Fundamentals of Information Systems

Python Programming (for Data Science)

Master's Degree in Data Science

Gabriele Tolomei

gtolomei@math.unipd.it
University of Padua, Italy
2018/2019
November, 8 2018

Lecture 6: Numerical Python (`numpy`)

What is `numpy`?

- It stands for `num`erical `py`thon, and is one of the core packages for numerical/scientific computing in Python.
- Most computational packages providing scientific functionality use `numpy` **array objects** as the building block for data exchange.
- You can find more about `numpy` on the official website.

In [1]:

```python
"""
As any other third-party module, the numpy module has to be imported before it can be used.
If you installed Python with Anaconda, numpy would be just available to you.
This is usually how numpy is imported and aliased. Although you could also
use another syntax like 'from numpy import *', I strongly encourage you to define an alias,
as this will help you to identify numpy's functions in your code.
"""
import numpy as np
```

What is inside `numpy`?

- `ndarray`: an efficient multi-dimensional array providing fast array-oriented arithmetic operations.
- Mathematical functions for fast operations on entire arrays of data without having to write loops.
- Tools for reading array data from (writing array data to) disk and working with memory-mapped files.
- Linear algebra, random number generation, and Fourier transform capabilities.
- A C API for connecting `numpy` with libraries written in C, C++, or FORTRAN.

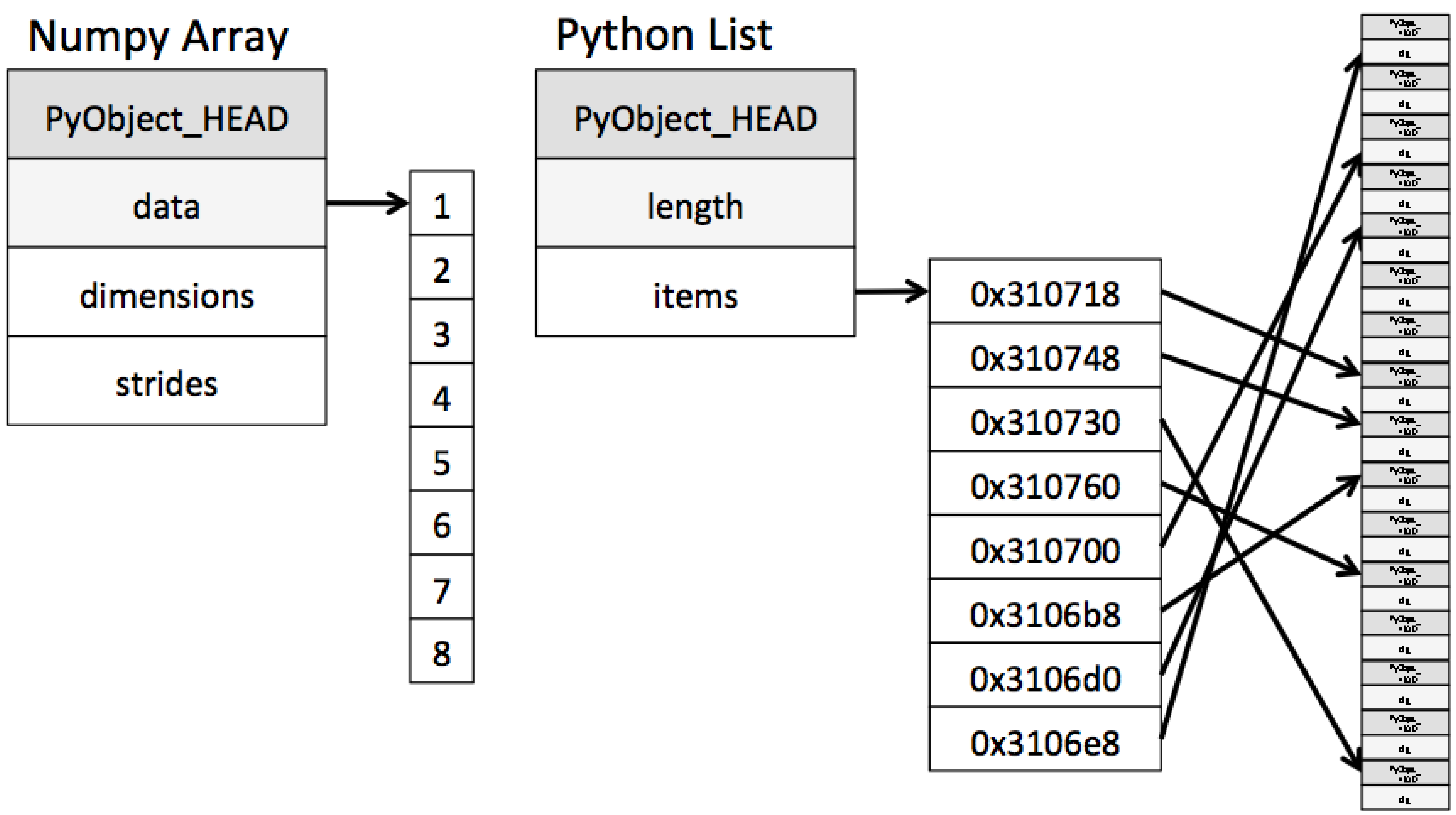
General purpose `numpy`

- Because `numpy` provides an easy-to-use C API, it is straightforward to pass data to/from external libraries written in a low-level language.
- `numpy` itself does not provide modeling nor scientific functionality, but knowing of `numpy` basics will help you use tools with array-oriented semantics, like `pandas`.
- In this class, we will in fact use `pandas`, which is tailored to tabular data and also provides some more domain-specific functionality like time series manipulation, which is not present in `numpy`.

Space Efficiency of `numpy`'s `ndarray`

- `numpy`'s importance for numerical computations in Python is due to its design for efficiency (especially when operating on large arrays of data).
- It internally stores data in a **contiguous** block of memory, independent of other built-in Python objects.

Space Efficiency of `numpy`'s `ndarray`



Time Efficiency of `numpy`'s `ndarray`

- `ndarray`s efficient memory occupation implies also computational **time efficiency**.
- Its library of algorithms mostly written in low-level C can operate on this memory without introducing any overhead due to type checking.
- `numpy` operations perform complex computations on entire arrays without the need for Python `for` loops (i.e., knowing the address of the memory block and the data type, it is just simple arithmetic).
- Spatial locality in memory access patterns results in performance gains notably due to the CPU cache (sequential locality, or locality of reference).
- Since items are stored contiguously in memory, `numpy` can take advantage of **vectorized instructions** provided by modern CPUs.

Efficiency of `numpy`: a real example

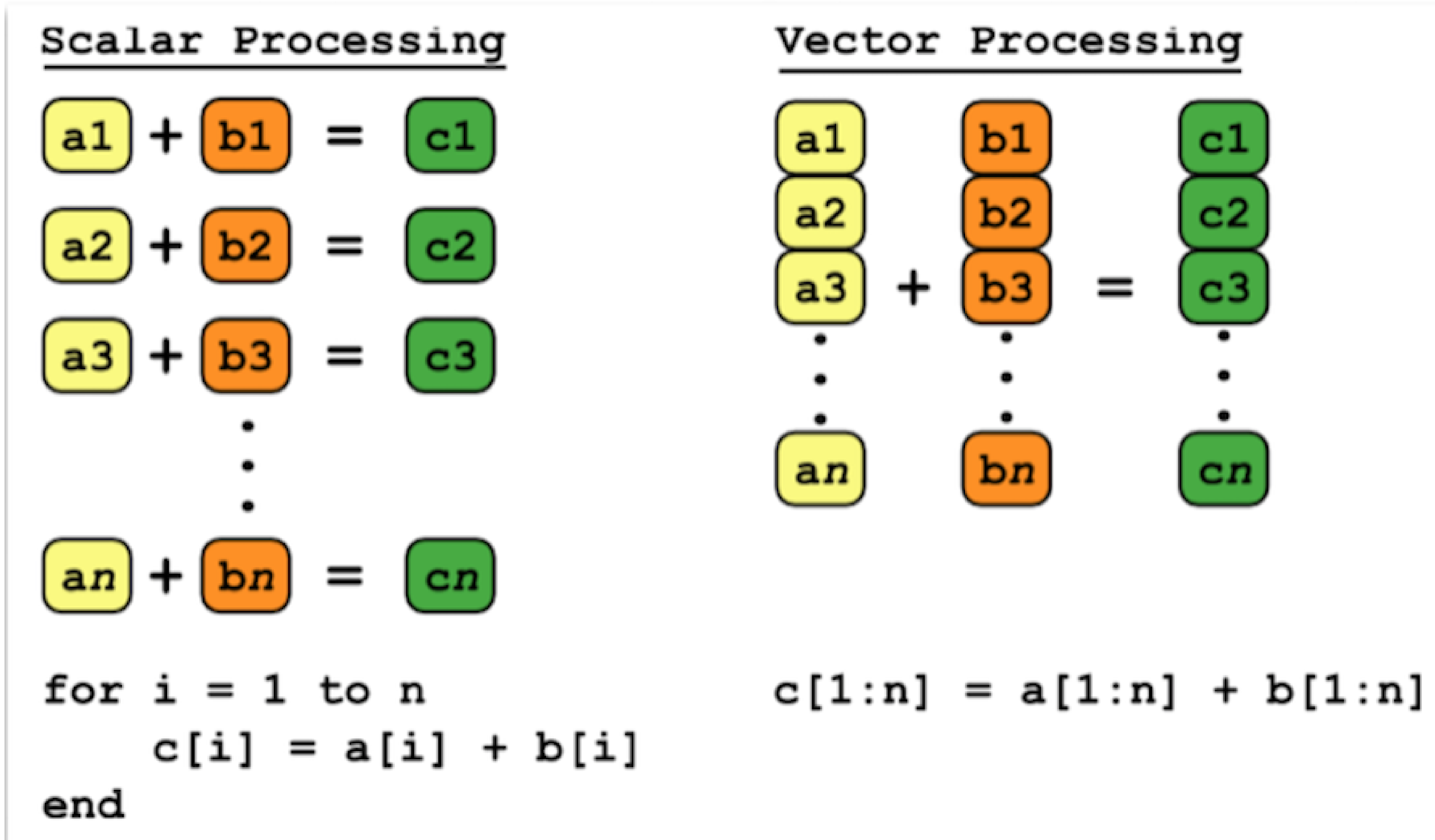
- To validate the efficiency of `numpy` in contrast with built-in Python list, just try to run the code snippet below:

```
# create a numpy array with 1M integers
my_arr = np.arange(1000000)
# create a built-in list with 1M integers
my_list = list(range(1000000))
# double each element of the numpy array
my_arr2 = my_arr * 2
# double each element of the built-in list
my_list2 = [x * 2 for x in my_list]
```

- `numpy`-based algorithms are expected to be **10 to 100** times faster than their pure Python counterparts and use significantly less memory.

Scalar vs. Vector Processing

Vector processing is also known as **S**ingle **I**nstruction **M**ultiple **D**ata (**SIMD**)



`ndarray`: A Multidimensional Array Object

Properties of `ndarray`

- A fast, flexible, generic multidimensional container for large homogeneous data sets in Python.
- Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalars.
- All the elements of an `ndarray` must be of the **same** type.
- Every array has a `shape`, a tuple indicating the size of each dimension, and a `dtype`, an object describing the data type of the array.

In [2]:

```python
# Generate some random data over a 2x3 array (i.e., a matrix)
data = np.random.randn(2, 3)
# Print out data
print("Original data matrix =\n{}".format(data))
# Multiply each element of the matrix by a constant 10
data10 = data * 10
# Print out the new data matrix
print("data matrix * 10 =\n{}".format(data10))
# Sum two data matrices (i.e., the same as multiply each element by 2)
data2 = data + data
# Print out the new data matrix
print("(data matrix + data matrix) =\n{}".format(data2))
```

```
Original data matrix =
[[ 0.60412659  1.20847991 -1.03453903]
 [ 0.76247938 -0.6180134   1.12874354]]
data matrix * 10 =
[[  6.04126594  12.0847991  -10.34539031]
 [  7.62479376  -6.18013404  11.28743543]]
(data matrix + data matrix) =
[[ 1.20825319  2.41695982 -2.06907806]
 [ 1.52495875 -1.23602681  2.25748709]]
```

In [3]:

```
# Showing the shape of the ndarray object
print("The shape of data is: {}".format(data.shape))
# Showing the type of objects contained in the ndarray object
print("The type of objects contained in data is: {}".format(data.dtype))
```

```
The shape of data is: (2, 3)
The type of objects contained in data is: float64
```

Creating `ndarray`

In [4]:

```python
# Start from a built-in Python list
data = [42, 2.5, 73, 0, 3, 1.0]
# The corresponding numpy array can be obtained by calling the np.array function
arr = np.array(data)
arr
```

Out[4]:

```
array([ 42. ,   2.5,  73. ,   0. ,   3. ,   1. ])
```

In [5]:

```python
# Nested sequences, like a list of equal-length lists,
# will be converted into a multidimensional array
multi_data = [[1, 2, 3, 4], [5, 6, 7, 8]]
# Convert the list of list into a (multidimensional) numpy array
multi_arr = np.array(multi_data)
print("Multidimensional array:\n{}".format(multi_arr))
print("Number of dimensions of the array: {}".format(multi_arr.ndim))
print("Shape of the array: {}".format(multi_arr.shape))
# Unless explicitly specified (more on this later), np.array tries to infer
# a good data type for the array that it creates.
# The data type is stored in a special dtype metadata object's field.
print("Shape of the unidimensional array: {}".format(arr.dtype))
print("Shape of the multidimensional array: {}".format(multi_arr.dtype))
```

```
Multidimensional array:
[[1 2 3 4]
 [5 6 7 8]]
Number of dimensions of the array: 2
Shape of the array: (2, 4)
Shape of the unidimensional array: float64
Shape of the multidimensional array: int64
```

In [6]:

```python
"""
In addition to np.array, there are a number of other functions for creating new arrays.
As examples, 'zeros' and 'ones' create arrays of 0's or 1's, respectively,
with a given length or shape.
'empty' creates an array without initializing its values to any particular value.
To create a higher dimensional array with these methods, pass a tuple for the shape.
"""
print("Creating a unidimensional array with 5 zeros: {}".format(np.zeros(5)))
print("Creating a multidimensional array (i.e., 3x2 matrix) with all zeros:\n{}"\
      .format(np.zeros((3,2))))
print("Creating two empty multidimensional arrays (i.e., 3x4 matrix):\n{}"\
      .format(np.empty((2, 3, 4))))
```

```
Creating a unidimensional array with 5 zeros: [ 0.  0.  0.  0.  0.]
Creating a multidimensional array (i.e., 3x2 matrix) with all zeros:
[[ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]]
Creating two empty multidimensional arrays (i.e., 3x4 matrix):
[[[  2.31584178e+077   2.31584178e+077   6.42285340e-323   0.00000000e+000]
  [  0.00000000e+000   0.00000000e+000   0.00000000e+000   0.00000000e+000]
  [  0.00000000e+000   0.00000000e+000   0.00000000e+000   0.00000000e+000]]

 [[  0.00000000e+000   0.00000000e+000   0.00000000e+000   1.14411728e-308]
  [  2.31584178e+077   2.31584178e+077   2.47032823e-323   0.00000000e+000]
  [  0.00000000e+000   0.00000000e+000   2.31584178e+077   2.00390288e+000]]]
```

In [7]:
```python
# 'arange' is an array-valued version of the built-in Python 'range' function
np.arange(16)
```

Out[7]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
```

Table of `numpy` Functions to Create `ndarray`

Function	Description
`array`	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype. 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 1's with the given shape and dtype. `ones_like` takes another array and produces a ones array of the same shape and dtype.
`zeros`, `zeros_like`	Like `ones` and `ones_like` but producing arrays of 0's 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 dtype with all values set to the indicated "fill value". `full_like` takes another array and produces a a filled array of the same shape and dtype.
`eye`, `identity`	Create a square N x N identity matrix (1's on the diagonal and 0's elsewhere)

Data Types for `ndarray`

Data Type: `dtype`

- The data type or `dtype` is a special object containing the information (or **metadata**) that the ndarray needs to interpret a chunk of memory as a particular type of data.
- In most cases, `dtype`s provide a mapping directly onto an underlying disk or memory representation, which makes it easy to read and write binary streams of data.
- Numerical dtypes are named the same way as built-in numerics, yet they also contain the number of bits per element. E.g., `float64` is the `numpy` equivalent of a standard double-precision floating point.

In [8]:

```python
# Explicitly declare the dtype of the array at definition time
# float64
arr1 = np.array([1, 2, 3], dtype=np.float64)
# int32
arr2 = np.array([1, 2, 3], dtype=np.int32)
print("arr1 data type is: {}".format(arr1.dtype))
print("arr2 data type is: {}".format(arr2.dtype))
```

```
arr1 data type is: float64
arr2 data type is: int32
```

Table of `dtype` (1 of 2)

Type	Type Code	Description
`int8, uint8`	`i1, u1`	Signed and unsigned 8-bit (1 byte) integer types
`int16, uint16`	`i2, u2`	Signed and unsigned 16-bit integer types
`int32, uint32`	`i4, u4`	Signed and unsigned 32-bit integer types
`int64, uint64`	`i8, u8`	Signed and unsigned 32-bit integer types
`float16`	`f2`	Half-precision floating point
`float32`	`f4 or f`	Standard single-precision floating point. Compatible with C float
`float64`	`f8 or d`	Standard double-precision floating point. Compatible with C double and Python `float` object
`float128`	`f16 or g`	Extended-precision floating point

Table of `dtype` (2 of 2)

`complex64`, `complex128`, `complex256`	`c8`, `c16`, `c32`	Complex numbers represented by two 32, 64, or 128 floats, respectively
`bool`	?	Boolean type storing `True` and `False` values
`object`	O	Python object type, a value can be any Python object
`string_`	S	Fixed-length ASCII string type (1 byte per character). For example, to create a string dtype with length 10, use `'S10'`.
`unicode_`	U	Fixed-length unicode type (number of bytes platform specific). Same specification semantics as `string_` (e.g. `'U10'`).

Casting to a specific `dtype` using `astype`

In [9]:

```python
"""
Sometimes it may be useful to explicitly convert or cast an array
from one dtype to another using ndarray's 'astype' method.
"""
# Let's define an array using the numpy's array method
arr = np.array([1, 2, 3, 4, 5])
print("The (inferred) dtype for the just defined numpy array is: {}".format(arr.dtype))
# Now, let's convert the inferred dtype (int64) into float64 using 'astype'
float_arr = arr.astype(np.float64)
print("The dtype for the cast numpy array is: {}".format(float_arr.dtype))
```

```
The (inferred) dtype for the just defined numpy array is: int64
The dtype for the cast numpy array is: float64
```

In [10]:

```python
"""
Sometimes it may be useful to explicitly convert or cast an array
from one dtype to another using ndarray's 'astype' method.
"""
# In the example above, integers are cast to floating point.
# What if we cast some floating point numbers to be of integer dtype?
# Let's create the numpy array from a list of float numbers
arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
print("The original array is: {}".format(arr))
print("The array cast to integer is: {}".format(arr.astype(np.int32)))
```

```
The original array is: [  3.7  -1.2  -2.6   0.5  12.9  10.1]
The array cast to integer is: [ 3 -1 -2  0 12 10]
```

In [11]:

```python
"""
If you have an array of strings representing numbers,
you can use 'astype' to convert them to numeric form.
"""
numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)
# Note that we use 'float' instead of 'np.float64',
# as numpy aliases the Python types to its own equivalent dtypes.
print("The original array cast to string is: {}".format(numeric_strings.astype(float)))
# If casting fails for some reason (like a string that cannot be converted to float64),
# a ValueError will be raised.
wrong_numeric_strings = np.array(['1.25', '-9.6', 'h7-25', '42'], dtype=np.string_)
print("The original array cast to string is: {}".format(wrong_numeric_strings.astype(float)))
```

```
The original array cast to string is: [  1.25  -9.6   42.  ]

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-11-e39493940eea> in <module>()
     10 # a ValueError will be raised.
     11 wrong_numeric_strings = np.array(['1.25', '-9.6', 'h7-25', '42'], dtype=np.string_
)
---> 12 print("The original array cast to string is: {}".format(wrong_numeric_strings.asty
pe(float)))

ValueError: could not convert string to float: 'h7-25'
```

About `astype`

Calling `astype` **always** returns a copy of the original numpy array (even if we apply a "dummy" casting, i.e., if the new `dtype` we want to cast the array to is the same of the original, old `dtype`).

Operations between arrays and scalars

- `numpy` arrays enables you to express many kinds of "batched" data processing tasks as concise array expressions, instead of writing `for` loops.
- This practice is commonly referred to as **vectorization**.
- In general, vectorized array operations is one or two (or more) orders of magnitude faster than their pure Python equivalents.
- Any arithmetic operations between equal-size arrays applies the operation elementwise.
- Operations between differently sized arrays is called **broadcasting** but won't be further discussed here.

In [12]:

```python
# Let's define a simple 2x3 numpy array
arr = np.array([[1., 2., 3.], [4., 5., 6.]])
print("Consider the following {}x{} array:\n{}"\
      .format(arr.shape[0], arr.shape[1], arr))
# Square the values contained in the original array
arr_squared = arr * arr
print("Square the elements of the original array:\n{}"\
      .format(arr_squared))

# Arithmetic operations with scalars are as you would expect,
# propagating the value to each element
reciprocal_arr = 1/arr
print("Compute the reciprocal of the elements of the original array:\n{}"\
      .format(reciprocal_arr))
sqrt_arr = arr ** 0.5
print("Compute the square root of the elements of the original array:\n{}"\
      .format(sqrt_arr))
```

```
Consider the following 2x3 array:
[[ 1.  2.  3.]
 [ 4.  5.  6.]]
Square the elements of the original array:
[[  1.   4.   9.]
 [ 16.  25.  36.]]
Compute the reciprocal of the elements of the original array:
[[ 1.          0.5         0.33333333]
 [ 0.25        0.2         0.16666667]]
Compute the square root of the elements of the original array:
[[ 1.          1.41421356  1.73205081]
 [ 2.          2.23606798  2.44948974]]
```

Basic Indexing and Slicing

In [13]:

```python
"""
numpy array indexing is a rich topic, as there are many ways
you may want to select a subset of your data or individual elements.
One-dimensional arrays are simple; on the surface they act similarly to Python lists.
"""
# Create an ndarray of 10 random integers in the range [0, 50)
arr = np.random.randint(low=0, high=50, size=10)
print("Original numpy array is: {}".format(arr))
print("Accessing the 6-th element of the array: {}".format(arr[5]))
print("Extracting from the 6-th to the 8-th element of the array: {}".format(arr[5:8]))
# Assigning a new value to a slice is "broadcasted" to all the elements of the slice.
arr[5:8] = 12
print("Now the numpy array is: {}".format(arr))
```

```
Original numpy array is: [33 16 49 21 11 17 36 45 35 43]
Accessing the 6-th element of the array: 17
Extracting from the 6-th to the 8-th element of the array: [17 36 45]
Now the numpy array is: [33 16 49 21 11 12 12 12 35 43]
```

In [14]:

```python
"""
Differently from Python's built-in lists, numpy array slices are views on the original array
This means that the data is not (shallow-)copied,
and any modifications to the view will be reflected in the source array.
"""
# Define a Python standard list containing the first 10 non-negative integers [0, 1, ..., 9]
py_list = [x for x in range(10)]
# Define a numpy array with the same elements
arr = np.arange(10)
# Slicing the Python list
sliced_list = py_list[5:8]
print("Sliced Python list = {}".format(sliced_list))
# Slicing the numpy array
sliced_arr = arr[5:8]
print("Sliced numpy array = {}".format(sliced_arr))
# Changing references of the Python sliced list won't change the original list
sliced_list = [12 for i in sliced_list]
print("Sliced Python list = {}".format(sliced_list))
print("Original Python list = {}".format(py_list))
# Changing references of the sliced numpy array will reflect to the original array
sliced_arr[:] = 12
print("Sliced numpy array = {}".format(sliced_arr))
print("Original numpy arr = {}".format(arr))
```

```
Sliced Python list = [5, 6, 7]
Sliced numpy array = [5 6 7]
Sliced Python list = [12, 12, 12]
Original Python list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Sliced numpy array = [12 12 12]
Original numpy arr = [ 0  1  2  3  4 12 12 12  8  9]
```

In [15]:

```python
"""
If you want a copy of a slice of a numpy array instead of a view,
you will need to explicitly copy the array; for example arr[5:8].copy()
"""
arr = np.arange(10)
sliced_arr = arr[5:8].copy()
# Changing references of the sliced numpy array will NOT reflect to the original array
sliced_arr[:] = 12
print("Sliced numpy array = {}".format(sliced_arr))
print("Original numpy arr = {}".format(arr))
```

```
Sliced numpy array = [12 12 12]
Original numpy arr = [0 1 2 3 4 5 6 7 8 9]
```

In [16]:

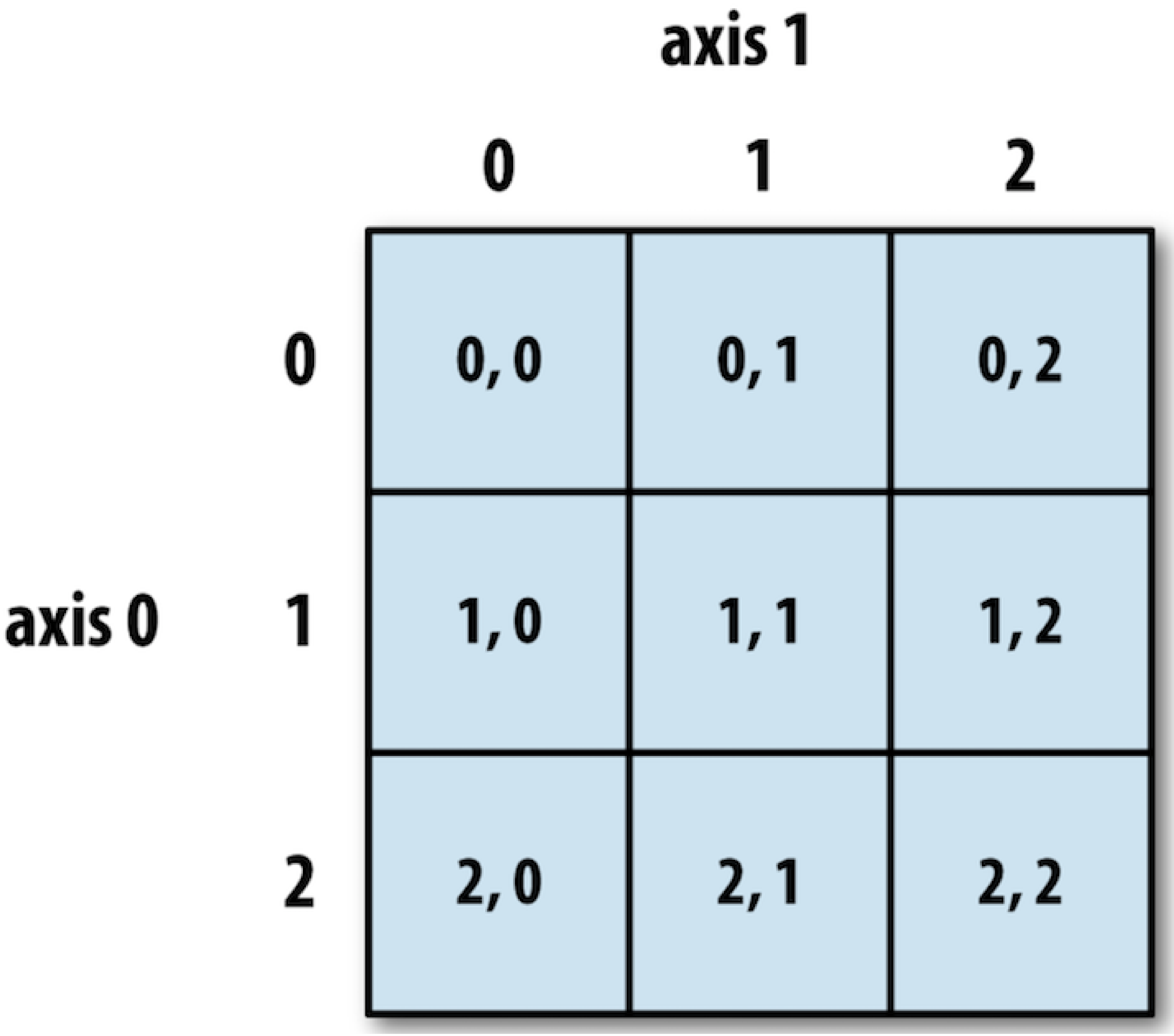
```python
"""
With higher dimensional arrays, you have many more options.
In a two-dimensional array, the elements at each index are no longer scalars
but rather one-dimensional arrays.
"""
# Consider the following 3x3 matrix defined as a two-dimensional array
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("Original numpy array is:\n{}".format(arr2d))
# Accessing the 2-nd element of the matrix above
print("The third element of the original array is: {}".format(arr2d[2]))
# Thus, individual elements can be accessed recursively.
# This is a bit too cumbersome, so you can pass a comma-separated list of indices.
print("The third element of the first array is: {}".format(arr2d[0][2]))
arr2d[0][2] == arr2d[0, 2]
```

```
Original numpy array is:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
The third element of the original array is: [7 8 9]
The third element of the first array is: 3
```

Out[16]:

```
True
```

Indexing on a 2-d Array



In [17]:

```python
"""
Higher dimensional numpy arrays give you more options,
as you can slice one or more axes and also mix integers.
Consider the 2D array above, arr2d. Slicing this array is a bit different
"""
print("Sliced array (matrix):\n{}".format(arr2d[:2]))
"""
As you can see, it has sliced along axis 0, the first axis.
A slice, therefore, selects a range of elements along an axis.
You can pass multiple slices just like you can pass multiple indexes
"""
# Extract the first two elements along axis 0 (i.e., the first two rows)
# and every element except the first along axis 1 (i.e., the second and third columns)
print("Sliced array (matrix):\n{}".format(arr2d[:2, 1:]))
# When slicing like this, you always obtain array views of the same number of dimensions.
# By mixing integer indexes and slices, you get lower dimensional slices
# Access the whole second element along axis 0 and the first two along axis 1.
print("Sliced array (matrix):\n{}".format(arr2d[1, :2]))
```

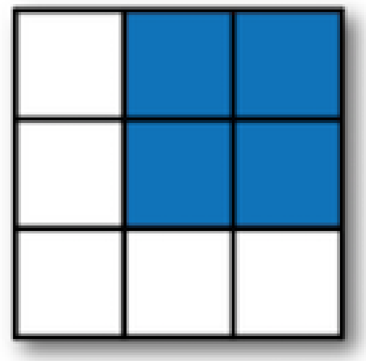
```
Sliced array (matrix):
[[1 2 3]
 [4 5 6]]
Sliced array (matrix):
[[2 3]
 [5 6]]
Sliced array (matrix):
[4 5]
```

In [18]:

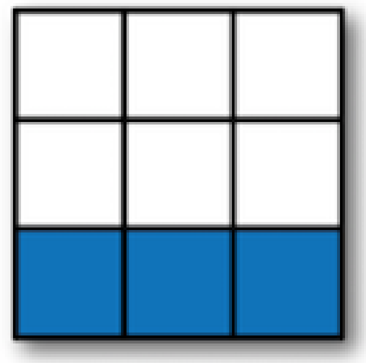
```python
"""
Note that a colon by itself means to take the entire axis,
so you can slice only higher dimensional axes by doing as follows
"""
# Extract the first column
print("Sliced array (matrix):\n{}".format(arr2d[:, :1]))
# Of course, assigning to a slice expression assigns to the whole selection
arr2d[:2, 1:] = 0
print("New array (matrix):\n{}".format(arr2d))
```

```
Sliced array (matrix):
[[1]
 [4]
 [7]]
New array (matrix):
[[1 0 0]
 [4 0 0]
 [7 8 9]]
```

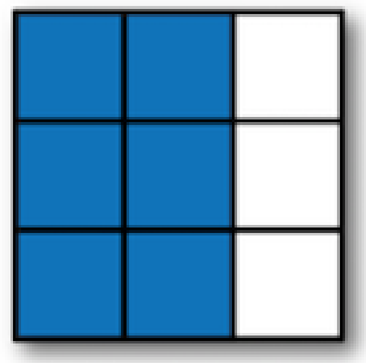
Slicing on a 2-d Array



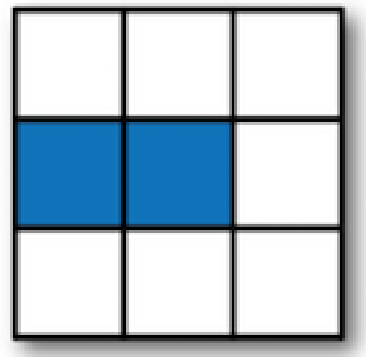
Expression	Shape
arr[:2, 1:]	(2, 2)



Expression	Shape
arr[2]	(3,)
arr[2, :]	(3,)
arr[2:, :]	(1, 3)



Expression	Shape
arr[:, :2]	(3, 2)



Expression	Shape
arr[1, :2]	(2,)
arr[1:2, :2]	(1, 2)

Boolean Indexing

In [19]:

```python
"""
Let's consider an array containing some data and an array of names with duplicates.
We generate some random normally distributed data with the 'randn' function in numpy.random
"""
# Random, normally distributed 7x4 data matrix
data = np.random.randn(7, 4)
print("The original input data is:\n{}".format(data))
# numpy array containing "names"
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
"""
Suppose each name corresponds to a row in the data array,
and we want to select all the rows with corresponding name 'Bob'.
Like arithmetic operations, comparisons (such as ==) with arrays are also vectorized.
Thus, comparing names with the string 'Bob' yields a boolean array
"""
names == 'Bob'
```

```
The original input data is:
[[-0.7719735  -0.54500994  1.08807583  1.49070231]
 [-0.98309067 -0.70861066 -1.93274927 -1.79011088]
 [ 1.63070203 -0.53315322  0.58725735  0.17322339]
 [-0.9051374  -0.6947978  -1.96765755  0.53994032]
 [-0.24636087  0.33501854  0.71853993  0.89458756]
 [-1.23836331 -1.69812044  0.50381849 -1.61415737]
 [ 0.7691585   0.8033165   2.0940385  -0.49227147]]
```

Out[19]:

```
array([ True, False, False,  True, False, False, False], dtype=bool)
```

In [20]:

```
"""
The boolean array above can be passed when indexing the array
"""
data[names == 'Bob']
```

Out[20]:

```
array([[-0.7719735 , -0.54500994,  1.08807583,  1.49070231],
       [-0.9051374 , -0.6947978 , -1.96765755,  0.53994032]])
```

In [21]:

```
"""
The boolean array must be of the same length as the axis it is indexing.
You can even mix and match boolean arrays with slices or integers (or sequences of integers)
"""
# Extract all the rows indexed by the boolean array yet limited to 3rd and 4th columns
print("Boolean indexing, 3rd and 4th columns only:\n{}".format(data[names == 'Bob', 2:]))
# Extract all the rows indexed by the boolean array yet limited to 2nd column
print("Boolean indexing, 2nd column only:\n{}".format(data[names == 'Bob', 1]))
```

```
Boolean indexing, 3rd and 4th columns only:
[[ 1.08807583  1.49070231]
 [-1.96765755  0.53994032]]
Boolean indexing, 2nd column only:
[-0.54500994 -0.6947978 ]
```

In [22]:
```python
"""
To select everything but 'Bob', you can either use '!=' or negate the condition using '~'
"""
data[~(names == 'Bob')]
```

Out[22]:
```
array([[-0.98309067, -0.70861066, -1.93274927, -1.79011088],
       [ 1.63070203, -0.53315322,  0.58725735,  0.17322339],
       [-0.24636087,  0.33501854,  0.71853993,  0.89458756],
       [-1.23836331, -1.69812044,  0.50381849, -1.61415737],
       [ 0.7691585 ,  0.8033165 ,  2.0940385 , -0.49227147]])
```

In [23]:

```
"""
To select more than one names to combine multiple boolean conditions,
use boolean arithmetic operators like '&' (and) and '|' (or)
NOTE: The Python keywords 'and' and 'or' DO NOT work with boolean arrays!!!
Selecting data from an array by boolean indexing always creates a copy of the data,
even if the returned array is unchanged.
"""
mask = (names == 'Bob') | (names == 'Will')
print("Masked data:\n{}".format(data[mask]))
```

```
Masked data:
[[-0.7719735  -0.54500994  1.08807583  1.49070231]
 [ 1.63070203 -0.53315322  0.58725735  0.17322339]
 [-0.9051374  -0.6947978  -1.96765755  0.53994032]
 [-0.24636087  0.33501854  0.71853993  0.89458756]]
```

In [24]:
```python
"""
Setting values with boolean arrays works in a common-sense way.
To set all of the negative values in 'data' to '0' we need only to do the following.
"""
data[data < 0] = 0
data
```

Out[24]:
```
array([[ 0.        ,  0.        ,  1.08807583,  1.49070231],
       [ 0.        ,  0.        ,  0.        ,  0.        ],
       [ 1.63070203,  0.        ,  0.58725735,  0.17322339],
       [ 0.        ,  0.        ,  0.        ,  0.53994032],
       [ 0.        ,  0.33501854,  0.71853993,  0.89458756],
       [ 0.        ,  0.        ,  0.50381849,  0.        ],
       [ 0.7691585 ,  0.8033165 ,  2.0940385 ,  0.        ]])
```

In [25]:

```python
"""
Setting whole rows or columns using a 1D boolean array is also easy.
"""
data[names != 'Joe'] = 5
data
```

Out[25]:

```
array([[ 5.        ,  5.        ,  5.        ,  5.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ],
       [ 5.        ,  5.        ,  5.        ,  5.        ],
       [ 5.        ,  5.        ,  5.        ,  5.        ],
       [ 5.        ,  5.        ,  5.        ,  5.        ],
       [ 0.        ,  0.        ,  0.50381849,  0.        ],
       [ 0.7691585 ,  0.8033165 ,  2.0940385 ,  0.        ]])
```

Transposing Arrays and Swapping Axes

In [26]:

```python
"""
Transposing is a special form of reshaping which returns a view on the underlying data
without copying anything.
Arrays have the transpose method and also the special 'T' attribute.
"""
# Let's define a 1-d numpy array
arr = np.arange(15)
print("The original numpy array is: {}".format(arr))
reshaped_arr = arr.reshape((3, 5))
print("The reshaped numpy array is:\n{}".format(reshaped_arr))
transposed_arr = reshaped_arr.T
print("The transposed numpy array is:\n{}".format(transposed_arr))
```

```
The original numpy array is: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
The reshaped numpy array is:
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
The transposed numpy array is:
[[ 0  5 10]
 [ 1  6 11]
 [ 2  7 12]
 [ 3  8 13]
 [ 4  9 14]]
```

In [27]:

```python
"""
When doing matrix computations, you will do this very often,
like for example computing the dot matrix product XTX using np.dot
"""
# Create a 4x3 matrix
matrix = np.random.randn(4, 3)
print("The original matrix is:\n{}".format(matrix))
dot_product = np.dot(matrix.T, matrix)
print("The result of the dot product is:\n{}".format(dot_product))
```

```
The original matrix is:
[[ 0.53942833 -0.60514056  0.17707711]
 [ 1.35496875 -0.61242005  1.15398291]
 [ 1.78242713  0.02099927  0.3451653 ]
 [-0.5397399   0.55667071  2.01884349]]
The result of the dot product is:
[[ 5.59528889 -1.41926772  1.18471281]
 [-1.41926772  1.05157666  0.31720044]
 [ 1.18471281  0.31720044  5.55790097]]
```

Universal Functions: Fast Element-wise Array Functions

- A universal function, or `ufunc`, is a function that performs elementwise operations on data in `ndarray`s.
- You can think of them as fast vectorized wrappers for simple functions that take one or more scalar values and produce one or more scalar results.
- In case of binary universal functions, the shape of the input arrays **must be the same**.

In [28]:

```python
"""
Many ufuncs are simple elementwise unary transformations, like 'sqrt' or 'exp'.
"""
arr = np.arange(10)
print("The original array is: {}".format(arr))
sqrt_arr = np.sqrt(arr)
print("The squared-root array is: {}".format(sqrt_arr))
exp_arr = np.exp(arr)
print("The exp array is: {}".format(exp_arr))
"""
Other functions, such as 'add' or 'maximum', take 2 arrays (thus, binary ufuncs)
and return a single array as the result
"""
# Define two random arrays
x = np.random.randn(5)
y = np.random.randn(5)
print("x = {}".format(x))
print("y = {}".format(y))
print("Element-wise maximum between x's and y's elements: {}".format(np.maximum(x, y)))
```

```
The original array is: [0 1 2 3 4 5 6 7 8 9]
The squared-root array is: [ 0.          1.          1.41421356  1.73205081  2.
2.23606798
  2.44948974  2.64575131  2.82842712  3.        ]
The exp array is: [  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]
x = [-1.60788496  1.3027705   1.29542805 -0.5510918   1.55611235]
y = [-0.32853292 -0.36326418  0.19170485  0.15990394  0.76457901]
Element-wise maximum between x's and y's elements: [-0.32853292  1.3027705   1.29542805
0.15990394  1.55611235]
```

Universal Unary Functions (1 of 2)

Function	Description
`abs, fabs`	Compute the absolute value element-wise for integer, floating point, or complex values. Use `fabs` as a faster alternative for non-complex-valued data
`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 each element
`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`

Universal Unary Functions (2 of 2)

modf	Return fractional and integral parts of array as separate array
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 Binary 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, logical_or, logical_xor`	Compute element-wise truth value of logical operation. Equivalent to infix operators `& |, ^`

Mathematical and Statistical Methods

- A set of mathematical functions which compute statistics about an entire array or about the data along an axis are accessible as array methods.
- Aggregations (often called reductions) like `sum`, `mean`, and `std` (standard deviation) can be invoked:
    - by calling the array instance method;
    - using the top level `numpy` function.

In [29]:

```python
# Consider the following normally-distributed random 5x4 matrix data
matrix = np.random.randn(5, 4)
print("The original matrix is:\n{}".format(matrix))
print("The mean of the matrix is: {}".format(matrix.mean()))
print("The mean of the matrix is: {}".format(np.mean(matrix)))
"""
Functions like 'mean' and 'sum' take an optional axis argument,
which computes the statistic over the given axis,
resulting in an array with one fewer dimension
"""
print("The mean of the matrix along the columns is: {}".format(matrix.mean(axis=1)))
print("The sum of the matrix along the rows is: {}".format(matrix.sum(axis=0)))
```

```
The original matrix is:
[[ 0.80315271  0.43595353  0.25475635 -1.41916843]
 [ 0.08995756  0.24721462  2.36291207  0.36666885]
 [ 0.24413369 -0.2324963   0.709209    0.05656641]
 [-1.49674819  0.25793226 -1.04658819  0.23984137]
 [-1.52227821  0.13571985  0.81259401  0.86914029]]
The mean of the matrix is: 0.1084236629677177
The mean of the matrix is: 0.1084236629677177
The mean of the matrix along the columns is: [ 0.01867354  0.76668828  0.1943532  -0.51139
069  0.07379399]
The sum of the matrix along the rows is: [-1.88178244  0.84432396  3.09288324  0.1130485 ]
```

Table of `numpy` Statistical Methods

Method	Description
`sum`	Sum of all the elements in the array or along an axis. Zero-length arrays have sum 0.
`mean`	Arithmetic mean. Zero-length arrays have `NaN` mean.
`std, var`	Standard deviation and variance, respectively, with optional degrees of freedom adjustment (default denominator `n`).
`min, max`	Minimum and maximum.
`argmin, argmax`	Indices of minimum and maximum elements, respectively.
`cumsum`	Cumulative sum of elements starting from 0
`cumprod`	Cumulative product of elements starting from 1

Table of `numpy` Set Methods

Method	Description
`unique(x)`	Compute the sorted, unique elements in `x`
`intersect1d(x, y)`	Compute the sorted, common elements in `x` and `y`
`union1d(x, y)`	Compute the sorted union of elements
`in1d(x, y)`	Compute a boolean array indicating whether each element of `x` is contained in `y`
`setdiff1d(x, y)`	Set difference, elements in `x` that are not in `y`
`setxor1d(x, y)`	Set symmetric differences; elements that are in either of the arrays, but not both

I/O with `numpy` Arrays

- `numpy` is able to **save** and **load** data to and from disk either in ***text*** or ***binary*** format.
- We only discuss built-in binary format, since we will use `pandas` for loading text or tabular data.
- `np.save` and `np.load` are the two workhorse functions for efficiently saving and loading array data on disk.
- Arrays are saved by default in an ***uncompressed raw binary*** format with file extension `.npy`

In [30]:

```python
# Consider the following numpy array
arr = np.random.randn(5)
print("The original array is: {}".format(arr))
# Persist the above array out to disk to the specified path on disk
np.save("./data/np_array", arr) # if no '.npy' extension is specified it will be appended
# Load the array back from the specified path on disk
arr_loaded = np.load("./data/np_array.npy")
print("The array loaded from disk is: {}".format(arr_loaded))

# NOTE: If you need to save multiple arrays in a zip archive
# use 'np.savez' and pass the arrays as keyword arguments:
#    np.savez("path/to/arr_archive.npz", a=arr_a, b=arr_b)
# When this is loaded back with:
#    arr_archive = np.load("./data/np_array.npy")
# You get back a dict-like object which loads the individual arrays lazily:
#    arr_archive['b'] # refers to the second array in the archive
```

```
The original array is: [-0.00948631 -0.18875794 -0.49668335 -0.08650266  1.06290762]
The array loaded from disk is: [-0.00948631 -0.18875794 -0.49668335 -0.08650266  1.0629076
2]
```

Table of `numpy` Linear Algebra 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 pseudo-inverse inverse 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

Table of `numpy.random` Functions

Function	Description
`seed`	Seed the random number generator
`permutation`	Return a random permutation of a sequence, or return a permuted range
`shuffle`	Randomly permute a sequence in place
`rand`	Draw samples from a uniform distribution
`randint`	Draw random integers from a given low-to-high range
`randn`	Draw samples from a normal distribution with mean 0 and standard deviation 1 (MATLAB-like interface)
`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