

Python for Data Analysis, 3rd edition

Data Wrangling with pandas, NumPy, and Jupyter

Stephen CUI¹

January 4, 2022

¹cuixuanStephen@gmail.com

Contents

1	NumPy Basics: Arrays and Vectorized Computation	3
1.1	The NumPy ndarray: A Multidimensional Array Object	4
1.1.1	Creating ndarrays	5
1.1.2	Data Types for ndarrays	6
1.2	Pseudorandom Number Generation	6
1.3	Universal Functions: Fast Element-Wise Array Functions	6
1.4	Array-Oriented Programming with Arrays	6
1.5	Linear Algebra	6
1.6	Example: Random Walks	6
1.7	Conclusion	6

Chapter 1

NumPy Basics: Arrays and Vectorized Computation

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

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

For most data analysis applications, the main areas of functionality I'll focus on are:

- Fast array-based operations for data munging and cleaning, subsetting and filtering, transformation, and any other kind of computation
- 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 heterogeneous datasets
- Expressing conditional logic as array expressions instead of loops with `if-elif-else` branches
- Group-wise data manipulations (aggregation, transformation, and function application)

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, which can be slow for large sequences. NumPy is faster than regular Python code because its C-based algorithms avoid overhead present with regular interpreted Python code.

```
1 import numpy as np
2 my_arr = np.arange(1_000_000)
3 my_list = list(range(1_000_000))
4
5 %timeit my_arr2 = my_arr * 2
6 %timeit my_list2 = [x * 2 for x in my_list]
```

1.1 The NumPy ndarray: A Multidimensional Array Object

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.

```
1 import numpy as np
2
3 data = np.array([[1.5, -.1, 3], [0, -3, 6.5]])
4 data * 10
5 data + data
```

Notes

In this chapter and throughout the book, I use the standard NumPy convention of always using `import numpy as np`. It would be possible to put `from numpy import *` in your code to avoid having to write `np.`, but I advise against making a habit of this. The `numpy` namespace is large and contains a number of functions whose names conflict with built-in Python functions (like `min` and `max`). Following standard conventions like these is almost always a good idea.

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:

Notes

Whenever you see “array”, “NumPy array,” or “ndarray”, in most cases they all refer to the ndarray object.

1.1.1 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.

```
1 data1 = [6, 7.5, 8, 0, 1]
2 arr1 = np.array(data1)
3 arr1
```

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

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

Unless explicitly specified (discussed in [Subsection 1.1.2](#)), `numpy.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.

```
1 print(arr1.dtype) # float64
2 print(arr2.dtype) # int32
```

In addition to `numpy.array`, there are a number of other functions for creating new arrays. To create a higher dimensional array with these methods, pass a tuple for the shape:

```
1 np.zeros(10)
2 np.zeros((3, 6))
3 np.empty((1, 2, 3))
```

Warnings

It's not safe to assume that `numpy.empty` will return an array of all zeros. This function returns uninitialized memory and thus may contain nonzero “garbage” values. You should use this function only if you intend to populate(填充) the new array with data.

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

Table 1.1: Some important NumPy array creation functions

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

```

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

```

See [Table 1.1](#) for a short list of standard array creation functions. Since NumPy is focused on numerical computing, the data type, if not specified, will in many cases be float64 (floating point).

1.1.2 Data Types for ndarrays

1.2 Pseudorandom Number Generation

1.3 Universal Functions: Fast Element-Wise Array Functions

1.4 Array-Oriented Programming with Arrays

1.5 Linear Algebra

1.6 Example: Random Walks

1.7 Conclusion