SCIENTIFIC COMPUTING

CS 3030: Python

Instructor: Damià Fuentes Escoté

UCCS University of Colorado Colorado Springs

Scientific computing

- Modules
  - *Numpy*
  - *SciPy*

Data visualization

- Modules
  - *matplotlib*

NUMPY

Numpy

- NumPy is an acronym for "Numeric Python" or "Numerical Python". It is pronounced /ˈnʌmpaɪ/ (NUM-py) or less often /ˈnʌmpi (NUM-pee)).
- It is an extension module for Python, mostly **written in C**. This makes sure that the precompiled mathematical and numerical functions and functionalities of Numpy guarantee **great execution speed**.
- Furthermore, NumPy enriches the programming language Python with powerful data structures, implementing **multi-dimensional arrays and matrices**. These data structures guarantee efficient calculations with matrices and arrays.

https://www.python-course.eu/numpy.php

Core Python vs Numpy

- The advantages of Core Python:
  - *high-level number objects: integers, floating point*
  - *containers: lists with cheap insertion and append methods, dictionaries with fast lookup*

- Advantages of using Numpy with Python:
  - *array oriented computing*
  - *efficiently implemented multi-dimensional arrays*
  - *designed for scientific computation*

Numpy

- Numpy is usually renamed to np:

```
import numpy as np
```

Numpy array

```
# Temperatures in Celsius
cvalues = [20.1, 20.8, 21.9, 22.5, 22.7, 22.3, 21.8, 21.2, 20.9, 20.1]
print(cvalues)
# [20.1, 20.8, 21.9, 22.5, 22.7, 22.3, 21.8, 21.2, 20.9, 20.1]

C = np.array(cvalues)
print(type(C))
# <class 'numpy.ndarray'>
print(C)
# [20.1 20.8 21.9 22.5 22.7 22.3 21.8 21.2 20.9 20.1]
```

Numpy array

- Let's assume, we want **to turn the values into degrees Fahrenheit**. This is very easy to accomplish with a numpy array. The solution to our problem can be achieved by simple scalar multiplication (like in Matlab):

```
print(C * 9 / 5 + 32)
# [68.18 69.44 71.42 72.5  72.86 72.14 71.24 70.16 69.62 68.18]

# With a Python list we would need to do the following:
print([x*9/5 + 32 for x in cvalues])
# [68.18, 69.44, 71.42, 72.5, 72.86, 72.14, 71.24, 70.16, 69.62, 68.18]
```

Creating Arrays

- There are functions provided by Numpy to create arrays with evenly spaced values within a given interval.
  - *arange() uses a given distance*
  - *linspace() needs the number of elements and creates the distance automatically*

arange(start, stop, step, dtype=None)

- arange returns evenly spaced values within a given interval.
- It is nearly equivalent to the Python built-in function range, but arange returns an ndarray rather than a list iterator as range does.
- If the 'start' parameter is not given, it will be set to 0.
- The end of the interval is determined by the parameter 'stop'. Usually, the interval will not include this value, except in some cases where 'step' is not an integer and floating point round-off affects the length of output ndarray.
- The default value for 'step' is 1. If the parameter 'step' is given, the 'start' parameter cannot be optional
- When using a non-integer step, such as 0.1, the results will often not be consistent. It is better to use numpy.linspace() for these cases.

arange(start, stop, step, dtype=None)

```
print(np.arange(10))
# [0 1 2 3 4 5 6 7 8 9]
print(np.arange(1, 10))
# [1 2 3 4 5 6 7 8 9]
print(range(1, 10))
# range(1, 10), this is an iterator
print(list(range(1, 10)))
# [1, 2, 3, 4, 5, 6, 7, 8, 9]
print(np.arange(10.4))
# [ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
print(np.arange(0.5, 10.4, 0.8))
# [ 0.5  1.3  2.1  2.9  3.7  4.5  5.3  6.1  6.9  7.7  8.5  9.3 10.1]
print(np.arange(0.5, 10.4, 0.8, int))
# [ 0  1  2  3  4  5  6  7  8  9 10 11 12] -> Not consistent
```

linspace(start, stop, num=50, endpoint=True, retstep=False)

- linspace returns an ndarray, consisting of 'num' equally spaced samples in the closed interval [start, stop] or the half-open interval [start, stop).
- If a closed or a half-open interval will be returned, depends on whether 'endpoint' is True or False. 'stop' will be the end value of the sequence, unless 'endpoint' is set to False.
- Note that the step size changes when 'endpoint' is False.
- The number of samples to be generated can be set with 'num', which defaults to 50.
- If the optional parameter '**retstep=True**' is set, the function will also return the value of the spacing between adjacent values. So, the function will return a tuple **('samples', 'step')**

linspace(start, stop, num=50, endpoint=True, retstep=False)

```
print(np.linspace(1, 10))
# [ 1.          1.18367347  1.36734694  1.55102041  1.73469388  1.91836735
#   2.10204082  2.28571429  2.46938776  2.65306122  2.83673469  3.02040816
                                  ...
#   8.71428571  8.89795918  9.08163265  9.26530612  9.44897959  9.63265306
#   9.81632653 10.        ]
print(np.linspace(1, 10, 7))
# [ 1.   2.5  4.   5.5  7.   8.5 10. ]
print(np.linspace(1, 10, 7, endpoint=False))
# [1.         2.28571429 3.57142857 4.85714286 6.14285714 7.42857143 8.71428571]
```

linspace(start, stop, num=50, endpoint=True, retstep=False)

```
samples, spacing = np.linspace(1, 10, retstep=True)
print(spacing, samples)
# 0.1836734693877551
# [ 1.          1.18367347  1.36734694  1.55102041  1.73469388  1.91836735
# ...           9.81632653 10.        ]
samples, spacing = np.linspace(1, 10, 20, endpoint=True, retstep=True)
print(spacing, samples)
# 0.47368421052631576
# [ 1.          1.47368421  1.94736842  2.42105263  2.89473684  3.36842105
# ...           9.52631579 10.        ]
samples, spacing = np.linspace(1, 10, 20, endpoint=False, retstep=True)
print(spacing, samples)
# 0.45
# [1.   1.45 1.9  2.35 2.8  3.25 3.7  4.15 4.6  5.05 ... 8.2  8.65 9.1  9.55]
```

Zero-dimensional Arrays in Numpy

- It's possible to create multidimensional arrays in numpy. **Scalars are zero dimensional**. In the following example, we will create the scalar 42. Applying the ndim method to our scalar, we get the dimension of the array. We can also see that the type is a "numpy.ndarray" type.

```
x = np.array(42)
print("x: ", x)          # x:  42
print(type(x))           # <class 'numpy.ndarray'>#
print("The dimension of x:", np.ndim(x)) # The dimension of x: 0
```

One-dimensional Arrays

- Better known to some as vectors.
- Numpy arrays are containers of items of the same type, e.g. only integers

```
I = np.array([1, 1, 2, 3, 5, 8, 13, 21])
F = np.array([3.4, 6.9, 99.8, 12.8])
print(I)           # [ 1  1  2  3  5  8 13 21]
print(F)           # [ 3.4  6.9 99.8 12.8]
print(I.dtype)     # int64
print(F.dtype)     # float64
print(np.ndim(I))  # 1
print(np.ndim(F))  # 1
```

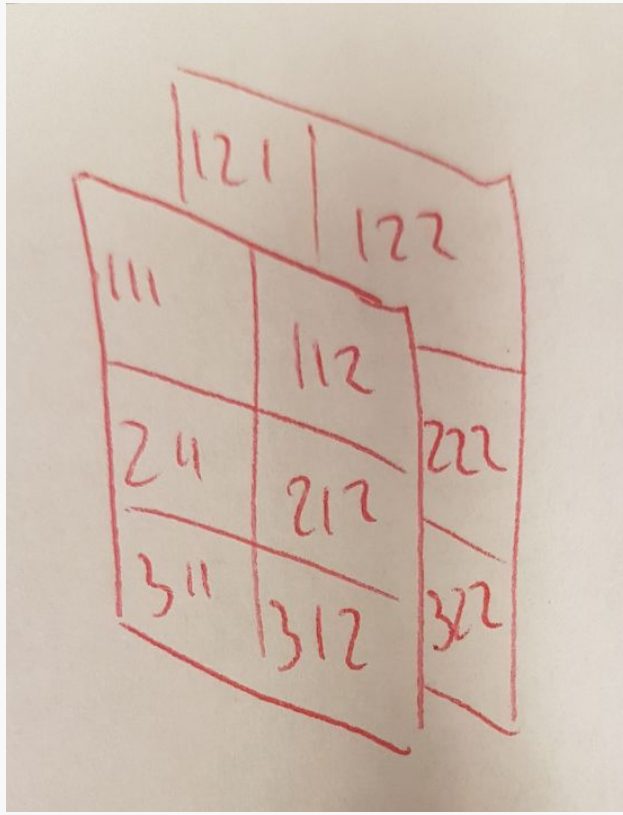
Two- and Multidimensional Arrays

- Arrays of NumPy are not limited to one dimension. They are of arbitrary dimension. We create them by passing nested lists (or tuples) to the array method of numpy.

```
A = np.array([ [3.4, 8.7, 9.9],
               [1.1, -7.8, -0.7],
               [4.1, 12.3, 4.8]])
print(A)
# [[  3.4   8.7   9.9]
#  [  1.1  -7.8  -0.7]
#  [  4.1  12.3   4.8]]
print(A.ndim)
# 2
```

Two- and Multidimensional Arrays

```
B = np.array([ [[111, 112], [121, 122]],
               [[211, 212], [221, 222]],
               [[311, 312], [321, 322]] ])
print(B)
# [[[111 112]
#   [121 122]]
#  [[211 212]
#   [221 222]]
#  [[311 312]
#   [321 322]]]
print(B.ndim)
# 3
```



Shape of an Array

- The "shape" of an array is a tuple with the number of elements per axis (dimension).

```
x = np.array([ [67, 63, 87],
               [77, 69, 59],
               [85, 87, 99],
               [79, 72, 71],
               [63, 89, 93],
               [68, 92, 78]])
print(np.shape(x))  # (6, 3)
print(x.shape)      # (6, 3)
```

(row, column, depth…)

Shape of an Array

- "shape" can also be used to change the shape of an array.
  - *the total size of the new array must be the same as the old one.*

```
x.shape = (3, 6)
print(x)
# [[67 63 87 77 69 59]
#  [85 87 99 79 72 71]
#  [63 89 93 68 92 78]]
x.shape = (2, 9)
print(x)
# [[67 63 87 77 69 59 85 87 99]
#  [79 72 71 63 89 93 68 92 78]]
```

Shape of an Array

- We can also use the .reshape() method to modify the shape and assign to another variable:

```
y = x.reshape(3, 6)
print(y)
# [[67 63 87 77 69 59]
#  [85 87 99 79 72 71]
#  [63 89 93 68 92 78]]
```

Indexing and Slicing

- Assigning to and accessing the elements of an array is similar to other sequential data types of Python, i.e. lists and tuples

```
A = np.array([ [3.4, 8.7, 9.9],
               [1.1, -7.8, -0.7],
               [4.1, 12.3, 4.8]])
print(A[1][0]) # Highly inefficient: We create an intermediate
array A[1] from which we access the element with the index 0.
print(A[1, 0]) # More efficient
```

Indexing and Slicing

- Assigning to and accessing the elements of an array is similar to other sequential data types of Python, i.e. lists and tuples

```
S = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
print(S[2:5])      # [2 3 4]
print(S[:4])       # [0 1 2 3]
print(S[6:])       # [6 7 8 9]
print(S[:])        # [0 1 2 3 4 5 6 7 8 9]
```

Indexing and Slicing

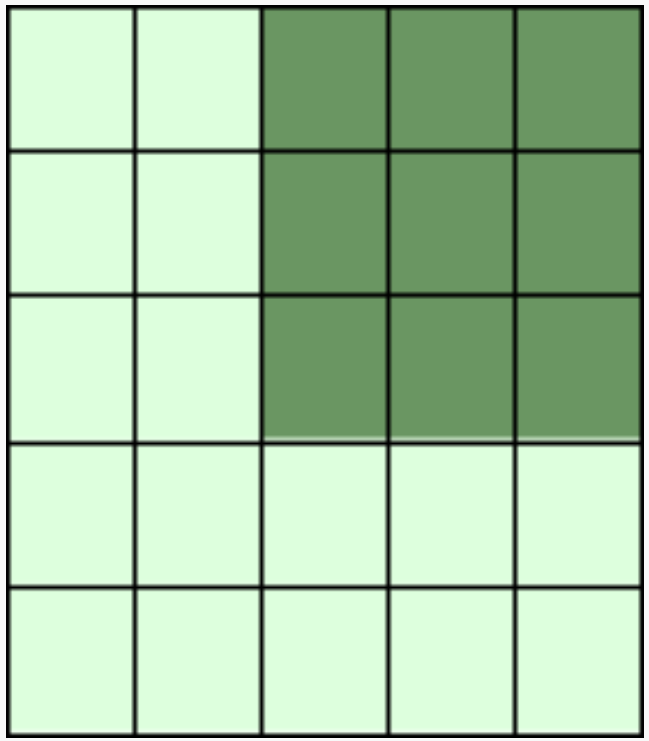
- In multidimensional slicing, the ranges for each dimension are separated by commas:

```
A = np.array([
    [11, 12, 13, 14, 15],
    [21, 22, 23, 24, 25],
    [31, 32, 33, 34, 35],
    [41, 42, 43, 44, 45],
    [51, 52, 53, 54, 55]])
print(A[:3, 2:])
# [[13 14 15]
#  [23 24 25]
#  [33 34 35]]
```

Indexing and Slicing

- In multidimensional slicing, the ranges for each dimension are separated by commas:

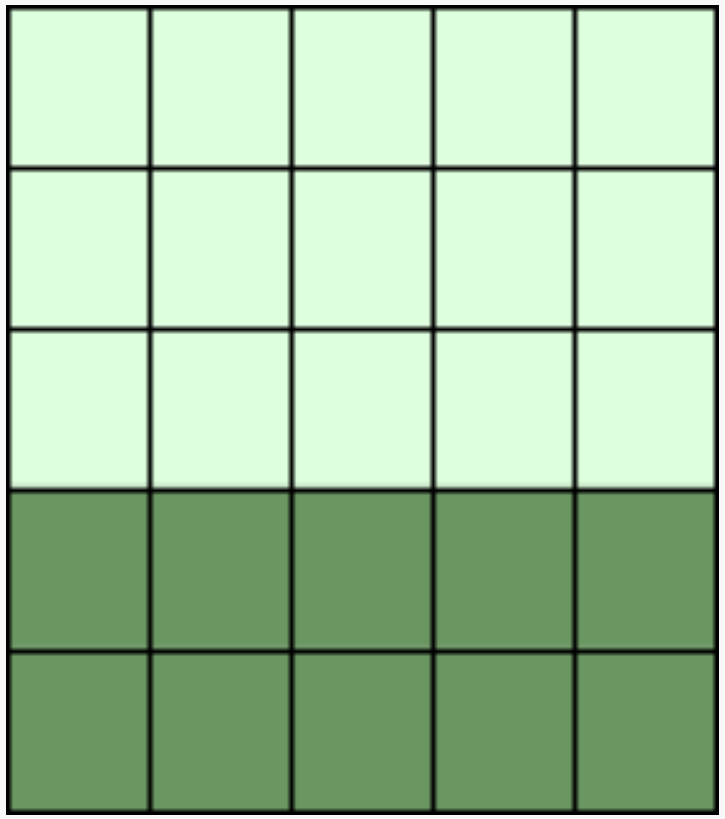
```
A = np.array([
    [11, 12, 13, 14, 15],
    [21, 22, 23, 24, 25],
    [31, 32, 33, 34, 35],
    [41, 42, 43, 44, 45],
    [51, 52, 53, 54, 55]])


print(A[:3, 2:])
# [[13 14 15]
#  [23 24 25]
#  [33 34 35]]
```



Indexing and Slicing

- In multidimensional slicing, the ranges for each dimension are separated by commas:

```
A = np.array([
    [11, 12, 13, 14, 15],
    [21, 22, 23, 24, 25],
    [31, 32, 33, 34, 35],
    [41, 42, 43, 44, 45],
    [51, 52, 53, 54, 55]])

print(A[3:, :])
# [[41 42 43 44 45]
#  [51 52 53 54 55]]
```



Indexing and Slicing

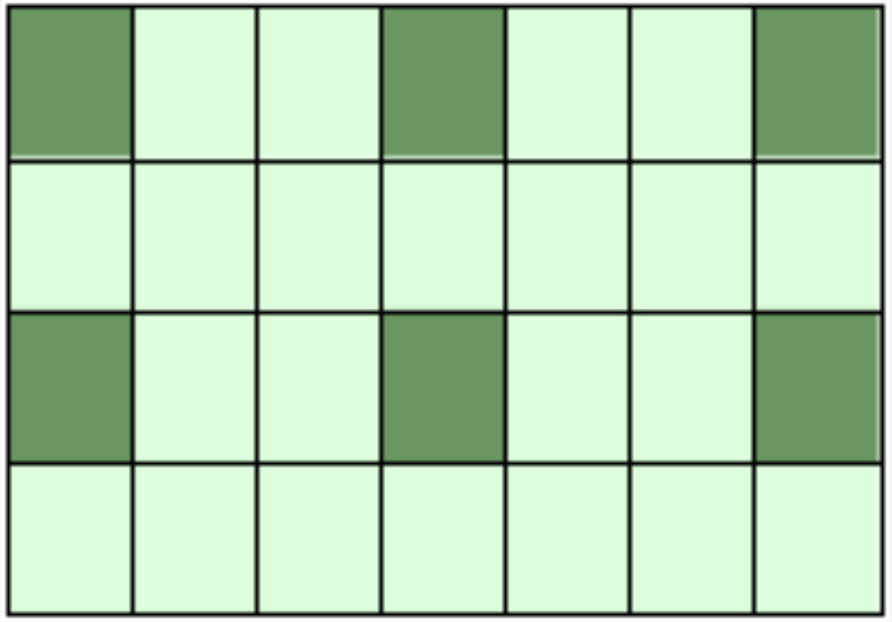
- In multidimensional slicing, the ranges for each dimension are separated by commas:

```
X = np.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]])

print(X[::2, ::3])
```

Indexing and Slicing

- In multidimensional slicing, the ranges for each dimension are separated by commas:

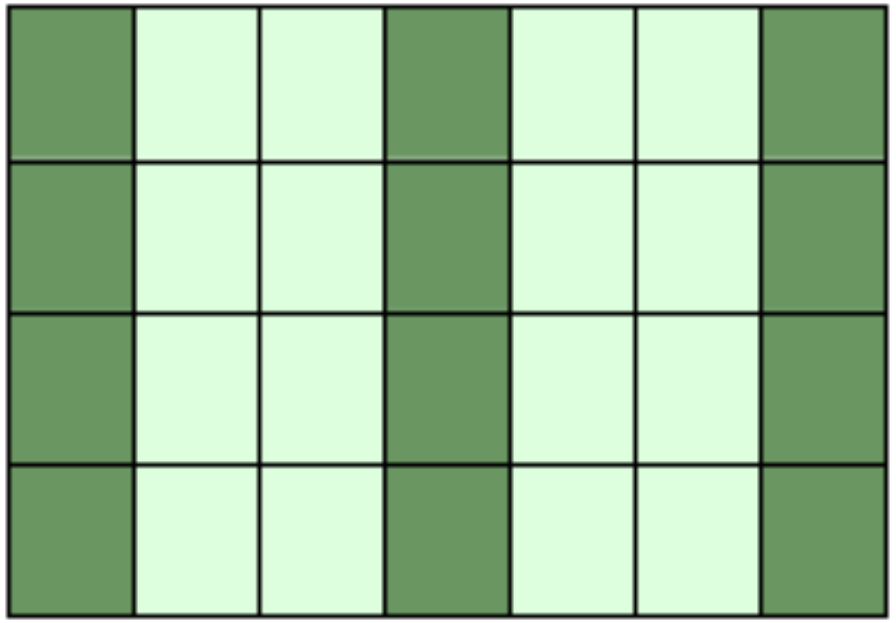
```
X = np.array([
    [0,  1,  2,   3,  4,  5,  6],
    [7,  8,  9,  10, 11, 12, 13],
    [14, 15, 16, 17, 18, 19, 20],
    [21, 22, 23, 24, 25, 26, 27]])

print(X[::2, ::3])
# [[ 0  3  6]
#  [14 17 20]]
```



Indexing and Slicing

- In multidimensional slicing, the ranges for each dimension are separated by commas:

```
X = np.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]])

print(X[:, ::3])
# [[ 0  3  6]
#  [ 7 10 13]
#  [14 17 20]
#  [21 24 27]]
```



Indexing and Slicing

- If the number of objects in the selection tuple is less than the dimension N, then : is assumed for any subsequent dimensions:

```
X = np.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]])

print(X[:2]) # Same as X[:2,:]
# [[ 0  1  2  3  4  5  6]
#  [ 7  8  9 10 11 12 13]]
```

Indexing and Slicing

- Attention: Whereas slicing on lists and tuples create new objects, a slicing operation on an array creates a view on the original array. So we get another possibility to access the array, or better a part of the array. From this follows that if we modify a view, the original array will be modified as well.

```
A = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
S = A[2:6]
S[0] = 22
S[1] = 23
print(A)
# [ 0  1 22 23  4  5  6  7  8  9]
```

Indexing and Slicing

- To determine if two arrays A and B can share memory the memory-bounds of A and B are computed. The function returns True, if they overlap and False otherwise.

```
A = np.arange(12)
print(A)
# [ 0  1  2  3  4  5  6  7  8  9 10 11]
B = A.reshape(3, 4)
A[0] = 42
print(B)
# [[42  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]
print(np.may_share_memory(A, B))      # True
```

Copying arrays

```
x = np.array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
y = x
y[1] = 10
print(x)
# [ 1. 10.  1.  1.  1.  1.  1.  1.  1.  1.]
```

Copying arrays

```
x = np.array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
y = x.copy()
y[1] = 10
print(x)
# [ 1. 10.  1.  1.  1.  1.  1.  1.  1.  1.]
```

Creating Arrays with Ones and Zeros

```
print(np.ones((2,3)))
# [[1. 1. 1.]
#  [1. 1. 1.]]
print(np.ones((3,4), dtype=int))
# [[1 1 1 1]
#  [1 1 1 1]
#  [1 1 1 1]]
print(np.zeros((2,4)))
# [[0. 0. 0. 0.]
#  [0. 0. 0. 0.]]
```

```
x = np.array([2,5,18,14,4])
print(np.ones_like(x))
# [1 1 1 1 1]
print(np.zeros_like(x))
# [0 0 0 0 0]
```

Identity Array

- In linear algebra, the identity matrix, or unit matrix, of size n is the n × n square matrix with ones on the main diagonal and zeros elsewhere.

```
print(np.identity(4))
# [[1. 0. 0. 0.]
#  [0. 1. 0. 0.]
#  [0. 0. 1. 0.]
#  [0. 0. 0. 1.]]
```

Using Scalars

- Without numpy:

```
lst = [2, 3, 7.9, 3.3, 6.9, 0.11, 10.3, 12.9]
res = [val + 2 for val in lst]
print(res)
# [4, 5, 9.9, 5.3, 8.9, 2.11, 12.3, 14.9]
```

Using Scalars

```
@timer
def withoutNumpy():
    lst = np.random.randint(0, 100, (1000, 1000))
    lst = [val + 2 for val in lst]

@timer
def withNumpy():
    lst = np.random.randint(0, 100, (1000, 1000))
    lst += 2

withoutNumpy()
withNumpy()
# Finished 'withoutNumpy' in 0.0176 secs
# Finished 'withNumpy' in 0.0082 secs
```

Using Scalars

- With Numpy:

```
lst = [2,3, 7.9, 3.3, 6.9, 0.11, 10.3, 12.9]
v = np.array(lst)
v = v + 2
print(v)
# [ 4.    5.    9.9   5.3   8.9   2.11 12.3  14.9 ]
```

Using Scalars

- Multiplication, Subtraction, Division and exponentiation are as easy as the previous addition.

```
print(v)
# [ 4.    5.    9.9   5.3   8.9   2.11 12.3  14.9 ]
print(v * 2.2)
# [ 8.8   11.    21.78  11.66  19.58   4.642 27.06  32.78 ]
print(v - 1.38)
# [ 2.62  3.62  8.52  3.92  7.52  0.73 10.92 13.52]
print(v ** 2)
# [ 16.      25.      98.01    28.09    79.21     4.4521 151.29
222.01  ]
```

Arithmetic Operations with two Arrays

```
A = np.array([ [11, 12, 13], [21, 22, 23], [31, 32, 33] ])
B = np.ones((3,3))
print(A + B)
# [[12. 13. 14.]
#  [22. 23. 24.]
#  [32. 33. 34.]]
print(A * (B + 1))
# [[22. 24. 26.]
#  [42. 44. 46.]
#  [62. 64. 66.]]
```

Matrix Multiplication

```
A = np.array([[11, 12, 13], [21, 22, 23], [31, 32, 33]])
B = np.ones((3, 3))
print(np.dot(A, B))
# [[36. 36. 36.]
#  [66. 66. 66.]
#  [96. 96. 96.]]
```

Matrices vs. Two-Dimensional Arrays

```
A = np.array([[1, 2, 3], [2, 2, 2], [3, 3, 3]])
B = np.array([[3, 2, 1], [1, 2, 3], [-1, -2, -3]])
print(np.dot(A, B))
# [[ 2  0 -2]
#  [ 6  4  2]
#  [ 9  6  3]]
print(A*B)
# [[ 3  4  3]
#  [ 2  4  6]
#  [-3 -6 -9]]
print(np.mat(A) * np.mat(B))
# [[ 2  0 -2]
#  [ 6  4  2]
#  [ 9  6  3]]
```

Two-dimensional arrays will be only multiplied component-wise

We can turn a two-dimensional array into a matrix by applying the "mat" function

Comparison Operators

```
A = np.array([ [11, 12,  13], [21,  22,  23], [31, 32,  33] ])
B = np.array([ [11, 102, 13], [201, 22, 203], [31, 32, 303] ])
print(A == B)
# [[ True False  True]
#  [False  True False]
#  [ True  True False]]
```

Comparison Operators

```
print(np.array_equal(A, B))
print(np.array_equal(A, A))
# False
# True
```

Logical Operators

```
a = np.array([ [True, True], [False, False]])
b = np.array([ [True, False], [True, False]])
print(np.logical_or(a, b))
# [[ True  True]
#  [ True False]]
print(np.logical_and(a, b))
# [[ True False]
#  [False False]]
```

Flatten Arrays

```
A = np.array([[[ 0,  1],
               [ 2,  3],
               [ 4,  5],
               [ 6,  7]],
              [[ 8,  9],
               [10, 11],
               [12, 13],
               [14, 15]]])
print(A.flatten())
# [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
print(A.flatten(order="C"))                    # Default
# [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
print(A.flatten(order="F"))
# [ 0  8  2 10  4 12  6 14  1  9  3 11  5 13  7 15]
```

Concatenating Arrays

```
x = np.array([11,22])
y = np.array([18,7,6])
z = np.array([1,3,5])
c = np.concatenate((x,y,z))
print(c)
# [11 22 18  7  6  1  3  5]
```

Concatenating Arrays

- If we are concatenating multidimensional arrays, we can concatenate the arrays according to axis. Arrays must have the same shape to be concatenated with concatenate(). In the case of multidimensional arrays, we can arrange them according to the axis. The default value is axis = 0:

```
x = np.array(range(24))
x = x.reshape((3,4,2))
y = np.array(range(100,124))
y = y.reshape((3,4,2))
z = np.concatenate((x,y))
print(z)
```

```
# [[[  0   1]
#  [  2   3]
#  [  4   5]
#  [  6   7]]

# [[  8   9]
#  [ 10  11]
#  [ 12  13]
#  [ 14  15]]

# [[ 16  17]
#  [ 18  19]
#  [ 20  21]
#  [ 22  23]]

# [[100 101]
#  [102 103]
#  [104 105]
#  [106 107]]

# [[108 109]
#  [110 111]
#  [112 113]
#  [114 115]]
#
# ...
```

Concatenating Arrays

- If we are concatenating multidimensional arrays, we can concatenate the arrays according to axis. Arrays must have the same shape to be concatenated with concatenate(). In the case of multidimensional arrays, we can arrange them according to the axis. The default value is axis = 0:

```
x = np.array(range(24))
x = x.reshape((3,4,2))
y = np.array(range(100,124))
y = y.reshape((3,4,2))
z = np.concatenate((x,y),axis = 1)
print(z)
```

```
# [[[  0   1]
#  [  2   3]
#  [  4   5]
#  [  6   7]
#  [100 101]
#  [102 103]
#  [104 105]
#  [106 107]]
#
# [[  8   9]
#  [ 10  11]
#  [ 12  13]
#  [ 14  15]
#  [108 109]
#  [110 111]
#  [112 113]
#  [114 115]]
#
# [[ 16  17]
#  [ 18  19]
#  [ 20  21]
#  [ 22  23]
#  [116 117]
#  ...
```

Adding New Dimensions

```
x = np.array([2,5,18,14,4])
print(x)
# [ 2  5 18 14  4]
y = x[:, np.newaxis]
print(y)
# [[ 2]
#  [ 5]
#  [18]
#  [14]
#  [ 4]]
```

SciPy

- SciPy (pronounced “Sigh Pie”) is a Python-based ecosystem of open-source software for mathematics, science, and engineering.
- SciPy (Scientific Python) is often mentioned in the same breath with NumPy. SciPy needs Numpy, as it is based on the data structures of Numpy and furthermore its basic creation and manipulation functions. It extends the capabilities of NumPy with further useful functions for minimization, regression, Fourier-transformation and many others.
- NumPy has to be installed before installing SciPy.