

Day 4: NumPy

This day is dedicated to NumPy, a *big* Python module to handle numerical datasets efficiently.

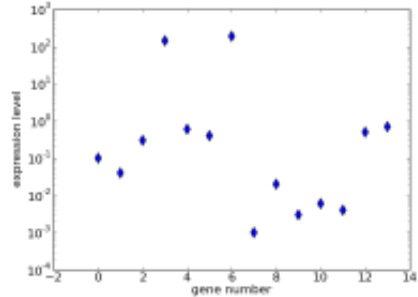
Why NumPy? (aka “Why are lists so bad?”)

When you work with numerical data, especially lists or tables (Excel people listen up!), NumPy is a must.

Example 1: you have a list of gene expression levels:

```
levels = [0.1, 0.04, 0.3, 143.4, 0.6, 0.4, 189.0, 0.001,
          0.02, 0.003, 0.006, 0.004, 0.5, 0.7]
```

and you want to keep only the highly expressed genes. Compare:



Python:

```
high = []
for level in levels:
    if level > 1:
        high.append(level)
```

NumPy:

```
high = levels[levels > 1]
```

The latter is weird-looking, I agree 😊 ; on the other hand, it is much shorter and **much** faster for the computer to calculate.

Example 2: you have a table, and want to switch rows and columns.

Python:

```
new_table = []
for i, row in enumerate(old_table):
    for j, element in enumerate(row):
        if i == 0:
            new_table.append([])
        new_table[j].append(element)
```

NumPy:

```
new_table = old_table.T
```

Plus, it is much faster to execute.

Types and length!

The price to pay for such efficiency and conciseness is twofold: types and length.

1. **types:** your data must be of a single kind, e.g. real numbers, or letters (e.g. for a DNA sequence). You cannot have a list that mixes up letters, numbers, and apples; *you cannot have tables (matrices, see below) that contain both integers and floating point numbers.*
2. **fixed length:** changing the length of data is a slow operation. If your data is not

predetermined but grows as you do the calculations, NumPy will require some care.

Import NumPy et al.

There are two popular ways of importing NumPy into your script/shell:

- directly: the functions will live under the `numpy/np` umbrella

```
import numpy as np
np.array([0,5,2,7], dtype='int')
```

- via `pylab`: the functions will be global (handy, but more dangerous)

```
from pylab import *
array([0,5,2,7], dtype='int')
```

PyLab also imports statistical functions and plotting functions, but tends to mess up the shell a little bit. If you feel more like a quick-thinker, use `pylab`; if you feel more at ease with clean rules around you, use the direct import.

Note: a typical problem is that you import via `pylab` and then execute stuff via `np.XXX` statements. Alternatively, you import directly and then try to execute `array(...)` and it does not work. Please **check the import statements** at the beginning of your script.



Note: in the following, I will always use *direct imports* and assume the following:

```
import numpy as np
```

Array, who art thou?

The basic object (data structure, thing) you manipulate is an **array**. Arrays are similar to Python lists, but they:

1. have a fixed **type** (e.g. integer numbers, characters)
2. have a fixed **length** (they have no `append` or `insert`)
3. are much more efficient

`['apple', 'orange', ['berry', 'lime']]`  `'pear'`
`np.array([4.0, 89.0, 1.0, 3.65])`  `4.5`

There are many ways of creating an array:

```
# From an existing list
l = ['A', 'G', 'C', 'T', 'A']
a = np.array(l)

# From an existing list, recast type
l = [1.0, 4, 6, 7.0, 9]
a = np.array(l, dtype='int')
```

```
# With zeroes/ones and length 10
a = np.zeros(10)
a = np.ones(10)

# With all numbers from 0 to 9
a = np.arange(9)

# 10 numbers uniformly spaced between 15 and 20
a = np.linspace(15, 20, 10)

# 12 numbers logarithmically spaced between 0.01 and 100
a = np.logspace(-2, 2, 12)
```

Finally, you might want to recover a Python list out of a NumPy array:

```
arr = np.linspace(1, 6, 55)
a = list(arr)
a = tuple(arr)
```

Q: how do you make an array of letters out of a string, e.g. for DNA sequences? And vice versa?

Basic operations

So how do you sum two arrays elementwise?

```
a1 = np.array([1, 2, 4, 6, 8, 4])
a2 = np.array([6, 9, 2, 1, 2, 0])
a3 = a1 + a2
```

How do you calculate products, differences, divisions, powers?

```
a1 * a2
a1 - a2
a1 / a2
a1 ** a2
```

How do you calculate the mean, standard deviation, and variance of an array?

```
np.mean(a)
np.std(a)
np.var(a)
```

Note: remember: all operations are by default **elementwise** in NumPy arrays. If you do lots of linear algebra, google for *Numpy matrices* or the `np.dot` function.

Q: what happens if you try to add arrays of *different* length?

Q: how do you add a constant to all elements of a **Python** list?

Basic indexing and slicing

How do you get the elements 4th to 8th of a

Python list?

```
l[3:8]
```

a **NumPy** array?

```
a[3:8]
```

No big surprise! The same is true for single elements. And to get the **reversed** array?

```
a_reversed = a[::-1]
```

Q: how do you get every other element (e.g. only the even ones)?

Matrices (aka "tables")

We can use arrays now instead of lists, and be more efficient. How about **tables**? In Python, a table is a list of lists:

```
table = [['A', 'G', 'T', 'G', 'T', 'T'],
         ['A', 'G', 'T', 'G', 'C', 'T'],
         ['G', 'G', 'T', 'G', 'T', 'T']]
```

In NumPy, tables are called **2 dimensional arrays** or, more sloppily, **matrices**. You can create a matrix just like you create an array:

```
# From an existing Python table
table = [['A', 'G', 'T', 'G', 'T', 'T'],
         ['A', 'G', 'T', 'G', 'C', 'T'],
         ['G', 'G', 'T', 'G', 'T', 'T']]
matrix = np.array(table)

# A zero- or one-filled matrix (5 rows, 2 columns)
matrix = np.ones((5, 2))
matrix = np.zeros((5, 2), dtype='S1')
```

If you want to find out the number of rows/columns of a matrix:

```
matrix.shape
```

If you want to swap rows and columns:

```
matrix_swapped = matrix.T
```

Indexing a matrix can be a challenging experience, but the basic ideas work out of the box:

```
# Get the 3rd to 5th rows
matrix[2:5]

# Get the 2nd to 4th columns
matrix[:, 2:4]

# Get only the element 4th row, 6th column
matrix[3, 5]
```

Arrays to matrices and back

You can create a matrix by repeating an array several times:

```
# Repeating an array 5 times
a = np.array([0, 6, 8, 3, 5])
matrix = np.tile(a, (5, 1))
```

Conversely, you can take a matrix and line up one row after the other, a process called **raveling** or **flattening**:

```
lined_up = np.ravel(matrix)
```

Note: there are two ways of flattening a matrix, row-wise or column-wise.

IO - Files stuff

Your data is usually sitting in some Excel table – oh, no! – and you want to read that table in. Conversely, you might want to export your Python results back to – ouch! – Excel.

Unfortunately, Excel is a closed format so nobody quite knows how it exactly works. Export/import goes over two steps:

1. Excel ↔ text file
2. text file ↔ Python

Although we cannot help the first step (ask your Excel solicitor about that), we can learn how to do the second step.

First of all, your **text file** must be well formatted, i.e. a table and not a messy collection of things. A good example looks like the following:

```
# Animal number, number of legs, number of heads
1254, 5, 1
1743, 4, 1
753, 2, 1
459, 8, 2
```

A bad example, instead:

```
Remember to call mum
1254, 5, 1
1743 4 1
753, 2, 1 ah and by the way the first column is animal number
459, 8, 2, 5,
and the other columns are legs and heads
```

As you can easily imagine, bad examples are a nightmare and should be avoided. For the good example, however, you can read its contents easily. Compare:

read in **Python**:

```
matrix = []
with open('my_samples.txt') as f:
    for line in f:
        fields = line[:-1].split()
        row = []
        for field in fields:
            row.append(float(field))
```

read in **Numpy**:

```
matrix = np.loadtxt('my_samples.txt')
```

and write back to file in **Python**:

```
with open('my_results.txt', 'w') as f:
    for row in matrix:
        rowstring = []
        for item in row:
            rowstring.append(str(item))
        line = '\t'.join(rowstring) + '\n'
```

write back your results in **Numpy**:

```
np.savetxt('my_results.txt', results_matrix)
```

```
f.write(line)
```

The downside of NumPy files functions is that they are quite rigid, i.e. they simply *digest a well-formatted file into matrices* without asking too many questions, but cannot handle more complex situations. For instance, if you want to read a file into a **dictionary**, NumPy cannot do it for you.

Bonus Q: can you parse the atom positions from the file `unknown_protein.pdb`? What protein is it? **Hint:** `np.loadtxt` is not sufficient, but...

Random numbers

NumPy can generate random numbers:

```
random_numbers = np.random.rand(50)
```

If you want to produce random numbers according to specific distributions, however, the module `scipy.stats` is recommended.

Line up!

Take again the example with expression levels. How do you sort the array in increasing order?

```
np.sort(a)
```

How to sort in reverse order? Well, you know that:

```
np.sort(a)[::-1]
```

What about the argument list?

```
np.argsort(a)
```

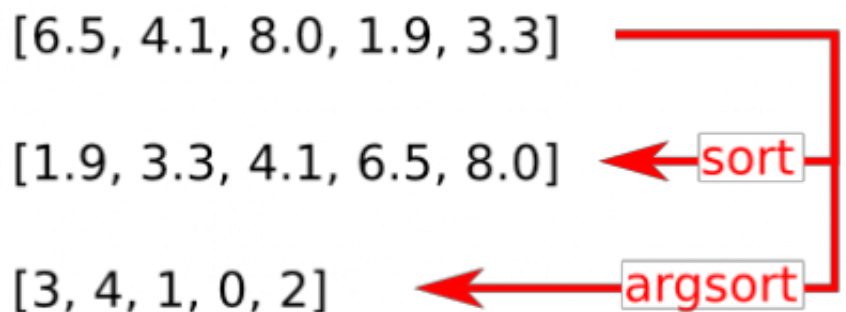
Note: if you are not following this point, do not worry!

How to find the minimum/maximum in a numerical array?

```
np.min(a)  
np.max(a)
```

And what about the index of the min/max?

```
np.argmin(a)  
np.argmax(a)
```



Where art thou, my index?

Take your expression levels again:

```
levels = [0.1, 0.04, 0.3, 143.4, 0.6, 0.4, 189, 0.001]
```

We see that only the 4th and 7th are highly expressed, but how can we get the computer tell us?

```
np.where(levels > 1)[0]
```

Note: the `[0]` is a trick only, because `np.where` is mostly used for matrices.

To get the *levels* of those genes only, we can simply use

```
high_genes = np.where(levels > 1)[0]  
high_levels = levels[high_genes]
```

Fancy indexing

Now if you look at the beginning of the page, we were able to find the levels of highly expressed genes *without* using any `np.where` function or the likes. How is that possible? That's called **fancy indexing**.

Note: fancy indexing is very powerful, yet very confusing at first. It's fine if you get lost repeatedly here 😊

The starting point is: what happens if we specify a logical condition to an array? For instance:

```
levels > 1
```

It comes back with an array of `True` and `False`, depending whether or not the condition was satisfied.

The trick is that you can use `True/False (boolean)` arrays as indices. For instance:

```
# Keep only 1st and 3rd element of the array  
arr = np.array([4, 6, 53, 2])  
kept = arr[[True, False, True, False]]  
  
# Equivalent syntax  
kept = arr[[0,2]]
```

Of course, nobody would dream of preferring the first syntax, the boolean one, over the second. However, fancy indexing is useful when you filter an array using conditions coming from other arrays:

```
high_genes = levels > 1  
high_levels = levels[high_genes]  
  
# Equivalent syntax  
high_levels = levels[levels > 1]
```

This requires some practice, but take home: boolean indexing is a powerful contraption!

Statistics: short intro

Everything you need in **not** in NumPy, but in `scipy.stats`. So start your scripts with:

```
import scipy.stats as stats
```

and everything is going to be fine! At least – until you really have to breed more damn'd fish!

