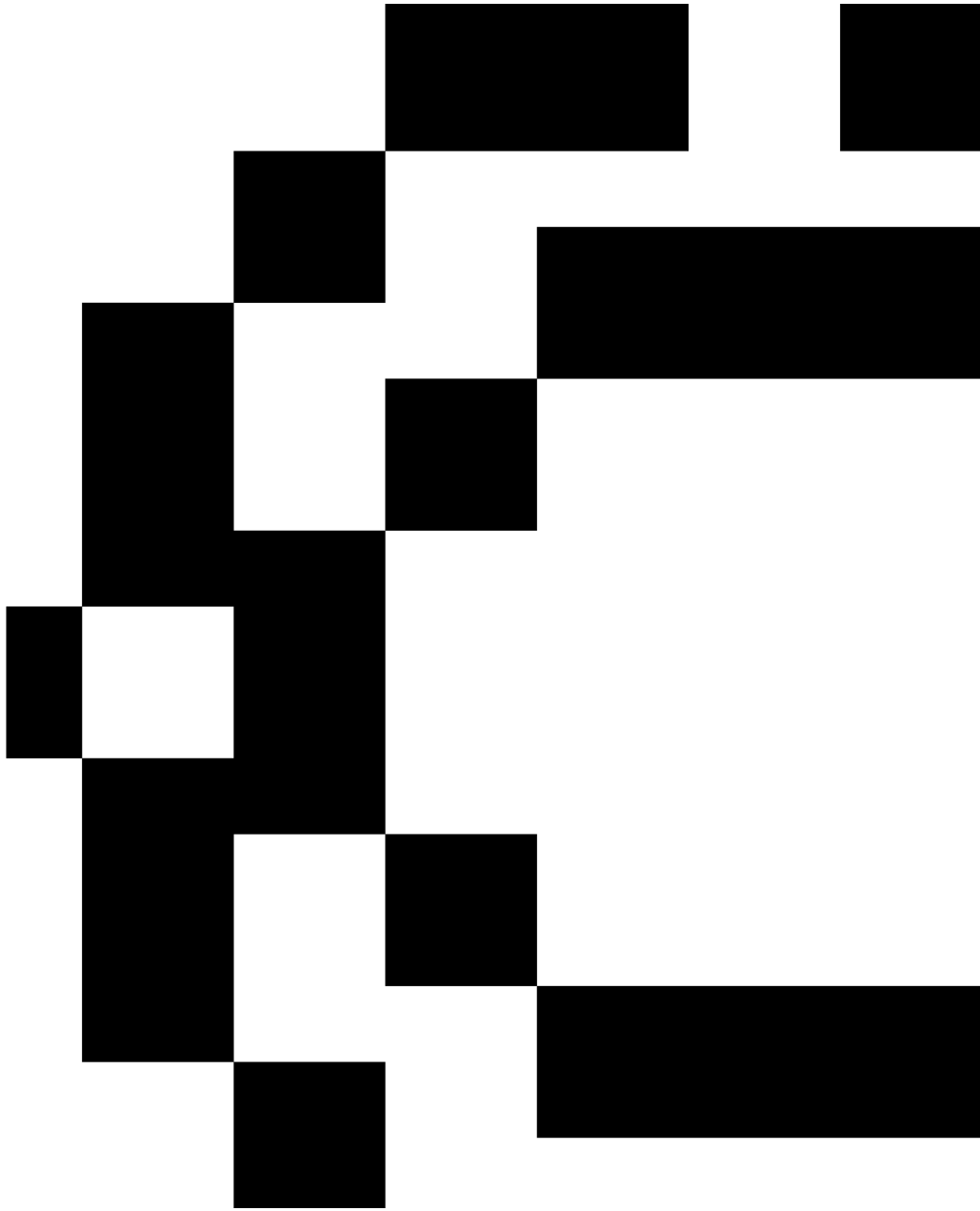
NOVA IMS INFORMATION MANAGEMENT SCHOOL

Programming for Data Science

Lecture 6

Flávio L. Pinheiro

fpinheiro@novaims.unl.pt | www.flaviolpp.com
www.linkedin.com/in/flaviolpp | X @flavio_lpp



https://www.socrative.com

Login as a student

Room Name: **PDS2025**

Student ID: **Student Number**

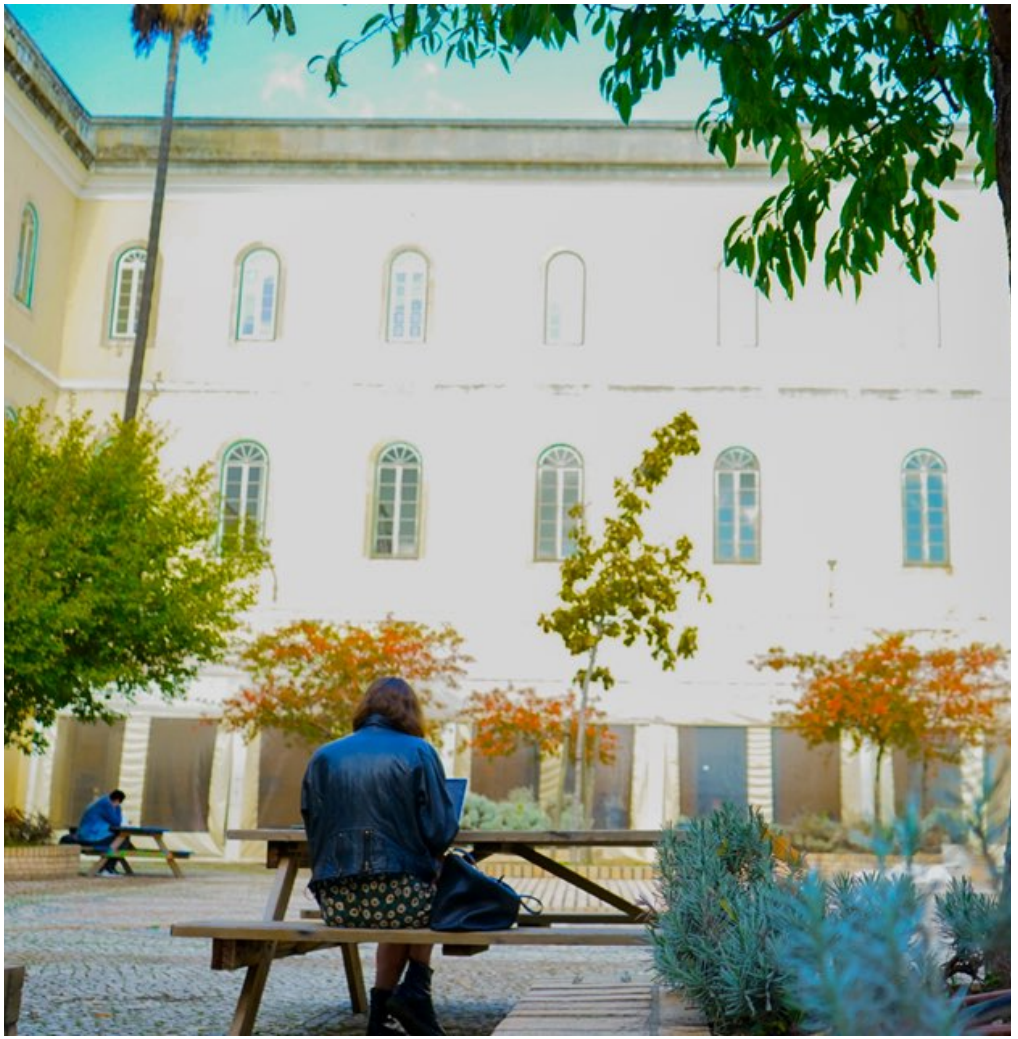
or

https://api.socrative.com/rc/5NpKRX

Student ID: **Student Number**

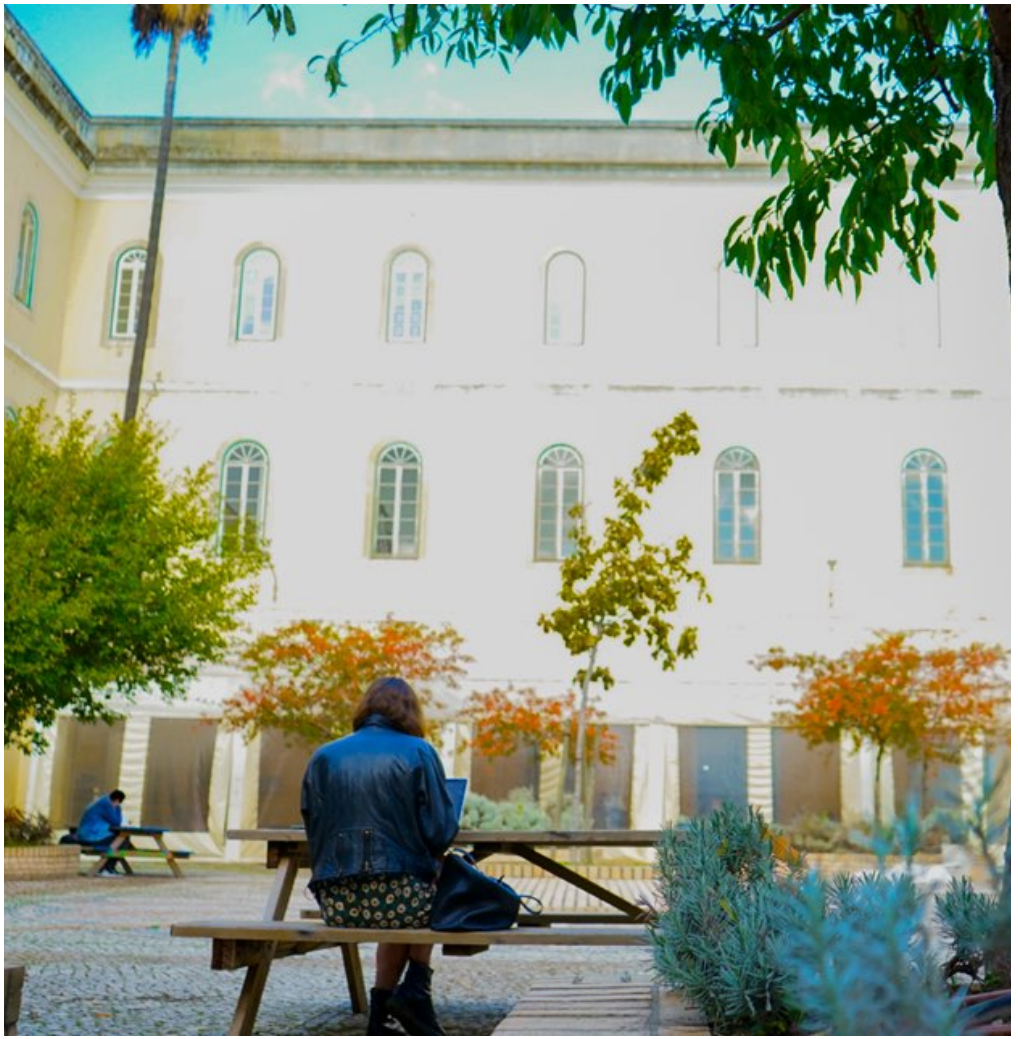
Lecture 6

- Intro to Numpy
- Array Creation and Basic Ops
- Array indexing, slicing, and reshaping
- Vectorization and Broadcasting
- Misc
- Intro to Scipy



Lecture 6

- **Intro to Numpy**
- Array Creation and Basic Ops
- Array indexing, slicing, and reshaping
- Vectorization and Broadcasting
- Misc
- Intro to Scipy



starting numpy

```
import numpy as np
```

```
np.__version__
```

```
'2.3.2'
```

NumPy (Numerical Python) is an open source Python library that's widely used in science and engineering. The NumPy library contains multidimensional array data structures, such as the homogeneous, N-dimensional **ndarray**, and a large library of functions that operate efficiently on these data structures.

Data Structures

Variables
Lists
Dictionaries
Tuples
Sets

ndarray

Series
DataFrames

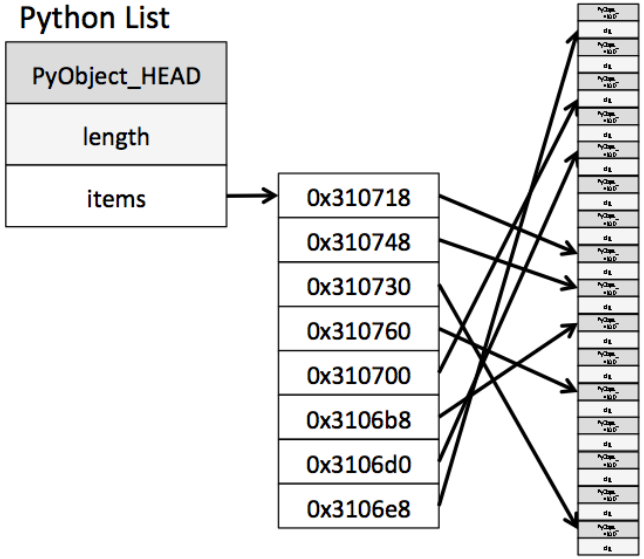
But **why** do we need ndarrays?

Lists have a problem!

for us they look like this

```
l = [“string”,2, 2.5, 2-5j,”string”]
```

but the computer
stores them like this



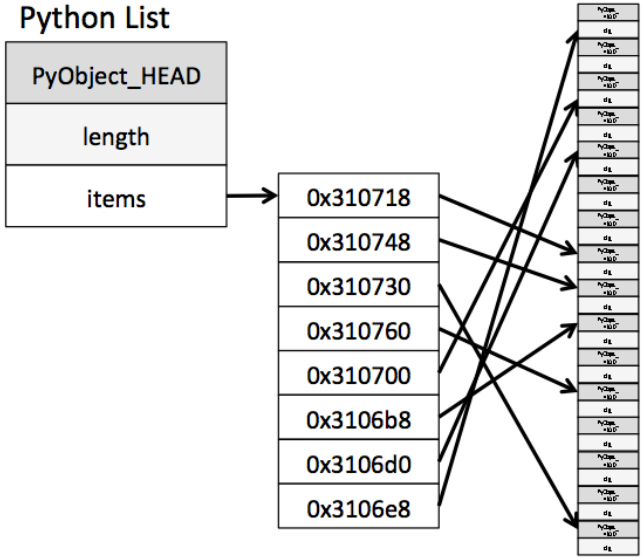
But **why** do we need ndarrays?

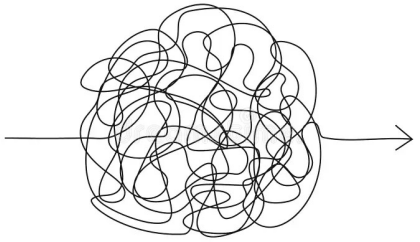
Lists have a problem!

for us they look like this

```
l = ["string",2, 2.5, 2-5j,"string"]
```

but the computer stores them like this



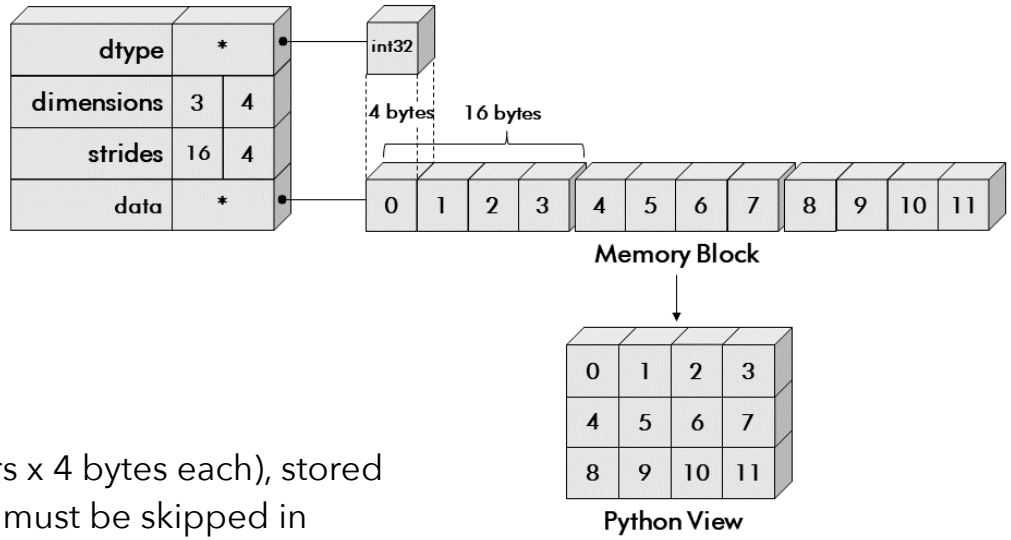


This is a mess!

the anatomy of ndarrays

Unlike Python lists, NumPy arrays are stored in a single continuous place in memory so that processes can access and manipulate them very efficiently.

This behavior, so-called **locality of reference**, is the main reason why NumPy ndarrays are faster than Python lists.

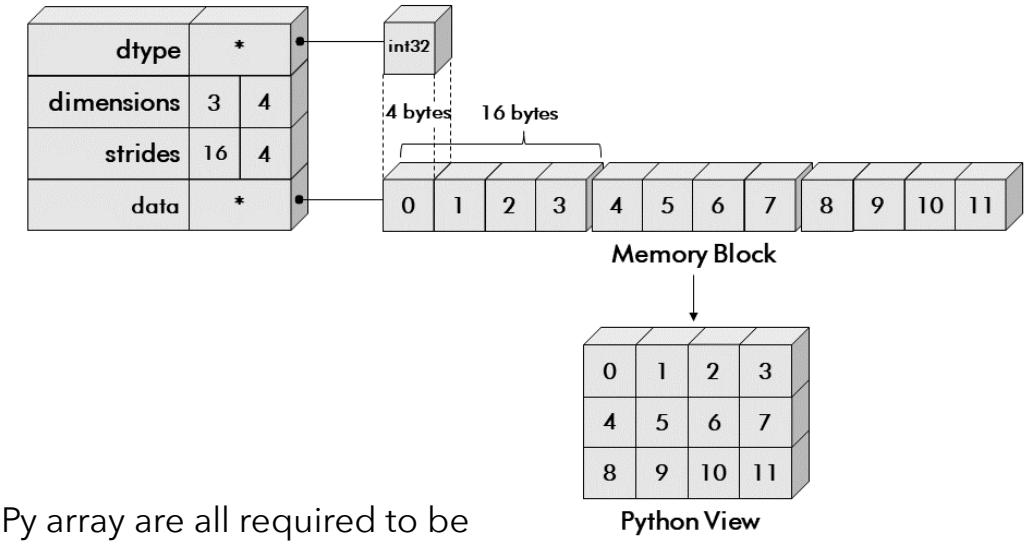


In the figure, the memory block consists of 48 bytes (12 integers x 4 bytes each), stored one after the other. The *strides* head indicates how many bytes must be skipped in memory to move to the next position along a certain axis. For example, we must skip 4 bytes (1 integer) to reach the next column, but 16 bytes (4 integers) to move to the same position in the next row. Thus, the strides for the array are (16, 4).

the anatomy of ndarrays

Unlike Python lists, NumPy arrays are stored in a single continuous place in memory so that processes can access and manipulate them very efficiently.

This behavior, so-called **locality of reference**, is the main reason why NumPy ndarrays are faster than Python lists.



Fixed Size

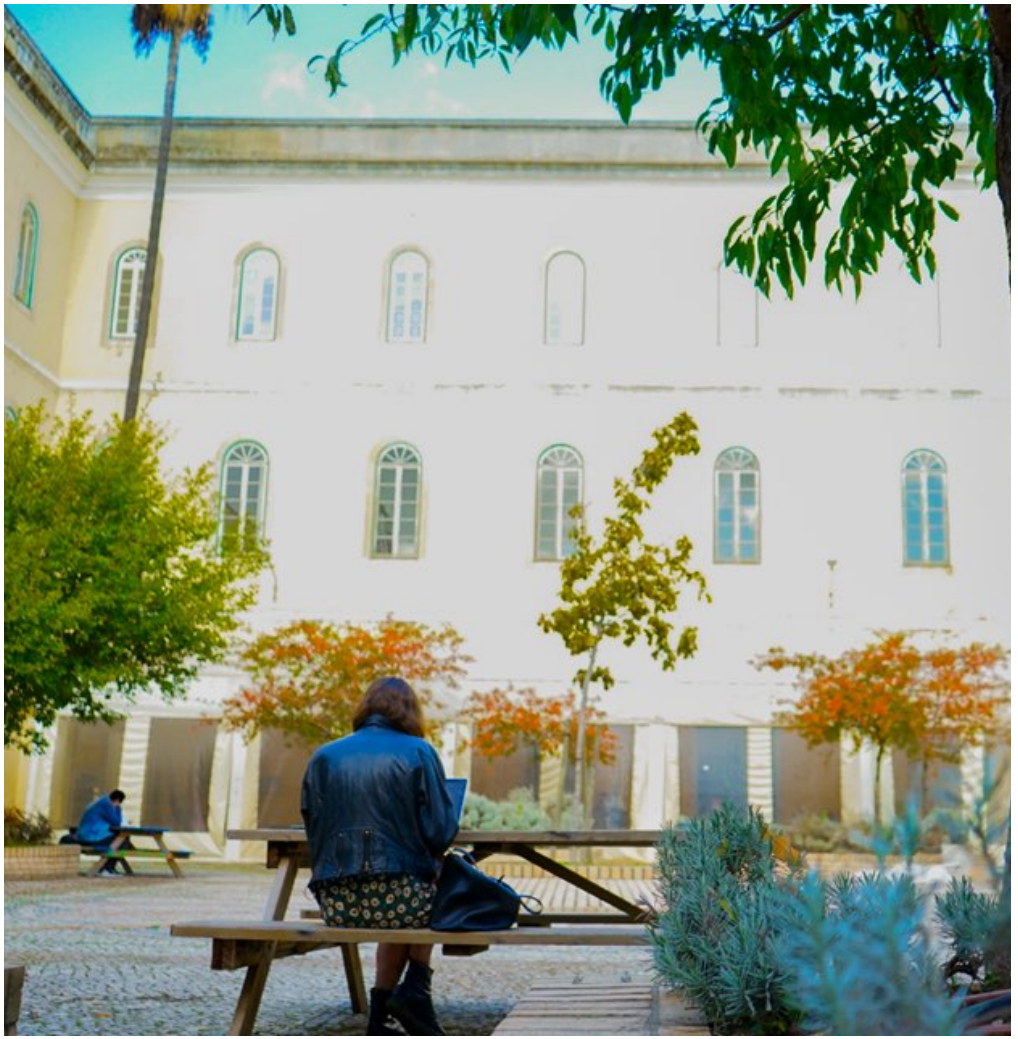
Unlike NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an **ndarray** will create a new array and delete the original.

Type Homogeneous

The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.

Lecture 6

- ~~Intro to Numpy~~
- **Array Creation and Basic Ops**
- Array indexing, slicing, and reshaping
- Vectorization and Broadcasting
- Misc
- Intro to Scipy



Creating Arrays

You can create numpy arrays from lists

```
a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b.shape)                     # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

```
<class 'numpy.ndarray'>
(3,)
1 2 3
[5 2 3]
(2, 3)
1 2 4
```

Creating Arrays

You can create numpy arrays from lists

```
a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])   # Create a rank 2 array
print(b.shape)                    # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])  # Prints "1 2 4"
```

```
<class 'numpy.ndarray'>
(3,)
1 2 3
[5 2 3]
(2, 3)
1 2 4
```

Vector

2x3 Matrix

Creating Arrays

Mutidimensional from lists of lists

```
listlist = [[2,3,5],[9,3,4],[1,2,4]]
print(listlist)
print(type(listlist))
```

```
[[2, 3, 5], [9, 3, 4], [1, 2, 4]]
<class 'list'>
```

```
marray = np.array(listlist)
print(marray)
print(type(marray))
```

```
[[2 3 5]
 [9 3 4]
 [1 2 4]]
<class 'numpy.ndarray'>
```

Arrays Attributes

```
marray.shape # returns a tuple with the number of elements per dimension
```

```
(3, 3)
```

```
marray.ndim # returns an integer with the number of dimensions, which is the same as the length of .shape
```

```
2
```

```
marray.size # returns the total numer of elements hold by the ndarray
```

```
9
```

```
marray.dtype # data type of the ndarray
```

```
dtype('int64')
```

Data Types and Conversion

Examples of numpy dtypes

Data type	Description
bool_	Boolean (True or False) stored as a byte
int_	Default integer type (same as C long; normally either int64 or in32
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float_	Shorthand for float64
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa

Data Types and Conversion

generating an array with random integers.

```
X = np.random.randint(1, 101, size=(3, 4))
```

```
X
```

```
array([[52,  1, 27, 41],
       [82, 77, 46,  2],
       [39, 25, 75, 40]])
```

```
X.dtype
```

```
dtype('int64')
```

We can convert the type of data on an array with .astype()

```
X = X.astype(float)
```

```
X.dtype
```

```
dtype('float64')
```

Generating functions

.arange() and .linspace() to create arrays

```
e = np.linspace(0,10,3)
print(e)
```

Returns a List with values between 0 and 10 with three evenly spaced values

```
[ 0.  5. 10.]
```

```
e = np.arange(0,10,2)
print(e)
```

Returns a List with values between 0 and 10 in steps of 2

```
[0 2 4 6 8]
```

Generating functions

functions fo generate Matrices

```
print(np.zeros((3,3)))
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

matrix of zeros

```
print(np.eye(3))
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

square matrix of with diagonal of ones

```
print(np.ones((3,3)))
```

```
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

matrix of ones

```
print(np.full((3,3),2))
```

```
[[2 2 2]
 [2 2 2]
 [2 2 2]]
```

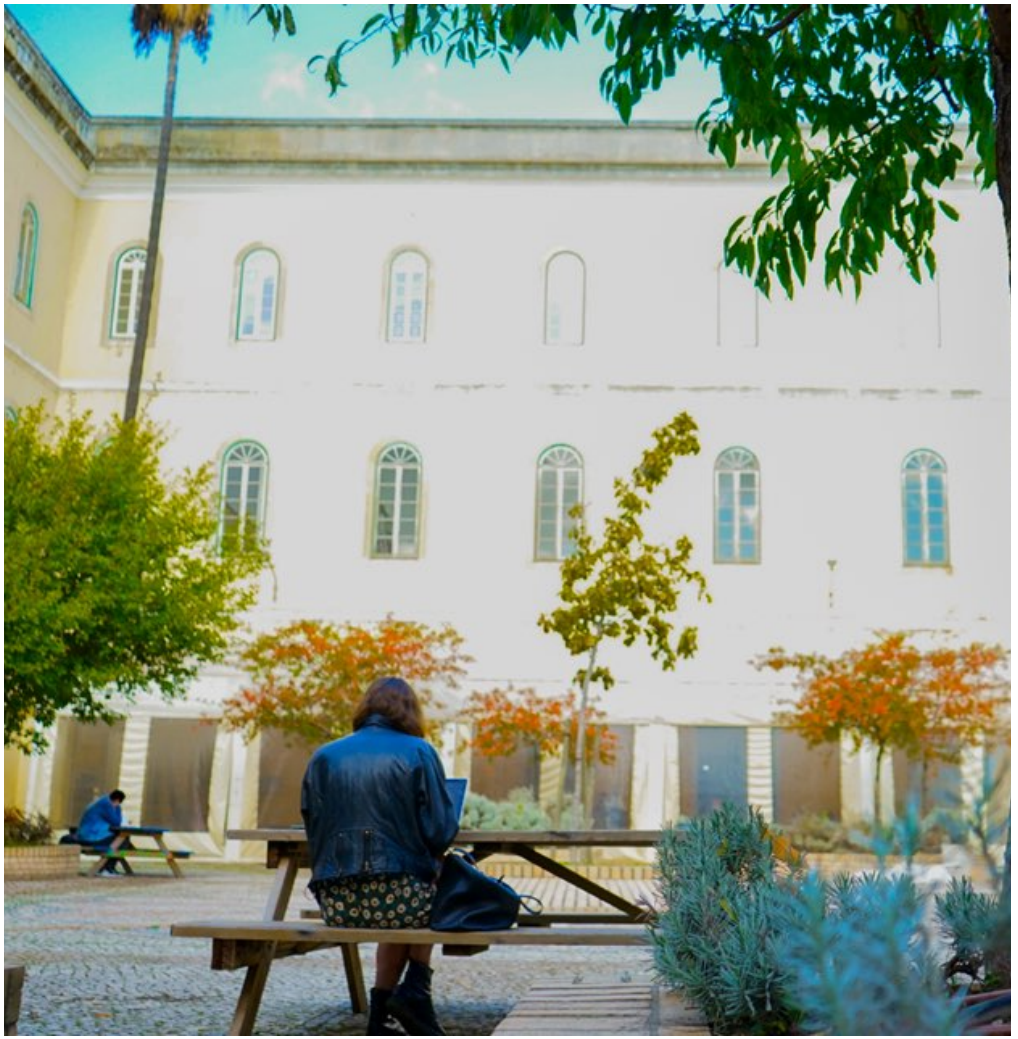
matrix filled with constant value

Generating functions

Function	Description
`array`	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a data type or explicitly specifying a data type; copies the input data by default
`asarray`	Convert input to ndarray, but do not copy if the input is already an ndarray
`arange`	Like the built-in `range` but returns an ndarray instead of a list
`ones, ones_like`	Produce an array of all 1s with the given shape and data type; `ones_like` takes another array and produces a `ones` array of the same shape and data type
`zeros, zeros_like`	Like `ones` and `ones_like` but producing arrays of 0s instead
`empty, empty_like`	Create new arrays by allocating new memory, but do not populate with any values like `ones` and `zeros`
`full, full_like`	Produce an array of the given shape and data type with all values set to the indicated "fill value"; `full_like` takes another array and produces a filled array of the same shape and data type
`eye, identity`	Create a square N × N identity matrix (1s on the diagonal and 0s elsewhere)

Lecture 6

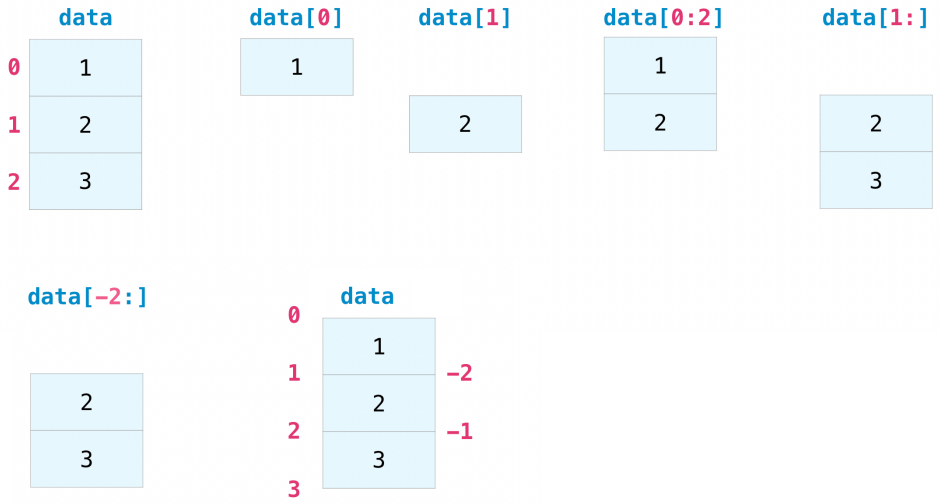
- ~~Intro to Numpy~~
- ~~Array Creation and Basic Ops~~
- **Array indexing, slicing, and reshaping**
- Vectorization and Broadcasting
- Misc
- Intro to Scipy



1D and multidimensional indexing

You can index and slice 1D NumPy arrays in the same ways you can slice Python lists

visually you can think of it as




```
data = np.array([1, 2, 3])
```

```
data[1]
```

```
np.int64(2)
```

```
data[0:2]
```

```
array([1, 2])
```

```
data[1:]
```

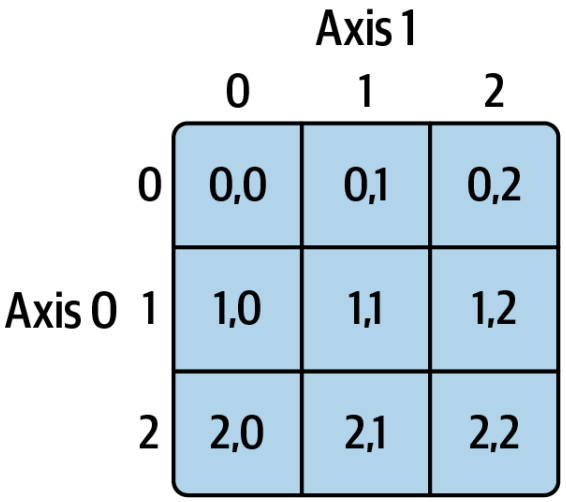
```
array([2, 3])
```

```
data[-2:]
```

```
array([2, 3])
```

1D and multidimensional indexing

With higher dimensional arrays, you have many more options. In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays




```
arr2d
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
arr2d[2] #returns the third row
```

```
array([7, 8, 9])
```

```
arr2d[0][2] #access single elements recursively
```

```
np.int64(3)
```

```
arr2d[0,2] #access with a comma-separated list of indices
```

1D and multidimensional indexing

Slicing Examples in multidimensions

	0	1	2
0	2	3	5
1	9	3	4
2	1	2	4

```
marray[0:2,0:2]
```

```
array([[2, 3],
       [9, 3]])
```

```
marray[0:2,1:3]
```

```
array([[3, 5],
       [3, 4]])
```

```
marray[1:3,1:2]
```

```
array([[3],
       [2]])
```

```
marray[2,1:3]
```

```
array([2, 4])
```

1D and multidimensional indexing

Slicing Examples

	0	1	2
0	2	3	5
1	9	3	4
2	1	2	4

```
marray[0:2,0:2]
```

```
array([[2, 3],
       [9, 3]])
```

```
marray[0:2,1:3]
```

```
array([[3, 5],
       [3, 4]])
```

```
marray[1:3,1:2]
```

```
array([[3],
       [2]])
```

```
marray[2,1:3]
```

```
array([2, 4])
```

1D and multidimensional indexing

Slicing Examples

	0	1	2
0	2	3	5
1	9	3	4
2	1	2	4

```
marray[0:2,0:2]
```

```
array([[2, 3],
       [9, 3]])
```

```
marray[0:2,1:3]
```

```
array([[3, 5],
       [3, 4]])
```

```
marray[1:3,1:2]
```

```
array([[3],
       [2]])
```

```
marray[2,1:3]
```

```
array([2, 4])
```

1D and multidimensional indexing

Slicing Examples

	0	1	2
0	2	3	5
1	9	3	4
2	1	2	4

```
marray[0:2,0:2]
```

```
array([[2, 3],
       [9, 3]])
```

```
marray[0:2,1:3]
```

```
array([[3, 5],
       [3, 4]])
```

```
marray[1:3,1:2]
```

```
array([[3],
       [2]])
```

```
marray[2,1:3]
```

```
array([2, 4])
```

1D and multidimensional indexing

Slicing Examples

	0	1	2
0	2	3	5
1	9	3	4
2	1	2	4

```
marray[0:2,0:2]
```

```
array([[2, 3],
       [9, 3]])
```

```
marray[0:2,1:3]
```

```
array([[3, 5],
       [3, 4]])
```

```
marray[1:3,1:2]
```

```
array([[3],
       [2]])
```

```
marray[2,1:3]
```

```
array([2, 4])
```

1D and multidimensional indexing

Boolean Indexing

```
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

```
a[a < 5] # get elements that meet a criteria
```

```
array([1, 2, 3, 4])
```

```
a[a%2==0] # even get elements that are divisible by two
```

```
array([ 2,  4,  6,  8, 10, 12])
```

```
a[(a>2) & (a<11)] # satisfying two conditions using the operators & or |
```

```
array([ 3,  4,  5,  6,  7,  8,  9, 10])
```

Selecting data from an array by Boolean indexing and assigning the result to a new variable always creates a copy of the data, even if the returned array is unchanged.

```
data
```

```
array([[  4,   7],
       [  0,   2],
       [ -5,   6],
       [  0,   0],
       [  1,   2],
       [-12,  -4],
       [  3,   4]])
```

```
data[data < 0]
```

```
array([ -5, -12,  -4])
```

We can use the boolean indexing to replace values that meet a certain criteria

```
data[data < 0] = 0
```

```
data
```

```
array([[4, 7],
       [0, 2],
       [0, 6],
       [0, 0],
       [1, 2],
       [0, 0],
       [3, 4]])
```

Fancy Indexing

is a term adopted by NumPy to describe indexing using integer arrays. Fancy Indexing always **generates a copy** of the data into a new array!

```
arr
```

```
array([[0., 0., 0., 0.],
       [1., 1., 1., 1.],
       [2., 2., 2., 2.],
       [3., 3., 3., 3.],
       [4., 4., 4., 4.],
       [5., 5., 5., 5.],
       [6., 6., 6., 6.],
       [7., 7., 7., 7.]])
```

```
arr[[4, 3, 0, 6]]
```

```
array([[4., 4., 4., 4.],
       [3., 3., 3., 3.],
       [0., 0., 0., 0.],
       [6., 6., 6., 6.]])
```

we can pass a list of indicies of the rows we want to sample

```
arr
```

```
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]])
```

```
arr[[1, 5, 7, 2], [0, 3, 1, 2]]
```

```
array([ 4, 23, 29, 10])
```

```
arr[[1, 5, 7, 2], [0, 3, 1, 2]] = 0
```

```
arr
```

```
array([[ 0,  1,  2,  3],
       [ 0,  5,  6,  7],
       [ 8,  9,  0, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22,  0],
       [24, 25, 26, 27],
       [28,  0, 30, 31]])
```

Passing multiple index arrays does something slightly different; it selects a one-dimensional array of elements corresponding to each tuple of indices

Reshapping

In many cases, you can convert an array from one shape to another without copying any data.

```
arr = np.arange(8)
```

```
arr
```

```
array([0, 1, 2, 3, 4, 5, 6, 7])
```

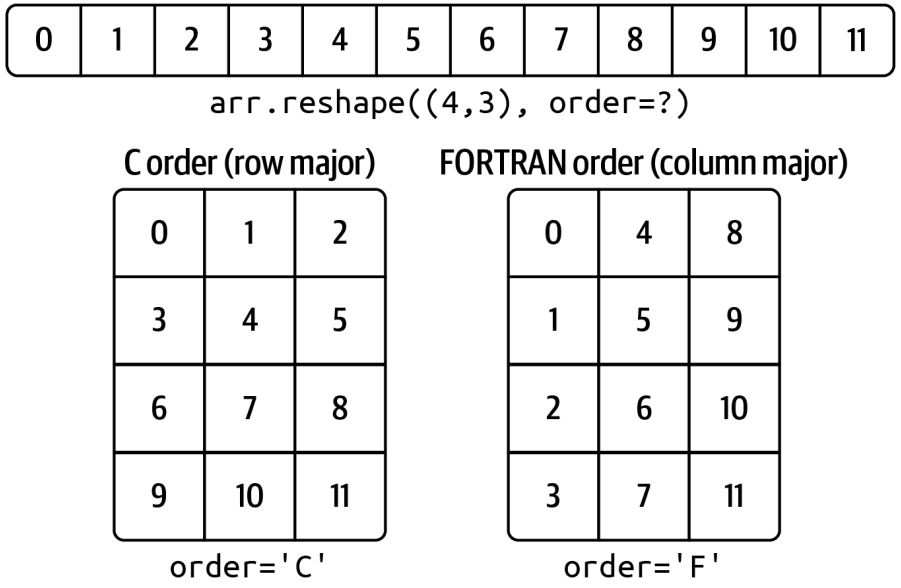
```
arr.reshape((4, 2))
```

```
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

```
arr.reshape((4, 2)).reshape((2, 4))
```

```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

You can pass the order of reshaping row-first or column first.



Reshapping

In many cases, you can convert an array from one shape to another without copying any data.

```
arr = np.arange(8)
```

```
arr
```

```
array([0, 1, 2, 3, 4, 5, 6, 7])
```

```
arr.reshape((4, 2))
```

```
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

```
arr.reshape((4, 2)).reshape((2, 4))
```

```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

The opposite operation of reshape from one-dimensional to a higher dimensional is known as flattening or raveling

```
arr = np.arange(15).reshape((5, 3))
arr
```

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

```
arr.ravel()
```

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

```
arr.flatten()
```

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

Copies and Views

While working with NumPy, you may notice that some operations return a **copy**, while others return a **view**. A copy creates a new, independent array with its own memory, while a view shares the same memory as the original array. As a result, changes made to a view also affect the original and vice versa.

View

(shallow copy)

A **view** lets you access the same data without making a copy. Changes made in the view affect the original array and vice versa, because they share memory. This makes views memory-efficient for working with large datasets. They're known as shallow copies and can be created using the .view() method.

Copy

(deep copy)

A **copy** creates a new, independent array with its own memory. Any change to the copied array won't affect the original one, and vice versa. This is useful when you want to modify data safely without touching the original. A copy is also called a deep copy and can be created using the .copy() method.

Copies and Views

Example of creating a view

```
x = np.arange(10)
x
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
y = x[1:3] # creates a view
y
```

```
array([1, 2])
```

```
x[1:3] = [10,11]
x
```

```
array([ 0, 10, 11,  3,  4,  5,  6,  7,  8,  9])
```

```
y
```

```
array([10, 11])
```

here y gets changed when x is change because it is a view!

In NumPy, you can check whether an array is a view or a copy using the .base attribute.

```
y.base
```

```
array([ 0, 10, 11,  3,  4,  5,  6,  7,  8,  9])
```

```
x.base is None
```

```
True
```

If .base returns None, the array owns the data (i.e., it's a copy).
If .base returns another array (typically the original), then it's a view and does not own the data.

Copies and Views

```
x = np.arange(10)
x
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
y = x
```

```
print(id(y))
print(id(x))
```

```
4489622416
4489622416
```

```
y[0]=10
```

```
x
```

```
array([10,  1,  2,  3,  4,  5,  6,  7,  8,  9])
```

```
y
```

```
array([10,  1,  2,  3,  4,  5,  6,  7,  8,  9])
```

When you assign an array to another variable using =, you're not creating a copy or a view. You're just creating a new reference (alias) to the same array. Both variables point to the same data, so changes made through one will appear in the other.

y is not a new object, it just points to the same data as x. Their IDs are the same. Changing y[0] to 10 also changes x[0], because they're the same array under different names.

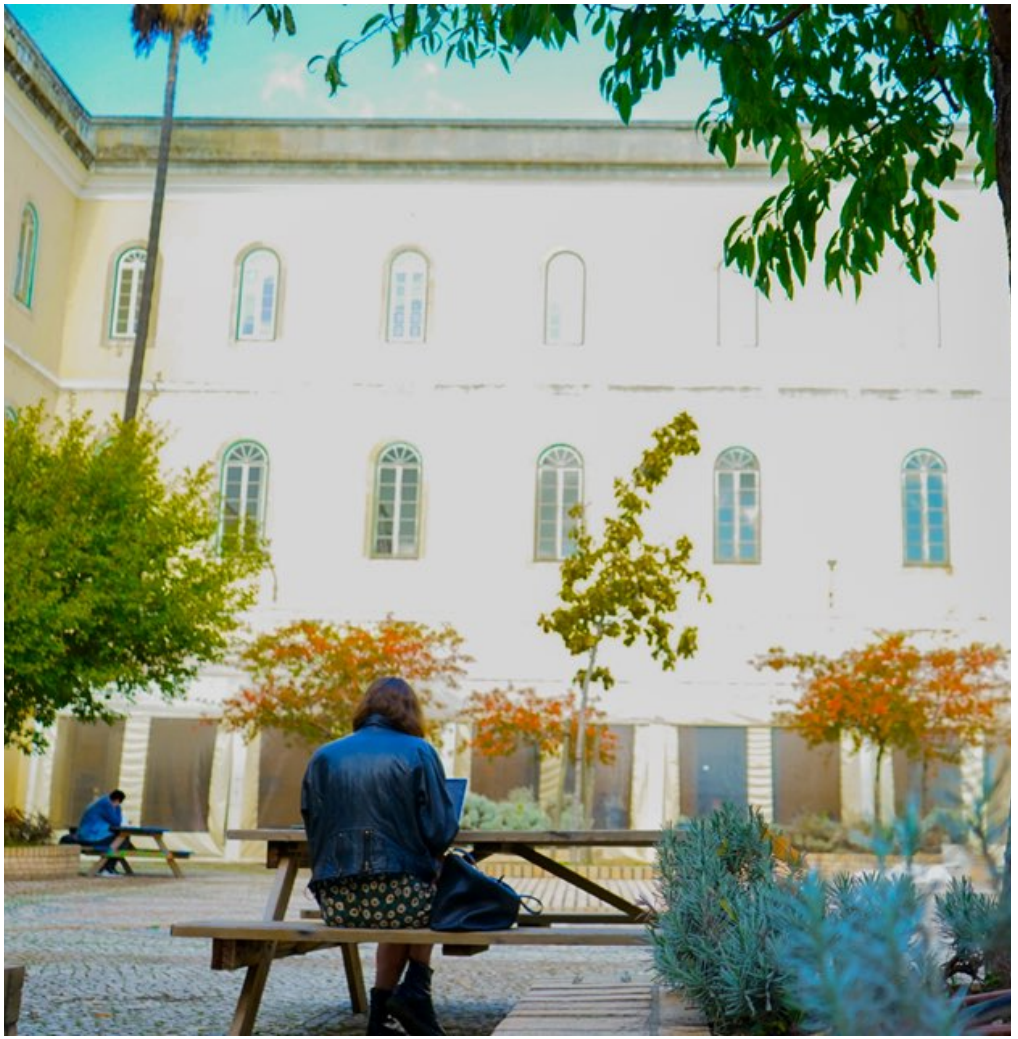
You can use the .copy() method to force a deep copy and create a different object.

```
y = x.copy()
print(id(x))
print(id(y))
```

```
4489622416
4488382096
```

Lecture 6

- ~~Intro to Numpy~~
- ~~Array Creation and Basic Ops~~
- ~~Array indexing, slicing, and reshaping~~
- **Vectorization and Broadcasting**
- Misc
- Intro to Scipy



What is vectorization and why it matters

Vectorization describes the absence of any explicit looping, indexing, etc., in the code - these things are taking place, of course, just “behind the scenes” in optimized, pre-compiled C code. Vectorized code has many advantages, among which are:

- vectorized code is more concise and easier to read
- fewer lines of code generally means fewer bugs
- the code more closely resembles standard mathematical notation (making it easier, typically, to correctly code mathematical constructs)
- vectorization results in more “Pythonic” code. Without vectorization, our code would be littered with inefficient and difficult to read for loops..

Universal Functions

A universal function (or ufunc for short) is a function that operates on ndarrays in an element-by-element fashion, supporting array broadcasting, type casting, and several other standard features. That is, a ufunc is a “vectorized” wrapper for a function that takes a fixed number of specific inputs and produces a fixed number of specific outputs.

In NumPy, universal functions are instances of the numpy.ufunc class. Many of the built-in functions are implemented in compiled C code. The basic ufuncs operate on scalars, but there is also a generalized kind for which the basic elements are sub-arrays (vectors, matrices, etc.), and broadcasting is done over other dimensions. One can also produce custom ufunc instances using the frompyfunc factory function.

Universal Functions

Many ufuncs are simple element-wise transformations.
These are referred to as unary ufuncs

```
arr = np.arange(10)
arr
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
np.sqrt(arr)
```

```
array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
       2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.        ])
```

```
np.exp(arr)
```

```
array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
       5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,
       2.98095799e+03, 8.10308393e+03])
```

```
np.sin(arr)
```

```
array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ,
       -0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849])
```

Universal Functions

Table 4.4: Some unary universal functions

Function	Description
`abs, fabs`	Compute the absolute value element-wise for integer, floating-point, or complex values
`sqrt`	Compute the square root of each element (equivalent to `arr ** 0.5`)
`square`	Compute the square of each element (equivalent to `arr ** 2`)
`exp`	Compute the exponent e^x of each element
`log, log10, log2, log1p`	Natural logarithm (base *e*), log base 10, log base 2, and log(1 + x), respectively
`sign`	Compute the sign of each element: 1 (positive), 0 (zero), or −1 (negative)
`ceil`	Compute the ceiling of each element (i.e., the smallest integer greater than or equal to that number)
`floor`	Compute the floor of each element (i.e., the largest integer less than or equal to each element)
`rint`	Round elements to the nearest integer, preserving the `dtype`
`modf`	Return fractional and integral parts of array as separate arrays
`isnan`	Return Boolean array indicating whether each value is `NaN` (Not a Number)
`isfinite, isinf`	Return Boolean array indicating whether each element is finite (non-`inf`, non-`NaN`) or infinite, respectively
`cos, cosh, sin, sinh, tan, tanh`	Regular and hyperbolic trigonometric functions
`arccos, arccosh, arcsin, arcsinh, arctan, arctanh`	Inverse trigonometric functions
`logical_not`	Compute truth value of `not x` element-wise (equivalent to `~arr`)

Universal Functions

Many ufuncs are simple element-wise transformations. These are referred to as unary ufuncs

Others, such as numpy.add or numpy.maximum, take two arrays (thus, binary ufuncs) and return a single array as the result

```
x = np.random.standard_normal(8)
y = np.random.standard_normal(8)
```

```
np.maximum(x, y)
```

```
array([ 0.41399566, -0.28529872,  0.49113275,  0.59101698,  0.64658696,
        0.23798213, -0.91840197,  0.12962196])
```

```
np.add(x,y)
```

```
array([-0.21858542, -1.48636861, -0.62138696, -0.36532498,  1.29109079,
        0.35834392, -2.33629218, -1.92964489])
```

Universal Functions

Table 4.5: Some binary universal functions

Function	Description
`add`	Add corresponding elements in arrays
`subtract`	Subtract elements in second array from first array
`multiply`	Multiply array elements
`divide, floor_divide`	Divide or floor divide (truncating the remainder)
`power`	Raise elements in first array to powers indicated in second array
`maximum, fmax`	Element-wise maximum; `fmax` ignores `NaN`
`minimum, fmin`	Element-wise minimum; `fmin` ignores `NaN`
`mod`	Element-wise modulus (remainder of division)
`copysign`	Copy sign of values in second argument to values in first argument
`greater, greater_equal, less, less_equal, equal, not_equal`	Perform element-wise comparison, yielding Boolean array (equivalent to infix operators `>, >=, <, <=, ==, !=`)
`logical_and`	Compute element-wise truth value of AND (`&`) logical operation
`logical_or`	Compute element-wise truth value of OR (`|`) logical operation
`logical_xor`	Compute element-wise truth value of XOR (`^`) logical operation

Element Wise Operations

```
d = np.eye(10)
```

```
np.multiply(m,d)
```

```
array([[0.69, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.47, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.4 , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.86, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.23, 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.9 , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.96, 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.89, 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.8 , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.68]])
```

Element Wise Operations

```
np.divide(m,np.full((10,10),0.5))
```

```
array([[1.38, 0.78, 1.36, 1.16, 0.7 , 0.74, 1.74, 1.58, 1.62, 0.1 ],
       [0.6 , 0.94, 0.86, 0.22, 1.94, 0.86, 1.4 , 1.46, 0.86, 0.24],
       [1.94, 1.12, 0.8 , 1.26, 0.9 , 0.74, 0.18, 1.8 , 0.72, 0.56],
       [0.6 , 1.48, 1.34, 1.72, 0.46, 1.7 , 0.5 , 1.3 , 0.1 , 1.36],
       [0.32, 1.68, 1.74, 1.1 , 0.46, 0.2 , 1.64, 1.9 , 0.28, 0.84],
       [0.78, 1.84, 1.54, 0.9 , 1.68, 1.8 , 0.02, 1.08, 0.68, 1.54],
       [1.52, 1.96, 1.  , 1.24, 0.54, 1.06, 1.92, 0.44, 0.82, 0.68],
       [0.88, 1.22, 0.44, 1.78, 0.24, 1.4 , 0.92, 1.78, 0.02, 0.18],
       [1.86, 0.8 , 1.54, 0.58, 0.58, 1.6 , 1.26, 0.84, 1.6 , 0.78],
       [0.4 , 0.52, 1.4 , 0.5 , 0.6 , 0.6 , 1.14, 1.14, 1.  , 1.36]])
```

Element Wise Operations

```
np.subtract(np.full((10,10),1),m)
```

```
array([[0.31, 0.61, 0.32, 0.42, 0.65, 0.63, 0.13, 0.21, 0.19, 0.95],
       [0.7 , 0.53, 0.57, 0.89, 0.03, 0.57, 0.3 , 0.27, 0.57, 0.88],
       [0.03, 0.44, 0.6 , 0.37, 0.55, 0.63, 0.91, 0.1 , 0.64, 0.72],
       [0.7 , 0.26, 0.33, 0.14, 0.77, 0.15, 0.75, 0.35, 0.95, 0.32],
       [0.84, 0.16, 0.13, 0.45, 0.77, 0.9 , 0.18, 0.05, 0.86, 0.58],
       [0.61, 0.08, 0.23, 0.55, 0.16, 0.1 , 0.99, 0.46, 0.66, 0.23],
       [0.24, 0.02, 0.5 , 0.38, 0.73, 0.47, 0.04, 0.78, 0.59, 0.66],
       [0.56, 0.39, 0.78, 0.11, 0.88, 0.3 , 0.54, 0.11, 0.99, 0.91],
       [0.07, 0.6 , 0.23, 0.71, 0.71, 0.2 , 0.37, 0.58, 0.2 , 0.61],
       [0.8 , 0.74, 0.3 , 0.75, 0.7 , 0.7 , 0.43, 0.43, 0.5 , 0.32]])
```

```
m+np.subtract(np.full((10,10),1),m)
```

```
array([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]])
```

Overloading arithmetic operators always returns the element-wise operations

```
m*np.eye(10)
```

```
array([[0.69, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.47, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.4 , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.86, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.23, 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.9 , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.96, 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.89, 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.8 , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.68]])
```

Universal Functions

Operator	Equivalent ufunc	Description
`+`	`np.add`	Addition (e.g., `1 + 1 = 2`)
`-`	`np.subtract`	Subtraction (e.g., `3 - 2 = 1`)
`-`	`np.negative`	Unary negation (e.g., `-2`)
`*`	`np.multiply`	Multiplication (e.g., `2 * 3 = 6`)
`/`	`np.divide`	Division (e.g., `3 / 2 = 1.5`)
`//`	`np.floor_divide`	Floor division (e.g., `3 // 2 = 1`)
`**`	`np.power`	Exponentiation (e.g., `2 ** 3 = 8`)
`%`	`np.mod`	Modulus/remainder (e.g., `9 % 4 = 1`)

Universal Functions

```
a = np.arange(1,5,1)
print(a)
print("addition a + 2:", np.add(a,10))
print("addition a + a:", np.add(a,a))
print("e^a:", np.exp(a))
print("2^a:", np.exp2(a))
print("ln(a):", np.log(a))
print("log10(a):", np.log10(a))
```

```
[1 2 3 4]
addition a + 2: [11 12 13 14]
addition a + a: [2 4 6 8]
e^a: [ 2.71828183  7.3890561  20.08553692 54.59815003]
2^a: [ 2.  4.  8. 16.]
ln(a): [0.         0.69314718 1.09861229 1.38629436]
log10(a): [0.         0.30103    0.47712125 0.60205999]
```

sum, min, max, mean

Operations like sum, mean, max are optimized with much faster than the traditional Python approach of looping through elements. These are aggregation operations-

vectorized

```
%timeit np.sum(np.arange(15000))
```

```
5.16 µs ± 53.9 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

non-vectorized

```
%timeit sum(range(15000))
```

```
120 µs ± 1.36 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

```
a = np.array([[0, 1, 1], [1, 0, 2], [1, 2, 0]])
```

```
a.sum()
```

```
np.int64(8)
```

```
a.mean()
```

```
np.float64(0.8888888888888888)
```

```
a.sum(axis=0)
```

```
array([2, 3, 3])
```

```
a.max()
```

```
np.int64(2)
```

```
a.max(axis=1)
```

```
array([1, 2, 2])
```

```
a.max(axis=0)
```

```
array([1, 2, 2])
```

matrix multiplications

```
np.round(np.random.random((10,10)),decimals=2)
```

```
array([[0.13, 0.72, 0.38, 0.7 , 0.91, 0.55, 0.33, 0.23, 0.41, 0.43],
       [0.54, 0.67, 0.51, 0.94, 0.82, 0.95, 0.09, 0.8 , 0.18, 0.34],
       [0.64, 0.23, 0.07, 0.57, 0.35, 0.53, 0.  , 0.93, 0.32, 0.3 ],
       [0.22, 0.74, 0.12, 0.83, 0.73, 0.61, 0.96, 0.37, 0.39, 0.93],
       [0.49, 0.93, 0.49, 0.9 , 0.9 , 0.62, 0.34, 0.2 , 0.92, 0.52],
       [0.59, 0.7 , 0.89, 0.2 , 0.84, 0.55, 0.82, 0.45, 0.25, 0.96],
       [0.86, 0.39, 0.05, 0.51, 0.04, 0.08, 0.06, 0.27, 0.41, 0.6 ],
       [0.46, 0.13, 0.34, 0.47, 0.98, 0.23, 0.56, 0.8 , 0.87, 0.43],
       [0.21, 0.78, 0.76, 0.31, 0.76, 0.64, 0.32, 0.27, 0.62, 0.5 ],
       [0.81, 0.68, 0.21, 0.93, 0.11, 0.42, 0.05, 0.84, 0.56, 0.55]])
```

matrix multiplications

dot product

$$\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \cdot \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} = \begin{bmatrix} a_{00}*b_{00} + a_{01}*b_{10} & a_{00}*b_{01} + a_{01}*b_{11} \\ a_{10}*b_{00} + a_{11}*b_{10} & a_{10}*b_{01} + a_{11}*b_{11} \end{bmatrix}$$

matrix multiplications

m.T returns the transpose of m

```
np.round(m.dot(m.T),decimals=2)
```

dot product between m and its transpose

```
array([[3.73, 2.79, 2.91, 2.65, 3.06, 2.79, 3.25, 2.62, 3.43, 2.46],
       [2.79, 2.86, 2.3 , 2.16, 2.52, 2.77, 2.51, 2.02, 2.59, 2.04],
       [2.91, 2.3 , 3.18, 2.75, 2.56, 3.02, 2.72, 2.6 , 2.88, 1.96],
       [2.65, 2.16, 2.75, 3.55, 2.98, 3.55, 2.97, 2.87, 2.82, 2.26],
       [3.06, 2.52, 2.56, 2.98, 3.62, 2.93, 3.03, 2.62, 2.65, 2.46],
       [2.79, 2.77, 3.02, 3.55, 2.93, 4.31, 3.1 , 2.59, 3.22, 2.5 ],
       [3.25, 2.51, 2.72, 2.97, 3.03, 3.1 , 3.78, 2.67, 3.32, 2.26],
       [2.62, 2.02, 2.6 , 2.87, 2.62, 2.59, 2.67, 2.92, 2.38, 1.7 ],
       [3.43, 2.59, 2.88, 2.82, 2.65, 3.22, 3.32, 2.38, 3.79, 2.49],
       [2.46, 2.04, 1.96, 2.26, 2.46, 2.5 , 2.26, 1.7 , 2.49, 2.2 ]])
```

matrix multiplications

```
print(marray)
```

```
[[2 3 5]
 [9 3 4]
 [1 2 4]]
```

```
print(marray.T)
```

```
[[2 9 1]
 [3 3 2]
 [5 4 4]]
```

```
print(marray.transpose())
```

```
[[2 9 1]
 [3 3 2]
 [5 4 4]]
```

Broadcasting

Broadcasting governs how operations work between arrays of different shapes. It can be a powerful feature, but it can cause confusion, even for experienced users. The simplest example of broadcasting occurs when combining a scalar value with an array:

(Here we say that the scalar value 5 has been broadcast to all of the other elements in the multiplication operation)

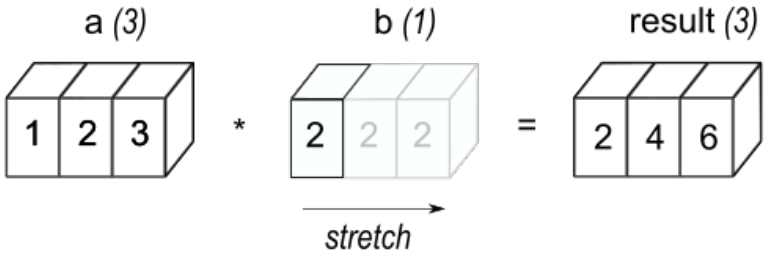
```
arr = np.arange(5)
```

```
arr
```

```
array([0, 1, 2, 3, 4])
```

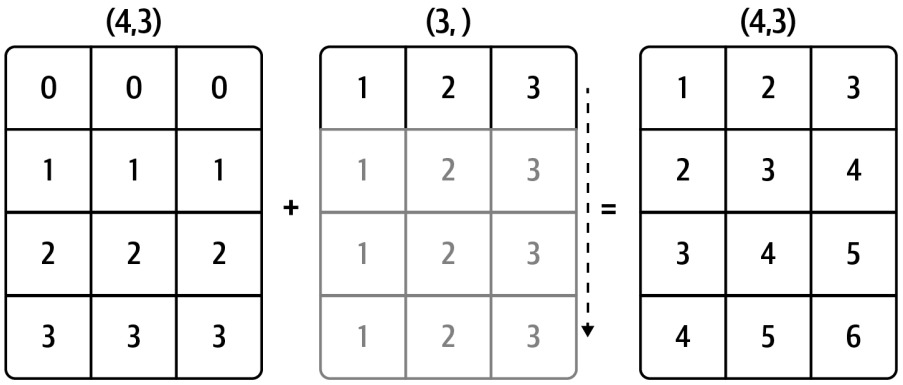
```
arr*5
```

```
array([ 0,  5, 10, 15, 20])
```



Broadcasting

T**he broadcasting rule:** Two arrays are compatible for broadcasting if for each trailing dimension (i.e., starting from the end) the axis lengths match or if either of the lengths is 1. Broadcasting is then performed over the missing or length 1 dimensions.




```
arr = rng.standard_normal((4, 3))
arr
```

```
array([[ 0.468,  0.891,  1.023],
       [ 0.312, -0.062, -0.359],
       [-0.749, -0.965,  0.36 ],
       [-0.245, -1.996, -0.155]])
```

```
arr.mean(0)
```

```
array([-0.053, -0.533,  0.217])
```

```
demeaned = arr-arr.mean(0)
demeaned
```

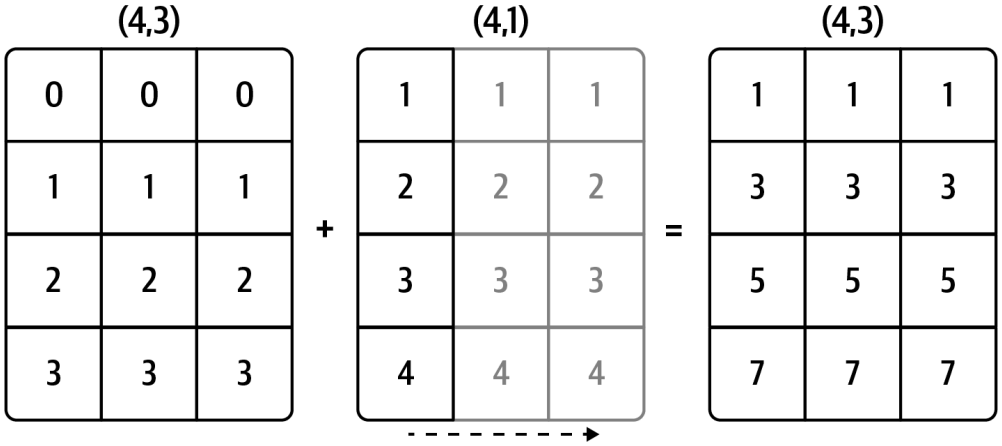
```
array([[ 0.521,  1.424,  0.806],
       [ 0.366,  0.471, -0.577],
       [-0.695, -0.432,  0.143],
       [-0.191, -1.463, -0.372]])
```

```
demeaned.mean(0)
```

```
array([-0.,  0., -0.])
```

Broadcasting

T**he broadcasting rule:** Two arrays are compatible for broadcasting if for each trailing dimension (i.e., starting from the end) the axis lengths match or if either of the lengths is 1. Broadcasting is then performed over the missing or length 1 dimensions.




```
row_means = arr.mean(1)
```

```
row_means.shape
```

```
(4,)
```

```
row_means.reshape((4,1))
```

```
array([[ 0.794],
       [-0.036],
       [-0.451],
       [-0.799]])
```

```
demeaned = arr -row_means.reshape((4,1))
demeaned
```

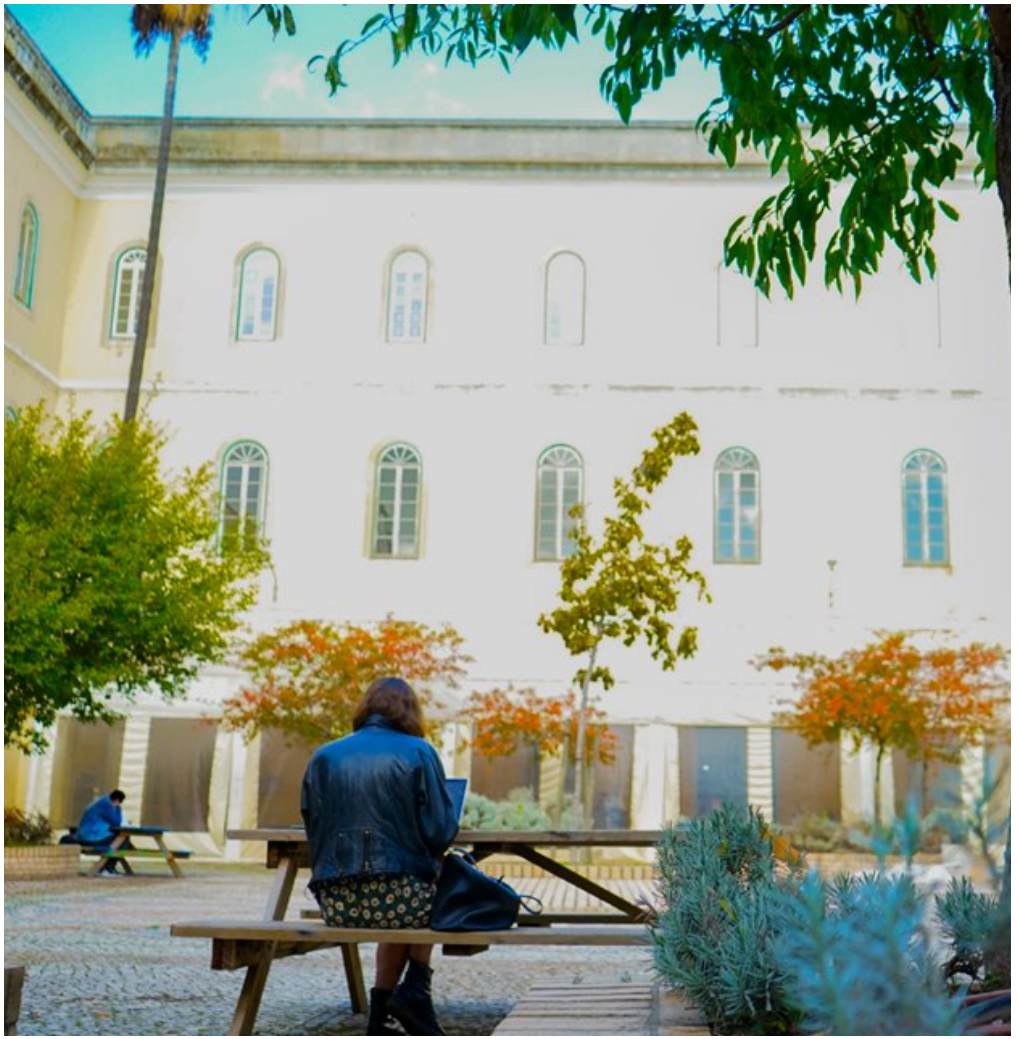
```
array([[-0.326,  0.097,  0.229],
       [ 0.349, -0.026, -0.323],
       [-0.297, -0.514,  0.811],
       [ 0.554, -1.197,  0.643]])
```

```
demeaned.mean(1)
```

```
array([0., 0., 0., 0.])
```

Lecture 6

- ~~Intro to Numpy~~
- ~~Array Creation and Basic Ops~~
- ~~Array indexing, slicing, and reshaping~~
- ~~Vectorization and Broadcasting~~
- **Misc**
- Intro to Scipy



numpy.random

The numpy.random module supplements the built-in Python random module with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions. For example, you can get a 4 × 4 array of samples from the standard normal distribution using numpy.random.standard_normal

```
samples = np.random.standard_normal(size=(4,4))
```

```
samples
```

```
array([[ 1.15720818e+00,  1.57651625e+00,  5.48107680e-01,
        -7.92443543e-01],
       [ 8.77013891e-01, -6.61391081e-01, -1.07780023e+00,
        -6.22621436e-02],
       [ 2.68076422e-01, -1.88888724e-03, -3.78623451e-01,
        -1.20479768e+00],
       [ 2.20133778e+00,  7.61080949e-01,  6.57339925e-01,
         5.80112857e-01]])
```

numpy.random

Method	Description
`permutation`	Return a random permutation of a sequence, or return a permuted range
`shuffle`	Randomly permute a sequence in place
`uniform`	Draw samples from a uniform distribution
`integers`	Draw random integers from a given low-to-high range
`standard_normal`	Draw samples from a normal distribution with mean 0 and standard deviation 1
`binomial`	Draw samples from a binomial distribution
`normal`	Draw samples from a normal (Gaussian) distribution
`beta`	Draw samples from a beta distribution
`chisquare`	Draw samples from a chi-square distribution
`gamma`	Draw samples from a gamma distribution
`uniform`	Draw samples from a uniform [0, 1) distribution

numpy.random

The numpy.random module supplements the built-in Python random module with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions. For example, you can get a 4 × 4 array of samples from the standard normal distribution using numpy.random.standard_normal

```
from random import normalvariate
N = 1000000
```

```
%timeit samples = [normalvariate(0, 1) for _ in range(N)]
```

```
282 ms ± 2.77 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%timeit np.random.standard_normal(N)
```

```
9.95 ms ± 10.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

These random numbers are not truly random (rather, pseudorandom) but instead are generated by a configurable random number generator that determines deterministically what values are created. Functions like numpy.random.standard_normal use the numpy.random module's default random number generator, but your code can be configured to use an explicit generator

numpy.random

The numpy.random module supplements the built-in Python random module with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions. For example, you can get a 4 × 4 array of samples from the standard normal distribution using numpy.random.standard_normal

These random numbers are not truly random (rather, pseudorandom) but instead are generated by a configurable random number generator that determines deterministically what values are created. Functions like numpy.random.standard_normal use the numpy.random module's default random number generator, but your code can be configured to use an explicit generator

```
rng = np.random.default_rng(seed=12345)
```

```
data = rng.standard_normal((2, 3))
data
```

```
array([[-1.42382504,  1.26372846, -0.87066174],
       [-0.25917323, -0.07534331, -0.74088465]])
```

```
data = rng.standard_normal((2, 3))
data
```

```
array([[-1.3677927 ,  0.6488928 ,  0.36105811],
       [-1.95286306,  2.34740965,  0.96849691]])
```

```
rng = np.random.default_rng(seed=12345)
data = rng.standard_normal((2, 3))
data
```

```
array([[-1.42382504,  1.26372846, -0.87066174],
       [-0.25917323, -0.07534331, -0.74088465]])
```

numpy.linalg

Linear algebra operations, like matrix multiplication, decompositions, determinants, and other square matrix math, are an important part of many array libraries. Multiplying two two-dimensional arrays with * is an element-wise product, while matrix multiplications require either using the dot function or the @ infix operator. dot is both an array method and a function in the numpy namespace for doing matrix multiplication:

```
x = np.array([[1., 2., 3.], [4., 5., 6.]])
y = np.array([[6., 23.], [-1, 7], [8, 9]])
```

```
x.dot(y)
```

```
array([[ 28.,  64.],
       [ 67., 181.]])
```

```
np.dot(x,y)
```

```
array([[ 28.,  64.],
       [ 67., 181.]])
```

```
x@y
```

```
array([[ 28.,  64.],
       [ 67., 181.]])
```

numpy.linalg

numpy.linalg has a standard set of matrix decompositions and things like inverse and determinant

Table 4.8: Commonly used `numpy.linalg` functions

Function	Description
`diag`	Return the diagonal (or off-diagonal) elements of a square matrix as a 1D array, or convert a 1D array into a square matrix with zeros on the off-diagonal
`dot`	Matrix multiplication
`trace`	Compute the sum of the diagonal elements
`det`	Compute the matrix determinant
`eig`	Compute the eigenvalues and eigenvectors of a square matrix
`inv`	Compute the inverse of a square matrix
`pinv`	Compute the Moore-Penrose pseudoinverse of a matrix
`qr`	Compute the QR decomposition
`svd`	Compute the singular value decomposition (SVD)
`solve`	Solve the linear system Ax = b for x, where A is a square matrix
`lstsq`	Compute the least-squares solution to `Ax = b`

```
from numpy.linalg import inv, qr
```

```
X = rng.standard_normal((5, 5))
```

```
mat = X.T @ X
```

```
inv(mat)
```

```
array([[ 7.45126546, 10.13993956,  3.72453623,  1.29512485,  1.95005349],
       [10.13993956, 14.29440803,  5.25721034,  1.81021024,  2.59207062],
       [ 3.72453623,  5.25721034,  2.15055078,  0.58673381,  0.88577989],
       [ 1.29512485,  1.81021024,  0.58673381,  0.3364715 ,  0.36398142],
       [ 1.95005349,  2.59207062,  0.88577989,  0.36398142,  0.66237081]])
```

```
np.round(mat@inv(mat),2)
```

```
array([[ 1., -0., -0., -0., -0.],
       [ 0.,  1., -0.,  0.,  0.],
       [ 0., -0.,  1.,  0., -0.],
       [ 0., -0.,  0.,  1., -0.],
       [-0.,  0.,  0., -0.,  1.]])
```

Numpy Application Example

Towards Building a Simple Recommendation System

Suppose you are given a table with n columns and m rows (n << m).
Columns refer to activities that tourists participate in when visiting a country.
Each Row is the report of a single tourist activity.
The entries of the table are one if a given tourist participated in a particular activity, and zero otherwise.

columns, activities

rows, tourists

1 **1** **1**

0 **1** **1**

0 **0** **1**

Numpy & Scipy Challenge

Towards Building a Simple Recommendation System

The exercise involves computing from that initial table, a new one of size m x m that summarizes the number of activities co-visited by the same tourist.

Write a function, that given a rectangular binary matrix
returns a square matrix with the number of co-occurrences of activities by row?

Start by generating a random matrix of activity participation with 5000 tourists records and 100 activities and run your code. You can use the following code to generate such table.

```
import random as rnd
M = [ [ 0 if rnd.random() <0.9 else 1 for column in range(1,100)] for row in range(1,5000) ]
```

tip: debug your function with a small matrix that you know the solution of beforehand

Numpy & Scipy Challenge

An alternative way to visualize our data is by drawing associations between the two sets of elements: individuals (tourists) and activities. See Figure to the right.

A link between an individual and an activity represents a 1 in our table, and the absence of a link is a 0.

Numpy & Scipy Challenge

Initial Table M_{ta}

Co-visits Table $O_{aa'}$

how many times are two activities visited by the same person?

Definition Co-Visit Table

$$O_{aa'} = \sum_{a} M_{ta} M_{ta'}$$

Number of times two activities are co-visited by the same person.

initial table, where t represents the index of a row (tourist), and a the index of a column (activity)

$O_{aa'}$ Co-Visit table summarizes how many times the same tourist visted a pair of activities a and a'

Numpy & Scipy Challenge

Generate a dummy bipartite matrix

```
import random
def bipartite(n,m,d):
    rnd = lambda : 1 if random.random() <= d else 0
    return [[rnd() for _ in range(n)] for _ in range(m)]
```

```
B = bipartite(1500,100, 0.25)
```

n defines the number of rows, m is the number of columns,
d is the density of ones in the matrix, we will set ~25% to ones
at random

Numpy & Scipy Challenge

Compute Cooccurrences

```
B = bipartite(1500,100, 0.25)
```

```
import time

t0 = time.time()
#Create a list of m x m
S = [[0 for _ in range(1500)] for _ in range(1500)]
for i in range(len(B)):
    for j in range(len(B[1])):
        for k in range(j+1,len(B[1])):
            if B[i][j] == 1 and B[i][k] == 1:
                S[j][k]+=1
                S[k][j]+=1
t1 = time.time()
print('Operation time: {:.2f} seconds'.format(round(t1 - t0, 2)))
```

```
Operation time: 13.19 seconds
```

Numpy & Scipy Challenge

Compute Cooccurrences

```
%%time
N = 15000
Spots = 100
Density = 0.25

B = bipartite(N,Spots,Density)
S = [[0 for _ in range(N)] for _ in range(N)]

for i in range(len(B)):
    for j in range(len(B[i])):
        for k in range(j+1,len(B[i])):
            if B[i][j] == 1 and B[i][k] == 1:
                S[j][k] += 1
                S[k][j] += 1
```

```
CPU times: user 14min 29s, sys: 637 ms, total: 14min 30s
Wall time: 14min 30s
```

Numpy & Scipy Challenge

```
%%time
N = 15000
Spots = 100
Density = 0.25

B = bipartite(N,Spots,Density)
S = [[0 for _ in range(N)] for _ in range(N)]

for i in range(len(B)):
    for j in range(len(B[i])):
        for k in range(j+1,len(B[i])):
            if B[i][j] == 1 and B[i][k] == 1:
                S[j][k] += 1
                S[k][j] += 1
```

```
CPU times: user 14min 29s, sys: 637 ms, total: 14min 30s
Wall time: 14min 30s
```

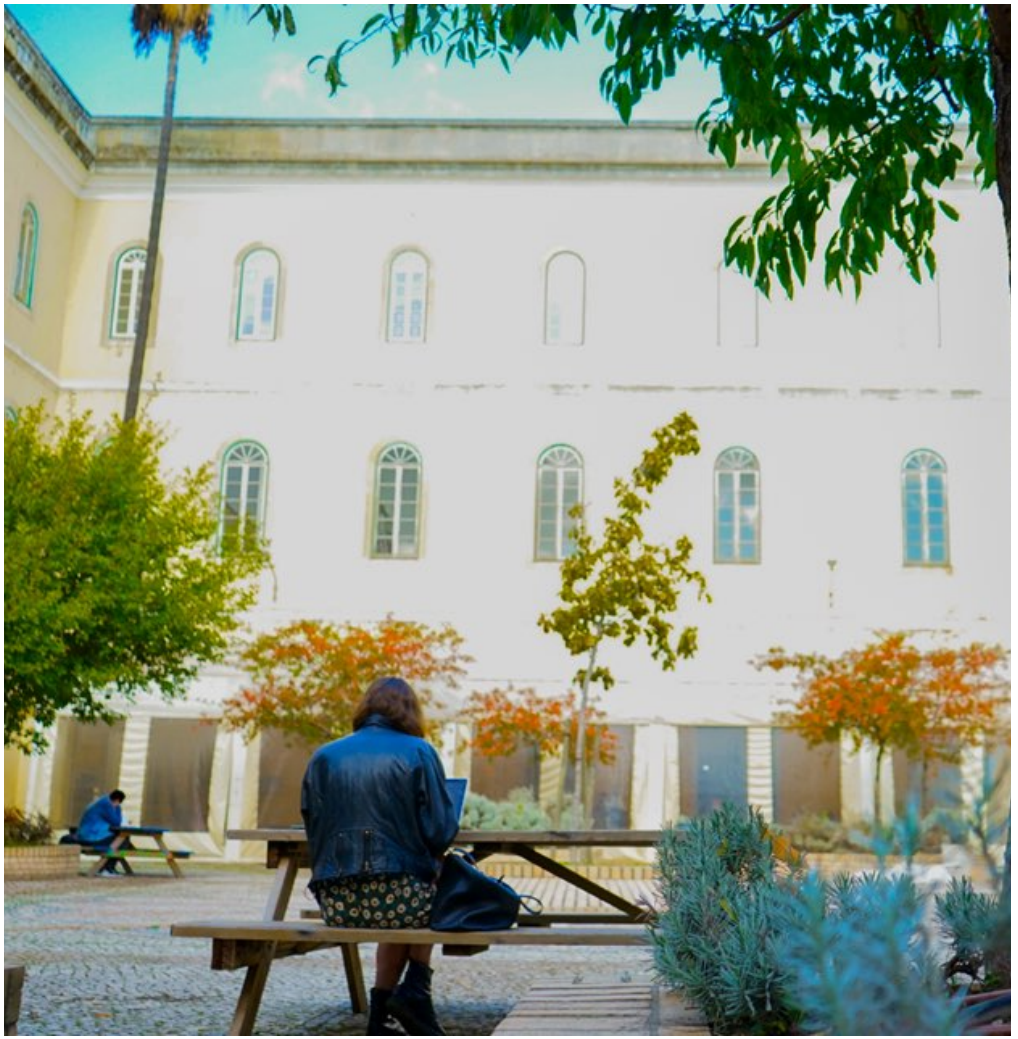
```
%%time

S = np.array(B)
SA = np.dot(S.T,S)
SA = SA - np.diag(np.diagonal(SA))
```

```
CPU times: user 11.8 s, sys: 318 ms, total: 12.2 s
Wall time: 12.2 s
```

Lecture 6

- ~~Intro to Numpy~~
- ~~Array Creation and Basic Ops~~
- ~~Array indexing, slicing, and reshaping~~
- ~~Vectorization and Broadcasting~~
- ~~Misc~~
- **Intro to Scipy**



starting scipy

```
import scipy as sc
```

```
sc.__version__
```

```
'1.16.1'
```

SciPy is a collection of mathematical algorithms and convenience functions built on NumPy . It adds significant power to Python by providing the user with high-level commands and classes for manipulating and visualizing data. SciPy is organized into subpackages covering different scientific computing domains.

Scipy modules

Subpackage	Description and User Guide
`cluster`	Clustering algorithms
`constants`	Physical and mathematical constants
`differentiate`	Finite difference differentiation tools
`fft`	Fourier Transforms (scipy.fft)
`fftpack`	Fast Fourier Transform routines (legacy)
`integrate`	Integration (scipy.integrate)
`interpolate`	Interpolation (scipy.interpolate)
`io`	File IO (scipy.io)
`linalg`	Linear Algebra (scipy.linalg)
`ndimage`	Multidimensional Image Processing (scipy.ndimage)
`odr`	Orthogonal distance regression
`optimize`	Optimization (scipy.optimize)
`signal`	Signal Processing (scipy.signal)
`sparse`	Sparse Arrays (scipy.sparse)
`spatial`	Spatial Data Structures and Algorithms (scipy.spatial)
`special`	Special Functions (scipy.special)
`stats`	Statistics (scipy.stats)

Scipy Linear Algebra

```
import scipy.linalg as linalg
```

```
linalg.inv(marray)
```

```
array([[-0.30769231,  0.15384615,  0.23076923],
       [ 2.46153846, -0.23076923, -2.84615385],
       [-1.15384615,  0.07692308,  1.61538462]])
```

Inverse of the Matrix

```
linalg.det(marray)
```

```
-13.000000000000012
```

Determinant of the Matrix

```
linalg.eigvals(marray)
```

```
array([10.24289364+0.j, -1.90805817+0.j,  0.66516452+0.j])
```

Eigenvalues of the Matrix

```
linalg.eig(marray,right=True)
```

```
(array([10.24289364+0.j, -1.90805817+0.j,  0.66516452+0.j]),
 array([[-0.49535751, -0.38143675, -0.00703753],
        [-0.80108329,  0.89327638, -0.8566417 ],
        [-0.33598587, -0.23783043,  0.51586381]]))
```

Eigenvalues and Eigenvectors of the Matrix

Scipy Linear Algebra

```
linalg.expm(marray)
```

```
array([[ 8980.52734029,  6932.22661506, 11634.12001016],
       [14524.77901037, 11210.86056025, 18811.63804243],
       [ 6089.96450292,  4701.8933951 ,  7892.97381095]])
```

Exponent of the Matrix

```
linalg.logm(marray)
```

```
array([[ 1.17435368+2.10970327j,  0.41464754-0.77609391j,
         0.71014533-1.25999787j],
       [-0.24321321-4.94065693j,  1.29350343+1.81751331j,
         2.82172934+2.95075487j],
       [ 1.03432257+1.31542554j,  0.29549779-0.48390395j,
         0.09709224-0.78562393j]])
```

Logarithm of the Matrix

Scipy Linear Algebra

$$2X + 3Y = 5$$
$$9X + 3Y = 4$$

2	3
9	3

x

Exponent of the Matrix

```
print(marray[0:2,0:2])
```

```
[[2 3]
 [9 3]]
```

```
print(marray[0:2,2])
```

```
[5 4]
```

```
linalg.solve(marray[0:2,0:2],marray[0:2,2])
```

```
array([-0.14285714,  1.76190476])
```

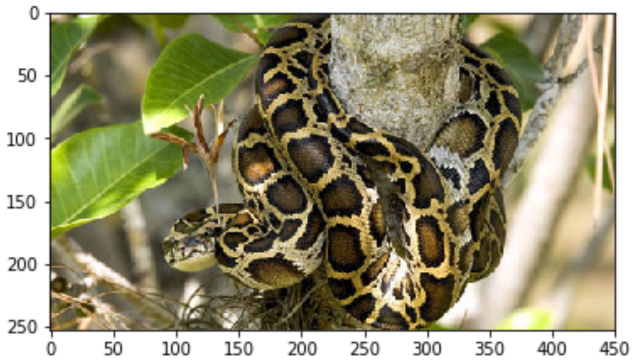
Solves the System of Linear Equations

Scipy Image Manipulation

```
import scipy.ndimage as ndimage
pic = ndimage.imread('python.jpg')
```

```
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:2: DeprecationWarning: `imread`
is deprecated!
`imread` is deprecated in SciPy 1.0.0.
Use ``matplotlib.pyplot.imread`` instead.
```

```
import matplotlib.pyplot as plt
plt.imshow(pic)
plt.show()
```



Scipy Image Manipulation

```
type(pic)
```

```
numpy.ndarray
```

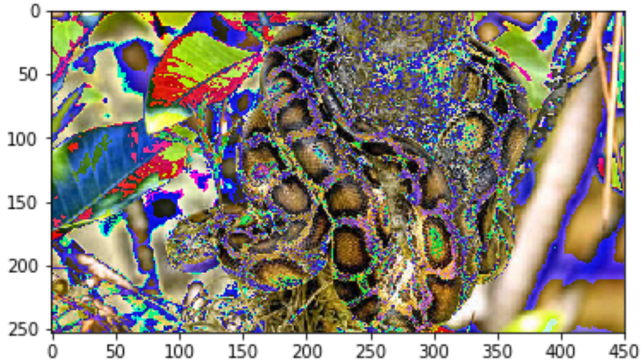
```
pic.shape
```

```
(253, 450, 3)
```

```
pic[0,0]
```

```
array([198, 178, 151], dtype=uint8)
```

```
plt.imshow(2*pic)
plt.show()
```



Scipy Image Manipulation

```
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import matplotlib.image as mimage

gs = gridspec.GridSpec(2, 2, top=1., bottom=0., right=1., left=0., hspace=0.,
        wspace=0.)

for i, g in enumerate(gs):
    ax = plt.subplot(g)
    ax.imshow(ndimage.gaussian_filter(pic, sigma=i))
    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_aspect('auto')
```

Final Considerations