FUNDAMENTALS

NumPy BASICS: ARRAYS & VECTORIZEDCOMPUTATION

NUMPY BASICS

ARRAYS AND VECTORIZED COMPUTATION

NumPy Basics

> NumPy, short for Numerical Python, is one of the most important foundational packages for numerical computing in Python.

> Most computational packages providing scientific functionality use NumPy's array objects as the *lingua franca* for data exchange.

NumPy Basics

> Here are some of the things you'll find in NumPy:

- **`ndarray`**, an efficient multidimensional array providing fast array-oriented arithmetic operations and flexible *broadcasting* capabilities.
- Mathematical functions for fast operations on entire arrays of data without having to write loops.
- Tools for reading/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

NumPy Basics

> Because NumPy provides an easy-to-use C API, it is straightforward to pass data to external libraries written in a low-level language and also for external libraries to return data to Python as NumPy arrays.

> This feature has made Python a language of choice for wrapping legacy C/C++/Fortran codebases and giving them a dynamic and easy-to-use interface.

> While NumPy by itself does not provide modeling or scientific functionality, having an understanding of NumPy arrays and array-oriented computing will help you use tools with array-oriented semantics, like pandas, much more effectively.

NumPy Focus Areas

> Fast vectorized array operations for data munging and cleaning, subsetting and filtering, transformation, and any other kinds of computations

> Array algorithms like sorting, unique, and set operations

> Efficient descriptive statistics and aggregating/summarizing data

> Data alignment and relational data manipulations for merging and joining together heterogeneous datasets

> Expressing conditional logic as array expressions instead of loops with if-elif-else branches

> Group-wise data manipulations (aggregation, transformation, function application)

Advantages of NumPy

> One of the reasons NumPy is so important for numerical computations in Python is because it is designed for efficiency on large arrays of data:
  - NumPy internally stores data in a contiguous block of memory, independent of other built-in Python objects.
  - NumPy's library of algorithms written in the C language can operate on this memory without any type checking or other overhead.
  - NumPy arrays also use much less memory than built-in Python sequences
  - NumPy operations perform complex computations on entire arrays without the need for Python for loops.

Advantages of NumPy

> To give you an idea of the performance difference, consider a NumPy array of one million integers, and the equivalent Python list:

```
In [7]: import numpy as np

In [8]: my_arr = np.arange(1000000)

In [9]: my_list = list(range(1000000))
```

> Now let's multiply each sequence by 2:

```
In [10]: %time for _ in range(10): my_arr2 = my_arr * 2
CPU times: user 20 ms, sys: 50 ms, total: 70 ms
Wall time: 72.4 ms

In [11]: %time for _ in range(10): my_list2 = [x * 2 for x in my_list]
CPU times: user 760 ms, sys: 290 ms, total: 1.05 s
Wall time: 1.05 s
```

Advantages of NumPy

- NumPy-based algorithms are generally 10 to 100 times faster (or more) than their pure Python counterparts
- NumPy-based algorithms use significantly less memory

`ndarray`: A Multidimensional Array Object

- `ndarray`
  - One of the key features of NumPy
  - N-dimensional array object
  - Fast, flexible container for large datasets
- Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements.

ndarray: A Multidimensional Array Object

> Generate a small array of random data

```
In [12]: import numpy as np

# Generate some random data
In [13]: data = np.random.randn(2, 3)

In [14]: data
Out[14]:
array([[-0.2047,  0.4789, -0.5194],
       [-0.5557,  1.9658,  1.3934]])

In [15]: data * 10
Out[15]:
array([[ -2.0471,   4.7894,  -5.1944],
       [ -5.5573,  19.6578,  13.9341]])

In [16]: data + data
Out[16]:
array([[-0.4094,  0.9579, -1.0389],
       [-1.1115,  3.9316,  2.7868]])
```


ndarray: A Multidimensional Array Object

> An **ndarray** is a generic multidimensional container for homogeneous data

> All of the elements must be the same type.

> Every array has
  – a **shape**: a tuple indicating the size of each dimension
  – a **dtype**: an object describing the *data type* of the array

```
In [17]: data.shape
Out[17]: (2, 3)

In [18]: data.dtype
Out[18]: dtype('float64')
```

Creating ndarrays

> The easiest way to create an array is to use the **`array`** function

– accepts any sequence-like object

– produces a new NumPy array containing the passed data

> Example: converting a list to **`ndarray`**

```
In [19]: data1 = [6, 7.5, 8, 0, 1]

In [20]: arr1 = np.array(data1)

In [21]: arr1
Out[21]: array([ 6. ,  7.5,  8. ,  0. ,  1. ])
```


Creating **`ndarrays`**

> Nested sequences, like a list of equal-length lists, will be converted into a multidimensional array:

```
In [22]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

In [23]: arr2 = np.array(data2)

In [24]: arr2
Out[24]:
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])

In [25]: arr2.ndim
Out[25]: 2

In [26]: arr2.shape
Out[26]: (2, 4)
```

Creating `ndarrays`

- `zeros` and `ones` create arrays of 0s or 1s, 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

```
In [30]: np.zeros((3, 6))
Out[30]:
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.]])
```


Creating `ndarrays`

```
In [29]: np.zeros(10)
Out[29]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

In [30]: np.zeros((3, 6))
Out[30]:
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.]])

In [31]: np.empty((2, 3, 2))
Out[31]:
array([[[ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.]],
       [[ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.]]])
```

Creating **ndarrays**

> **arange** is an array-valued version of the built-in Python range function:

```
In [32]: np.arange(15)
Out[32]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```


Array Creation Functions

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`	**`full`** produces 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 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 `ndarrays`

> The *data type* or `dtype` is a special object containing the information (*metadata*) the `ndarray` needs to interpret a chunk of memory as a particular type of data:

```
In [33]: arr1 = np.array([1, 2, 3], dtype=np.float64)

In [34]: arr2 = np.array([1, 2, 3], dtype=np.int32)

In [35]: arr1.dtype
Out[35]: dtype('float64')

In [36]: arr2.dtype
Out[36]: dtype('int32')
```


NumPy Data Types

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
complex64, complex128, complex256	c8, c16, c32	Complex numbers represented by two 32, 64, or 128 floats, respectively

NumPy Data Types

Type	Type Code	Description
bool	?	Boolean type storing True and False values
object	O	Python object type
string_	S	Fixed-length 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').

Data Types for **`ndarrays`**

> You can explicitly convert or cast an array from one **`dtype`** to another using ndarray's **`astype`** method:

```
In [37]: arr = np.array([1, 2, 3, 4, 5])

In [38]: arr.dtype
Out[38]: dtype('int64')

In [39]: float_arr = arr.astype(np.float64)

In [40]: float_arr.dtype
Out[40]: dtype('float64')
```

Data Types for **ndarrays**

> If you cast some floating-point numbers to be of integer **dtype**, the decimal part will be truncated:

```
In [41]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

In [42]: arr
Out[42]: array([  3.7,  -1.2,  -2.6,   0.5,  12.9,  10.1])

In [43]: arr.astype(np.int32)
Out[43]: array([ 3, -1, -2,  0, 12, 10], dtype=int32)
```


Data Types for **ndarrays**

> If you have an array of strings representing numbers, you can use **astype** to convert them to numeric form:

```
In [44]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)

In [45]: numeric_strings.astype(float)
Out[45]: array([  1.25,  -9.6 ,  42.  ])
```

Data Types for **`ndarrays`**

> You can also use another array's **`dtype`** attribute in conversion

```
In [46]: int_array = np.arange(10)

In [47]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)

In [48]: int_array.astype(calibers.dtype)
Out[48]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
```


Data Types for **`ndarrays`**

> There are shorthand type code strings you can also use to refer to a **`dtype`**:

```
In [49]: empty_uint32 = np.empty(8, dtype='u4')

In [50]: empty_uint32
Out[50]:
array([         0, 1075314688,          0, 1075707904,          0,
       1075838976,          0, 1072693248], dtype=uint32)
```

Arithmetic with NumPy Arrays

> Arrays are important because they enable you to express batch operations on data without writing any for loops.
  – NumPy users call this ***vectorization***.

Arithmetic with NumPy Arrays

> Any arithmetic operations between equal-size arrays applies the operation element-wise:

```
In [51]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])

In [52]: arr
Out[52]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])

In [53]: arr * arr
Out[53]:
array([[  1.,   4.,   9.],
       [ 16.,  25.,  36.]])

In [54]: arr - arr
Out[54]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

Arithmetic with NumPy Arrays

> Arithmetic operations with scalars propagate the scalar argument to each element in the array:

```
In [55]: 1 / arr
Out[55]:
array([[ 1.    ,  0.5   ,  0.3333],
       [ 0.25  ,  0.2   ,  0.1667]])

In [56]: arr ** 0.5
Out[56]:
array([[ 1.    ,  1.4142,  1.7321],
       [ 2.    ,  2.2361,  2.4495]])
```


Arithmetic with NumPy Arrays

> Comparisons between arrays of the same size yield boolean arrays:

```
In [57]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])

In [58]: arr2
Out[58]:
array([[  0.,   4.,   1.],
       [  7.,   2.,  12.]])

In [59]: arr2 > arr
Out[59]:
array([[False,  True, False],
       [ True, False,  True]], dtype=bool)
```

Basic Indexing and Slicing

> NumPy array indexing is a rich

> There are many ways you may want to select a subset of your data or individual elements.

Basic Indexing and Slicing

> One-dimensional arrays are simple and act similarly to Python lists:

```
In [60]: arr = np.arange(10)

In [61]: arr
Out[61]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [62]: arr[5]
Out[62]: 5

In [63]: arr[5:8]
Out[63]: array([5, 6, 7])

In [64]: arr[5:8] = 12

In [65]: arr
Out[65]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

Basic Indexing and Slicing

```
In [66]: arr_slice = arr[5:8]

In [67]: arr_slice
Out[67]: array([12, 12, 12])

In [68]: arr_slice[1] = 12345

In [69]: arr
Out[69]: array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,     9])
```

```
arr_slice[1] = 12345
```

```
In [70]: arr_slice[:] = 64

In [71]: arr
Out[71]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

The “bare” slice [:] will assign to all values in an array

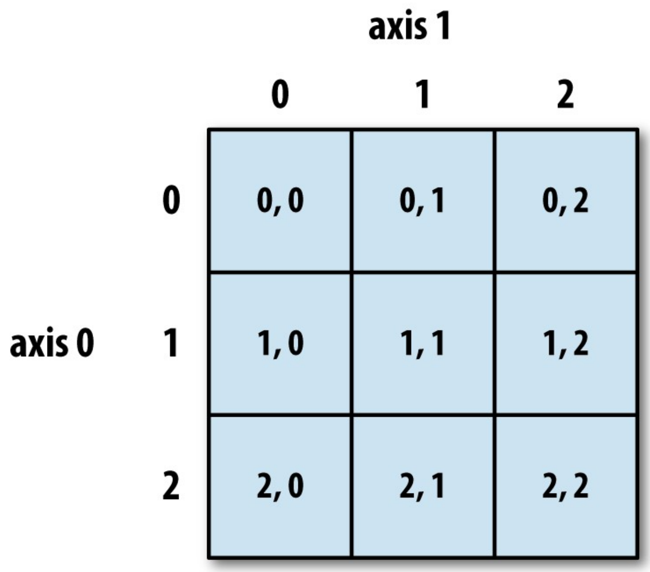
Basic Indexing and Slicing

> 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:

```
In [72]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

In [73]: arr2d[2]
Out[73]: array([7, 8, 9])

In [74]: arr2d[0][2]
Out[74]: 3

In [75]: arr2d[0, 2]
Out[75]: 3
```

Indexing elements in a NumPy array



Multidimensional arrays

> In multidimensional arrays, if you omit later indices, the returned object will be a lower dimensional **ndarray** consisting of all the data along the higher dimensions

```
In [76]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

In [77]: arr3d
Out[77]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])

In [78]: arr3d[0]
Out[78]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Multidimensional arrays

```
In [79]: old_values = arr3d[0].copy()

In [80]: arr3d[0] = 42

In [81]: arr3d
Out[81]:
array([[[42, 42, 42],
        [42, 42, 42]],
       [[ 7,  8,  9],
        [10, 11, 12]]])

In [82]: arr3d[0] = old_values

In [83]: arr3d
Out[83]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```


Multidimensional arrays

```
In [84]: arr3d[1, 0]
Out[84]: array([7, 8, 9])

In [85]: x = arr3d[1]

In [86]: x
Out[86]:
array([[ 7,  8,  9],
       [10, 11, 12]])

In [87]: x[0]
Out[87]: array([7, 8, 9])
```

Indexing with slices

> Like one-dimensional objects such as Python lists, **`ndarrays`** can be sliced with the familiar syntax:

```
In [88]: arr
Out[88]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])

In [89]: arr[1:6]
Out[89]: array([ 1,  2,  3,  4, 64])
```


Indexing with slices

> Slicing the two-dimensional array is a bit different:

```
In [90]: arr2d
Out[90]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [91]: arr2d[:2]
Out[91]:
array([[1, 2, 3],
       [4, 5, 6]])

In [92]: arr2d[:2, 1:]
Out[92]:
array([[2, 3],
       [5, 6]])
```

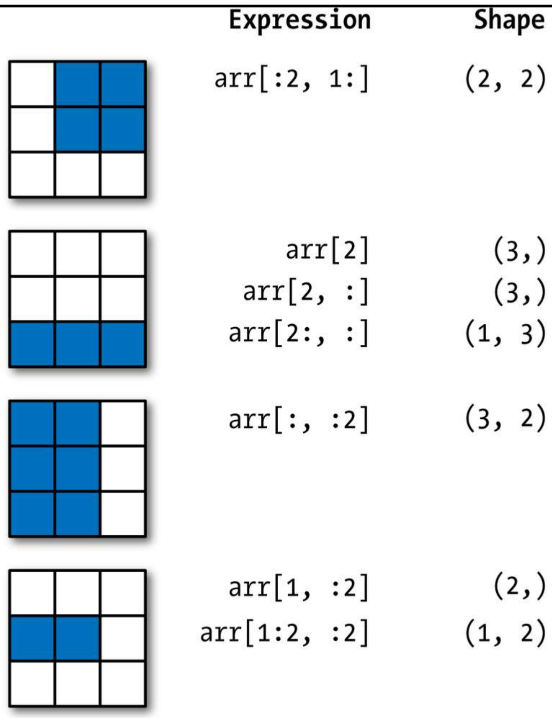
Indexing with slices

> By mixing integer indexes and slices, you get lower dimensional slices

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

```
In [93]: arr2d[1, :2]
Out[93]: array([4, 5])

In [94]: arr2d[:2, 2]
Out[94]: array([3, 6])

In [95]: arr2d[:, :1]
Out[95]:
array([[1],
       [4],
       [7]])

In [96]: arr2d[:2, 1:] = 0

In [97]: arr2d
Out[97]:
array([[1, 0, 0],
       [4, 0, 0],
       [7, 8, 9]])
```


Indexing with slices



Boolean Indexing

```
In [98]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [99]: data = np.random.randn(7, 4)

In [100]: names
Out[100]:
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'],
      dtype='<U4')

In [101]: data
Out[101]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397,  0.477 ],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```


Boolean Indexing

```
In [102]: names == 'Bob'
Out[102]: array([ True, False, False,  True, False, False, False], dtype=bool)

In [103]: data[names == 'Bob']
Out[103]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.669 , -0.4386, -0.5397,  0.477 ]])

In [104]: data[names == 'Bob', 2:]
Out[104]:
array([[ 0.769 ,  1.2464],
       [-0.5397,  0.477 ]])

In [105]: data[names == 'Bob', 3]
Out[105]: array([ 1.2464,  0.477 ])
```

Boolean Indexing

```
In [106]: names != 'Bob'
Out[106]: array([False,  True,  True, False,  True,  True,  True], dtype=bool)

In [107]: data[~(names == 'Bob')]
Out[107]:
array([[ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])

In [108]: cond = names == 'Bob'

In [109]: data[~cond]
Out[109]:
array([[ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

45

Boolean Indexing

```
[[ 0.0929,  0.2817,  0.769 ,  1.2464],
 [ 1.0072, -1.2962,  0.275 ,  0.2289],
 [ 1.3529,  0.8864, -2.0016, -0.3718],
 [ 1.669 , -0.4386, -0.5397,  0.477 ],
 [ 3.2489, -1.0212, -0.5771,  0.1241],
 [ 0.3026,  0.5238,  0.0009,  1.3438],
 [-0.7135, -0.8312, -2.3702, -1.8608]]
```

```
['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe']
```

```
In [110]: mask = (names == 'Bob') | (names == 'Will')

In [111]: mask
Out[111]: array([ True, False,  True,  True,  True, False, False], dtype=bool)

In [112]: data[mask]
Out[112]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397,  0.477 ],
       [ 3.2489, -1.0212, -0.5771,  0.1241]])
```

46

Boolean Indexing

```
[[ 0.0929,  0.2817,  0.769 ,  1.2464],
 [ 1.0072, -1.2962,  0.275 ,  0.2289],
 [ 1.3529,  0.8864, -2.0016, -0.3718],
 [ 1.669 , -0.4386, -0.5397,  0.477 ],
 [ 3.2489, -1.0212, -0.5771,  0.1241],
 [ 0.3026,  0.5238,  0.0009,  1.3438],
 [-0.7135, -0.8312, -2.3702, -1.8608]]
```

```
In [113]: data[data < 0] = 0

In [114]: data
Out[114]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.0072,  0.    ,  0.275 ,  0.2289],
       [ 1.3529,  0.8864,  0.    ,  0.    ],
       [ 1.669 ,  0.    ,  0.    ,  0.477 ],
       [ 3.2489,  0.    ,  0.    ,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [ 0.    ,  0.    ,  0.    ,  0.    ]])
```


Boolean Indexing

```
['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe']
```

```
[[ 0.0929,  0.2817,  0.769 ,  1.2464],
 [ 1.0072,  0.    ,  0.275 ,  0.2289],
 [ 1.3529,  0.8864,  0.    ,  0.    ],
 [ 1.669 ,  0.    ,  0.    ,  0.477 ],
 [ 3.2489,  0.    ,  0.    ,  0.1241],
 [ 0.3026,  0.5238,  0.0009,  1.3438],
 [ 0.    ,  0.    ,  0.    ,  0.    ]]
```

```
In [115]: data[names != 'Joe'] = 7

In [116]: data
Out[116]:
array([[ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 1.0072,  0.    ,  0.275 ,  0.2289],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [ 0.    ,  0.    ,  0.    ,  0.    ]])
```

Fancy Indexing

> Fancy indexing is a term adopted by NumPy to describe indexing using integer arrays.

```
In [117]: arr = np.empty((8, 4))

In [118]: for i in range(8):
   .....:     arr[i] = i

In [119]: arr
Out[119]:
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.]])
```


Fancy Indexing

> To select out a subset of the rows in a particular order, you can simply pass a list or **ndarray** of integers specifying the desired order:

```
[[ 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.]]
```

```
In [120]: arr[[4, 3, 0, 6]]
Out[120]:
array([[ 4.,  4.,  4.,  4.],
       [ 3.,  3.,  3.,  3.],
       [ 0.,  0.,  0.,  0.],
       [ 6.,  6.,  6.,  6.]])

In [121]: arr[[-3, -5, -7]]
Out[121]:
array([[ 5.,  5.,  5.,  5.],
       [ 3.,  3.,  3.,  3.],
       [ 1.,  1.,  1.,  1.]])
```

Fancy Indexing

> Passing multiple index arrays does something slightly different

> It selects a one-dimensional array of elements corresponding to each tuple of indices:

```
In [122]: arr = np.arange(32).reshape((8, 4))

In [123]: arr
Out[123]:
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]])

In [124]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[124]: array([ 4, 23, 29, 10])
```


Transposing Arrays and Swapping Axes

> Transposing is a special form of reshaping that similarly returns a view on the underlying data without copying anything.

> Arrays have the **`transpose`** method and also the special T attribute:

```
In [126]: arr = np.arange(15).reshape((3, 5))

In [127]: arr
Out[127]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

In [128]: arr.T
Out[128]:
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

Transposing Arrays and Swapping Axes

> Computing the inner matrix product using **np.dot**:

```
In [129]: arr = np.random.randn(6, 3)

In [130]: arr
Out[130]:
array([[-0.8608,  0.5601, -1.2659],
       [ 0.1198, -1.0635,  0.3329],
       [-2.3594, -0.1995, -1.542 ],
       [-0.9707, -1.307 ,  0.2863],
       [ 0.378 , -0.7539,  0.3313],
       [ 1.3497,  0.0699,  0.2467]])

In [131]: np.dot(arr.T, arr)
Out[131]:
array([[ 9.2291,  0.9394,  4.948 ],
       [ 0.9394,  3.7662, -1.3622],
       [ 4.948 , -1.3622,  4.3437]])
```


Transposing Arrays and Swapping Axes

> For higher dimensional arrays, transpose will accept a tuple of axis numbers to permute the axes

```
In [132]: arr = np.arange(16).reshape((2, 2, 4))

In [133]: arr
Out[133]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

In [134]: arr.transpose((1, 0, 2))
Out[134]:
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],
       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])
```

Transposing Arrays and Swapping Axes

> **`ndarray`** has the method **`swapaxes`**, which takes a pair of axis numbers and switches the indicated axes to rearrange the data:

```
In [135]: arr
Out[135]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])
In [136]: arr.swapaxes(1, 2)
Out[136]:
array([[[ 0,  4],
        [ 1,  5],
        [ 2,  6],
        [ 3,  7]],
       [[ 8, 12],
        [ 9, 13],
        [10, 14],
        [11, 15]]])
```


Universal Functions: Fast Element-Wise Array Functions

> A universal function, or ***ufunc***, is a function that performs element-wise operations on data in **`ndarrays`**.

> 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.

Universal Functions: Fast Element-Wise Array Functions

> Many ***ufuncs*** are simple element-wise transformations, like **`sqrt`** or **`exp`**:

```
In [137]: arr = np.arange(10)

In [138]: arr
Out[138]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [139]: np.sqrt(arr)
Out[139]:
array([ 0.    ,  1.    ,  1.4142,  1.7321,  2.    ,  2.2361,  2.4495,
        2.6458,  2.8284,  3.    ])

In [140]: np.exp(arr)
Out[140]:
array([    1.    ,     2.7183,     7.3891,    20.0855,    54.5982,
         148.4132,   403.4288,  1096.6332,  2980.958 ,  8103.0839])
```


Universal Functions: Fast Element-Wise Array Functions

> These are referred to as unary ***ufuncs***.

> Others, such as **`add`** or **`maximum`**, take two arrays (binary ***ufuncs***) and return a single array as the result:

```
In [141]: x = np.random.randn(8)

In [142]: y = np.random.randn(8)

In [143]: x
Out[143]:
array([-0.0119,  1.0048,  1.3272, -0.9193, -1.5491,  0.0222,  0.7584, -0.6605])

In [144]: y
Out[144]:
array([ 0.8626, -0.01  ,  0.05  ,  0.6702,  0.853 , -0.9559, -0.0235, -2.3042])

In [145]: np.maximum(x, y)
Out[145]:
array([ 0.8626,  1.0048,  1.3272,  0.6702,  0.853 ,  0.0222,  0.7584, -0.6605])
```

Using **numpy.vectorize()** for Custom Functions

```
import numpy as np

def fun(x):
    return x ** 2 + 3

arr = np.array([1, 2, 3, 4])

vectorized_fun = np.vectorize(fun)
result = vectorized_fun(arr)
print(result)  # Output: [4 7 12 19]
```


Using **numpy.vectorize()** for Custom Functions

```
def multiply_add(x, y):
    return x * y + 5

vectorized_gun = np.vectorize(multiply_add)

arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([10, 20, 30, 40])

result = vectorized_gun(arr1, arr2)
print(result)  # Output: [15 45 95 165]
```

Use **apply_along_axis()** for 1D Operations on Higher-Dimensional Arrays

```
arr2d = np.array([[1, 2, 3], [4, 5, 6]])


def sum_square(x):
    return np.sum(x ** 2)


result = np.apply_along_axis(sum_square, axis=1, arr=arr2d)
print(result)  # Output: [14 77]
```