

SELECTED TOPICS IN ENGINEERING

INTR. TO PROG. FOR DATA SCIENCE
ENGR 350

Tuesday–Thursday 10:00–12:45

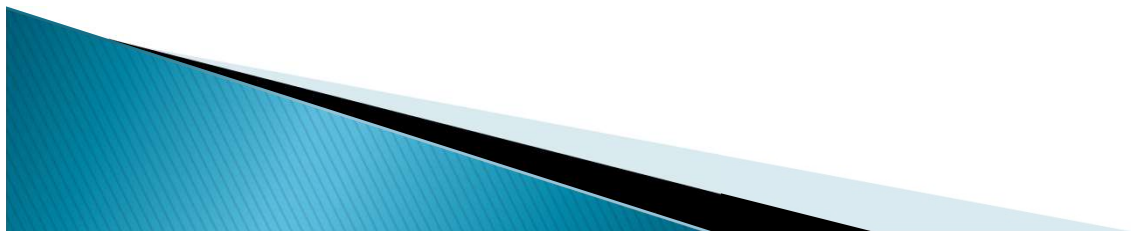
ENG B05

2019 Summer

Dr. Banu Yobaş

Numpy

- ▶ Numpy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.
- ▶ NumPy's main object is the **homogeneous multidimensional** array.
- ▶ It is a table of elements (usually numbers), all of the **same type**, **indexed by a tuple** of positive integers.



Axis

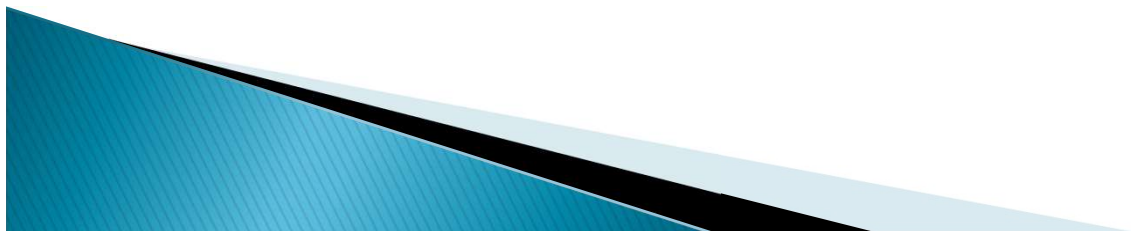
- ▶ In NumPy dimensions are called *axes*.
- ▶ For example, the coordinates of a point in 3D space `[1, 2, 1]` has one axis. That axis has 3 elements in it, so we say it has a length of 3.
- ▶ In the example pictured below, the array has 2 axes. The first axis has a length of 2, the second axis has a length of 3.

```
[[ 1., 0., 0.],  
 [ 0., 1., 2.]]
```



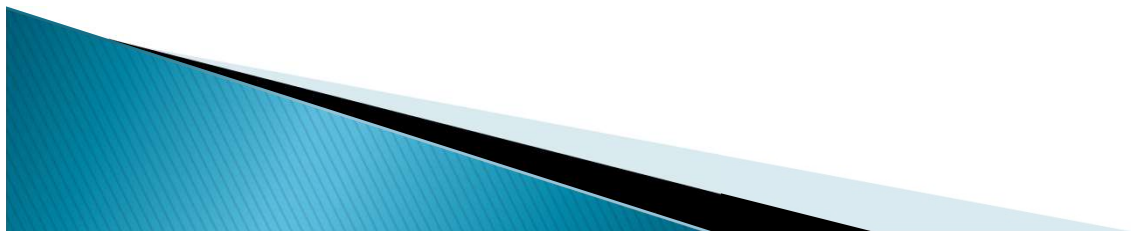
Numpy arrays

- ▶ A range is an array of consecutive numbers
 - `np.arange(end)`:
- ▶ An array of increasing integers from 0 up to end
 - `np.arange(start, end)`:
- ▶ An array of increasing integers from start up to end
 - `np.arange(start, end, step)`:
- ▶ A range with step between consecutive values
- ▶ The range always includes start but excludes end



Numpy arrays vs Python lists

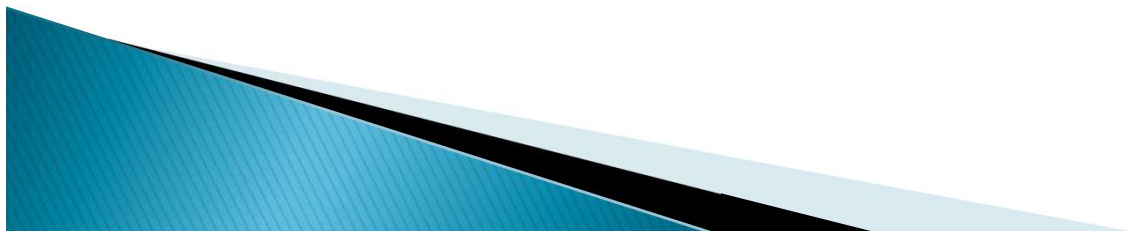
- ▶ A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers.
- ▶ The number of dimensions is the *rank* of the array;
- ▶ the *shape* of an array is a tuple of integers giving the size of the array along each dimension.
- ▶ The Python core library provided Lists. A list is the Python equivalent of an array, but is resizable and can contain elements of different types.



Numpy arrays

- ▶ Suppose we have a function $f(x)$ and want to evaluate this function at a number of x points x_0, x_1, \dots, x_{n-1} .
- ▶ We could collect the n pairs $(x_i, f(x_i))$ in a list, or we could collect all the x_i values, for $i = 0, \dots, n-1$, in a list and all the associated $f(x_i)$ values in another list.

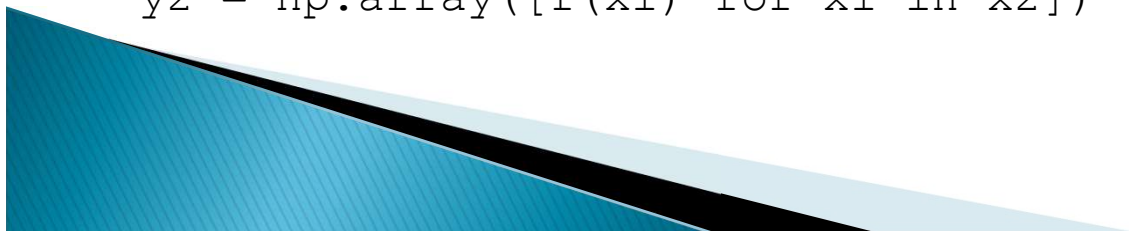
```
def f(x):  
    return x**3 # sample function  
  
n = 5 # no of points along the x axis  
dx = 1.0/(n-1) # spacing between x points in [0,1]  
xlist = [i*dx for i in range(n)]  
ylist = [f(x) for x in xlist]  
pairs = [[x, y] for x, y in zip(xlist, ylist)]
```



Numpy arrays

- ▶ Suppose we have a function $f(x)$ and want to evaluate this function at a number of x points x_0, x_1, \dots, x_{n-1} .
- ▶ List comprehensions do not work with arrays because the list comprehension creates a list, not an array.
- ▶ We can, of course, compute the y coordinates with a list comprehension and then turn the resulting list into an array:

```
x2 = np.linspace(0, 1, n)
y2 = np.array([f(xi) for xi in x2])
```



Numpy arrays

- ▶ an array to have n elements with uniformly distributed values in an interval $[p, q]$. The numpy function `linspace` creates such arrays:

`a = linspace(p, q, n)`

- ▶ Array elements are accessed by square brackets as for lists: `a[i]`.
- ▶ Slices also work as for lists, for example, `a[1:-1]` picks out all elements except *the first and the last*, but contrary to lists,

`a[1:-1]` is not a copy of the data in `a`.

- ▶ Hence,

`b = a[1:-1]`

`b[2] = 0.1`

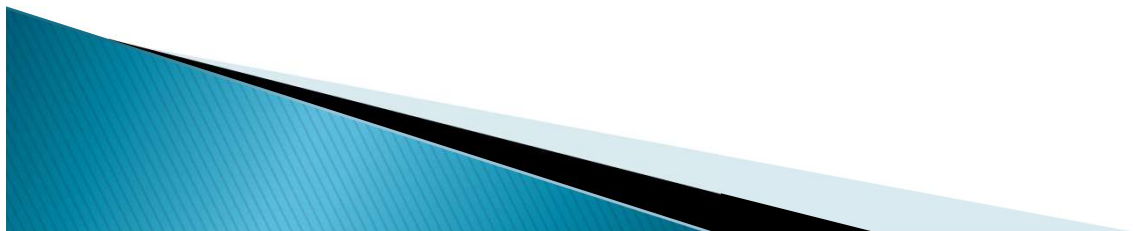
will also change `a[3]` to 0.1

- ▶ A slice `a[i:j:s]` picks out the elements starting with index i and stepping s indices at the time up to, but not including, j .



Numpy arrays

- ▶ Omitting i implies $i=0$, and omitting j implies $j=n$ if n is the number of elements in the array.
- ▶ For example, `a[0:-1:2]` picks out every two elements up to, but not including, the last element, while
- ▶ `a[::4]` picks out every four elements in the whole array.



Vectorization

- ▶ Loops over very long arrays may run slowly. A great advantage with arrays is that we can get rid of the loops and apply $f(x)$ directly to the whole array:

```
y2 = f(x2)
```

```
y2
```

```
array([ 0. , 0.015625, 0.125 , 0.421875, 1. ])
```

- ▶ The magic that makes $f(x2)$ work builds on the vector computing concepts

```
r = sin(x)*cos(x)*(-x**2) + 2 + x**2
```

works perfectly for an array x . The resulting array is the same as if we apply the formula to each array entry:

```
r = np.zeros(len(x))
```

```
for i in range(len(x)):
```

```
    r[i] = sin(x[i])*cos(x[i])*(-x[i]**2) + 2 + x[i]**2
```

- ▶ Replacing a loop like the one above by a vector/array expression (like $\sin(x)*\cos(x)*\exp(-x**2) + 2 + x**2$) is what we call vectorization.



What is the real difference btw lists and numpy.arrays?

- ▶ The answer is performance.

Numpy data structures perform better in:

- ▶ Size – Numpy data structures take up less space
- ▶ Performance – they have a need for speed and are faster than lists
- ▶ Functionality – SciPy and NumPy have optimized functions such as linear algebra operations built in.

