Python for biologists

Tutorial 4 - Numpy tutorial

by Tobias Andermann

In the following tutorials we will cover some data science operations in Python that are particularly useful for the natural sciences. The course is 'advanced' in the sense that it assumes that you know how to use the basic data structures and operations in Python, such as loops, strings, lists, functions, etc. If you have never been in contact with Python before, you can start with our Python introduction tutorial (https://github.com/tobiashofmann88/python_for_biologists/blob/master/tutorials/tutorial_1.ipynb).

Let's start getting you more familiar with the numpy library. We already used numpy briefly in the first tutorial, but here we will cover this very useful and essential package in more detail and point out the most useful numpy functions.

Installation

In this tutorial we will need to have these libraries installed. If you already installed them during the previous tutorials, there is no need to repeat this step.

```
pip install numpy
pip install matplotlib
```

Basic numpy math operations

Numpy is used for defining and manipulating arrays, and is one of the most powerful and commonly used tools in python. This is how you define a new, empty array:

In [22]:
```
import numpy as np
my_array = np.array([])
my_array
```

Out[22]:
```
array([], dtype=float64)
```

This is how you populate the array with values:

In [23]:
```
my_array = np.array([1,2,3,4,5,6,7])
my_array
```

Out[23]:
```
array([1, 2, 3, 4, 5, 6, 7])
```

Arrays are different to lists in python, even though they may look similar on first sight. The strength of arrays is that they allow a multitude of mathematical operations, while lists don't. Lists are more used to store a sequence of values (numbers, strings objects, lists, anything really). Arrays on the other hand usually contain integers or floats, but can also contain text-strings, which is not very common though and defies the purpose (since no mathematical operations can be executed on strings).

See the following example to understand the different betwee **lists** and **arrays**:

In [24]:
```
my_list = [1,2,3,4,5,6,7]
my_list
```

Out[24]:
```
[1, 2, 3, 4, 5, 6, 7]
```

Now let's try some mathematical operations:

In [25]:
```
my_array+4
```

Out[25]:
```
array([ 5,  6,  7,  8,  9, 10, 11])
```

In [26]:
```
my_list+4
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-26-aa5f4a7d3a83> in <module>()
----> 1 my_list+4

TypeError: can only concatenate list (not "int") to list
```

Even though both of our objects contain the same exact elements, adding a value to all elements in a list does not work. How about multiplications?

In [30]:
```
my_array*2
```

Out[30]:
```
array([ 2,  4,  6,  8, 10, 12, 14])
```

In [29]:
```
my_list*2
```

Out[29]:
```
[1, 2, 3, 4, 5, 6, 7, 1, 2, 3, 4, 5, 6, 7]
```

No error this time, but what happened to the list? You can also subtract, divide, etc.

In [39]:
```
my_array/2
```

Out[39]:
```
array([0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5])
```

In [40]:
```
my_array-2
```

Out[40]:
```
array([-1,  0,  1,  2,  3,  4,  5])
```

If you want to add the values of two arrays, these arrays need to have the same dimensions:

In [41]:
```
array_a = np.array([1,2,3,4,5])
array_b = np.array([2,3,4,5])

array_a+array_b
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-41-e086d53ce4a2> in <module>()
      2 array_b = np.array([2,3,4,5])
      3
----> 4 array_a+array_b

ValueError: operands could not be broadcast together with shapes (5,) (4,)
```

Doesn't work because of array dimensions, but this works:

In [37]:
```
array_a = np.array([1,2,3,4,5])
array_b = np.array([2,3,4,5,6])

array_a+array_b
```

Out[37]:
```
array([ 3,  5,  7,  9, 11])
```

Raise your array to the power of 2:

In [59]:
```
# You can also perform scalar operations elementwise on the entire array
my_array**2
```

Out[59]:
```
array([ 1,  4,  9, 16, 25, 36, 49])
```

Or take the log value of the array elements:

In [4]:
```
# log with base 10
b = np.array([1,2,3,4])
np.log10(b)
```

Out[4]:
```
array([0.        , 0.30103   , 0.47712125, 0.60205999])
```

In [61]:
```
# log with base e
np.log(b)
```

Out[61]:
```
array([0.        , 0.69314718, 1.09861229, 1.38629436])
```

You can try a multitude of mathematical operations [see here for an overview (https://numpy.org/doc/stable/reference/routines.math.html)](https://numpy.org/doc/stable/reference/routines.math.html). I recommend that you play around with this for a bit, using different input arrays and values. It is important to have a good grip on these basic numpy array operations.

2D arrays

So far we have only worked with 1-dimensional arrays. Arrays can have any number of dimensions, let's work with 2D arrays for now. This is how you define a 2D array:

In [42]:
```
np_array = np.array([[ 0,  1,  2,  3,  4],
                     [ 5,  6,  7,  8,  9],
                     [10, 11, 12, 13, 14]])
np_array
```

Out[42]:
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

Work through the following examples of basic and useful numpy operations. Try to understand them. Play around with different values. Ask if something is unclear.

In [73]:
```
# Creates a 3x4 array of 0's
np.zeros((3,4))
```

Out[73]:
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

In [72]:
```
# Creates a 3x4 array of int 1's
np.ones((3,4))
```

Out[72]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

In [75]:
```
### You can also create arrays with certain patterns like so
# Creating a 1D array of numbers from 10 to 100 in increments of 5
np.arange( 10, 100, 5 )
```

Out[75]:
```
array([10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90,
       95])
```

In [15]:
```
# Creating a 1D array of numbers from 0 to 2 in increments of 0.3
np.arange( 0, 2, 0.3 )
```

Out[15]:
```
array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])
```

In [76]:
```
# Creating a 1D array of 100 numbers equally spaced from 0 to 2
a = np.linspace( 0, 2, 100 )
a
```

Out[76]:
```
array([0.        , 0.02020202, 0.04040404, 0.06060606, 0.08080808,
       0.1010101 , 0.12121212, 0.14141414, 0.16161616, 0.18181818,
       0.2020202 , 0.22222222, 0.24242424, 0.26262626, 0.28282828,
       0.3030303 , 0.32323232, 0.34343434, 0.36363636, 0.38383838,
       0.4040404 , 0.42424242, 0.44444444, 0.46464646, 0.48484848,
       0.50505051, 0.52525253, 0.54545455, 0.56565657, 0.58585859,
       0.60606061, 0.62626263, 0.64646465, 0.66666667, 0.68686869,
       0.70707071, 0.72727273, 0.74747475, 0.76767677, 0.78787879,
       0.80808081, 0.82828283, 0.84848485, 0.86868687, 0.88888889,
       0.90909091, 0.92929293, 0.94949495, 0.96969697, 0.98989899,
       1.01010101, 1.03030303, 1.05050505, 1.07070707, 1.09090909,
       1.11111111, 1.13131313, 1.15151515, 1.17171717, 1.19191919,
       1.21212121, 1.23232323, 1.25252525, 1.27272727, 1.29292929,
       1.31313131, 1.33333333, 1.35353535, 1.37373737, 1.39393939,
       1.41414141, 1.43434343, 1.45454545, 1.47474747, 1.49494949,
       1.51515152, 1.53535354, 1.55555556, 1.57575758, 1.5959596 ,
       1.61616162, 1.63636364, 1.65656566, 1.67676768, 1.6969697 ,
       1.71717172, 1.73737374, 1.75757576, 1.77777778, 1.7979798 ,
       1.81818182, 1.83838384, 1.85858586, 1.87878788, 1.8989899 ,
       1.91919192, 1.93939394, 1.95959596, 1.97979798, 2.        ])
```

The above array has exactly 100 values, as we defined in the `.linspace()` function. We can reshape this array into a 2D array with a specified number of rows and columns (needs to add up to 100 in this case, though). For example if we want this to be changed into an array with 5 rows and 20 columns, we do the following:

In [79]:
```
a.reshape(5,20)
```

Out[79]:
```
array([[0.        , 0.02020202, 0.04040404, 0.06060606, 0.08080808,
        0.1010101 , 0.12121212, 0.14141414, 0.16161616, 0.18181818,
        0.2020202 , 0.22222222, 0.24242424, 0.26262626, 0.28282828,
        0.3030303 , 0.32323232, 0.34343434, 0.36363636, 0.38383838],
       [0.4040404 , 0.42424242, 0.44444444, 0.46464646, 0.48484848,
        0.50505051, 0.52525253, 0.54545455, 0.56565657, 0.58585859,
        0.60606061, 0.62626263, 0.64646465, 0.66666667, 0.68686869,
        0.70707071, 0.72727273, 0.74747475, 0.76767677, 0.78787879],
       [0.80808081, 0.82828283, 0.84848485, 0.86868687, 0.88888889,
        0.90909091, 0.92929293, 0.94949495, 0.96969697, 0.98989899,
        1.01010101, 1.03030303, 1.05050505, 1.07070707, 1.09090909,
        1.11111111, 1.13131313, 1.15151515, 1.17171717, 1.19191919],
       [1.21212121, 1.23232323, 1.25252525, 1.27272727, 1.29292929,
        1.31313131, 1.33333333, 1.35353535, 1.37373737, 1.39393939,
        1.41414141, 1.43434343, 1.45454545, 1.47474747, 1.49494949,
        1.51515152, 1.53535354, 1.55555556, 1.57575758, 1.5959596 ],
       [1.61616162, 1.63636364, 1.65656566, 1.67676768, 1.6969697 ,
        1.71717172, 1.73737374, 1.75757576, 1.77777778, 1.7979798 ,
        1.81818182, 1.83838384, 1.85858586, 1.87878788, 1.8989899 ,
        1.91919192, 1.93939394, 1.95959596, 1.97979798, 2.        ]])
```

You can also fill the array with random values between 0 and 1:

In [83]:
```
np.random.random(100)
```

Out[83]:
```
array([0.74001436, 0.12984161, 0.92778301, 0.07672625, 0.49570568,
       0.29320163, 0.40045479, 0.63617792, 0.05503776, 0.96416947,
       0.19864421, 0.36233258, 0.34499518, 0.06154136, 0.34290884,
       0.6699999 , 0.83829242, 0.57262143, 0.19558209, 0.29897307,
       0.79416663, 0.48187567, 0.93008834, 0.35445644, 0.02559039,
       0.46161158, 0.79312959, 0.82078239, 0.24850885, 0.42411661,
       0.91611583, 0.56073588, 0.64324351, 0.93828452, 0.24900303,
       0.07652539, 0.36782962, 0.0325134 , 0.99287605, 0.85477226,
       0.98906067, 0.844119  , 0.65769997, 0.08520813, 0.63447829,
       0.27977886, 0.86549087, 0.80351969, 0.35697466, 0.22088469,
       0.83605937, 0.78550207, 0.88556182, 0.19212274, 0.25627411,
       0.474251  , 0.02981747, 0.09122409, 0.67423951, 0.18574742,
       0.13208999, 0.56917704, 0.16530051, 0.13654349, 0.33718535,
       0.41659657, 0.68446419, 0.49199955, 0.51236923, 0.36189268,
       0.16257048, 0.40652984, 0.15069797, 0.77403348, 0.5475443 ,
       0.41195002, 0.36140081, 0.59266048, 0.97228396, 0.8117099 ,
       0.80752831, 0.44812699, 0.98951169, 0.77180613, 0.04412954,
       0.62532721, 0.04144324, 0.65331049, 0.41081668, 0.13815054,
       0.73194921, 0.76116973, 0.69337699, 0.95367449, 0.31849352,
       0.13289499, 0.1520524 , 0.87268957, 0.25180628, 0.88028717])
```

Getting array information

There are also many functions that help you getting information about your arrays. Here are some of the most important.

Try the following commands `.max()`, `.min()`, and `.sum()` and see what they are doing.

In [ ]:
```
a = np.array( [20,30,40,50] )
a.max()
a.min()
a.sum()
```

If you have a multi-dimensional array, use the "axis" parameter to specify how to calculate the sums, for example take this array:

In [91]:
```
b = np.arange(12).reshape(3,4)
b
```

Out[91]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Now if we want to calculate the sum of each column we can do this by specifying axis=0:

In [92]:
```
b.sum(axis=0)
```

Out[92]:
```
array([12, 15, 18, 21])
```

If instead you want to calculate the sum of each row, use axis=1:

In [98]:
```
b.sum(axis=1)
```

Out[98]:
```
array([ 6, 22, 38])
```

The sum of the whole 2D array is simply:

In [97]:
```
b.sum()
```

Out[97]:
```
66
```

TASK: Using these tools, can you produce the sum of each 4th number between 0 and 999? I.e. if you have the number from 0 to 999 `[0,1,2,3,4,5,6,7,8,...] `, add up the numbers `0+4+8+...`, `[1+5+9+...]`, `[2+6+10+...]`, and `[3+7+11+...]` to print four sums at the end.

In [102]:
```

```

Out[102]:
```
array([124500, 124750, 125000, 125250])
```

Array slicing

Python in general, but numpy particularly, makes the subselection of parts of the array rather simple and straight forward (much easier than in R for example).

You take a selection of an array by stating the index you are interested in. Remember that python starts counting with index 0.

To demonstrate the python index logic look at the following example:

```
array/list:       [10 , 20 , 30 , 40 , 50]

python index:     [0], [1], [2], [3], [4]

"human" language: 1st, 2nd, 3rd, 4th, 5th element of array
```

For example if I want to print the 5th value of an array:

In [127]:
```
a = np.array([0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100])
a[4]
```

Out[127]:
```
40
```

Note that counting in python starts with the index 0!

If I want to take the 2nd to the 4th element of an array I need to define the starting index (2) and the final index (5) separated by a colon. Python doesn't include the right boundary when ranges are specified, i.e. if you would put index 4 it would not include the 4th element but only go to the third. It's something to get used to but it has good reasons that python is handling it this way:

In [128]:
```
a[2:5]
```

Out[128]:
```
array([20, 30, 40])
```

You can print the last element by specifying the index `-1` , the second last with `-2` etc. Try the following subselections and pay attention what is being selected. Play around a bit with different indeces:

In [ ]:
```
a[2]
a[1:4]
a[-1]
a[:8]
a[2:]
a[-5:-2]
```

You can use this slicing for example to assign a new value to the selected elements, e.g.:

In [130]:
```
a = np.array([0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100])
a[1:4] = 999
a
```

Out[130]:
```
array([  0, 999, 999, 999,  40,  50,  60,  70,  80,  90, 100])
```

For a 2D array, the slicing works similarly, except that we need to specify indeces in two dimensions (rows and columns). Take for example the following array:

In [131]:
```
b = np.arange(12).reshape(3,4)
b
```

Out[131]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

If we only want to select the 2nd and third value in the last row, we provide first the row index and then the column indeces separated by a comma:

In [134]:
```
b[-1,1:3]
```

Out[134]:
```
array([ 9, 10])
```

If you just want to select all rows or all columns you can dot his with only using `:`:

In [135]:
```
b[:,1:3]
```

Out[135]:
```
array([[ 1,  2],
       [ 5,  6],
       [ 9, 10]])
```

Sometimes you need to flip an array, i.e. turn rows into columns and vice versa. You can do this by using `.T`:

In [140]:
```
b.T
```

Out[140]:
```
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
```

TASK: Use array slicing to produce the following 2D array with the number `999` in the central 4 cells.

In [139]:
```
```

Out[139]:
```
array([[  0,   1,   2,   3,   4,   5,   6,   7,   8,   9],
       [ 10,  11,  12,  13,  14,  15,  16,  17,  18,  19],
       [ 20,  21,  22,  23,  24,  25,  26,  27,  28,  29],
       [ 30,  31,  32,  33,  34,  35,  36,  37,  38,  39],
       [ 40,  41,  42,  43, 999, 999,  46,  47,  48,  49],
       [ 50,  51,  52,  53, 999, 999,  56,  57,  58,  59],
       [ 60,  61,  62,  63,  64,  65,  66,  67,  68,  69],
       [ 70,  71,  72,  73,  74,  75,  76,  77,  78,  79],
       [ 80,  81,  82,  83,  84,  85,  86,  87,  88,  89],
       [ 90,  91,  92,  93,  94,  95,  96,  97,  98,  99]])
```

Sampling random values

Besides just generating random numbers between 0 and 1, as we did earlier with the `np.random.random()` function, numpy has a whole range of random sampling methods.

One method that is often used is the `np.random.choice()` function. Here you can defined a list that numpy randomly samples values from. For example we can simulate 100 dice throws by first defining the list containing all available options (numbers 1-6) and then randomely sampling from it.

In [157]:
```
possible_values = [1,2,3,4,5,6]
np.random.choice(possible_values,100)
```

Out[157]:
```
array([3, 1, 4, 3, 1, 3, 2, 6, 4, 3, 1, 6, 1, 1, 4, 6, 6, 6, 4, 6, 6, 6,
       2, 2, 3, 6, 5, 4, 3, 6, 6, 5, 2, 6, 4, 5, 5, 4, 4, 6, 1, 2, 1, 3,
       2, 3, 3, 3, 5, 6, 5, 3, 5, 3, 1, 5, 1, 6, 2, 2, 2, 5, 5, 3, 5, 2,
       3, 5, 5, 4, 3, 5, 2, 3, 6, 2, 6, 3, 2, 6, 5, 3, 6, 3, 1, 1, 3, 4,
       4, 1, 1, 2, 2, 5, 2, 4, 4, 1, 3, 6])
```

In the example above you see that the values are drawn with replacement (the same value can be drawn multiple times from the list of possible values).

Let's say you want to randomely determine the presentation order for your next research group meeting, then you normally don't want to draw the same name multiple times and you want to ensure that every name is drawn. Therefore you need to turn off replacement, using the `replace=False` option and set the number of draws to be equal to the number of names in the 'bag':

In [162]:
```
names = ['Adam','Bert','Christina','Dora','Emil','Fernanda']
presentation_order = np.random.choice(names,len(names),replace=False)
print(presentation_order)
```

```
['Christina' 'Bert' 'Dora' 'Emil' 'Adam' 'Fernanda']
```

Insteead of using predefined lists of possible values, we can also sample from distributions.

In this example I randomely sample values from a uniform distribution between 0 and 100.

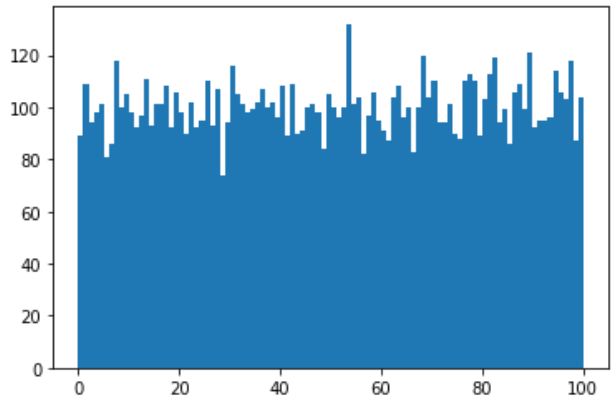
In [163]:
```
samples = np.random.uniform(0,100,10000)
```

When sampling random values it always helps to visualize what is happening. A good tool for this is plotting histograms of the sampled values. In Python you can do this easily using the matplotlib library, which we already used in yesterdays tutorial.

In [165]:
```
import matplotlib.pyplot as plt
```

Now we plot a histogram using the `hist()` function. We can specify how many bins the histogram should have, in this case you can choose 100:

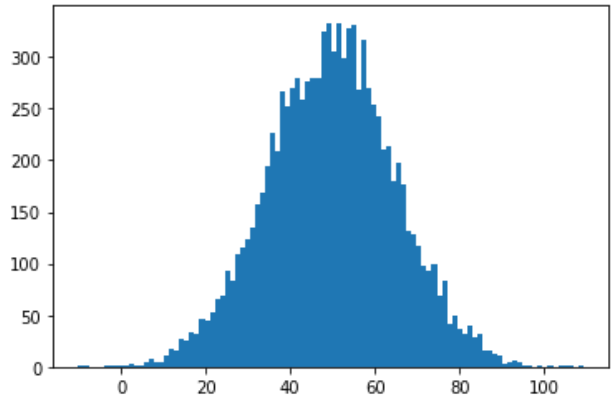
In [166]:
```
plt.hist(samples,100);
```



You see that the values are sampled with equal probability from across the defined range. Let's try a different distibution.

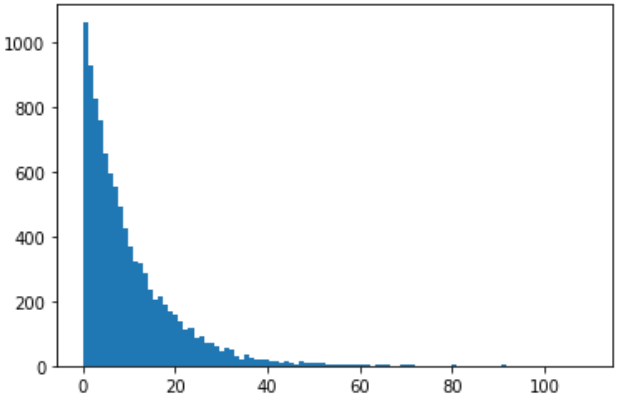
Now we sample instead from a normal distribution. For this we need to specify the mean and standard deviation of the distribution we want to sample from, I choose mean=50 and std=15 in the example.

In [170]:
```
samples = np.random.normal(50,15,10000)
plt.hist(samples,100);
```



Similarly you can sample from an exponential distribution by specifying the scale parameter (inverse of the rate parameter). I choose `scale = 10` here:

In [169]:
```
samples = np.random.exponential(10,10000)
plt.hist(samples,100);
```



Let's move on to tutorial 5 (https://github.com/tobiashofmann88/python_for_biologists/blob/master/tutorials/tutorial_5.ipynb)