NumPy Basics: Arrays and Vectorized Computation

Part 1

- 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.

- 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.

- 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.

- For most data analysis applications, the main areas of functionality I'll focus on are:
  - Fast vectorized array operations for data munging and cleaning, subsetting and filtering, transformation, and any other kinds of computations
  - Common 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)

- While NumPy provides a computational foundation for general numerical data processing, many readers will want to use pandas as the basis for most kinds of statistics or analytics, especially on tabular data.
- pandas also provides some more domain-specific functionality like time series manipulation, which is not present in 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.
- There are a number of reasons for this:
  - 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.

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

```
In [1]: import numpy as np
        my_arr = np.arange(1000000)
        my_list = list(range(1000000))
```

- Now let’s multiply each sequence by 2:

```
In [2]: %time for _ in range(10): my_arr2 = my_arr * 2

        CPU times: user 4.11 ms, sys: 25.5 ms, total: 29.6 ms
        Wall time: 29.5 ms

In [3]: %time for _ in range(10): my_list2 = [x * 2 for x in my_list]

        CPU times: user 361 ms, sys: 105 ms, total: 466 ms
        Wall time: 464 ms
```

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

The NumPy ndarray: A Multidimensional Array Object

Part 1

- One of the key features of NumPy is its N-dimensional array object, or ndarray, which is a fast, flexible container for large datasets in Python.
- Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements.

- To give you a flavor of how NumPy enables batch computations with similar syntax to scalar values on built-in Python objects, I first import NumPy and generate a small array of random data:

```
In [4]: import numpy as np
        # Generate some random data
        data = np.random.randn(2, 3)
        data

Out[4]: array([[-0.96739214, -0.27428928, -0.65430991],
               [ 1.42501939, -1.18751685,  1.86248932]])
```

- I then write mathematical operations with data:

```
In [5]: data * 10

Out[5]: array([[ -9.67392141,  -2.74289281,  -6.54309912],
               [ 14.25019394, -11.87516851,  18.62489317]])

In [6]: data + data

Out[6]: array([[-1.93478428, -0.54857856, -1.30861982],
               [ 2.85003879, -2.3750337 ,  3.72497863]])
```

- An ndarray is a generic multidimensional container for homogeneous data; that is, all of the elements must be 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 [7]: data
Out[7]: array([[-0.96739214, -0.27428928, -0.65430991],
               [ 1.42501939, -1.18751685,  1.86248932]])

In [8]: data.shape
Out[8]: (2, 3)

In [9]: data.dtype
Out[9]: dtype('float64')
```

Creating ndarrays

- The easiest way to create an array is to use the `array` function.
- This accepts any sequence-like object (including other arrays) and produces a new NumPy array containing the passed data.
- For example, a list is a good candidate for conversion:

```
In [10]: data1 = [6, 7.5, 8, 0, 1]
         arr1 = np.array(data1)
         arr1

Out[10]: array([6. , 7.5, 8. , 0. , 1. ])
```

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

```
In [11]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
         arr2 = np.array(data2)
         arr2

Out[11]: array([[1, 2, 3, 4],
                [5, 6, 7, 8]])
```

- Since `data2` was a list of lists, the NumPy array `arr2` has two dimensions with shape inferred from the data.
- We can confirm this by inspecting the `ndim` and `shape` attributes:

```
In [12]: arr2.ndim

Out[12]: 2

In [13]: arr2.shape

Out[13]: (2, 4)
```

- 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; for example, in the previous two examples we have:

```
In [14]: arr1
Out[14]: array([6. , 7.5, 8. , 0. , 1. ])

In [15]: arr1.dtype
Out[15]: dtype('float64')

In [16]: arr2
Out[16]: array([[1, 2, 3, 4],
               [5, 6, 7, 8]])

In [17]: arr2.dtype
Out[17]: dtype('int64')
```

- 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:

```
In [18]: np.zeros(10)
Out[18]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

In [19]: np.ones((3, 6))
Out[19]: array([[1., 1., 1., 1., 1., 1.],
               [1., 1., 1., 1., 1., 1.],
               [1., 1., 1., 1., 1., 1.]])

In [20]: np.empty((2, 3, 2))
Out[20]: array([[[5.95525570e+228, 9.40039426e-154],
                [1.23475616e-259, 3.68777421e+180],
                [4.47593816e-091, 7.13637443e+159]],

               [[9.16526748e+242, 3.45392512e-086],
                [9.91599552e-096, 6.98348255e-077],
                [9.91599515e-096, 3.64464665e-086]]])
```

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

```
In [21]: np.arange(15)
Out[21]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

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 1s 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 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 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 × N identity matrix (1s on the diagonal and 0s elsewhere)