Python for Data Analysis, 3rd edition

Data Wrangling with pandas, NumPy, and Jupyter

Stephen CUI[1]

January 4, 2022

[1]cuixuanStephen@gmail.com

Contents

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.

```
import numpy as np
my_arr = np.arange(1_000_000)
my_list = list(range(1_000_000))

%timeit my_arr2 = my_arr * 2
%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.

```
import numpy as np

data = np.array([[1.5, -.1, 3], [0, -3, 6.5]])
data * 10
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.

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

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

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

```
print(arr1.dtype) # float64
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:

```
np.zeros(10)
np.zeros((3, 6))
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:

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

Table 1.1: Some important NumPy array creation functions

Function	Description
`array`	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
`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 data type; `ones_like` takes another array and produces a ones array of the same shape and data type
`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 data type with all values set to the indicated "fill value"; `full_like` takes another array and produces a filled array of the same shape and data type
`eye`, `identity`	Create a square $N \times N$ identity matrix (1s on the diagonal and 0s elsewhere)

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

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

```
arr1 = np.array([1, 2, 3], dtype=np.float64)
arr2 = np.array([1, 2, 3], dtype=np.int32)
print(arr1.dtype) # float64
print(arr2.dtype) # int32
```

Notes

Don't worry about memorizing the NumPy data types, especially if you're a new user. It's often only necessary to care about the general kind of data you're dealing with, whether floating point, complex, integer, Boolean, string, or general Python object. When you need more control over how data is stored in memory and on disk, especially large datasets, it is good to know that you have control over the storage type.

See Table 1.2 for a full listing of NumPy's supported data types.

Table 1.2: 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 64-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
`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 astring data type with length 10, use ’S10’
`unicode_`	`U`	Fixed-length Unicode type (number of bytes platform specific); same specification semantics as `string_` (e.g., ’U10’)

Notes

There are both signed and unsigned integer types, and many readers will not be familiar with this terminology. A signed integer can represent both positive and negative integers, while an unsigned integer can only represent nonnegative integers. For example, int8 (signed 8-bit integer) can represent integers from -128 to 127 (inclusive), while uint8 (unsigned 8-bit integer) can represent 0 through 255.

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

```
arr = np.array([1, 2, 3, 4, 5])
print(arr) # [1 2 3 4 5]
print(arr.dtype) # int32
float_arr = arr.astype(np.float64)
print(float_arr) # [1. 2. 3. 4. 5.]
print(float_arr.dtype) # float64
```

If I cast some floating-point numbers to be of integer data type, the decimal part will be truncated:

```
arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
print(arr)
arr.astype(np.int32)
# array([ 3, -1, -2,  0, 12, 10])
```

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_)
numeric_strings.astype(float)
# array([ 1.25, -9.6 , 42.  ])
```

Warnings

Be cautious when using the numpy.string_ type, as string data in NumPy is fixed size and may truncate input without warning. pandas has more intuitive out-of-the-box behavior on non-numeric data.

If casting were to fail for some reason (like a string that cannot be converted to float64), a ValueError will be raised. Before, I was a bit lazy and wrote float instead of np.float64; NumPy aliases the Python types to its own equivalent data types.

You can also use another array's dtype attribute:

```
int_array = np.arange(10)

calibers = np.array([.22, .270, .357, .44, .50], dtype=np.float64)
```

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

There are shorthand type code strings you can also use to refer to a dtype.

Notes

Calling astype always creates a new array (a copy of the data), even if the new data type is the same as the old data type.

1.1.3 Arithmetic with NumPy Arrays

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