which is equivalent to the above:

In [ ]:
```
np.arange(3) + 4
```

Out[ ]:
```
array([4, 5, 6])
```

To get can idea of what operations are allowed, i.e. what shapes of the two arrays are *compatible*, it can be useful to think that before the binary operation is performed, NumPy tries to stretch the arrays to have the same shape. For example in above NumPy first stretched the array `np.array([4])` (or the scalar 4), to the array `np.array([4,4,4])` and then performed the element-wise addition. In NumPy this stretching is called *broadcasting*.


broadcast

The argument arrays can of course have higher dimensions, as the next example shows:

In [ ]:
```
a=np.full((3,3), 5)
b=np.arange(3)
print("a:", a, sep="\n")
print("b:", b)
print("a+b:", a+b, sep="\n")
```

```
a:
[[5 5 5]
 [5 5 5]
 [5 5 5]]
b: [0 1 2]
a+b:
[[5 6 7]
 [5 6 7]
 [5 6 7]]
```

In this example the second argument was first broadcasted to the array

In [ ]:
```
np.array([[0, 1, 2],
       [0, 1, 2],
       [0, 1, 2]])
```

Out[ ]:
```
array([[0, 1, 2],
       [0, 1, 2],
       [0, 1, 2]])
```

and then the addition was performed. And it may be that both of the argument arrays need to be broadcasted as in the next example:

In [ ]:
```
a=np.arange(3)
b=np.arange(3).reshape((3,1))
info("a", a)
info("b", b)
info("a+b", a+b)
```

```
a has dim 1, shape (3,), size 3, and dtype int64:
[0 1 2]
b has dim 2, shape (3, 1), size 3, and dtype int64:
[[0]
 [1]
 [2]]
a+b has dim 2, shape (3, 3), size 9, and dtype int64:
[[0 1 2]
 [1 2 3]
 [2 3 4]]
```

To see what the arguments were broadcasted to before the binary operation, the function `np.broadcast_arrays` can be used:

In [ ]:
```
broadcasted_a, broadcasted_b = np.broadcast_arrays(a,b)
info("broadcasted_a", broadcasted_a)
info("broadcasted_b", broadcasted_b)
```

```
broadcasted_a has dim 2, shape (3, 3), size 9, and dtype int64:
[[0 1 2]
 [0 1 2]
 [0 1 2]]
broadcasted_b has dim 2, shape (3, 3), size 9, and dtype int64:
[[0 0 0]
 [1 1 1]
 [2 2 2]]
```

So, both arrays were broadcasted, but in different ways. Let's next go through the rules how the broadcasting work.

1. All input arrays with `ndim` smaller than the input array of largest `ndim`, have 1's prepended to their shapes.
2. The size in each dimension of the output shape is the maximum of all the input sizes in that dimension.
3. An input can be used in the calculation if its size in a particular dimension either matches the output size in that dimension, or has value exactly 1.
4. If an input has a dimension size of 1 in its shape, the first data entry in that dimension will be used for all calculations along that dimension. In other words, the stepping machinery of the ufunc will simply not step along that dimension (the stride will be 0 for that dimension).

Finally an example of a situation where the two array are not compatible:

In [ ]:
```
a=np.array([1,2,3])
b=np.array([4,5])
try:
    a+b                 # This does not work since it violates the rule 3 above.
except ValueError as e:
    import sys
    print(e, file=sys.stderr)
```

```
operands could not be broadcast together with shapes (3,) (2,)
```

Exercise 16 (multiplication table revisited)

Write function `multiplication_table` that gets a positive integer `n` as parameter. The function should return an array with shape (n,n). The element at index `(i,j)` should be `i*j`. Don't use `for` loops! In your solution, rely on broadcasting, the `np.arange` function, reshaping and vectorized operators. Example of usage:

```
print(multiplication_table(4))
[[0 0 0 0]
 [0 1 2 3]
 [0 2 4 6]
 [0 3 6 9]]
```

Summary (week 2)

- We have learned how regular expressions can be used to specify regular sets of strings
  - We know how to find out if a string matches a regular expression
  - We know how to extract pieces of text that match a RE
  - We can replace matches to a RE with another string
- We can read (and write) a text file either line by line or whole file at the same time
- We also know how to specify the encoding of the file. The utf-8 encoding is a very common and can represent nearly every character or symbol of any (natural) language
- Program's parameters are in the array `sys.argv`, and we can return a value from the program with the function `sys.exit`
- The file streams `sys.stdin`, `sys.stdout`, and `sys.stderr` allow basic textual input and output to and from the program
- Every value in Python is an object
  - Classes are user defined data types, they tell how to construct instances (objects) of that type
  - Relationship between objects and classes: `isinstance`
  - Relationship between classes: `issubclass`
- Exceptions signal exceptional situations, not necessarily errors
  - We know how catch and raise exceptions
- The efficiency of NumPy is based on the fact that the same operations can be performed on elements fast, if all the elements have the same type. These are called vectorized operations
- We know how to create, reshape, perform basic access, combine, split, and aggregate arrays