

Module 3 Part 1: NumPy Basics

Introduction

The key topic we are discussing in this module is NumPy, or Numerical Python package. We will continue working with Python and learn commonly used functions in the NumPy package.

This module consists of 2 parts:

- **Part 1** - NumPy Basics
- **Part 2** - Operations with NumPy Arrays

Each part is provided in a separate notebook file. It is recommended that you follow the order of the notebooks.

Learning Outcomes

In this module, you will:

- Continue to build your Python skills
- Use the NumPy data analysis library to work with large arrays

Readings and Resources

The majority of the notebook content draws from the recommended readings. We invite you to further supplement this notebook with the following recommended texts:

1. McKinney, W. (2017). NumPy Basics: Arrays and Vectorized Computation In *Python for Data Analysis* (pp 85-119). O-Reilly: Boston
2. SciPy community (2018). *Quickstart tutorial*. Retrieved from the following [web page](#).


Table of Contents

- [Module 3 Part 1: NumPy Basics](#)
- [Introduction](#)

- Learning Outcomes
- Readings and Resources
- Table of Contents
- NumPy Basics
 - Preparing the data: NumPy
 - NumPy introduction
 - The NumPy ndarray: a multidimensional array object
 - Creating NumPy arrays
 - Creating NumPy arrays from lists, an overview of array properties
 - EXERCISE 1: Creating NumPy arrays from lists
 - Creating NumPy array using `arange()`, `linspace()` and `random()` functions
 - EXERCISE 2: Creating NumPy arrays of random numbers and ranges
 - Creating NumPy arrays of 1s and 0s
 - Review of array data types
- References

NumPy Basics

Preparing the data: NumPy

 This image shows the stages of creating a model from start to finish. (Course Authors, 2018) *This image shows the stages of creating a model from start to finish.*

In the last module, you learned about the basic syntax of Python, including various data types, and how to read / write files. In the rest of the course, you will learn how to use libraries within Python to help you understand and analyze your data.

In this module, we will be learning about NumPy. This library allows you to create large matrices and arrays from imported data. For example, if you import data in the form of a list, by using NumPy you can turn it into a matrix or array. This will allow you to easily organize and understand what data points make up your sample. Further, once your data is organized in a matrix or array you can apply the mathematical functions available within NumPy to draw some preliminary conclusions about your dataset.

NumPy introduction

NumPy stands for **Numerical Python** or **Numeric Python**. NumPy is the core Python library for scientific computing. Most computational packages providing scientific functionality use NumPy's array objects.

Some of the features of NumPy:

- NumPy arrays are n-dimensional array objects, a core component of scientific and numerical computation in Python
- NumPy provides tools for integrating with C, C++ and Fortran code
- NumPy also provides tools to perform linear algebra, generate random numbers and tools for reading/writing array data to disk and working with memory-mapped files

More about NumPy can be found at numpy.org (NumPy developers, 2018).

One of the reasons NumPy is so important for numerical computations is that the package is designed for efficiently handling 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 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.

The NumPy ndarray: a multidimensional array object

NumPy's main tool which is used for scientific computations is a multidimensional array object, N-dimensional, or `ndarray`, a high-performance data structure build for efficient computation of arrays and matrices.

NumPy arrays allow us to perform mathematical operations on blocks of data using syntax that we would use for the same type of operations between scalars.

It is important to note that NumPy's arrays are **homogeneous structures**. Usually the elements in the arrays are numbers, but this doesn't have to be the case. However, all elements of the array should be of the same type.

Here are some important terms that will help us to work with NumPy arrays:

- NumPy `dimensions` are called `axes`.
- The number of dimensions is the `rank` of the array. You can say that the number of axes defines the rank of the array.
- The `shape` of an array is a tuple of integers showing the size of the array along each dimension.

NOTE: These terms are relative to the NumPy array *instance* in question. It is possible to nest general Python objects as elements of Numpy arrays. Numpy is blind to the data-dimensions of the object in each NumPy array element, unless they are of a type recognized by NumPy.

For example, a scalar (one dimensional element), string (categorical data, thus zero or one dimension), or a NumPy array (arbitrary dimension) would be consistent with the terms above. These terms would no longer be consistent if used with tuples. "*Dimension*" in a data context refers to the number of array elements used to span a finite numerical space. "*Dimension*" in an array context refers to the number of indexes needed to traverse the `ndarray`. Keep this in mind when going forward.

We will review all these features in the next section.

Creating NumPy arrays

NumPy arrays can be created:

- From other Python structures, for example lists or tuples using an `array()` function
- Using `random` functions to create an array of random data, or using the `range()` function to create an array of a sequence of numbers
- Using one of the `zeros()`, `ones()` or `empty()` functions to create arrays of `0`s or `1`s or an array of uninitialized empty elements

Let's create several arrays using these functions.

Creating NumPy arrays from lists, an overview of array properties

The main NumPy library is `numpy` which we need to import to be able to use NumPy functionality.

```
In [1]: import numpy as np
```

Numerical data arranged in an array-like structure in Python can be converted to arrays using the `array()` function. The most obvious examples of array-like structures are lists and tuples.

```
In [2]: # First, we will use a simple list of 4 integer numbers to create an array

list_int = [8, 3, 34, 111]
a_int = np.array(list_int)
a_int
```

```
Out[2]: array([ 8,  3, 34, 111])
```

The **rank** of the array `a_int` is 1 — it is a one-dimensional array of numbers. To check the rank of the array, use `ndim`, which outputs the number of axes (dimensions) of the array.

To see what those dimensions are use `.shape`. The output is a tuple of integers indicating the size of the array in each dimension.

```
In [3]: # The rank of the array, or the number of dimensions (axes):  
a_int.ndim
```

Out[3]: 1

```
In [4]: # The size of the array in each dimension.  
# For a 1-dimensional array, this represents the number of elements along a single  
a_int.shape
```

Out[4]: (4,)

We can check the type of the object using `type()` function — just like we did in the previous modules:

```
In [5]: type(a_int) # NumPy's array class is called ndarray.
```

Out[5]: numpy.ndarray

dtype returns an object describing the type of the elements in the array.

```
In [6]: a_int.dtype
```

Out[6]: dtype('int64')

This array was created from a list of integers, hence the array is of integer type.

To create a 2-dimensional array, we can use a list of lists. Below, we will create an array with 2 rows and 3 columns.

```
In [7]: list_2dim = [[1.5,2,3], [4,5,6]]  
a_2dim = np.array(list_2dim)  
a_2dim
```

Out[7]: array([[1.5, 2. , 3.],
 [4. , 5. , 6.]])

For a matrix with **n** rows and **m** columns the shape will be **(n,m)** :

```
In [8]: print(a_2dim.ndim) # number of dimensions  
print(a_2dim.shape) # the size of the array in each dimension
```

2
(2, 3)

The `a_2dim` array has been created from a nested list of integers and floating point numbers. You can see above that each element of the array has a decimal point after the

number indicating the floating point data type.

```
In [9]: a_2dim.dtype
```

```
Out[9]: dtype('float64')
```

We can create a NumPy array from a mix of tuples and lists:

```
In [10]: a_mix = np.array([[1, 2.0], [0, 0], (5.78, 3.)])
a_mix
```

```
Out[10]: array([[1. , 2.  ],
                [0. , 0.  ],
                [5.78, 3.  ]])
```

The type of the array can be explicitly specified at creation time:

```
In [11]: a_intfloat = np.array([[1, 2], [3, 4]], dtype = float)
a_intfloat
```

```
Out[11]: array([[1., 2.],
                [3., 4.]])
```

Even though the elements of the nested list were all integers, the array created using this list is of `float64` type:

```
In [12]: a_intfloat.dtype
```

```
Out[12]: dtype('float64')
```

EXERCISE 1: Creating NumPy arrays from lists

1). Create a 1-dimensional array of numbers from a list `[2, 5.5, 3.1415, 0]`. What type will be the resulting array?

```
In [9]: # Type your code here:

a_list = np.array([2, 5.5, 3.1415, 0])
print(a_list)
a_list.dtype
```

```
[[2.    5.5   3.1415 0.    ]]
```

```
Out[9]: dtype('float64')
```

2). Create a 2-dimensional array of integers

```
In [12]: # Type your code here:
b_list = np.array([[1,2,3],[4,5,6.0]],dtype = int)
```

```
In [13]: b_list
```

```
Out[13]: array([[1, 2, 3],
               [4, 5, 6]])
```

```
In [ ]:
```

```
In [15]: # 1). Solution:
```

```
list_numbers = [2, 5.5, 3.1415, 0]
array_numbers = np.array(list_numbers)
array_numbers
```

```
Out[15]: array([2.    , 5.5    , 3.1415, 0.    ])
```

```
In [16]: ''' Created array will have dtype = 'float64'
           because the second and third elements of the list used to create an array are f

array_numbers.dtype
```

```
Out[16]: dtype('float64')
```

```
In [17]: # 2). Solution:
```

```
ar_int_numbers1 = np.array([[2, 5.5, 3.1415, 0], [2, 5.5, 3.1415, 0]], dtype=int)
ar_int_numbers1
```

```
Out[17]: array([[2, 5, 3, 0],
               [2, 5, 3, 0]])
```

```
In [18]: # We could have created an array using only integers as an input:
```

```
ar_int_numbers2 = np.array([[2, 5, 3], [5, 12, 24]])
ar_int_numbers2
```

```
Out[18]: array([[ 2,  5,  3],
               [ 5, 12, 24]])
```

```
In [19]: ar_int_numbers1.dtype
```

```
Out[19]: dtype('int64')
```

```
In [20]: ar_int_numbers2.dtype
```

```
Out[20]: dtype('int64')
```

Creating NumPy array using arange(), linspace() and random() functions

To create sequences of numbers, NumPy provides a function analogous to `range()` that returns arrays. The function `arange()` returns arrays with regularly incrementing values.

```
In [21]: # This will create an array of integers from 0 to 9
```

```
np.arange(10)
```

```
Out[21]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [22]: # An array of floating point numbers, from 2.0 to 9.0 (10 is excluded), with increment
np.arange(2, 10, dtype=float)
```

```
Out[22]: array([2., 3., 4., 5., 6., 7., 8., 9.])
```

```
In [23]: # An array of numbers from 0 to 10 (exclusive) with 0.3 increment
np.arange(0, 10, 0.3)    # start, end (exclusive), step
```

```
Out[23]: array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8, 2.1, 2.4, 2.7, 3. , 3.3, 3.6,
               3.9, 4.2, 4.5, 4.8, 5.1, 5.4, 5.7, 6. , 6.3, 6.6, 6.9, 7.2, 7.5,
               7.8, 8.1, 8.4, 8.7, 9. , 9.3, 9.6, 9.9])
```

As we can see from the example above, when the `arange()` function is used with floating point arguments, it is difficult to estimate the number of elements in the resulting array.

If you need to create an array with a predefined number of elements between a start and an end point, use the function `linspace()`. The `linspace()` function receives as an argument the number of elements that we want instead of the step: `linspace(start, stop, number of elements)`.

```
In [24]: np.linspace(0.4, 1.1, 7)    # create an array of 7 elements from 0.4 to 1.1 (inclus
```

```
Out[24]: array([0.4          , 0.51666667, 0.63333333, 0.75          , 0.86666667,
               0.98333333, 1.1          ])
```

```
In [25]: # If the end point should not be included, use endpoint=False parameter
np.linspace(0.4, 1.1, 7, endpoint = False)
```

```
Out[25]: array([0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

To generate an array with some random data, we can use `np.random.randn()` function. The sequence of numbers created using this function will be normally distributed around its mean. For a complete list of functions for random sampling, refer to the NumPy documentation, [numpy.random package](#). (SciPy community, 2018)

```
In [26]: np.random.randn(4)    # 1-dimensional array of 4 normally distributed numbers
```

```
Out[26]: array([ 0.05939082, -0.49575749,  0.54485679, -1.07434219])
```

```
In [27]: # creating 2x3 2-dimensional array of random numbers:
np.random.randn(2, 3)
```

```
Out[27]: array([[ 0.94412535,  2.16498226, -1.30516753],
               [-0.53715055, -0.96661248,  0.09073315]])
```


EXERCISE 2: Creating NumPy arrays of random numbers and ranges

1). Create two arrays of 5 numbers from 5 to 25 using `arange()` and `linspace()` functions

```
In [16]: # type your code here (`arange`)  
a_array = np.arange(5,26,5)
```

```
In [17]: a_array
```

```
Out[17]: array([ 5, 10, 15, 20, 25])
```

```
In [23]: # type your code here (`linspace`)  
b_array = np.linspace(5,25,5,dtype=int)
```

```
In [24]: b_array
```

```
Out[24]: array([ 5, 10, 15, 20, 25])
```

2). Create an array of random numbers with dimensions 2x3

```
In [30]: # Type your code here:
```

```
In [25]: c_array = np.random.randn(2, 3)
```

```
In [26]: c_array
```

```
Out[26]: array([[ -1.27009613, -1.36635576,  0.6291594 ],  
                [ -1.47270768,  0.64734961,  0.43854825]])
```

```
In [ ]:
```

```
In [31]: # 1). Solution:  
arange_array = np.arange(5,26,5)  
arange_array
```

```
Out[31]: array([ 5, 10, 15, 20, 25])
```

In the solution above, we had to use a number greater than 25 as an endpoint, if we wanted 25 to be in the resulting array, per the exercise requirement.

```
In [32]: linspace_array = np.linspace(5, 25, 5, dtype = int)  
linspace_array
```

```
Out[32]: array([ 5, 10, 15, 20, 25])
```

```
In [33]: # 2). Solution:
np.random.randn(2, 3)
```

```
Out[33]: array([[ -1.5059236 , -1.62596245, -1.27357065],
                [-1.30847285,  0.45047557, -0.91428171]])
```

Creating NumPy arrays of 1s and 0s

Often, in scientific computations, the elements of an array are originally unknown, but its size is known. For this scenario, NumPy has several functions to create arrays with initial placeholder content.

The function `zeros()` creates an array full of zeros, the function `ones()` creates an array full of ones, and the function `empty()` creates an array without populating its elements with any predefined numbers. Hence, the initial content of this array is random and depends on the state of the memory. By default, the `dtype` of the created array is `float64`.

In all cases, note that to define the shape of future array, we are passing tuples. For example, in the line below, we are passing a tuple (4,7) to create a 2-dimensional array of zeros with shape (4,7):

```
In [34]: array_zeros = np.zeros((4,7))
array_zeros
```

```
Out[34]: array([[0., 0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0., 0.]])
```

```
In [35]: array_zeros.shape
```

```
Out[35]: (4, 7)
```

```
In [36]: # Create a 3-dimensional array, 2x3x4, of ones, of type integer
np.ones((2, 3, 4), dtype = np.int16)
```

```
Out[36]: 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]]], dtype=int16)
```

```
In [37]: # Create an empty array, a placeholder
np.empty((3, 5))
```

```
Out[37]: array([[ 2.68156159e+154, -2.68678775e+154,  3.95252517e-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]])
```

Review of array data types

In the previous section, we were able to create an array of a certain data type. For example, by using parameter `dtype=float`, we can create an array of all floating point numbers regardless of what is the data type of the numbers used to create an array.

What if we need to convert (or *cast*) an array from one data type to another data type after it was created? It can be done with `astype()` method:

```
In [38]: arr_int = np.array([1, 2, 3, 4, 5])
arr_int.dtype
```

```
Out[38]: dtype('int64')
```

```
In [39]: arr_float = arr_int.astype(np.float64)
arr_float.dtype
```

```
Out[39]: dtype('float64')
```

NOTE: Be careful when casting an array of floating point numbers to integers — NumPy will simply truncate the decimal part of the number:

```
In [40]: arr_float2 = np.array ([4.3, 5.34, 6.89, 0.3])
arr_float2.dtype
```

```
Out[40]: dtype('float64')
```

```
In [41]: arr_int2 = arr_float2.astype(np.int32)
arr_int2
```

```
Out[41]: array([4, 5, 6, 0], dtype=int32)
```

`astype()` method can be used to convert numbers represented as strings to numeric form

```
In [42]: arr_str = np.array(['3.14', '4.56', '7.89', '32'])
arr_str
```

```
Out[42]: array(['3.14', '4.56', '7.89', '32'], dtype='<U4')
```

```
In [43]: arr_str.astype(np.float)
```

```
Out[43]: array([ 3.14,  4.56,  7.89, 32.  ])
```

End of Part 1

This notebook makes up one part of this module. Now that you have completed this part, please proceed to the next notebook in this module.

If you have any questions, please reach out to your peers using the discussion boards. If you and your peers are unable to come to a suitable conclusion, do not hesitate to reach out to your instructor on the designated discussion board.

References

NumPy developers (2018). *NumPy*. Retrieved from <http://www.numpy.org/#>

SciPy community (2018). *Random Sampling*. Retrieved from <https://docs.scipy.org/doc/numpy/reference/routines.random.html>