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.

```python
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.

```python
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:

```python
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:

```python
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:

```python
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:

```python
int_array = np.arange(10)

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

```
4    int_array.astype(calibers.dtype)
5    # 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

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

- Any arithmetic operations between equal-size arrays apply the operation element-wise

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

- Comparisons between arrays of the same size yield Boolean arrays

```
1    arr = np.array([[1., 2., 3.], [4., 5., 6.]])
2    arr * arr
3    arr - arr
4
5    1 / arr
6    arr ** 2
7
8    arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])
9    arr2 > arr
```

Evaluating operations between differently sized arrays is called broadcasting and will be discussed in more detail in Appendix A.

1.1.4   Basic Indexing and Slicing

NumPy array indexing is a deep topic, as there are many ways you may want to select a subset of your data or individual elements.

```
1    arr = np.arange(10)
2    arr[5]
3    # 5
4
```

```
5   arr[5: 8]
6   # array([5, 6, 7])
7
8   arr[5: 8] = 12
9   arr
10  # array([ 0,   1,   2,   3,   4, 12, 12, 12,   8,   9])
```

> **Notes**
>
> An important first distinction from Python's built-in lists is that **array slices are views on the original array**. This means that the data is not copied, and any modifications to the view will be reflected in the source array.

```
1   arr_slice = arr[5: 8]
2   arr_slice
3   # array([12, 12, 12])
4   arr_slice[1] = 123
5   arr
6   # array([ 0,    1,    2,    3,    4,  12, 123,  12,    8,    9])
7
8   arr_slice[:] = 64
9   arr
10  # array([ 0,   1,   2,   3,   4, 64, 64, 64,   8,   9])
```

Pythonn内置的列表

```
1   a = list(range(10))
2   print(a)
3   # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
4   b = a[2: 5]
5   # [2, 3, 4]
6   print(b)
7   b[1] = 1234
8   print(b)
9   # [2, 1234, 4]
10  print(a)
11  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Figure 1.1: Indexing elements in a NumPy array

As NumPy has been designed to be able to work with very large arrays, you could imagine performance and memory problems if NumPy insisted on always copying data.

> **Warnings**
>
> If you want a copy of a slice of an ndarray instead of a view, you will need to explicitly copy the array—for example, arr[5:8].copy(). As you will see, pandas works this way, too.

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. Thus, individual elements can be accessed recursively. But that is a bit too much work, so you can pass a comma-separated list of indices to select individual elements.

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


# these are equivalent:
arr2d[0][2]
arr2d[0, 2]
```

See Figure 1.1 for an illustration of indexing on a two-dimensional array. I find it helpful to think of axis 0 as the "rows" of the array and axis 1 as the "columns."

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.
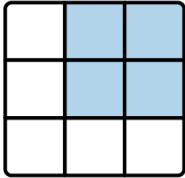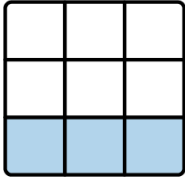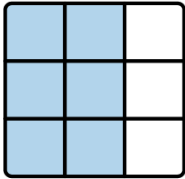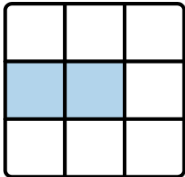
Expression	Shape
arr[:2,1:]	(2,2)
arr[2]	(3,)
arr[2, :]	(3,)
arr[2:, :]	(1,3)
arr[:, :2]	(3,2)
arr[1, :2]	(2,)
arr[1:2, :2]	(1,2)

Figure 1.2: Two-dimensional array slicing

Note that in all of these cases where subsections of the array have been selected, the returned arrays are views.

> **Warnings**
>
> This multidimensional indexing syntax for NumPy arrays will not work with regular Python objects, such as lists of lists.

Indexing with slices

Like one-dimensional objects such as Python lists, ndarrays can be sliced with the familiar syntax.

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

```
lower_dim_slice = arr2d[1, :2]
lower_dim_slice.shape

```

```
4   arr2d[:2, 2]
5   arr2d[:, :1]
6
7   arr2d[:2, 1:] = 0
8   arr2d
```

1.1.5 Boolean Indexing

The Boolean array must be of the same length as the array axis it's indexing. You can even mix and match Boolean arrays with slices or integers (or sequences of integers).

```
1   names = np.array(["Bob", "Joe", "Will", "Bob", "Will", "Joe", "Joe"])
2   print(names)
3   data = np.array([[4, 7], [0, 2], [-5, 6], [0, 0], [1, 2], [-12, 4], [3, 4]])
4   print(data)
5   print(names == "Bob")
6   print(data[names == "Bob"])
7   print(data[names == "Bob", 1:])
8   print(data[names == "Bob", 1])
```

- To select everything but "Bob" you can either use != or negate the condition using ∼

- The ∼ operator can be useful when you want to invert a Boolean array referenced by a variable

- To select two of the three names to combine multiple Boolean conditions, use Boolean arithmetic operators like & (and) and |(or).

```
1   names = np.array(["Bob", "Joe", "Will", "Bob", "Will", "Joe", "Joe"])
2   names
3   data = np.array([[4, 7], [0, 2], [-5, 6], [0, 0], [1, 2], [-12, 4], [3, 4]])
4   data
5   names == "Bob"
6   data[names == "Bob"]
7   data[names == "Bob", 1:]
8   data[names == "Bob", 1]
9
10  names != "Bob"
11  ~(names == "Bob")
12  data[~(names == "Bob")]
```

```
13  cond = names == "Bob"
14  data[~cond]
15
16  mask = (names == "Bob") | (names == "Will")
17  data[mask]
```

Selecting data from an array by Boolean indexing and assigning the result to a new variable always creates a copy of the data, even if the returned array is unchanged.

> **Warnings**
>
> The Python keywords and and or do not work with Boolean arrays. Use & (and) and |(or) instead.

- Setting values with Boolean arrays works by substituting the value or values on the righthand side into the locations where the Boolean array's values are True.

- You can also set whole rows or columns using a one-dimensional Boolean array

```
1  data[data < 0] = 0
2  data
3
4  data[names != "Joe"] = 7
5  data
```

1.1.6   Fancy Indexing

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

- To select a subset of the rows in a particular order, you can simply pass a list or ndarray of integers specifying the desired order. Using negative indices selects rows from the end

- Passing multiple index arrays does something slightly different; it selects a one-dimensional array of elements corresponding to each tuple of indices.(返回的是一个一维的数组，这超乎了我的想象)

```
1  arr = np.zeros((8, 4))
2  for i in range(8):
3      arr[i] = i
4  arr
5
6  arr[[4, 3, 0, 6]]
7
```

```
8   arr[[-3, -5, -7]]
9
10  arr = np.arange(32).reshape((8, 4))
11  arr[[1, 5, 7, 2], [0, 3, 1, 2]]
```

To learn more about the reshape method, have a look at Appendix A. Here the elements (1, 0), (5, 3), (7, 1), and (2, 2) were selected. The result of fancy indexing with as many integer arrays as there are axes is always one-dimensional.

如果你想要返回一个二维的数组，你写的索引位置应该也得是一个矩形数组，下面是一种写法：

```
1   arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
```

Keep in mind that fancy indexing, unlike slicing, always copies the data into a new array when assigning the result to a new variable. If you assign values with fancy indexing, the indexed values will be modified.

```
1   a = arr[[1, 5, 7, 2], [0, 3, 1, 2]]
2   a[0] = 10000
3   print(a)
4   print(arr)
```

1.1.7 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 the special T attribute. When doing matrix computations, you may do this very often. The @ infix operator is another way to do matrix multiplication.

- Simple transposing with .T is a special case of swapping axes. ndarray has the method swapaxes, which takes a pair of axis numbers and switches the indicated axes to rearrange the data. 只返回源数据的视图，不会复制数据。

- 对于高维数组，transpose需要得到一个由轴编号组成的元组才能对这些轴进行转置（费脑子，不要用图像理解）。用代数的方法理解，原本$x_{103} = 11$，轴变换后$x_{310} = 11$或者$(1,0,3) = x_{103}$，变换后$(3,1,0) = x_{103}$。

```
1   arr = np.arange(15).reshape((3, 5))
2   arr.T
3
4   arr = np.array([[0, 1, 0], [1, 2, -2], [6, 3, 2], [-1, 0, -1], [1, 0, 1]])
5   np.dot(arr.T, arr)
```

```
6   arr.T @ arr
7
8   arr.swapaxes(0, 1)
9
10  arr = np.arange(16).reshape((2, 2, 4))
11  arr.transpose((2, 0, 1))
```

1.2   Pseudorandom Number Generation(伪随机数生成)

1. The `numpy.random` module supplements the built-in Python random module with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions.

2. Python's built-in `random` module, by contrast, samples only one value at a time. As you can see from this benchmark, `numpy.random` is well over an order of magnitude faster for generating very large samples(要快一个数量级).

3. These random numbers are not truly random (rather, pseudorandom) but instead are generated by a configurable random number generator that determines deterministically what values are created. Functions like `numpy.random.standard_normal` use the numpy.random module's default random number generator, but your code can be configured to use an explicit generator.

4. The seed argument is what determines the initial state of the generator, and the state changes each time the rng object is used to generate data. The generator object rng is also isolated from other code which might use the numpy.random module. 和随机数种子不一样，每次还是会产生不同的随机数，仅仅是修改一些随机数生成器的配置文件。

```
1   samples = np.random.standard_normal(size=(4, 4))
2   samples
3
4   from random import normalvariate
5   N = 1_000_000
6
7   %timeit samples = [normalvariate(0, 1) for _ in range(N)]
8
9   %timeit np.random.standard_normal(N)
10
11  rng = np.random.default_rng(seed=42)
12  data = rng.standard_normal((2, 3))
13
```

Table 1.3: NumPy random number generator methods

Methods	Description
permutation	Return a random permutation of a sequence, or return a permuted range
shuffle	Randomly permute a sequence in place(没有任何的返回None，需要接收一个可迭代对象，这个对象将被随机打乱)
uniform	Draw samples from a uniform distribution
randint	Draw random integers from a given low-to-high range
standard_normal	Draw samples from a normal distribution with mean 0 and standard deviation 1
binomial	Draw samples from a binomial distribution
normal	Draw samples from a normal (Gaussian) distribution
beta	Draw samples from a beta distribution
chisquare	Draw samples from a chi-square distribution
gamma	Draw samples from a gamma distribution

```
14  type(rng)
15  # numpy.random._generator.Generator
```

See Table 1.3 for a partial list of methods available on random generator objects like rng(random number generator).

1.3 Universal Functions: Fast Element-Wise Array Functions

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

1. Many ufuncs are simple element-wise transformations, like numpy.sqrt or numpy.exp. These are referred to as *unary*(一元的) ufuncs.

2. Others, such as numpy.add or numpy.maximum, take two arrays (thus, binary ufuncs) and return a single array as the result.

3. While not common, a ufunc can return multiple arrays. numpy.modf is one example: a vectorized version of the built-in Python math.modf, it returns the fractional and integral parts of a floating-point array

4. Ufuncs accept an optional out argument that allows them to assign their results into an existing array rather than create a new one

```
1  arr = np.arange(10)
2  np.sqrt(arr)
3  np.exp(arr)
4
```

```python
x = rng.standard_normal(8)
y = rng.standard_normal(8)

np.maximum(x, y)

arr = rng.standard_normal(7) * 5
remainder, whole_part = np.modf(arr)
remainder
whole_part

arr
# 7 * 1
print(arr)
out = np.zeros_like(arr)
out = np.add(arr, 1)

np.add(arr, 1, out=out)
out
```

See Table 1.4 and Table 1.4 for a listing of some of NumPy's ufuncs. New ufuncs continue to be added to NumPy, so consulting the online NumPy documentation is the best way to get a comprehensive listing and stay up to date.

1.4   Array-Oriented Programming with Arrays

Using NumPy arrays enables you to express many kinds of data processing tasks as concise array expressions that might otherwise require writing loops. This practice of replacing explicit loops with array expressions is referred to by some people as vectorization. In general, vectorized array operations will usually be significantly faster than their pure Python equivalents, with the biggest impact in any kind of numerical computations.

The numpy.meshgrid function takes two one-dimensional arrays and produces two two-dimensional matrices corresponding to all pairs of (x, y) in the two arrays:

```python
points = np.arange(-5, 5, .01)
xs, ys = np.meshgrid(points, points)

z = np.sqrt(xs ** 2 + ys ** 2)

import matplotlib.pyplot as plt
plt.imshow(z, cmap=plt.cm.gray, extent=[-5, 5, -5, 5])
```

Table 1.4: Some unary universal functions

Function	Description
abs, fabs	Compute the absolute value element-wise for integer, floating-point, or complex values
sqrt	Compute the square root of each element (equivalent to arr ** 0.5)
square	Compute the square of each element (equivalent to arr ** 2)
exp	Compute the exponent e^x of each element
log, log10 log2, log1p	Natural logarithm (base e), log base 10, log base 2, and log(1 + x), respectively
sign	Compute the sign of each element: 1 (positive), 0 (zero), or –1 (negative)
ceil	Compute the ceiling of each element (i.e., the smallest integer greater than or equal to that number)
floor	Compute the floor of each element (i.e., the largest integer less than or equal to each element)
rint	Round elements to the nearest integer, preserving the dtype
modf	Return fractional and integral parts of array as separate arrays
isnan	Return Boolean array indicating whether each value is NaN (Not a Number)
isfinite, isinf	Return Boolean array indicating whether each element is finite (non-inf, non-NaN) or infinite, respectively
cos, cosh, sin sinh, tan, tanh	Regular and hyperbolic trigonometric functions
arccos, arccosh arcsin, arcsinh arctan, arctanh	Inverse trigonometric functions
logical_not	Compute truth value of not x element-wise (equivalent to ˜arr)

Table 1.5: Some unary universal functions

Function	Description
`add`	Add corresponding elements in arrays
`subtract`	Subtract elements in second array from first array
`multiply`	Multiply array elements
`divide`, `floor_divide`	Divide or floor divide (truncating the remainder)
`power`	Raise elements in first array to powers indicated in second array
`maximum`, `fmax`	Element-wise maximum; fmax ignores NaN
`minimum`, `fmin`	Element-wise minimum; fmin ignores NaN
`mod`	Element-wise modulus (remainder of division)
`copysign`	Copy sign of values in second argument to values in first argument
`greater`, `greater_equal`, `less`,	Perform element-wise comparison, yielding Boolean array (equivalent to infix operators $>, >=, <, <=, ==, !=$)
`less_equal`, `equal`, `not_equal`	
`logical_and`	Compute element-wise truth value of AND (&) logical operation
`logical_or`	Compute element-wise truth value of OR (|) logical operation
`logical_xor`	Compute element-wise truth value of XOR (^) logical operation

```
8   plt.colorbar()
9   plt.title('Image plot of $\sqrt{x^2 + y^2}$ for a grid of values')
10  save_fig("Plot of function evaluated on a grid")
```

> **Notes**
>
> The term *vectorization* is used to describe some other computer science concepts, but in this book I use it to describe operations on whole arrays of data at once rather than going value by value using a Python for loop.

1.4.1   Expressing Conditional Logic as Array Operations

The `numpy.where` function is a vectorized version of the ternary expression `x if condition else y`.

```
1   xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
2   yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
3   cond = np.array([True, False, True, True, False])
```

Suppose we wanted to take a value from xarr whenever the corresponding value in cond is True, and otherwise take the value from yarr.
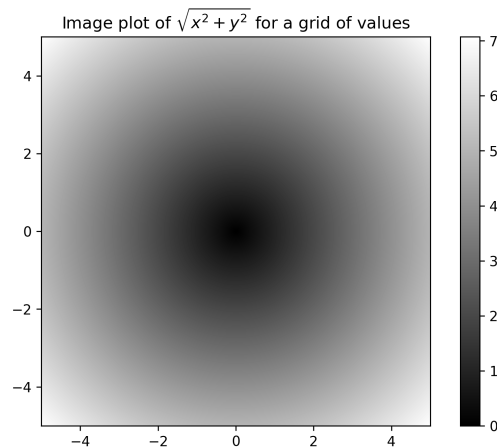
Figure 1.3: Plot of function evaluated on a grid

```
result = [(x if c else y) for x, y, c in zip(xarr, yarr, cond)]
result
# [1.1, 2.2, 1.3, 1.4, 2.5]
```

This has multiple problems. First, it will not be very fast for large arrays (because all the work is being done in interpreted Python code). Second, it will not work with multidimensional arrays. With numpy.where you can do this with a single function call:

```
result = np.where(cond, xarr, yarr)
```

The second and third arguments to numpy.where don't need to be arrays; one or both of them can be scalars. A typical use of where in data analysis is to produce a new array of values based on another array. You can combine scalars and arrays when using numpy.where.

```
arr = rng.standard_normal((4, 4))
np.where(arr > 0, 2, -2)


np.where(arr > 0, 2, arr)
```

1.4.2 Mathematical and Statistical Methods

A set of mathematical functions that compute statistics about an entire array or about the data along an axis are accessible as methods of the array class. You can use aggregations (sometimes called reductions) like sum, mean, and std (standard deviation) either by calling the array instance method or using the top-level NumPy function. When you use the NumPy function, like numpy.sum, you have to pass the array you want to aggregate as the first argument.

Table 1.6: Basic array statistical methods

Methods	Description
sum	Sum of all the elements in the array or along an axis; zero-length arrays have sum 0
mean	Arithmetic mean; invalid (returns NaN) on zero-length arrays
std, var	Standard deviation and variance, respectively
min, max	Minimum and maximum
argmin, argmax	Indices of minimum and maximum elements, respectively
cumsum	Cumulative sum of elements starting from 0
cumprod	Cumulative product of elements starting from 1

1. Functions like mean and sum take an optional axis argument that computes the statistic over the given axis, resulting in an array with one less dimension. `arr.mean(axis=1)` means "compute mean across the columns," where `arr.sum(axis=0)` means "compute sum down the rows."

2. Other methods like cumsum and cumprod do not aggregate, instead producing an array of the intermediate results. In multidimensional arrays, accumulation functions like cumsum return an array of the same size but with the partial aggregates computed along the indicated axis according to each lower dimensional slice.

```
arr = rng.standard_normal((5, 4))
arr.mean()
np.mean(arr)
arr.sum()

arr.mean(axis=1)
# compute mean across the columns
arr.mean(axis=0)
# compute sum down the rows

arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])
arr.cumsum()

arr2d = np.arange(9).reshape(-1, 3)
arr2d.cumsum(axis=0)
arr2d.cumsum(axis=1)
```

See Table 1.6 for a full listing.

1.4.3   Methods for Boolean Arrays

1. Boolean values are coerced to 1 (True) and 0 (False) in the preceding methods.

2. Two additional methods, any and all, are useful especially for Boolean arrays. any tests whether one or more values in an array is True(存在), while all checks if every value is True(任意).

3. These methods also work with non-Boolean arrays, where nonzero elements are treated as True.

```python
arr = rng.standard_normal(100)
(arr > 0).sum()
(arr < 0).sum()

bools = np.array([False, False, True, False])
bools.any()
# True
bools.all()
# False

arr = np.array([0, 1, 1, 2, 2, 4])
arr.any()
# True
arr.all()
# False
```

1.4.4   Sorting

1. Like Python's built-in list type, NumPy arrays can be sorted in place with the sort method.

2. You can sort each one-dimensional section of values in a multidimensional array in place along an axis by passing the axis number to sort.

3. The top-level method numpy.sort returns a sorted copy of an array (like the Python built-in function sorted) instead of modifying the array in place.

```python
arr = rng.standard_normal(5)
arr.sort()
arr

arr2d = rng.standard_normal((5, 3))
arr2d
arr2d.sort(axis=0)
arr2d
arr2d.sort(axis=1)
```

```
10    arr2d

11

12    arr2 = np.arange(10)
13    np.random.shuffle(arr2)

14

15    sorted_arr = np.sort(arr2)
16    sorted_arr

17

18    (sorted_arr == arr2).all()
```

For more details on using NumPy's sorting methods, and more advanced techniques like indirect sorts, see Appendix A.

1.4.5   Unique and Other Set Logic

NumPy has some basic set operations for one-dimensional ndarrays.

1. A commonly used one is numpy.unique, which returns the sorted unique values in an array. In many cases, the NumPy version is faster and returns a NumPy array rather than a Python list.

2. Another function, numpy.in1d, tests membership of the values in one array in another, returning a Boolean array

```
1     names = np.array(["Bob", "Will", "Joe", "Bob", "Will", "Joe", "Joe"])
2     np.unique(names)
3     # array(['Bob', 'Joe', 'Will'], dtype='<U4')

4

5     ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])
6     np.unique(ints)
7     # array([1, 2, 3, 4])

8

9     # Contrast numpy.unique with the pure Python alternative
10    sorted(set(names))
11    # ['Bob', 'Joe', 'Will']

12

13    values = np.array([6, 0, 0, 3, 2, 5, 6])
14    np.in1d(values, [2, 3, 6])
15    # array([ True, False, False,  True,  True, False,  True])
```

See Table 1.7 for a listing of array set operations in NumPy.

Table 1.7: Array set operations

Methods	Description
`unique(x)`	Compute the sorted, unique elements in x
`intersect1d(x, y)`	Compute the sorted, common elements in x and y
`union1d(x, y)`	Compute the sorted union of elements
`unique(x)`	Compute the sorted, unique elements in x
`intersect1d(x, y)`	Compute the sorted, common elements in x and y
`union1d(x, y)`	Compute the sorted union of elements

1.5 File Input and Output with Arrays

NumPy is able to save and load data to and from disk in some text or binary formats.

1.6 Linear Algebra

1.7 Example: Random Walks

1.8 Conclusion

Chapter 2

Appendix A