Lesson 4: Play with NumPy Arrays

Welcome to **Lesson 4** of the first course module in *"Engineering Computations*." You have come a long way!

Remember, this course assumes no coding experience, so the first three lessons were focused on creating a foundation with Python programming constructs using essentially *no mathematics*. The previous lessons are:

- Lesson 1: Interacting with Python
- Lesson 2: Play with data in Jupyter
- Lesson 3: Strings and lists in action

In engineering applications, most computing situations benefit from using *arrays*: they are sequences of data all of the *same type*. They behave a lot like lists, except for the constraint in the type of their elements. There is a huge efficiency advantage when you know that all elements of a sequence are of the same type—so equivalent methods for arrays execute a lot faster than those for lists.

The Python language is expanded for special applications, like scientific computing, with **libraries**. The most important library in science and engineering is **NumPy**, providing the *n-dimensional array* data structure (a.k.a, `ndarray`) and a wealth of functions, operations and algorithms for efficient linear-algebra computations.

In this lesson, you'll start playing with NumPy arrays and discover their power. You'll also meet another widely loved library: **Matplotlib**, for creating two-dimensional plots of data.

1 Importing libraries

First, a word on importing libraries to expand your running Python session. Because libraries are large collections of code and are for special purposes, they are not loaded automatically when you launch Python (or IPython, or Jupyter). You have to import a library using the `import` command. For example, to import **NumPy**, with all its linear-algebra goodness, we enter:

```
import numpy
```

Once you execute that command in a code cell, you can call any NumPy function using the dot notation, prepending the library name. For example, some commonly used functions are:

- `numpy.linspace()`
- `numpy.ones()`
- `numpy.zeros()`

- `numpy.empty()`
- `numpy.copy()`

Follow the links to explore the documentation for these very useful NumPy functions!

Warning: You will find *a lot* of sample code online that uses a different syntax for importing. They will do:

```
import numpy as np
```

All this does is create an alias for `numpy` with the shorter string `np`, so you then would call a **NumPy** function like this: `np.linspace()`. This is just an alternative way of doing it, for lazy people that find it too long to type `numpy` and want to save 3 characters each time. For the not-lazy, typing `numpy` is more readable and beautiful.

We like it better like this:

```
In [1]: import numpy
```

2 Creating arrays

To create a NumPy array from an existing list of (homogeneous) numbers, we call `numpy.array()`, like this:

```
In [2]: numpy.array([3, 5, 8, 17])
```

```
Out[2]: array([ 3,  5,  8, 17])
```

NumPy offers many ways to create arrays in addition to this. We already mentioned some of them above.

Play with `numpy.ones()` and `numpy.zeros()`: they create arrays full of ones and zeros, respectively. We pass as an argument the number of array elements we want.

```
In [3]: numpy.ones(5)
```

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

```
In [4]: numpy.zeros(3)
```

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

Another useful one: `numpy.arange()` gives an array of evenly spaced values in a defined interval.

Syntax:

```
numpy.arange(start, stop, step)
```

where `start` by default is zero, `stop` is not inclusive, and the default for `step` is one. Play with it!

```
In [5]: numpy.arange(4)
```

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

```
In [6]: numpy.arange(2, 6)
```

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

In [7]: numpy.arange(2, 6, 2)

Out[7]: array([2, 4])

In [8]: numpy.arange(2, 6, 0.5)

Out[8]: array([ 2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,  5.5])
```

`numpy.linspace()` is similar to `numpy.arange()`, but uses number of samples instead of a step size. It returns an array with evenly spaced numbers over the specified interval.

Syntax:

```
numpy.linspace(start, stop, num)
```

`stop` is included by default (it can be removed, read the docs), and `num` by default is 50.

```
In [9]: numpy.linspace(2.0, 3.0)

Out[9]: array([ 2.        ,  2.02040816,  2.04081633,  2.06122449,  2.08163265,
                2.10204082,  2.12244898,  2.14285714,  2.16326531,  2.18367347,
                2.20408163,  2.2244898 ,  2.24489796,  2.26530612,  2.28571429,
                2.30612245,  2.32653061,  2.34693878,  2.36734694,  2.3877551 ,
                2.40816327,  2.42857143,  2.44897959,  2.46938776,  2.48979592,
                2.51020408,  2.53061224,  2.55102041,  2.57142857,  2.59183673,
                2.6122449 ,  2.63265306,  2.65306122,  2.67346939,  2.69387755,
                2.71428571,  2.73469388,  2.75510204,  2.7755102 ,  2.79591837,
                2.81632653,  2.83673469,  2.85714286,  2.87755102,  2.89795918,
                2.91836735,  2.93877551,  2.95918367,  2.97959184,  3.        ])

In [10]: len(numpy.linspace(2.0, 3.0))

Out[10]: 50

In [11]: numpy.linspace(2.0, 3.0, 6)

Out[11]: array([ 2. ,  2.2,  2.4,  2.6,  2.8,  3. ])

In [12]: numpy.linspace(-1, 1, 9)

Out[12]: array([-1.  , -0.75, -0.5 , -0.25,  0.  ,  0.25,  0.5 ,  0.75,  1.  ])
```

3 Array operations

Let's assign some arrays to variable names and perform some operations with them.

```
In [13]: x_array = numpy.linspace(-1, 1, 9)
```

Now that we've saved it with a variable name, we can do some computations with the array. E.g., take the square of every element of the array, in one go:

```
In [14]: y_array = x_array**2
         print(y_array)
```

```
[ 1.      0.5625  0.25    0.0625  0.      0.0625  0.25    0.5625  1.    ]
```

We can also take the square root of a positive array, using the `numpy.sqrt()` function:

```
In [15]: z_array = numpy.sqrt(y_array)
         print(z_array)
```

```
[ 1.    0.75  0.5   0.25  0.    0.25  0.5   0.75  1.  ]
```

Now that we have different arrays `x_array`, `y_array` and `z_array`, we can do more computations, like add or multiply them. For example:

```
In [16]: add_array = x_array + y_array
         print(add_array)
```

```
[ 0.     -0.1875 -0.25   -0.1875  0.      0.3125  0.75    1.3125  2.    ]
```

Array addition is defined element-wise, like when adding two vectors (or matrices). Array multiplication is also element-wise:

```
In [17]: mult_array = x_array * z_array
         print(mult_array)
```

```
[-1.     -0.5625 -0.25   -0.0625  0.      0.0625  0.25    0.5625  1.    ]
```

We can also divide arrays, but you have to be careful not to divide by zero. This operation will result in a `nan` which stands for *Not a Number*. Python will still perform the division, but will tell us about the problem.

Let's see how this might look:

```
In [18]: x_array / y_array
```

```
//anaconda/envs/future/lib/python3.5/site-packages/ipykernel/__main__.py:1:
RuntimeWarning: invalid value encountered in true_divide
  if __name__ == '__main__':
```

```
Out[18]: array([-1.        , -1.33333333, -2.        , -4.        ,         nan,
                 4.        ,  2.        ,  1.33333333,  1.        ])
```

4 Multidimensional arrays

4.1 2D arrays

NumPy can create arrays of N dimensions. For example, a 2D array is like a matrix, and is created from a nested list as follows:

```
In [19]: array_2d = numpy.array([[1, 2], [3, 4]])
         print(array_2d)
```

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

2D arrays can be added, subtracted, and multiplied:

```
In [20]: X = numpy.array([[1, 2], [3, 4]])
         Y = numpy.array([[1, -1], [0, 1]])
```

The addition of these two matrices works exactly as you would expect:

```
In [21]: X + Y
```

```
Out[21]: array([[2, 1],
                [3, 5]])
```

What if we try to multiply arrays using the `'*'`operator?

```
In [22]: X * Y
```

```
Out[22]: array([[ 1, -2],
                [ 0,  4]])
```

The multiplication using the `'*'` operator is element-wise. If we want to do matrix multiplication we use the `'@'` operator:

```
In [23]: X @ Y
```

```
Out[23]: array([[1, 1],
                [3, 1]])
```

Or equivalently we can use `numpy.dot()`:

```
In [24]: numpy.dot(X, Y)
```

```
Out[24]: array([[1, 1],
                [3, 1]])
```

4.2 3D arrays

Let's create a 3D array by reshaping a 1D array. We can use `numpy.reshape()`, where we pass the array we want to reshape and the shape we want to give it, i.e., the number of elements in each dimension.

Syntax

```
numpy.reshape(array, newshape)
```

For example:

```
In [25]: a = numpy.arange(24)
```

```
In [26]: a_3D = numpy.reshape(a, (2, 3, 4))
         print(a_3D)
```

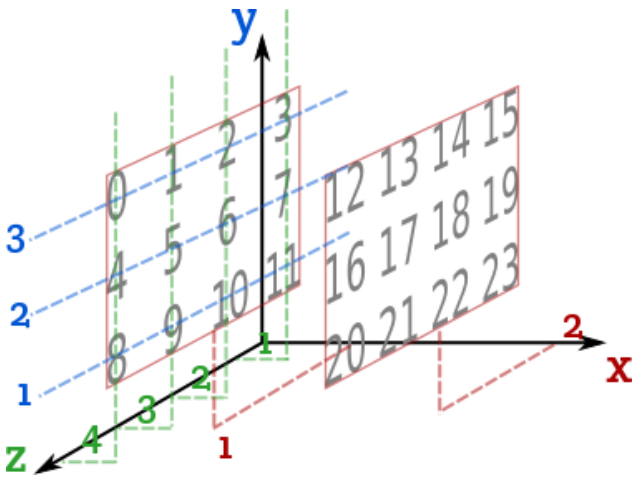
```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

We can check for the shape of a NumPy array using the function `numpy.shape()`:

```
In [27]: numpy.shape(a_3D)
```

```
Out[27]: (2, 3, 4)
```

Visualizing the dimensions of the `a_3D` array can be tricky, so here is a diagram that will help you to understand how the dimensions are assigned: each dimension is shown as a coordinate axis. For a 3D array, on the "x axis", we have the sub-arrays that themselves are two-dimensional (matrices). We have two of these 2D sub-arrays, in this case; each one has 3 rows and 4 columns. Study this sketch carefully, while comparing with how the array `a_3D` is printed out above.



When we have multidimensional arrays, we can access slices of their elements by slicing on each dimension. This is one of the advantages of using arrays: we cannot do this with lists.

Let's access some elements of our 2D array called `X`.

```
In [28]: X
```

```
Out[28]: array([[1, 2],
                [3, 4]])
```

```
In [29]: # Grab the element in the 1st row and 1st column
         X[0, 0]
```

```
Out[29]: 1
```

```
In [30]: # Grab the element in the 1st row and 2nd column
         X[0, 1]
```

```
Out[30]: 2
```

Exercises: From the X array:

1. Grab the 2nd element in the 1st column.
2. Grab the 2nd element in the 2nd column.

Play with slicing on this array:

```
In [31]: # Grab the 1st column
         X[:, 0]
```

```
Out[31]: array([1, 3])
```

When we don't specify the start and/or end point in the slicing, the symbol `':'` means "all". In the example above, we are telling NumPy that we want all the elements from the 0-th index in the second dimension (the first column).

```
In [32]: # Grab the 1st row
         X[0, :]
```

```
Out[32]: array([1, 2])
```

Exercises: From the X array:

1. Grab the 2nd column.
2. Grab the 2nd row.

Let's practice with a 3D array.

```
In [33]: a_3D
```

```
Out[33]: 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]]])
```

If we want to grab the first column of both matrices in our `a_3D` array, we do:

```
In [34]: a_3D[:, :, 0]
```

```
Out[34]: array([[ 0,  4,  8],
                [12, 16, 20]])
```

The line above is telling NumPy that we want:

- first `':'` : from the first dimension, grab all the elements (2 matrices).
- second `':'`: from the second dimension, grab all the elements (all the rows).
- `'0'` : from the third dimension, grab the first element (first column).

If we want the first 2 elements of the first column of both matrices:

```
In [35]: a_3D[:, 0:2, 0]
```

```
Out[35]: array([[ 0,  4],
                [12, 16]])
```

Below, from the first matrix in our `a_3D` array, we will grab the two middle elements (5,6):

```
In [36]: a_3D[0, 1, 1:3]
```

```
Out[36]: array([5, 6])
```

Exercises: From the array named `a_3D`:

1. Grab the two middle elements (17, 18) from the second matrix.
2. Grab the last row from both matrices.
3. Grab the elements of the 1st matrix that exclude the first row and the first column.
4. Grab the elements of the 2nd matrix that exclude the last row and the last column.

5 NumPy == Fast and Clean!

When we are working with numbers, arrays are a better option because the NumPy library has built-in functions that are optimized, and therefore faster than vanilla Python. Especially if we have big arrays. Besides, using NumPy arrays and exploiting their properties makes our code more readable.

For example, if we wanted to add element-wise the elements of 2 lists, we need to do it with a `for` statement. If we want to add two NumPy arrays, we just use the addtion `'+'` symbol!

Below, we will add two lists and two arrays (with random elements) and we'll compare the time it takes to compute each addition.

5.1 Element-wise sum of a Python list

Using the Python library `random`, we will generate two lists with 100 pseudo-random elements in the range [0,100), with no numbers repeated.

```
In [37]: #import random library
         import random
```

```
In [38]: lst_1 = random.sample(range(100), 100)
         lst_2 = random.sample(range(100), 100)
```

```
In [39]: #print first 10 elements
         print(lst_1[0:10])
         print(lst_2[0:10])
```

```
[69, 21, 55, 9, 12, 57, 75, 81, 15, 17]
[57, 29, 94, 67, 51, 71, 78, 55, 41, 72]
```

We need to write a `for` statement, appending the result of the element-wise sum into a new list we call `result_lst`.

For timing, we can use the IPython "magic" `%%time`. Writing at the beginning of the code cell the command `%%time` will give us the time it takes to execute all the code in that cell.

```
In [40]: %%time
         res_lst = []
         for i in range(100):
             res_lst.append(lst_1[i] + lst_2[i])
```

```
CPU times: user 36 µs, sys: 1 µs, total: 37 µs
Wall time: 38.9 µs
```

```
In [41]: print(res_lst[0:10])
```

```
[126, 50, 149, 76, 63, 128, 153, 136, 56, 89]
```

5.2 Element-wise sum of NumPy arrays

In this case, we generate arrays with random integers using the NumPy function `numpy.random.randint()`. The arrays we generate with this function are not going to be like the lists: in this case we'll have 100 elements in the range [0, 100) but they can repeat. Our goal is to compare the time it takes to compute addition of a *list* or an *array* of numbers, so all that matters is that the arrays and the lists are of the same length and type (integers).

```
In [42]: arr_1 = numpy.random.randint(0, 100, size=100)
         arr_2 = numpy.random.randint(0, 100, size=100)
```

```
In [43]: #print first 10 elements
         print(arr_1[0:10])
         print(arr_2[0:10])
```

```
[31 13 72 30 13 29 34 64 26 56]
[ 3 57 63 51 35 75 56 59 86 50]
```

Now we can use the `%%time` cell magic, again, to see how long it takes NumPy to compute the element-wise sum.

```
In [44]: %%time
         arr_res = arr_1 + arr_2
```

```
CPU times: user 20 µs, sys: 1 µs, total: 21 µs
Wall time: 26 µs
```

Notice that in the case of arrays, the code not only is more readable (just one line of code), but it is also faster than with lists. This time advantage will be larger with bigger arrays/lists.

(Your timing results may vary to the ones we show in this notebook, because you will be computing in a different machine.)

Exercise

1. Try the comparison between lists and arrays, using bigger arrays; for example, of size 10,000.
2. Repeat the analysis, but now computing the operation that raises each element of an array/list to the power two. Use arrays of 10,000 elements.

6 Time to Plot

You will love the Python library **Matplotlib**! You'll learn here about its module `pyplot`, which makes line plots.

We need some data to plot. Let's define a NumPy array, compute derived data using its square, cube and square root (element-wise), and plot these values with the original array in the x-axis.

```
In [45]: xarray = numpy.linspace(0, 2, 41)
         print(xarray)
```

```
[ 0.    0.05  0.1   0.15  0.2   0.25  0.3   0.35  0.4   0.45  0.5   0.55
  0.6   0.65  0.7   0.75  0.8   0.85  0.9   0.95  1.    1.05  1.1   1.15
  1.2   1.25  1.3   1.35  1.4   1.45  1.5   1.55  1.6   1.65  1.7   1.75
  1.8   1.85  1.9   1.95  2.  ]
```

```
In [46]: pow2 = xarray**2
         pow3 = xarray**3
         pow_half = numpy.sqrt(xarray)
```

To plot the resulting arrays as a function of the orginal one (`xarray`) in the x-axis, we need to import the module `pyplot` from **Matplotlib**.

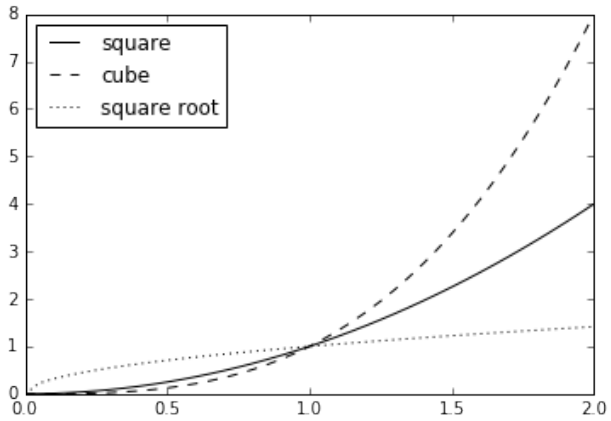
```
In [47]: from matplotlib import pyplot
         %matplotlib inline
```

The command `%matplotlib inline` is there to get our plots inside the notebook (instead of a pop-up window, which is the default behavior of `pyplot`).

We'll use the `pyplot.plot()` function, specifying the line color (`'k'` for black) and line style (`'-'`, `'--'` and `':'` for continuous, dashed and dotted line), and giving each line a label. Note that the values for `color`, `linestyle` and `label` are given in quotes.

```
In [48]: #Plot x^2
         pyplot.plot(xarray, pow2, color='k', linestyle='-', label='square')
         #Plot x^3
         pyplot.plot(xarray, pow3, color='k', linestyle='--', label='cube')
         #Plot sqrt(x)
         pyplot.plot(xarray, pow_half, color='k', linestyle=':', label='square root')
         #Plot the legends in the best location
         pyplot.legend(loc='best')
```

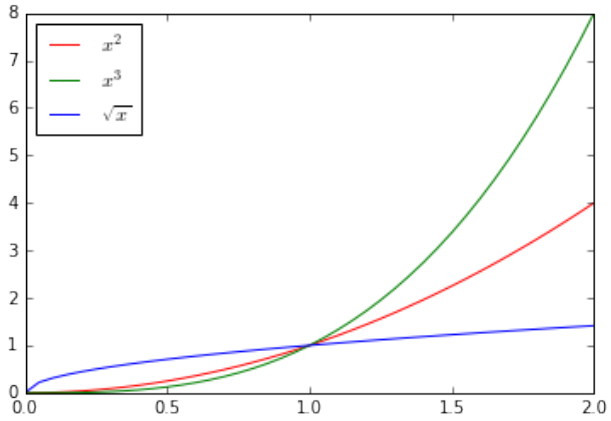
```
Out[48]: <matplotlib.legend.Legend at 0x10b224ac8>
```



To illustrate other features, we will plot the same data, but varying the colors instead of the line style. We'll also use LaTeX syntax to write formulas in the labels. If you want to know more about LaTeX syntax, there is a quick guide to LaTeX available online.

Adding a semicolon (';') to the last line in the plotting code block prevents that ugly output, like `<matplotlib.legend.Legend at 0x7f8c83cc7898>`. Try it.

```
In [49]: #Plot x^2
         pyplot.plot(xarray, pow2, color='red', linestyle='-', label='$x^2$')
         #Plot x^3
         pyplot.plot(xarray, pow3, color='green', linestyle='-', label='$x^3$')
         #Plot sqrt(x)
         pyplot.plot(xarray, pow_half, color='blue', linestyle='-', label='$\sqrt{x}$')
         #Plot the legends in the best location
         pyplot.legend(loc='best');
```



That's very nice! By now, you are probably imagining all the great stuff you can do with Jupyter notebooks, Python and its scientific libraries **NumPy** and **Matplotlib**. We just saw an introduction

to plotting but we will keep learning about the power of **Matplotlib** in the next lesson.

If you are curious, you can explore all the beautiful plots you can make by browsing the Matplotlib gallery.

Exercise: Pick two different operations to apply to the `xarray` and plot them the resulting data in the same plot.

7 What we've learned

- How to import libraries
- Multidimensional arrays using NumPy
- Accessing values and slicing in NumPy arrays
- `%%time` magic to time cell execution.
- Performance comparison: lists vs NumPy arrays
- Basic plotting with `pyplot`.

8 References

1. *Effective Computation in Physics: Field Guide to Research with Python* (2015). Anthony Scopatz & Kathryn D. Huff. O'Reilly Media, Inc.
2. *Numerical Python: A Practical Techniques Approach for Industry*. (2015). Robert Johansson. Appress.
3. "The world of Jupyter"—a tutorial. Lorena A. Barba - 2016