# Python Advanced

## NumPy Basics: Arrays and Vectorized Computation

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 arith‐ metic operations and flexible broadcasting capabilities.
* Mathematical functions for fast operations on entire arrays of data without hav‐ ing 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. Since NumPy is a large topic, I will cover many advanced NumPy features like broadcasting in more depth later (see Appendix A).

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 applica‐ tion)

While NumPy provides a computational foundation for general numerical data pro‐ cessing, 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.

Array-oriented computing in Python traces its roots back to 1995, when Jim Hugunin created the Numeric library. Over the next 10 years, many scientific programming communities began doing array programming in Python, but the library ecosystem had become fragmented in the early 2000s. In 2005, Travis Oliphant was able to forge the NumPy project from the then Numeric and Numarray projects to bring the community together around a sin‐ gle array computing framework.

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 lan‐ guage 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:

import numpy as np

my\_arr = np.arange(1000000)

my\_list = list(range(1000000))

Now let’s multiply each sequence by 2:

In [10]: %time for \_ in range(10): my\_arr2 = my\_arr \* 2 CPU times: user 20 ms, sys: 50 ms, total: 70 ms

**Wall time: 72.4 ms**

In [11]: %time for \_ in range(10): my\_list2 = [x \* 2 for x in my\_list] CPU times: user 760 ms, sys: 290 ms, total: 1.05 s

**Wall time: 1.05 s**

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

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:

import numpy as np

# Generate some random data

data = np.random.randn(2, 3)

data

*array([[-0.2047, 0.4789, -0.5194],*

*[-0.5557, 1.9658, 1.3934]])*

I then write mathematical operations with data:

data \* 10

*array([[ -2.0471, 4.7894, -5.1944],*

*[ -5.5573, 19.6578, 13.9341]])*

data + data

*array([[-0.4094, 0.9579, -1.0389],*

*[-1.1115, 3.9316, 2.7868]])*

In the first example, all of the elements have been multiplied by 10. In the second, the corresponding values in each “cell” in the array have been added to each other.

In this chapter and throughout the book, I use the standard NumPy convention of always using import numpy as np. You are, of course, welcome 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 func‐ tions whose names conflict with built-in Python functions (like min and max).

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:

data.shape

*(2, 3)*

data.dtype

*dtype('float64')*

This chapter will introduce you to the basics of using NumPy arrays, and should be sufficient for following along with the rest of the book. While it’s not necessary to have a deep understanding of NumPy for many data analytical applications, becom‐ ing proficient in array-oriented programming and thinking is a key step along the way to becoming a scientific Python guru.

Whenever you see “array,” “NumPy array,” or “ndarray” in the text, with few exceptions they all refer to the same thing: the ndarray object.

#### 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 con‐ taining the passed data. For example, a list is a good candidate for conversion:

data1 = [6, 7.5, 8, 0, 1]

arr1 = np.array(data1)

arr1

*array([ 6. , 7.5, 8. , 0. , 1. ])*

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

*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:

arr2.**ndim**

*2*

arr2.**shape**

*(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:

arr1.**dtype**

*dtype('float64')*

arr2.**dtype**

*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 0s or 1s, respectively, with a given length or shape. empty creates an array without initializing its values to any par‐ ticular value. To create a higher dimensional array with these methods, pass a tuple for the shape:

np.**zeros**(10)

*array([ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])*

np.**zeros**((3, 6)) Out[30]:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *array([[* | *0.,* | *0.,* | *0.,* | *0.,* | *0.,* | *0.],* |
| *[* | *0.,* | *0.,* | *0.,* | *0.,* | *0.,* | *0.],* |
| *[* | *0.,* | *0.,* | *0.,* | *0.,* | *0.,* | *0.]])* |

np.**empty**((2, 3, 2))

*array([[[ 0., 0.],*

*[ 0., 0.],*

*[ 0., 0.]],*

*[[ 0., 0.],*

*[ 0., 0.],*

*[ 0., 0.]]])*

It’s not safe to assume that **np.empty** will return an array of all zeros. In some cases, it may return uninitialized “garbage” values.

**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])*

See [Table 4-1](#_bookmark0) 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).

Table 4-1. Array creation functions

|  |  |
| --- | --- |
| **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 |
| arrange | Like the built-in range but returns an ndarray instead of a list |
| ones | 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. |
| ones\_like zeros | Like ones and ones\_like but producing arrays of 0s instead |
| zeros\_like empty | Create new arrays by allocating new memory, but do not populate with any values like  ones and zeros |
| empty\_like full | Produce an array of the given shape and dtype with all values set to the indicated “fill value” |
| full\_like | 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) |

#### 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)

arr1.dtype

dtype('float64')

arr2.dtype

dtype('int32')

dtypes are a source of NumPy’s flexibility for interacting with data coming from other systems. In most cases they provide a mapping directly onto an underlying disk or memory representation, which makes it easy to read and write binary streams of data to disk and also to connect to code written in a low-level language like C or Fortran. The numerical dtypes are named the same way: a type name, like float or int, fol‐ lowed by a number indicating the number of bits per element. A standard double- precision floating-point value (what’s used under the hood in Python’s float object) takes up 8 bytes or 64 bits. Thus, this type is known in NumPy as float64. See [Table 4-2](#_bookmark1) for a full listing of NumPy’s supported data types.

Don’t worry about memorizing the NumPy dtypes, 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 are stored in memory and on disk, especially large datasets, it is good to know that you have control over the storage type.

Table 4-2. 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, | c8, c16,  c32 | Complex numbers represented by two 32, 64, or 128 floats, respectively |
| complex256 |  |  |
| 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 a string dtype with length 10, use 'S10' |
| unicode\_ | U | Fixed-length Unicode type (number of bytes platform specific); same specification semantics as string\_ (e.g., 'U10') |

You can explicitly convert or cast an array from one dtype to another using ndarray’s astype method:

arr = np.**array**([1, 2, 3, 4, 5])

arr.**dtype**

*Out: dtype('int64')*

float\_arr = arr.**astype**(np.float64)

float\_arr.dtype

*Out: dtype('float64')*

In this example, integers were cast to floating point. If I cast some floating-point numbers to be of integer dtype, the decimal part will be truncated:

arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

arr

*Out: array([ 3.7, -1.2, -2.6, 0.5, 12.9, 10.1])*

arr.astype(np.int32)

*Out: array([ 3, -1, -2, 0, 12, 10], dtype=int32)*

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)

*Out: array([ 1.25, -9.6 , 42. ])*

It’s important to 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 behav‐ ior 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. Here I was a bit lazy and wrote float instead of np.float64; NumPy aliases the Python types to its own equivalent data dtypes.

You can also use another array’s dtype attribute:

int\_array = np.arange(10)

calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)

int\_array.astype(calibers.dtype)

*Out: 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:

empty\_uint32 = np.empty(8, dtype='u4')

empty\_uint32

*array([ 0, 1075314688, 0, 1075707904, 0,*

*1075838976, 0, 1072693248], dtype=uint32)*

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

#### 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 applies the operation element-wise:

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

arr

*array([[ 1., 2., 3.], [ 4., 5., 6.]])*

arr \* arr

*array([[ 1., 4., 9.], [ 16., 25., 36.]])*

arr - arr

array([[ 0., 0., 0.], [ 0., 0., 0.]])

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

1 / arr

array([[ 1. , 0.5 , 0.3333], [ 0.25 , 0.2 , 0.1667]])

arr \*\* 0.5

array( [[ 1. , 1.4142, 1.7321], [ 2. , 2.2361, 2.4495]] )

Comparisons between arrays of the same size yield boolean arrays:

arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])

arr2

*array([[ 0., 4., 1.], [ 7., 2., 12.]])*

*arr2 > arr*

*array([[False, True, False], [ True, False, True]], dtype=bool)*

Operations between differently sized arrays is called broadcasting and will be dis‐ cussed in more detail in Appendix A. Having a deep understanding of broadcasting is not necessary for most of this book.

#### Basic Indexing and Slicing

NumPy array indexing is a rich topic, as there are many ways you may want to select a subset of your data or individual elements. One-dimensional arrays are simple; on the surface they act similarly to Python lists:

arr = **np.arange(**10**)**

arr

*array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])*

**arr**[**5**]

*5*

**arr**[**5:8**]

*array([5, 6, 7])*

**arr**[**5:8**] = 12

**arr**

*array([ 0, 1, 2, 3, 4, 12, 12, 12, 8, 9])*

As you can see, if you assign a scalar value to a slice, as in arr[5:8] = 12, the value is propagated (or broadcasted henceforth) to the entire selection. An important first dis‐ tinction 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.

To give an example of this, I first create a slice of arr:

arr\_slice = arr[5:8]

**arr\_slice**

*array([12, 12, 12])*

Now, when I change values in arr\_slice, the mutations are reflected in the original array arr:

arr\_slice[1] = 12345

**arr**

*array([ 0, 1, 2, 3, 4, 12, 12345, 12, 8, 9])*

The “bare” slice **[:]** will assign to all values in an array:

arr\_slice**[:]** = 64

**arr**

*array([ 0, 1, 2, 3, 4, 64, 64, 64, 8, 9])*

If you are new to NumPy, you might be surprised by this, especially if you have used other array programming languages that copy data more eagerly. 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.

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()**.

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:

In [72]: arr2d = **np**.**array(**[[1, 2, 3], [4, 5, 6], [7, 8, 9]]**)**

In [73]: arr2d**[2]**

*array([7, 8, 9])*

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. So these are equivalent:

arr2d[0][2]

*3*

arr2d[0, 2]

*3*

See [Figure 4-1](#_bookmark2) 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.”

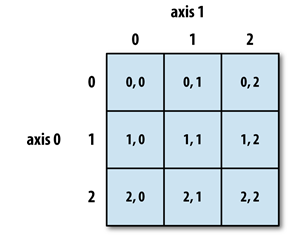


Figure 4-1. **Indexing elements in a NumPy array**

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. So in the 2 × 2 × 3 array arr3d:

In [76]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

In [77]: arr3d Out[77]:

array([[[ 1, 2, 3],

[ 4, 5, 6]],

[[ 7, 8, 9],

[10, 11, 12]]

])

**arr3d[0] is a 2 × 3 array:**

arr3d[0]

array([ [1, 2, 3],

[4, 5, 6] ])

Both scalar values and arrays can be assigned to arr3d[0]:

old\_values = arr3d[0].copy()

arr3d[0] = 42

arr3d

*array(*

*[[[42, 42, 42],*

*[42, 42, 42]],*

*[[ 7, 8, 9],*

*[10, 11, 12]]]*

*)*

arr3d[0] = old\_values

**arr3d**

*array( [[[ 1, 2, 3],*

*[ 4, 5, 6]],*

*[[ 7, 8, 9],*

*[10, 11, 12]]])*

Similarly, **arr3d[1, 0]** gives you all of the values whose indices start with (1, 0), forming a 1-dimensional array:

arr3d[1, 0]

*array([7, 8, 9])*

This expression is the same as though we had indexed in two steps:

**x = arr3d[1]**

**x**

*array([[ 7, 8, 9],*

*[10, 11, 12]])*

**x[0]**

*array([7, 8, 9])*

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

##### Indexing with slices

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

arr

*array([ 0, 1, 2, 3, 4, 64, 64, 64, 8, 9])*

arr[1:6]

*array([ 1, 2, 3, 4, 64])*

Consider the two-dimensional array from before, arr2d. Slicing this array is a bit different:

arr2d

*array([[1, 2, 3],*

*[4, 5, 6],*

*[7, 8, 9]])*

arr2d[:2]

Out[91]:

array([[1, 2, 3],

[4, 5, 6]])

As you can see, it has sliced along axis 0, the first axis. A slice, therefore, selects a range of elements along an axis. It can be helpful to read the expression arr2d[:2] as “select the first two rows of arr2d.”

You can pass multiple slices just like you can pass multiple indexes:

arr2d[:2, 1:]

*array([[2, 3], [5, 6]])*

When slicing like this, you always obtain array views of the same number of dimen‐ sions. By mixing integer indexes and slices, you get lower dimensional slices.

For example, I can select the second row but only the first two columns like so:

arr2d[1, :2]

*array([4, 5])*

Similarly, I can select the third column but only the first two rows like so:

arr2d[:2, 2]

*array([3, 6])*

See [Figure 4-2](#_bookmark3) for an illustration. Note that a colon by itself means to take the entire axis, so you can slice only higher dimensional axes by doing:

arr2d[:, :1]

*array([[1], [4], [7]])*

Of course, assigning to a slice expression assigns to the whole selection:

arr2d[:2, 1:] = 0

arr2d

*array([[1, 0, 0], [4, 0, 0], [7, 8, 9]])*

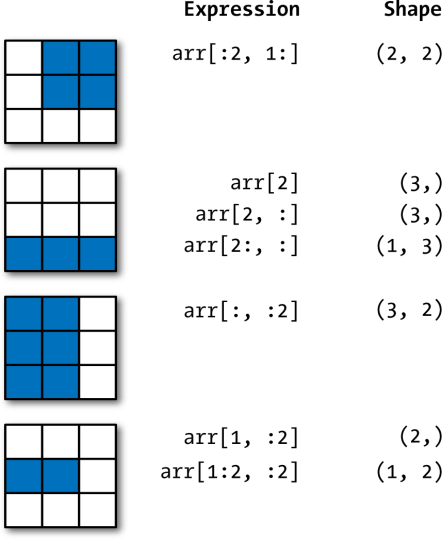


Figure 4-2. Two-dimensional array slicing

#### Boolean Indexing

Let’s consider an example where we have some data in an array and an array of names with duplicates. I’m going to use here the randn function in numpy.random to generate some random normally distributed data:

names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

data = np.random.randn(7, 4)

names

*array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')*

In [101]: data

*array([[ 0.0929, 0.2817, 0.769 , 1.2464],*

*[ 1.0072, -1.2962, 0.275 , 0.2289],*

*[ 1.3529, 0.8864, -2.0016, -0.3718],*

*[ 1.669 , -0.4386, -0.5397, 0.477 ],*

*[ 3.2489, -1.0212, -0.5771, 0.1241],*

*[ 0.3026, 0.5238, 0.0009, 1.3438],*

*[-0.7135, -0.8312, -2.3702, -1.8608]])*

Suppose each name corresponds to a row in the data array and we wanted to select all the rows with corresponding name 'Bob'. Like arithmetic operations, compari‐ sons (such as ==) with arrays are also vectorized. Thus, comparing names with the string 'Bob' yields a boolean array:

names == 'Bob'

*array([ True, False, False, True, False, False, False], dtype=bool)*

This boolean array can be passed when indexing the array:

data[names == 'Bob']

*array([[ 0.0929, 0.2817, 0.769 , 1.2464],*

*[ 1.669 , -0.4386, -0.5397, 0.477 ]])*

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; more on this later).

Boolean selection will not fail if the boolean array is not the correct length, so I recommend care when using this feature.

In these examples, I select from the rows where names == 'Bob' and index the col‐ umns, too:

data[names == 'Bob', 2:]

*array([[ 0.769 , 1.2464],*

*[-0.5397, 0.477 ]])*

data[names == 'Bob', 3]

*array([ 1.2464, 0.477 ])*

To select everything but 'Bob', you can either use != or negate the condition using ~:

names != 'Bob'

*array([False, True, True, False, True, True, True], dtype=bool)*

data[~(names == 'Bob')] Out[107]:

*array([[ 1.0072, -1.2962, 0.275 , 0.2289],*

*[ 1.3529, 0.8864, -2.0016, -0.3718],*

*[ 3.2489, -1.0212, -0.5771, 0.1241],*

*[ 0.3026, 0.5238, 0.0009, 1.3438],*

*[-0.7135, -0.8312, -2.3702, -1.8608]])*

The ~ operator can be useful when you want to invert a general condition:

cond = names == 'Bob'

data[~cond]

*array([[ 1.0072, -1.2962, 0.275 , 0.2289],*

*[ 1.3529, 0.8864, -2.0016, -0.3718],*

*[ 3.2489, -1.0212, -0.5771, 0.1241],*

*[ 0.3026, 0.5238, 0.0009, 1.3438],*

*[-0.7135, -0.8312, -2.3702, -1.8608]])*

Selecting two of the three names to combine multiple boolean conditions, use boolean arithmetic operators like & (and) and | (or):

mask = (names == 'Bob') | (names == 'Will')

mask

*array([ True, False, True, True, True, False, False], dtype=bool)*

data[mask]

*array([[ 0.0929, 0.2817, 0.769 , 1.2464],*

*[ 1.3529, 0.8864, -2.0016, -0.3718],*

*[ 1.669 , -0.4386, -0.5397, 0.477 ],*

*[ 3.2489, -1.0212, -0.5771, 0.1241]])*

Selecting data from an array by boolean indexing always creates a copy of the data, even if the returned array is unchanged.

The Python keywords and and or do not work with boolean arrays. Use & (and) and | (or) instead.

Setting values with boolean arrays works in a common-sense way. To set all of the negative values in data to 0 we need only do:

data[data < 0] = 0

*data*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *array([[* | *0.0929,* | *0.2817,* | *0.769 ,* | *1.2464],* |
| *[* | *1.0072,* | *0. ,* | *0.275 ,* | *0.2289],* |
| *[* | *1.3529,* | *0.8864,* | *0. ,* | *0. ],* |
| *[* | *1.669 ,* | *0. ,* | *0. ,* | *0.477 ],* |
| *[* | *3.2489,* | *0. ,* | *0. ,* | *0.1241],* |
| *[* | *0.3026,* | *0.5238,* | *0.0009,* | *1.3438],* |

*[ 0. , 0. , 0. , 0. ]])*

Setting whole rows or columns using a one-dimensional boolean array is also easy:

data[names != 'Joe'] = 7

data

*array([[ 7. , 7. , 7. , 7. ],*

*[ 1.0072, 0. , 0.275 , 0.2289],*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *[* | *7.* | *,* | *7.* | *,* | *7.* | *,* | *7.* | *],* |
| *[* | *7.* | *,* | *7.* | *,* | *7.* | *,* | *7.* | *],* |
| *[* | *7.* | *,* | *7.* | *,* | *7.* | *,* | *7.* | *],* |

*[ 0.3026, 0.5238, 0.0009, 1.3438],*

*[ 0. , 0. , 0. , 0. ]])*

As we will see later, these types of operations on two-dimensional data are convenient to do with pandas.

#### Fancy Indexing

Fancy indexing is a term adopted by NumPy to describe indexing using integer arrays. Suppose we had an 8 × 4 array:

arr = np.empty((8, 4))

for i in range(8):

.....: arr[i] = i

**arr**

*array([[ 0., 0., 0., 0.],*

*[ 1., 1., 1., 1.],*

*[ 2., 2., 2., 2.],*

*[ 3., 3., 3., 3.],*

*[ 4., 4., 4., 4.],*

*[ 5., 5., 5., 5.],*

*[ 6., 6., 6., 6.],*

*[ 7., 7., 7., 7.]])*

To select out a subset of the rows in a particular order, you can simply pass a list or ndarray of integers specifying the desired order:

arr[[4, 3, 0, 6]]

*array([[ 4., 4., 4., 4.],*

*[ 3., 3., 3., 3.],*

*[ 0., 0., 0., 0.],*

*[ 6., 6., 6., 6.]])*

Hopefully this code did what you expected! Using negative indices selects rows from the end:

arr[[-3, -5, -7]]

array([[ 5., 5., 5., 5.],

[ 3., 3., 3., 3.],

[ 1., 1., 1., 1.]])

Passing multiple index arrays does something slightly different; it selects a one- dimensional array of elements corresponding to each tuple of indices:

arr = **np.arange(32).reshape((8, 4))**

**arr**

*array([[ 0, 1, 2, 3],*

*[ 4, 5, 6, 7],*

*[ 8, 9, 10, 11],*

*[12, 13, 14, 15],*

*[16, 17, 18, 19],*

*[20, 21, 22, 23],*

*[24, 25, 26, 27],*

*[28, 29, 30, 31]*

*])*

**arr[**[1, 5, 7, 2], [0, 3, 1, 2]**]**

*array([ 4, 23, 29, 10])*

We’ll look at the reshape method in more detail in Appendix A.

Here the elements (1, 0), (5, 3), (7, 1), and (2, 2) were selected. Regardless of how many dimensions the array has (here, only 2), the result of fancy indexing is always one-dimensional.

The behavior of fancy indexing in this case is a bit different from what some users might have expected (myself included), which is the rectangular region formed by selecting a subset of the matrix’s rows and columns. Here is one way to get that:

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

*array([[ 4, 7, 5, 6],*

*[20, 23, 21, 22],*

*[28, 31, 29, 30],*

*[ 8, 11, 9, 10]])*

Keep in mind that fancy indexing, unlike slicing, always copies the data into a new array.

#### Transposing Arrays and Swapping Axes

Transposing is a special form of reshaping that similarly returns a view on the under‐ lying data without copying anything. Arrays have the transpose method and also the special T attribute:

arr = **np**.**arange**(15).**reshape((**3, 5**))**

**arr**

*array([[ 0, 1, 2, 3, 4],*

*[ 5, 6, 7, 8, 9],*

*[10, 11, 12, 13, 14]])*

**arr.T**

*array([[ 0, 5, 10],*

*[ 1, 6, 11],*

*[ 2, 7, 12],*

*[ 3, 8, 13],*

*[ 4, 9, 14]])*

When doing matrix computations, you may do this very often—for example, when computing the inner matrix product using np.dot:

arr = np.random.randn(6, 3)

arr

array([[-0.8608, 0.5601, -1.2659],

[ 0.1198, -1.0635, 0.3329],

[-2.3594, -0.1995, -1.542 ],

|  |  |
| --- | --- |
| [-0.9707, -1.307 , | 0.2863], |
| [ 0.378 , -0.7539, | 0.3313], |
| [ 1.3497, 0.0699, | 0.2467]]) |

np.dot(arr.T, arr)

*array([[ 9.2291, 0.9394, 4.948 ],*

*[ 0.9394, 3.7662, -1.3622],*

*[ 4.948 , -1.3622, 4.3437]])*

For higher dimensional arrays, transpose will accept a tuple of axis numbers to per‐ mute the axes (for extra mind bending):

arr = **np.arange(**16**)**.**reshape(**(2, 2, 4)**)**

**arr**

*array([[[ 0,*

*1, 2, 3],*

*[ 4, 5, 6, 7]],*

*[[ 8, 9, 10, 11],*

*[12, 13, 14, 15]]])*

In [134]: **arr**.**transpose(**(1, 0, 2)**)** Out[134]:

array([[[ 0, 1, 2, 3],

[ 8, 9, 10, 11]],

[[ 4, 5, 6, 7],

[12, 13, 14, 15]]])

Here, the axes have been reordered with the second axis first, the first axis second, and the last axis unchanged.

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 rear‐ range the data:

**arr**

*array([[[ 0, 1, 2, 3],*

*[ 4, 5, 6, 7]],*

*[[ 8, 9, 10, 11],*

*[12, 13, 14, 15]]])*

**arr.swapaxes(1, 2)**

*array([[[ 0, 4],*

*[ 1, 5],*

*[ 2, 6],*

*[ 3, 7]],*

*[[ 8, 12],*

*[ 9, 13],*

*[10, 14],*

*[11, 15]]])*

swapaxes similarly returns a view on the data without making a copy.

### Universal Functions: Fast Element-Wise Array Functions

A universal function, or ufunc, is a function that performs element-wise operations on data in ndarrays. You can think of them as fast vectorized wrappers for simple functions that take one or more scalar values and produce one or more scalar results.

Many ufuncs are simple element-wise transformations, like sqrt or exp:

arr = **np.arange(**10**)**

**arr**

*array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])*

**np**.**sqrt**(arr)

*array([ 0. , 1. , 1.4142, 1.7321, 2. , 2.2361, 2.4495, 2.6458, 2.8284, 3. ])*

**np.exp(**arr**)**

*array([ 1. , 2.7183, 7.3891, 20.0855, 54.5982, 148.4132, 403.4288, 1096.6332, 2980.958 , 8103.0839])*

These are referred to as unary ufuncs. Others, such as add or maximum, take two arrays (thus, binary ufuncs) and return a single array as the result:

x = **np.random.randn(**8**)**

y = **np.random.randn(**8**)**

**x**

*array([-0.0119, 1.0048, 1.3272, -0.9193, -1.5491, 0.0222, 0.7584, -0.6605])*

**y**

*array([ 0.8626, -0.01 , 0.05 , 0.6702, 0.853 , -0.9559, -0.0235, -2.3042])*

**np**.**maximum(**x, y**)** Out[145]:

*array([ 0.8626, 1.0048, 1.3272, 0.6702, 0.853 , 0.0222, 0.7584, -0.6605])*

Here, **numpy.maximum** computed the element-wise maximum of the elements in x and y.

While not common, a ufunc can return multiple arrays. modf is one example, a vec‐ torized version of the built-in Python divmod; it returns the fractional and integral parts of a floating-point array:

In [146]: arr = **np.random.randn(**7**)** \* 5

In [147]: **arr**

Out[147]: array([-3.2623, -6.0915, -6.663 , 5.3731, 3.6182, 3.45 , 5.0077])

In [148]: **remainder**, **whole\_part** = **np.modf(**arr**)** In [149]: remainder

Out[149]: array([-0.2623, -0.0915, -0.663 , 0.3731, 0.6182, 0.45 , 0.0077])

In [150]: **whole\_part**

Out[150]: array([-3., -6., -6., 5., 3., 3., 5.])

Ufuncs accept an optional out argument that allows them to operate in-place on arrays:

In [151]: **arr**

Out[151]: array([-3.2623, -6.0915, -6.663 , 5.3731, 3.6182, 3.45 , 5.0077])

In [152]: **np.sqrt(**arr**)**

Out[152]: array([ nan, nan, nan, 2.318 , 1.9022, 1.8574, 2.2378])

In [153]: **np**.**sqrt(**arr, arr**)**

*array([ nan, nan, nan, 2.318 , 1.9022, 1.8574, 2.2378])*

**arr**

*array([*

*nan,*

*nan,*

*nan, 2.318 , 1.9022, 1.8574, 2.2378])*

*array([ nan, nan, nan, 2.318 , 1.9022, 1.8574, 2.2378])*

**arr**

*array([*

*nan,*

*nan,*

*nan, 2.318 , 1.9022, 1.8574, 2.2378])*

See Tables [4-3](#_bookmark4) and [4-4](#_bookmark5) for a listing of available ufuncs.

**Table 4-3. Unary ufuncs**

**Functions Descriptions**

**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 ex of each element

**log**, **log10**, Natural logarithm (base e), log base 10, log base 2, and log(1 + x), respectively

**log2**, **log1p**

**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 a separate array

**isnan** Return boolean array indicating whether each value is NaN (Not a Number)

**isfinite**, Return boolean array indicating whether each element is finite (non-inf, non-NaN) or infinite, respectively

**isinf**

**cos, cosh, sin, sinh, tan, tanh** Regular and hyperbolic trigonometric functions

**arccos**, **arccosh**, Inverse trigonometric functions

**arcsin**, **arcsinh**, **arctan**, **arctanh**

**logical\_not** Compute truth value of not x element-wise (equivalent to ~arr).

**Table 4-4. Binary universal functions**

**Functions Descriptions**

**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, less\_equal, equal, not\_equal**

**logical\_and, logical\_or, logical\_xor**

Perform element-wise comparison, yielding boolean array (equivalent to infix operators >, >=, <, <=, ==, !=)

Compute element-wise truth value of logical operation (equivalent to infix operators & |, ^)

### 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 commonly referred to as vectoriza‐ tion. In general, vectorized array operations will often be one or two (or more) orders of magnitude faster than their pure Python equivalents, with the biggest impact in any kind of numerical computations. Later, in Appendix A, I explain broadcasting, a powerful method for vectorizing computations.

As a simple example, suppose we wished to evaluate the function sqrt(x^2 + y^2) across a regular grid of values. The np.meshgrid function takes two 1D arrays and produces two 2D matrices corresponding to all pairs of (x, y) in the two arrays:

points = np.arange(-5, 5, 0.01) # 1000 equally spaced points

xs, ys = np.meshgrid(points, points)

ys

array([[-5. , -5. , -5. , ..., -5. , -5. , -5. ],

[-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99],

[-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98],

...,

[ 4.97, 4.97, 4.97, ..., 4.97, 4.97, 4.97],

[ 4.98, 4.98, 4.98, ..., 4.98, 4.98, 4.98],

[ 4.99, 4.99, 4.99, ..., 4.99, 4.99, 4.99]])

Now, evaluating the function is a matter of writing the same expression you would write with two points:

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

**z**

array([[ 7.0711, 7.064 , 7.0569, ..., 7.0499, 7.0569, 7.064 ],

[ 7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569],

[ 7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499],

...,

[ 7.0499, 7.0428, 7.0357, ..., 7.0286, 7.0357, 7.0428],

[ 7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499],

[ 7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569]])

As a preview of Chapter 9, I use matplotlib to create visualizations of this two- dimensional array:

import matplotlib.pyplot as plt

plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()

*<matplotlib.colorbar.Colorbar at 0x7f715e3fa630>*

plt.title("Image plot of $\sqrt{x^2 + y^2}$ for a grid of values")

*<matplotlib.text.Text at 0x7f715d2de748>*

See [Figure 4-3](#_bookmark6). Here I used the matplotlib function imshow to create an image plot from a two-dimensional array of function values.

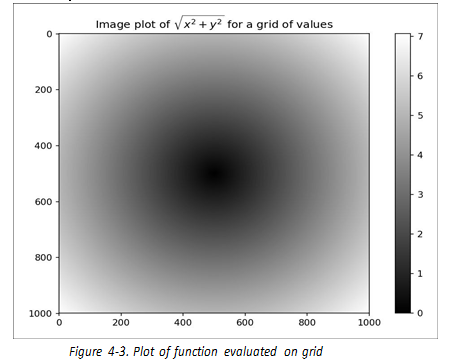


Figure 4-3. Plot of function evaluated on grid

#### Expressing Conditional Logic as Array Operations

The numpy.where function is a vectorized version of the ternary expression x if con dition else y. Suppose we had a boolean array and two arrays of values:

xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])

yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])

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. A list comprehension doing this might look like:

result = [(x if c else y) for x, y, c in zip(xarr, yarr, cond)]

result

*[1.1000000000000001, 2.2000000000000002, 1.3, 1.3999999999999999, 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 np.where you can write this very concisely:

In [170]: result = np.where(cond, xarr, yarr)

In [171]: result

Out[171]: array([ 1.1, 2.2, 1.3, 1.4, 2.5])

The second and third arguments to np.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. Suppose you had a matrix of randomly generated data and you wanted to replace all positive values with 2 and all negative values with

–2. This is very easy to do with np.where:

In [172]: arr = np.random.randn(4, 4)

In [173]: arr Out[173]:

array([[-0.5031, -0.6223, -0.9212, -0.7262],

[ 0.2229, 0.0513, -1.1577, 0.8167],

[ 0.4336, 1.0107, 1.8249, -0.9975],

[ 0.8506, -0.1316, 0.9124, 0.1882]])

In [174]: arr > 0 Out[174]:

array([[False, False, False, False], [ True, True, False, True], [ True, True, True, False],

[ True, False, True, True]], dtype=bool)

In [175]: np.where(arr > 0, 2, -2) Out[175]:

array([[-2, -2, -2, -2],

[ 2, 2, -2, 2],

[ 2, 2, 2, -2],

[ 2, -2, 2, 2]])

You can combine scalars and arrays when using np.where. For example, I can replace all positive values in arr with the constant 2 like so:

In [176]: np.where(arr > 0, 2, arr) # set only positive values to 2

Out[176]:

array([[-0.5031, -0.6223, -0.9212, -0.7262],

[ 2. , 2. , -1.1577, 2. ],

[ 2. , 2. , 2. , -0.9975],

[ 2. , -0.1316, 2. , 2. ]])

The arrays passed to np.where can be more than just equal-sized arrays or scalars.

#### 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 aggre‐ gations (often called reductions) like sum, mean, and std (standard deviation) either by calling the array instance method or using the top-level NumPy function.

Here I generate some normally distributed random data and compute some aggregate statistics:

In [177]: arr = np.random.randn(5, 4)

In [178]: arr Out[178]:

array([[ 2.1695, -0.1149, 2.0037, 0.0296],

[ 0.7953, 0.1181, -0.7485, 0.585 ],

[ 0.1527, -1.5657, -0.5625, -0.0327],

[-0.929 , -0.4826, -0.0363, 1.0954],

[ 0.9809, -0.5895, 1.5817, -0.5287]])

In [179]: arr.mean()

Out[179]: 0.19607051119998253

In [180]: np.mean(arr)

Out[180]: 0.19607051119998253

In [181]: arr.sum()

Out[181]: 3.9214102239996507

Functions like mean and sum take an optional axis argument that computes the statis‐ tic over the given axis, resulting in an array with one fewer dimension:

In [182]: arr.mean(axis=1)

Out[182]: array([ 1.022 , 0.1875, -0.502 , -0.0881, 0.3611])

In [183]: arr.sum(axis=0)

Out[183]: array([ 3.1693, -2.6345, 2.2381, 1.1486])

Here, arr.mean(1) means “compute mean across the columns” where arr.sum(0)

means “compute sum down the rows.”

Other methods like cumsum and cumprod do not aggregate, instead producing an array of the intermediate results:

In [184]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])

In [185]: arr.cumsum()

Out[185]: array([ 0, 1, 3, 6, 10, 15, 21, 28])

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 = **np.array(**[[0, 1, 2], [3, 4, 5], [6, 7, 8]]**)**

**arr**

array([[0, 1, 2],

[3, 4, 5],

[6, 7, 8]])

**arr.cumsum(axis=0)**

array([[ 0, 1, 2],

[ 3, 5, 7],

[ 9, 12, 15]])

**arr.cumprod(axis=1)**

array([[ 0, 0, 0],

[ 3, 12, 60],

[ 6, 42, 336]])

See [Table 4-5](#_bookmark7) for a full listing. We’ll see many examples of these methods in action in later chapters.

**Table 4-5. Basic array statistical methods**

sum Sum of all the elements in the array or along an axis; zero-length arrays have sum 0

mean Arithmetic mean; zero-length arrays have NaN mean

std, var Standard deviation & variance, respectively, with optional degrees of freedom adjustment

(default denominator n)

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

#### Methods for Boolean Arrays

Boolean values are coerced to 1 (True) and 0 (False) in the preceding methods. Thus, sum is often used as a means of counting True values in a boolean array:

In [190]: arr = np.random.randn(100)

In [191]: (arr > 0).sum() # Number of positive values

Out[191]: 42

There are two additional methods, any and all, useful especially for boolean arrays. any tests whether one or more values in an array is True, while all checks if every value is True:

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

bools.any() Out[193]: True

bools.all() Out[194]: False

These methods also work with non-boolean arrays, where non-zero elements evalu‐ ate to True.

#### Sorting

Like Python’s built-in list type, NumPy arrays can be sorted in-place with the sort method:

In [195]: arr = np.random.randn(6)

In [196]: arr

Out[196]: array([ 0.6095, -0.4938, 1.24 , -0.1357, 1.43 , -0.8469])

In [197]: arr.sort()

In [198]: arr

Out[198]: array([-0.8469, -0.4938, -0.1357, 0.6095, 1.24 , 1.43 ])

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:

arr = np.random.randn(5, 3)

arr

*array([[ 0.6033, 1.2636, -0.2555],*

*[-0.4457, 0.4684, -0.9616],*

*[-1.8245, 0.6254, 1.0229],*

*[ 1.1074, 0.0909, -0.3501],*

*[ 0.218 , -0.8948, -1.7415]])*

arr.sort(1)

arr

*array([[-0.2555, 0.6033, 1.2636],*

*[-0.9616, -0.4457, 0.4684],*

*[-1.8245, 0.6254, 1.0229],*

*[-0.3501, 0.0909, 1.1074],*

*[-1.7415, -0.8948, 0.218 ]])*

The top-level method np.sort returns a sorted copy of an array instead of modifying the array in-place. A quick-and-dirty way to compute the quantiles of an array is to sort it and select the value at a particular rank:

large\_arr = np.random.randn(1000) In [204]: large\_arr.sort()

large\_arr[int(0.05 \* len(large\_arr))] # 5% quantile

*-1.5311513550102103*

For more details on using NumPy’s sorting methods, and more advanced techniques like indirect sorts, see Appendix A. Several other kinds of data manipulations related to sorting (e.g., sorting a table of data by one or more columns) can also be found in pandas.

#### Unique and Other Set Logic

NumPy has some basic set operations for one-dimensional ndarrays. A commonly used one is np.unique, which returns the sorted unique values in an array:

In [206]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [207]: np.unique(names) Out[207]:

*array(['Bob', 'Joe', 'Will'], dtype='<U4')*

In [208]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])

np.unique(ints)

*Output: array([1, 2, 3, 4])*

Contrast np.unique with the pure Python alternative:

sorted(set(names)) Out[210]: ['Bob', 'Joe', 'Will']

Another function, np.in1d, tests membership of the values in one array in another, returning a boolean array:

values = np.array([6, 0, 0, 3, 2, 5, 6])

np.in1d(values, [2, 3, 6])

*array([ True, False, False, True, True, False, True], dtype=bool)*

See Table 4-6 for a listing of set functions in NumPy.

Table 4-6. Array set operations

**Functions Descriptions**

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

in1d(x, y) Compute a boolean array indicating whether each element of x is contained in y setdiff1d(x, y) Set difference, elements in x that are not in y

setxor1d(x, y) Set symmetric differences; elements that are in either of the arrays, but not both

### File Input and Output with Arrays

NumPy is able to save and load data to and from disk either in text or binary format. In this section I only discuss NumPy’s built-in binary format, since most users will prefer pandas and other tools for loading text or tabular data (see [Chapter 6](#_bookmark21) for much more).

np.save and np.load are the two workhorse functions for efficiently saving and load‐ ing array data on disk. Arrays are saved by default in an uncompressed raw binary format with file extension .npy:

In [213]: arr = np.arange(10)

In [214]: np.save('some\_array', arr)

If the file path does not already end in .npy, the extension will be appended. The array on disk can then be loaded with np.load:

In [215]: np.load('some\_array.npy')

Out[215]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

You save multiple arrays in an uncompressed archive using np.savez and passing the arrays as keyword arguments:

In [216]: np.savez('array\_archive.npz', a=arr, b=arr)

When loading an .npz file, you get back a dict-like object that loads the individual arrays lazily:

In [217]: arch = np.load('array\_archive.npz')

In [218]: arch['b']

Out[218]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

If your data compresses well, you may wish to use numpy.savez\_compressed instead:

In [219]: np.savez\_compressed('arrays\_compressed.npz', a=arr, b=arr)

### Linear Algebra

Linear algebra, like matrix multiplication, decompositions, determinants, and other square matrix math, is an important part of any array library. Unlike some languages like MATLAB, multiplying two two-dimensional arrays with \* is an element-wise product instead of a matrix dot product. Thus, there is a function dot, both an array method and a function in the numpy namespace, for matrix multiplication:

In [223]: x = np.array([[1., 2., 3.], [4., 5., 6.]])

In [224]: y = np.array([[6., 23.], [-1, 7], [8, 9]])

In [225]: x

Out[225]:

array([[ 1., 2., 3.],

[ 4., 5., 6.]])

|  |  |
| --- | --- |
| In [226]: y  Out[226]:  array([[ 6., | 23.], |
| [ -1., | 7.], |
| [ 8., | 9.]]) |

In [227]: x.dot(y) Out[227]:

array([[ 28., 64.],

[ 67., 181.]])

x.dot(y) is equivalent to np.dot(x, y):

In [228]: np.dot(x, y) Out[228]:

array([[ 28., 64.],

[ 67., 181.]])

A matrix product between a two-dimensional array and a suitably sized one- dimensional array results in a one-dimensional array:

In [229]: np.dot(x, np.ones(3))

Out[229]: array([ 6., 15.])

The @ symbol (as of Python 3.5) also works as an infix operator that performs matrix multiplication:

In [230]: x @ np.ones(3) Out[230]: array([ 6., 15.])

numpy.linalg has a standard set of matrix decompositions and things like inverse and determinant. These are implemented under the hood via the same industry- standard linear algebra libraries used in other languages like MATLAB and R, such as BLAS, LAPACK, or possibly (depending on your NumPy build) the proprietary Intel MKL (Math Kernel Library):

from numpy.linalg import inv, qr In [232]: X = np.random.randn(5, 5)

mat = X.T.dot(X)

inv(mat)

*array([[ 933.1189, 871.8258, -1417.6902, -1460.4005, 1782.1391],*

*[ 871.8258, 815.3929, -1325.9965, -1365.9242, 1666.9347],*

*[-1417.6902, -1325.9965, 2158.4424, 2222.0191, -2711.6822],*

*[-1460.4005, -1365.9242, 2222.0191, 2289.0575, -2793.422 ],*

*[ 1782.1391, 1666.9347, -2711.6822, -2793.422 , 3409.5128]])*

mat.dot(inv(mat)) Out[235]:

*array([[ 1., 0., -0., -0., -0.],*

*[-0., 1., 0., 0., 0.],*

*[ 0., 0., 1., 0., 0.],*

*[-0., 0., 0., 1., -0.],*

*[-0., 0., 0., 0., 1.]])*

q, r = qr(mat)

r

*array([[-1.6914, 4.38 , 0.1757, 0.4075, -0.7838],*

*[ 0. , -2.6436, 0.1939, -3.072 , -1.0702],*

*[ 0. , 0. , -0.8138, 1.5414, 0.6155],*

*[ 0. , 0. , 0. , -2.6445, -2.1669],*

*[ 0. , 0. , 0. , 0. , 0.0002]])*

The expression X.T.dot(X) computes the dot product of X with its transpose X.T. See [Table 4-7](#_bookmark9) for a list of some of the most commonly used linear algebra functions.

Table 4-7. Commonly used numpy.linalg functions

**Function Description**

**diag** Return the diagonal (or off-diagonal) elements of a square matrix as a 1D array, or convert a 1D array into

a square matrix with zeros on the off-diagonal

**dot** Matrix multiplication

**trace** Compute the sum of the diagonal elements

**det** Compute the matrix determinant

**Function Description**

**eig** Compute the eigenvalues and eigenvectors of a square matrix

**inv** Compute the inverse of a square matrix

**pinv** Compute the Moore-Penrose pseudo-inverse of a matrix

**qr** Compute the QR decomposition

**svd** Compute the singular value decomposition (SVD)

**solve** Solve the linear system Ax = b for x, where A is a square matrix

### Pseudorandom Number Generation

The numpy.random module supplements the built-in Python random with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions. For example, you can get a 4 × 4 array of samples from the standard normal distribution using normal:

samples = np.random.normal(size=(4, 4))

**samples**

*array([[ 0.5732, 0.1933, 0.4429, 1.2796],*

*[ 0.575 , 0.4339, -0.7658, -1.237 ],*

*[-0.5367, 1.8545, -0.92 , -0.1082],*

*[ 0.1525, 0.9435, -1.0953, -0.144 ]])*

Python’s built-in random module, by contrast, only samples 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:

from random import normalvariate

N = 1000000

%timeit samples = [normalvariate(0, 1) for \_ in range(N)]

*1.77 s +- 126 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)*

%timeit np.random.normal(size=N)

*61.7 ms +- 1.32 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)*

We say that these are pseudorandom numbers because they are generated by an algo‐ rithm with deterministic behavior based on the seed of the random number genera‐ tor. You can change NumPy’s random number generation seed using np.random.seed:

In [244]: np.random.seed(1234)

The data generation functions in numpy.random use a global random seed. To avoid global state, you can use numpy.random.RandomState to create a random number generator isolated from others:

In [245]: rng = np.random.RandomState(1234)

In [246]: rng.randn(10) Out[246]:

array([ 0.4714, -1.191 , 1.4327, -0.3127, -0.7206, 0.8872, 0.8596,

-0.6365, 0.0157, -2.2427])

See [Table 4-8](#_bookmark10) for a partial list of functions available in numpy.random. I’ll give some examples of leveraging these functions’ ability to generate large arrays of samples all at once in the next section.

Table 4-8. Partial list of numpy.random functions

**Function Description**

**seed** Seed the random number generator permutation Return a random permutation of a sequence,

or return a permuted range

**shuffle** Randomly permute a sequence in-place

**rand** Draw samples from a uniform distribution

**randint** Draw random integers from a given low-to-high range

**randn** Draw samples from a normal distribution with mean 0 and standard deviation 1 (MATLAB-like interface)

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 uniform Draw samples from a uniform [0, 1) distribution

#### Example: Random Walks

The simulation of [random walks](https://en.wikipedia.org/wiki/Random_walk) provides an illustrative application of utilizing array operations. Let’s first consider a simple random walk starting at 0 with steps of 1 and –1 occurring with equal probability.

Here is a pure Python way to implement a single random walk with 1,000 steps using the built-in random module:

import random

.....: position = 0

.....: walk = [position]

.....: steps = 1000

.....: for i in range(steps):

.....: step = 1 if random.randint(0, 1) else -1

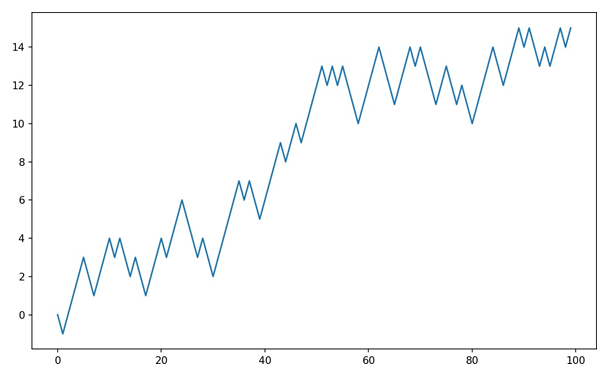
.....: position += step

.....: walk.append(position)

.....:

See [Figure 4-4](#_bookmark11) for an example plot of the first 100 values on one of these random walks:

plt.plot(walk[:100])



**Figure 4-4. A simple random walk**

You might make the observation that walk is simply the cumulative sum of the ran‐ dom steps and could be evaluated as an array expression. Thus, I use the np.random module to draw 1,000 coin flips at once, set these to 1 and –1, and compute the cumulative sum:

nsteps = 1000

draws = np.random.randint(0, 2, size=nsteps)

steps = np.where(draws > 0, 1, -1)

walk = steps.cumsum()

From this we can begin to extract statistics like the minimum and maximum value along the walk’s trajectory:

walk.min()

*-3*

walk.max()

*31*

A more complicated statistic is the first crossing time, the step at which the random walk reaches a particular value. Here we might want to know how long it took the random walk to get at least 10 steps away from the origin 0 in either direction. np.abs(walk) >= 10 gives us a boolean array indicating where the walk has reached or exceeded 10, but we want the index of the first 10 or –10. Turns out, we can com‐ pute this using argmax, which returns the first index of the maximum value in the boolean array (True is the maximum value):

(np.abs(walk) >= 10).argmax()

*37*

Note that using argmax here is not always efficient because it always makes a full scan of the array. In this special case, once a True is observed we know it to be the maxi‐ mum value.

##### Simulating Many Random Walks at Once

If your goal was to simulate many random walks, say 5,000 of them, you can generate all of the random walks with minor modifications to the preceding code. If passed a 2-tuple, the numpy.random functions will generate a two-dimensional array of draws, and we can compute the cumulative sum across the rows to compute all 5,000 ran‐ dom walks in one shot:

nwalks = 5000

nsteps = 1000

draws = np.random.randint(0, 2, size=(nwalks, nsteps)) # 0 or 1

steps = np.where(draws > 0, 1, -1)

walks = steps.cumsum(1)

walks

*array([[ 1, 0, 1, ..., 8, 7, 8],*

*[ 1, 0, -1, ..., 34, 33, 32],*

*[ 1, 0, -1, ..., 4, 5, 4],*

*...,*

*[ 1, 2, 1, ..., 24, 25, 26],*

*[ 1, 2, 3, ..., 14, 13, 14],*

*[ -1, -2, -3, ..., -24, -23, -22]])*

Now, we can compute the maximum and minimum values obtained over all of the walks:

walks.max()

138

walks.min()

-133

Out of these walks, let’s compute the minimum crossing time to 30 or –30. This is slightly tricky because not all 5,000 of them reach 30. We can check this using the any method:

hits30 = (np.abs(walks) >= 30).any(1)

hits30

*array([False, True, False, ..., False, True, False], dtype=bool)*

hits30.sum() # Number that hit 30 or -30

*3410*

We can use this boolean array to select out the rows of walks that actually cross the absolute 30 level and call argmax across axis 1 to get the crossing times:

crossing\_times = (np.abs(walks[hits30]) >= 30).argmax(1)

crossing\_times.mean()

*498.88973607038122*

Feel free to experiment with other distributions for the steps other than equal-sized coin flips. You need only use a different random number generation function, like normal to generate normally distributed steps with some mean and standard deviation:

steps = np.random.normal(loc=0, scale=0.25,

.....: size=(nwalks, nsteps))

### Conclusion

While much of the rest of the book will focus on building data wrangling skills with pandas, we will continue to work in a similar array-based style. In Appendix A, we will dig deeper into NumPy features to help you further develop your array comput‐ ing skills.

## Getting Started with Pandas

pandas will be a major tool of interest throughout much of the rest of the book. It contains data structures and data manipulation tools designed to make data cleaning and analysis fast and easy in Python. pandas is often used in tandem with numerical computing tools like NumPy and SciPy, analytical libraries like statsmodels and scikit-learn, and data visualization libraries like matplotlib. pandas adopts significant parts of NumPy’s idiomatic style of array-based computing, especially array-based functions and a preference for data processing without for loops.

While pandas adopts many coding idioms from NumPy, the biggest difference is that pandas is designed for working with tabular or heterogeneous data. NumPy, by con‐ trast, is best suited for working with homogeneous numerical array data.

Since becoming an open source project in 2010, pandas has matured into a quite large library that’s applicable in a broad set of real-world use cases. The developer community has grown to over 800 distinct contributors, who’ve been helping build the project as they’ve used it to solve their day-to-day data problems.

Throughout the rest of the book, I use the following import convention for pandas:

import pandas as pd

Thus, whenever you see pd. in code, it’s referring to pandas. You may also find it eas‐ ier to import Series and DataFrame into the local namespace since they are so fre‐ quently used:

from pandas import Series, DataFrame

### Introduction to pandas Data Structures

To get started with pandas, you will need to get comfortable with its two workhorse data structures: Series and DataFrame. While they are not a universal solution for every problem, they provide a solid, easy-to-use basis for most applications.

#### Series

A Series is a one-dimensional array-like object containing a sequence of values (of similar types to NumPy types) and an associated array of data labels, called its index. The simplest Series is formed from only an array of data:

obj = pd.Series([4, 7, -5, 3])

obj

0 4

1 7

2 -5

3 3

dtype: int64

The string representation of a Series displayed interactively shows the index on the left and the values on the right. Since we did not specify an index for the data, a default one consisting of the integers 0 through N - 1 (where N is the length of the data) is created. You can get the array representation and index object of the Series via its values and index attributes, respectively:

obj.values

*array([ 4, 7, -5, 3])*

obj.index # like range(4)

*RangeIndex(start=0, stop=4, step=1)*

Often it will be desirable to create a Series with an index identifying each data point with a label:

obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

obj2

d 4

b 7

a -5

c 3

dtype: int64

obj2.index

Index(['d', 'b', 'a', 'c'], dtype='object')

Compared with NumPy arrays, you can use labels in the index when selecting single values or a set of values:

obj2['a']

-5

obj2['d'] = 6

obj2[['c', 'a', 'd']]

c 3

a -5

d 6

dtype: int64

Here ['c', 'a', 'd'] is interpreted as a list of indices, even though it contains strings instead of integers.

Using NumPy functions or NumPy-like operations, such as filtering with a boolean array, scalar multiplication, or applying math functions, will preserve the index-value link:

obj2[obj2 > 0]

d 6

7

3

dtype: int64

obj2 \* 2

12

b 14

a -10

c 6

dtype: int64

np.exp(obj2)

d 403.428793

b 1096.633158

a 0.006738

c 20.085537

dtype: float64

Another way to think about a Series is as a fixed-length, ordered dict, as it is a mapping of index values to data values. It can be used in many contexts where you might use a dict:

'b' in obj2

True

'e' in obj2

False

Should you have data contained in a Python dict, you can create a Series from it by passing the dict:

sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

obj3 = pd.Series(sdata)

obj3

Ohio 35000

Oregon 16000

Texas 71000

Utah 5000

dtype: int64

When you are only passing a dict, the index in the resulting Series will have the dict’s keys in sorted order. You can override this by passing the dict keys in the order you want them to appear in the resulting Series:

states = ['California', 'Ohio', 'Oregon', 'Texas']

obj4 = pd.Series(sdata, index=states)

obj4

*California*

*NaN*

*Ohio 35000.0*

*Oregon 16000.0*

*Texas 71000.0*

*dtype: float64*

Here, three values found in sdata were placed in the appropriate locations, but since no value for 'California' was found, it appears as NaN (not a number), which is con‐ sidered in pandas to mark missing or NA values. Since 'Utah' was not included in states, it is excluded from the resulting object.

I will use the terms “missing” or “NA” interchangeably to refer to missing data. The

isnull and notnull functions in pandas should be used to detect missing data:

pd.isnull(obj4)

*California True*

*Ohio False*

*Oregon False*

*Texas False*

*dtype: bool*

pd.notnull(obj4)

California False

Ohio True

Oregon True

Texas True

dtype: bool

Series also has these as instance methods:

obj4.isnull()

*California True*

*Ohio False*

*Oregon False*

*Texas False dtype: bool*

I discuss working with missing data in more detail in Chapter 7.

A useful Series feature for many applications is that it automatically aligns by index label in arithmetic operations:

**obj3**

Ohio 35000

Oregon 16000

Texas 71000

Utah 5000

dtype: int64

obj4

*California NaN Ohio 35000.0*

*Oregon 16000.0*

*Texas 71000.0*

*dtype: float64*

*obj3 + obj4*

California NaN Ohio 70000.0

Oregon 32000.0

Texas 142000.0

Utah NaN

dtype: float64

Data alignment features will be addressed in more detail later. If you have experience with databases, you can think about this as being similar to a join operation.

Both the Series object itself and its index have a name attribute, which integrates with other key areas of pandas functionality:

obj4.name = 'population'

obj4.index.name = 'state'

obj4

*state California*

*NaN*

*Ohio 35000.0*

*Oregon 16000.0*

*Texas 71000.0*

Name: population, dtype: float64

A Series’s index can be altered in-place by assignment:

**obj**

0 4

1 7

2 -5

3 3

dtype: int64

obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']

obj

*Bob 4*

*Steve 7*

*Jeff -5*

*Ryan 3*

*dtype: int64*

#### DataFrame

A DataFrame represents a rectangular table of data and contains an ordered collec‐ tion of columns, each of which can be a different value type (numeric, string, boolean, etc.). The DataFrame has both a row and column index; it can be thought of as a dict of Series all sharing the same index. Under the hood, the data is stored as one or more two-dimensional blocks rather than a list, dict, or some other collection of one-dimensional arrays. The exact details of DataFrame’s internals are outside the scope of this book.

While a DataFrame is physically two-dimensional, you can use it to represent higher dimensional data in a tabular format using hier‐ archical indexing, a subject we will discuss in Chapter 8 and an ingredient in some of the more advanced data-handling features in pandas.

There are many ways to construct a DataFrame, though one of the most common is from a dict of equal-length lists or NumPy arrays:

data = { 'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],

'year': [2000, 2001, 2002, 2001, 2002, 2003],

'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2] }

frame = pd.DataFrame(data)

The resulting DataFrame will have its index assigned automatically as with Series, and the columns are placed in sorted order:

**frame**

pop state year

0 1.5 Ohio 2000

1 1.7 Ohio 2001

2 3.6 Ohio 2002

3 2.4 Nevada 2001

4 2.9 Nevada 2002

5 3.2 Nevada 2003

If you are using the Jupyter notebook, pandas DataFrame objects will be displayed as a more browser-friendly HTML table.

For large DataFrames, the head method selects only the first five rows:

frame.head()

***pop state year***

*0 1.5 Ohio 2000*

*1 1.7 Ohio 2001*

*2 3.6 Ohio 2002*

*3 2.4 Nevada 2001*

*4 2.9 Nevada 2002*

If you specify a sequence of columns, the DataFrame’s columns will be arranged in that order:

pd.DataFrame(data, columns=['year', 'state', 'pop'])

**year state pop**

0 2000 Ohio 1.5

1 2001 Ohio 1.7

2 2002 Ohio 3.6

3 2001 Nevada 2.4

4 2002 Nevada 2.9

5 2003 Nevada 3.2

If you pass a column that isn’t contained in the dict, it will appear with missing values in the result:

frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],

....: index=['one', 'two', 'three', 'four', 'five', 'six'])

**frame2**

year state pop debt

one 2000 Ohio 1.5 NaN

two 2001 Ohio 1.7 NaN

three 2002 Ohio 3.6 NaN

four 2001 Nevada 2.4 NaN

five 2002 Nevada 2.9 NaN

six 2003 Nevada 3.2 NaN

frame2.columns

Index(['year', 'state', 'pop', 'debt'], dtype='object')

A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute:

frame2['state']

*one Ohio*

*two Ohio*

*three Ohio*

*four Nevada*

*five Nevada*

*six Nevada*

*Name: state, dtype: object*

frame2.year

*one 2000*

*two 2001*

*three 2002*

*four 2001*

*five 2002*

*six 2003*

*Name: year, dtype: int64*

Attribute-like access (e.g., frame2.year) and tab completion of col‐ umn names in IPython is provided as a convenience. frame2[column] works for any column name, but frame2.column only works when the column name is a valid Python variable name.

Note that the returned Series have the same index as the DataFrame, and their name

attribute has been appropriately set.

Rows can also be retrieved by position or name with the special loc attribute (much more on this later):

frame2.loc['three']

*year 2002*

*state Ohio*

*pop 3.6*

*debt NaN*

*Name: three, dtype: object*

Columns can be modified by assignment. For example, the empty 'debt' column could be assigned a scalar value or an array of values:

frame2['debt'] = 16.5

frame2

year state pop debt

one 2000 Ohio 1.5 16.5

two 2001 Ohio 1.7 16.5

three 2002 Ohio 3.6 16.5

four 2001 Nevada 2.4 16.5

five 2002 Nevada 2.9 16.5

six 2003 Nevada 3.2 16.5

frame2['debt'] = np.arange(6.)

frame2

year state pop debt

one 2000 Ohio 1.5 0.0

two 2001 Ohio 1.7 1.0

three 2002 Ohio 3.6 2.0

four 2001 Nevada 2.4 3.0

five 2002 Nevada 2.9 4.0

six 2003 Nevada 3.2 5.0

When you are assigning lists or arrays to a column, the value’s length must match the length of the DataFrame. If you assign a Series, its labels will be realigned exactly to the DataFrame’s index, inserting missing values in any holes:

val = **pd.Series(**[-1.2, -1.5, -1.7], **index**=['two', 'four', 'five'])

**frame2['debt'] = val**

frame2

year state pop debt

one 2000 Ohio 1.5 NaN

two 2001 Ohio 1.7 -1.2

three 2002 Ohio 3.6 NaN

four 2001 Nevada 2.4 -1.5

five 2002 Nevada 2.9 -1.7

six 2003 Nevada 3.2 NaN

Assigning a column that doesn’t exist will create a new column. The del keyword will delete columns as with a dict.

As an example of del, I first add a new column of boolean values where the state column equals 'Ohio':

frame2['eastern'] = frame2.state == 'Ohio'

frame2

year state pop debt eastern

one 2000 Ohio 1.5 NaN True

two 2001 Ohio 1.7 -1.2 True

three 2002 Ohio 3.6 NaN True

four 2001 Nevada 2.4 -1.5 False

five 2002 Nevada 2.9 -1.7 False

six 2003 Nevada 3.2 NaN False

New columns cannot be created with the frame2.eastern syntax.

The del method can then be used to remove this column:

del frame2['eastern']

frame2.columns

*Index(['year', 'state', 'pop', 'debt'], dtype='object')*

The column returned from indexing a DataFrame is a view on the underlying data, not a copy. Thus, any in-place modifications to the Series will be reflected in the DataFrame. The column can be explicitly copied with the Series’s copy method.

Another common form of data is a nested dict of dicts:

pop = {'Nevada': {2001: 2.4, 2002: 2.9}, 'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}

If the nested dict is passed to the DataFrame, pandas will interpret the outer dict keys as the columns and the inner keys as the row indices:

frame3 = pd.DataFrame(pop)

frame3

*Nevada Ohio*

*2000 NaN 1.52001 2.4 1.7*

*2002 2.9 3.6*

You can transpose the DataFrame (swap rows and columns) with similar syntax to a NumPy array:

frame3**.T**

2000 2001 2002

Nevada NaN 2.4 2.9

Ohio 1.5 1.7 3.6

The keys in the inner dicts are combined and sorted to form the index in the result. This isn’t true if an explicit index is specified:

pd.**DataFrame**(pop, **index=[**2001, 2002, 2003**]**)

Nevada Ohio

2001 2.4 1.7

2002 2.9 3.6

2003 NaN NaN

Dicts of Series are treated in much the same way:

In [70]: pdata = {'Ohio': frame3['Ohio'][:-1],

....: 'Nevada': frame3['Nevada'][:2]}

pd.**DataFrame(**pdata**)**

Nevada Ohio

2000 NaN 1.5

2001 2.4 1.7

For a complete list of things you can pass the DataFrame constructor, see Table 5-1.

If a DataFrame’s index and columns have their name attributes set, these will also be displayed:

frame3.**index**.**name** = 'year'; frame3.**columns**.**name** = 'state'

**frame3**

state

year Nevada Ohio

2000 NaN 1.5

2001 2.4 1.7

2002 2.9 3.6

As with Series, the values attribute returns the data contained in the DataFrame as a two-dimensional ndarray:

frame3**.values**

*array([[ nan, 1.5],*

*[ 2.4, 1.7],*

*[ 2.9, 3.6]])*

If the DataFrame’s columns are different dtypes, the dtype of the values array will be chosen to accommodate all of the columns:

frame2**.values**

array([[2000, 'Ohio', 1.5, nan],

[2001, 'Ohio', 1.7, -1.2],

[2002, 'Ohio', 3.6, nan],

[2001, 'Nevada', 2.4, -1.5],

[2002, 'Nevada', 2.9, -1.7],

[2003, 'Nevada', 3.2, nan]], dtype=object)

Table 5-1. Possible data inputs to DataFrame constructor

**Type Notes**

2D ndarray A matrix of data, passing optional row and column labels

dict of arrays, lists Each sequence becomes a column in DataFrame; all sequences must be same length

, or tuples

NumPy structured/record array

Treated as the “dict of arrays” case

dict of Series Each value becomes a column; indexes from each Series are unioned together to form the

result’s row index if no explicit index is passed

dict of dicts Each inner dict becomes a column; keys are unioned to form the row index as in

the “dict of Series” case

List of dicts or Series Each item becomes a row in the DataFrame; union of dict keys or Series indexes

become the DataFrame’s column labels

List of lists or tuples Treated as the “2D ndarray” case

Another DataFrame The DataFrame’s indexes are used unless different ones are passed

NumPy MaskedArray Like the “2D ndarray” case except masked values become NA/missing in the DataFrame result

#### Index Objects

pandas’s Index objects are responsible for holding the axis labels and other metadata (like the axis name or names). Any array or other sequence of labels you use when constructing a Series or DataFrame is internally converted to an Index:

In [76]: obj = pd.Series(range(3), index=['a', 'b', 'c']) In [77]: index = obj.index

In [78]: index

Out[78]: Index(['a', 'b', 'c'], dtype='object')

In [79]: index[1:]

Out[79]: Index(['b', 'c'], dtype='object')

Index objects are immutable and thus can’t be modified by the user:

index[1] = 'd' # TypeError

Immutability makes it safer to share Index objects among data structures:

In [80]: labels = pd.Index(np.arange(3))

labels

*Int64Index([0, 1, 2], dtype='int64')*

obj2 = pd.Series([1.5, -2.5, 0], index=labels)

In [83]: obj2

*0 1.5*

*1 -2.5*

*2 0.0*

*dtype: float64*

obj2.index is labels

*True*

Some users will not often take advantage of the capabilities pro‐ vided by indexes, but because some operations will yield results containing indexed data, it’s important to understand how they work.

In addition to being array-like, an Index also behaves like a fixed-size set:

frame3

state year

Nevada Ohio

2000 NaN 1.5

2001 2.4 1.7

2002 2.9 3.6

frame3.columns

*Index(['Nevada', 'Ohio'], dtype='object', name='state')*

'Ohio' in frame3.columns

*True*

2003 in frame3.index

*False*

Unlike Python sets, a pandas Index can contain duplicate labels:

dup\_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])

dup\_labels

*Index(['foo', 'foo', 'bar', 'bar'], dtype='object')*

Selections with duplicate labels will select all occurrences of that label.

Each Index has a number of methods and properties for set logic, which answer other common questions about the data it contains. Some useful ones are summarized in [Table 5-2](#_bookmark13).

Table 5-2. Some Index methods and properties

**Method Description**

**append** Concatenate with additional Index objects, producing a new Index

**difference** Compute set difference as an Index intersection Compute set intersection

**union** Compute set union

**isin** Compute boolean array indicating whether each value is contained in the passed collection

**delete** Compute new Index with element at index i deleted

**drop** Compute new Index by deleting passed values

**insert** Compute new Index by inserting element at index i

**is\_monotonic** Returns True if each element is greater than or equal to the previous element

**is\_unique** Returns True if the Index has no duplicate values

**unique** Compute the array of unique values in the Index

### Essential Functionality

This section will walk you through the fundamental mechanics of interacting with the data contained in a Series or DataFrame. In the chapters to come, we will delve more deeply into data analysis and manipulation topics using pandas. This book is not intended to serve as exhaustive documentation for the pandas library; instead, we’ll focus on the most important features, leaving the less common (i.e., more esoteric) things for you to explore on your own.

#### Reindexing

An important method on pandas objects is reindex, which means to create a new object with the data conformed to a new index. Consider an example:

obj = pd.**Series**([4.5, 7.2, -5.3, 3.6], **index**=['d', 'b', 'a', 'c'])

obj

*d 4.5*

*b 7.2*

*a -5.3*

*c 3.6*

*dtype: float64*

Calling reindex on this Series rearranges the data according to the new index, intro‐ ducing missing values if any index values were not already present:

obj2 = obj.**reindex**(['a', 'b', 'c', 'd', 'e'])

obj2

*a -5.3*

*b 7.2*

*c 3.6*

*d 4.5*

*e NaN*

*dtype: float64*

For ordered data like time series, it may be desirable to do some interpolation or fill‐ ing of values when reindexing. The method option allows us to do this, using a method such as ffill, which forward-fills the values:

obj3 = pd.**Series**(['blue', 'purple', 'yellow'], **index**=[0, 2, 4])

obj3

*0 blue*

*2 purple*

*4 yellow dtype: object*

obj3**.reindex(range(**6**)**, **method**=**'ffill'**)

*blue*

*blue*

*purple*

*purple*

*yellow*

*yellow dtype: object*

With DataFrame, reindex can alter either the (row) index, columns, or both. When passed only a sequence, it reindexes the rows in the result:

frame = pd**.DataFrame( np.arange(**9)**.reshape(**(3, 3)), **index**=['a', 'c', 'd'],

**columns=[**'Ohio', 'Texas', 'California'])

**frame**

*Ohio Texas California*

*a 0 1 2*

*c 3 4 5*

*d 6 7 8*

frame2 = frame.**reindex**(['a', 'b', 'c', 'd'])

frame2

*Ohio Texas California*

*a 0.0 1.0 2.0*

*b NaN NaN NaN*

*c 3.0 4.0 5.0*

*d 6.0 7.0 8.0*

The columns can be reindexed with the columns keyword:

states = ['Texas', 'Utah', 'California']

frame.reindex(columns=states)

Texas Utah California

a 1 NaN 2

4 NaN 5

7 NaN 8

See [Table 5-3](#_bookmark14) for more about the arguments to reindex.

As we’ll explore in more detail, you can reindex more succinctly by label-indexing with loc, and many users prefer to use it exclusively:

frame.loc[['a', 'b', 'c', 'd'], states]

*Texas Utah California*

*a 1.0 NaN 2.0*

*b NaN NaN NaN*

*c 4.0 NaN 5.0*

*d 7.0 NaN 8.0*

Table 5-3. reindex function arguments

**Argument Description**

**index** New sequence to use as index. Can be Index instance or any other sequence-like Python data structure. An

Index will be used exactly as is without any copying.

method Interpolation (fill) method; 'ffill' fills forward, while 'bfill' fills backward.

**fill\_value** Substitute value to use when introducing missing data by reindexing.

**limit** When forward- or backfilling, maximum size gap (in number of elements) to fill.

**tolerance** When forward- or backfilling, maximum size gap (in absolute numeric distance) to fill for inexact matches.

**level** Match simple Index on level of MultiIndex; otherwise select subset of.

**copy** If True, always copy underlying data even if new index is equivalent to old index; if False, do not

copy the data when the indexes are equivalent.

#### Dropping Entries from an Axis

Dropping one or more entries from an axis is easy if you already have an index array or list without those entries. As that can require a bit of munging and set logic, the

drop method will return a new object with the indicated value or values deleted from an axis:

obj = pd**.Series(np.arange(**5.**)**, **index=[**'a', 'b', 'c', 'd', 'e'**])**

**obj**

*a 0.0*

*b 1.0*

*c 2.0*

*d 3.0*

*e 4.0*

*dtype: float64*

new\_obj = obj**.drop('c')**

new\_obj

*a 0.0*

*b 1.0*

*d 3.0*

*e 4.0*

*dtype: float64*

obj.drop(['d', 'c'])

*a 0.0*

*b 1.0*

*e 4.0*

*dtype: float64*

With DataFrame, index values can be deleted from either axis. To illustrate this, we first create an example DataFrame:

data = **pd.DataFrame(np.arange(**16**)**.**reshape((**4, 4**))**,

.....: **index=[**'Ohio', 'Colorado', 'Utah', 'New York'**],**

.....: **columns=[**'one', 'two', 'three', 'four'**])**

**data**

one two three four

Ohio 0 1 2 3

Colorado 4 5 6 7

Utah 8 9 10 11

New York 12 13 14 15

Calling drop with a sequence of labels will drop values from the row labels (axis 0):

data.drop(['Colorado', 'Ohio'])

one two three four

Utah 8 9 10 11

New York 12 13 14 15

You can drop values from the columns by passing axis=1 or axis='columns':

data.drop('two', axis=1)

one three four

Ohio 0 2 3

Colorado 4 6 7

Utah 8 10 11

New York 12 14 15

data.drop(['two', 'four'], axis='columns')

one three

Ohio 0 2

Colorado 4 6

Utah 8 10

New York 12 14

Many functions, like drop, which modify the size or shape of a Series or DataFrame, can manipulate an object in-place without returning a new object:

obj.drop('c', inplace=True)

obj

*a 0.0*

*b 1.0*

*d 3.0*

*e 4.0*

*dtype: float64*

Be careful with the inplace, as it destroys any data that is dropped.

#### Indexing, Selection, and Filtering

Series indexing (obj[...]) works analogously to NumPy array indexing, except you can use the Series’s index values instead of only integers. Here are some examples of this:

obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])

obj

*a 0.0*

*b 1.0*

*c 2.0*

*d 3.0*

*dtype: float64*

obj['b']

*1.0*

obj[1]

*1.0*

obj[2:4]

*c 2.0*

*d 3.0*

*dtype: float64*

In [122]: obj[['b', 'a', 'd']] Out[122]:

*b 1.0*

*a 0.0*

*d 3.0*

*dtype: float64*

In [123]: obj[[1, 3]]

*b 1.0*

*d 3.0*

*dtype: float64*

In [124]: obj[obj < 2]

*a 0.0*

*b 1.0*

*dtype: float64*

Slicing with labels behaves differently than normal Python slicing in that the end‐ point is inclusive:

obj['b':'c']

*b 1.0*

*c 2.0*

*dtype: float64*

Setting using these methods modifies the corresponding section of the Series:

obj['b':'c'] = 5

obj

*a 0.0*

*b 5.0*

*c 5.0*

*d 3.0*

*dtype: float64*

Indexing into a DataFrame is for retrieving one or more columns either with a single value or sequence:

data = pd**.DataFrame(np.arange(**16).**reshape(**(4, 4)**), index=[**'Ohio', 'Colorado', 'Utah', 'New York'**], columns=[**'one', 'two', 'three', 'four'**])**

data

*one two three four*

*Ohio 0 1 2 3*

*Colorado 4 5 6 7*

*Utah 8 9 10 11*

*New York 12 13 14 15*

data['two']

*Ohio 1*

*Colorado 5*

*Utah 9*

*New York 13*

*Name: two, dtype: int64*

data[['three', 'one']]

*three one*

*Ohio 2 0*

*Colorado 6 4*

*Utah 10 8*

*New York 14 12*

Indexing like this has a few special cases. First, slicing or selecting data with a boolean array:

data[:2]

one two three four

Ohio 0 1 2 3

Colorado 4 5 6 7

data[data['three'] > 5]

*one two three four*

*Colorado 4 5 6 7*

*Utah 8 9 10 11*

*New York 12 13 14 15*

The row selection syntax data[:2] is provided as a convenience. Passing a single ele‐ ment or a list to the [] operator selects columns.

Another use case is in indexing with a boolean DataFrame, such as one produced by a scalar comparison:

data < 5

one two three four

Ohio True True True True

Colorado True False False False

Utah False False False False

New York False False False False

data[data < 5] = 0

data

one two three four

Ohio 0 0 0 0

Colorado 0 5 6 7

Utah 8 9 10 11

New York 12 13 14 15

This makes DataFrame syntactically more like a two-dimensional NumPy array in this particular case.

#### Selection with loc and iloc

For DataFrame label-indexing on the rows, I introduce the special indexing operators loc and iloc. They enable you to select a subset of the rows and columns from a DataFrame with NumPy-like notation using either axis labels (loc) or integers (iloc).

As a preliminary example, let’s select a single row and multiple columns by label:

data.**loc**['Colorado', ['two', 'three']]

*two 5*

*three 6*

*Name: Colorado,*

*dtype: int64*

We’ll then perform some similar selections with integers using iloc:

data.**iloc**[2, [3, 0, 1]]

*four 11*

*one 8*

*two 9*

*Name: Utah, dtype: int64*

data.**iloc**[2]

*one 8*

*two 9*

*three 10*

*four 11*

*Name: Utah, dtype: int64*

data.**iloc**[[1, 2], [3, 0, 1]]

four one two

Colorado 7 0 5

Utah 11 8 9

Both indexing functions work with slices in addition to single labels or lists of labels:

data**.loc[**:'Utah', 'two'**]**

Ohio 0

Colorado 5

Utah 9

Name: two, dtype: int64

data**.iloc[:, :3][data.three > 5]**

*one two three*

*Colorado 0 5 6*

*Utah 8 9 10*

*New York 12 13 14*

So there are many ways to select and rearrange the data contained in a pandas object. For DataFrame, [Table 5-4](#_bookmark15) provides a short summary of many of them. As you’ll see later, there are a number of additional options for working with hierarchical indexes.

When originally designing pandas, I felt that having to type frame[:, col] to select a column was too verbose (and error- prone), since column selection is one of the most common opera‐ tions. I made the design trade-off to push all of the fancy indexing behavior (both labels and integers) into the ix operator. In practice, this led to many edge cases in data with integer axis labels, so the pandas team decided to create the loc and iloc operators to deal with strictly label-based and integer-based indexing, respectively.

The ix indexing operator still exists, but it is deprecated. I do not recommend using it.

Table 5-4. Indexing options with DataFrame

**Type Description**

df[val] Select single column or sequence of columns from the DataFrame; special case conveniences: boolean array (filter rows), slice (slice rows), or boolean DataFrame (set values based on some criterion)

df.loc[val] Selects single row or subset of rows from the DataFrame by label

df.loc[:, val] Selects single column or subset of columns by label

df.loc[val1, val2] Select both rows and columns by label

df.iloc[where] Selects single row or subset of rows from the DataFrame by integer position

**Type Description**

df.iloc[:, where] Selects single column or subset of columns by integer position

df.iloc[where\_i, where\_j] Select both rows and columns by integer position df.at[label\_i, label\_j] Select a single scalar value by row and column label df.iat[i, j] Select a single scalar value by row and column position (integers)

reindex method Select either rows or columns by labels

get\_value, set\_value methods Select single value by row and column label

#### Integer Indexes

Working with pandas objects indexed by integers is something that often trips up new users due to some differences with indexing semantics on built-in Python data structures like lists and tuples. For example, you might not expect the following code to generate an error:

ser = pd.Series(np.arange(3.))

ser

ser[-1]

In this case, pandas could “fall back” on integer indexing, but it’s difficult to do this in general without introducing subtle bugs. Here we have an index containing 0, 1, 2, but inferring what the user wants (label-based indexing or position-based) is difficult:

ser

*0 0.0*

*1 1.0*

*2 2.0*

*dtype: float64*

On the other hand, with a non-integer index, there is no potential for ambiguity:

ser2 = pd**.Series(np.arange(**3.**), index=[**'a', 'b', 'c'**])**

ser2[-1]

*2.0*

To keep things consistent, if you have an axis index containing integers, data selection will always be label-oriented. For more precise handling, use loc (for labels) or iloc (for integers):

ser[:1]

*0 0.0*

*dtype: float64*

ser.loc[:1]

*0 0.0*

*1 1.0*

*dtype: float64*

ser.iloc[:1]

*0 0.0*

*dtype: float64*

#### Arithmetic and Data Alignment

An important pandas feature for some applications is the behavior of arithmetic between objects with different indexes. When you are adding together objects, if any index pairs are not the same, the respective index in the result will be the union of the index pairs. For users with database experience, this is similar to an automatic outer join on the index labels. Let’s look at an example:

s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'**])**

s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],

.....: index=['a', 'c', 'e', 'f', 'g'])

s1

*a 7.3*

*c -2.5*

*d 3.4*

*e 1.5*

*dtype: float64*

s2

*a -2.1*

*c 3.6*

*e -1.5*

*f 4.0*

*g 3.1*

*dtype: float64*

Adding these together yields:

s1 + s2

*a 5.2*

*c 1.1*

*d NaN*

*e 0.0*

*f NaN*

*g NaN*

*dtype: float64*

The internal data alignment introduces missing values in the label locations that don’t overlap. Missing values will then propagate in further arithmetic computations.

In the case of DataFrame, alignment is performed on both the rows and the columns:

df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),

.....: index=['Ohio', 'Texas', 'Colorado'])

df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),

.....: index=['Utah', 'Ohio', 'Texas', 'Oregon'])

df1

*b c d*

*Ohio 0.0 1.0 2.0*

*Texas 3.0 4.0 5.0*

*Colorado 6.0 7.0 8.0*

df2

*b d e*

*Utah 0.0 1.0 2.0*

*Ohio 3.0 4.0 5.0*

*Texas 6.0 7.0 8.0*

*Oregon 9.0 10.0 11.0*

Adding these together returns a DataFrame whose index and columns are the unions of the ones in each DataFrame:

df1 + df2

*b c d e*

*Colorado NaN NaN NaN NaN*

*Ohio 3.0 NaN 6.0 NaN*

*Oregon NaN NaN NaN NaN*

*Texas 9.0 NaN 12.0 NaN*

*Utah NaN NaN NaN NaN*

Since the 'c' and 'e' columns are not found in both DataFrame objects, they appear as all missing in the result. The same holds for the rows whose labels are not common to both objects.

If you add DataFrame objects with no column or row labels in common, the result will contain all nulls:

df1 = pd.DataFrame({'A': [1, 2]})

df2 = pd.DataFrame({'B': [3, 4]})

df1

A

0 1

1 2

df2

B

0 3

1 4

df1 - df2

A B

0 NaN NaN

1

2 NaN NaN

3

#### Arithmetic methods with fill values

In arithmetic operations between differently indexed objects, you might want to fill with a special value, like 0, when an axis label is found in one object but not the other:

df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)),

.....: columns=list('abcd'))

df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),

.....: columns=list('abcde'))

df2.loc[1, 'b'] = np.nan

df1

A b c d

0 0.0 1.0 2.0 3.0

1 4.0 5.0 6.0 7.0

2 8.0 9.0 10.0 11.0

df2

a b c d e

0 0.0 1.0 2.0 3.0 4.0

1 5.0 NaN 7.0 8.0 9.0

2 10.0 11.0 12.0 13.0 14.0

3 15.0 16.0 17.0 18.0 19.0

Adding these together results in NA values in the locations that don’t overlap:

df1 + df2

a b c d e

0 0.0 2.0 4.0 6.0 NaN

1 9.0 NaN 13.0 15.0 NaN

2 18.0 20.0 22.0 24.0 NaN

3 NaN NaN NaN NaN NaN

Using the add method on df1, I pass df2 and an argument to fill\_value:

df1.add(df2, fill\_value=0)

a b c d e

0 0.0 2.0 4.0 6.0 4.0

1 9.0 5.0 13.0 15.0 9.0

2 18.0 20.0 22.0 24.0 14.0

3 15.0 16.0 17.0 18.0 19.0

See Table 5-5 for a listing of Series and DataFrame methods for arithmetic. Each of them has a counterpart, starting with the letter r, that has arguments flipped. So these two statements are equivalent:

**1 / df1**

a b c d

0 inf 1.000000 0.500000 0.333333

1 0.250000 0.200000 0.166667 0.142857

2 0.125000 0.111111 0.100000 0.090909

**df1.rdiv(1)**

a b c d

0 inf 1.000000 0.500000 0.333333

1 0.250000 0.200000 0.166667 0.142857

2 0.125000 0.111111 0.100000 0.090909

Relatedly, when reindexing a Series or DataFrame, you can also specify a different fill value:

**df1.reindex(columns=df2.columns**, fill\_value=0)

a b c d e

0 0.0 1.0 2.0 3.0 0

1 4.0 5.0 6.0 7.0 0

2 8.0 9.0 10.0 11.0 0

**Table 5-5. Flexible arithmetic methods**

Method Description

add, radd Methods for addition (+)

sub, rsub Methods for subtraction (-)

div, rdiv Methods for division (/) floordiv,

rfloordiv Methods for floor division (//)

mul, rmul Methods for multiplication (\*)

pow, rpow Methods for exponentiation (\*\*)

#### Operations between DataFrame and Series

As with NumPy arrays of different dimensions, arithmetic between DataFrame and Series is also defined. First, as a motivating example, consider the difference between a two-dimensional array and one of its rows:

arr = np.arange(12.).reshape((3, 4))

arr

*array([[ 0., 1., 2., 3.],*

*[ 4., 5., 6., 7.],*

*[ 8., 9., 10., 11.]])*

arr[0]

array([ 0., 1., 2., 3.])

arr - arr[0]

*array([[ 0., 0., 0., 0.],*

*[ 4., 4., 4., 4.],*

*[ 8., 8., 8., 8.]])*

When we subtract arr[0] from arr, the subtraction is performed once for each row. This is referred to as broadcasting and is explained in more detail as it relates to gen‐ eral NumPy arrays in Appendix A. Operations between a DataFrame and a Series are similar:

In [179]: frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),

.....: columns=list('bde'),

.....: index=['Utah', 'Ohio', 'Texas', 'Oregon'])

series = frame.iloc[0]

frame

*b d e*

*Utah 0.0 1.0 2.0*

*Ohio 3.0 4.0 5.0*

*Texas 6.0 7.0 8.0*

*Oregon 9.0 10.0 11.0*

**series**

*b 0.0*

*d 1.0*

*e 2.0*

Name: Utah, dtype: float64

By default, arithmetic between DataFrame and Series matches the index of the Series on the DataFrame’s columns, broadcasting down the rows:

**frame - series**

*b d e*

*Utah 0.0 0.0 0.0*

*Ohio 3.0 3.0 3.0*

*Texas 6.0 6.0 6.0*

*Oregon 9.0 9.0 9.0*

If an index value is not found in either the DataFrame’s columns or the Series’s index, the objects will be reindexed to form the union:

series2 = **pd.Series(range(3), index=[**'b', 'e', 'f'**])**

frame + series2

*b d e f*

*Utah 0.0 NaN 3.0 NaN*

*Ohio 3.0 NaN 6.0 NaN*

*Texas 6.0 NaN 9.0 NaN*

*Oregon 9.0 NaN 12.0 NaN*

If you want to instead broadcast over the columns, matching on the rows, you have to use one of the arithmetic methods. For example:

**series3 = frame['d']**

frame

*b d e*

*Utah 0.0 1.0 2.0*

*Ohio 3.0 4.0 5.0*

*Texas 6.0 7.0 8.0*

*Oregon 9.0 10.0 11.0*

series3

*Utah 1.0*

*Ohio 4.0*

*Texas 7.0*

*Oregon 10.0*

*Name: d, dtype: float64*

frame.sub(series3, axis='index')

*b d e*

*Utah -1.0 0.0 1.0*

*Ohio -1.0 0.0 1.0*

*Texas -1.0 0.0 1.0*

*Oregon -1.0 0.0 1.0*

The axis number that you pass is the axis to match on. In this case we mean to match on the DataFrame’s row index (axis='index' or axis=0) and broadcast across.

#### Function Application and Mapping

NumPy ufuncs (element-wise array methods) also work with pandas objects:

frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),

.....: index=['Utah', 'Ohio', 'Texas', 'Oregon'])

frame

*B d e*

*Utah -0.204708 0.478943 -0.519439*

*Ohio -0.555730 1.965781 1.393406*

*Texas 0.092908 0.281746 0.769023*

*Oregon 1.246435 1.007189 -1.296221*

np.abs(frame)

b d e

Utah 0.204708 0.478943 0.519439

Ohio 0.555730 1.965781 1.393406

Texas 0.092908 0.281746 0.769023

Oregon 1.246435 1.007189 1.296221

Another frequent operation is applying a function on one-dimensional arrays to each column or row. DataFrame’s apply method does exactly this:

f = lambda x: x.max() - x.min()

frame.apply(f)

*b 1.802165*

*d 1.684034*

*e 2.689627*

*dtype: float64*

Here the function f, which computes the difference between the maximum and mini‐ mum of a Series, is invoked once on each column in frame. The result is a Series hav‐ ing the columns of frame as its index.

If you pass axis='columns' to apply, the function will be invoked once per row instead:

frame**.apply(f, axis='columns')**

*Utah 0.998382*

*Ohio 2.521511*

*Texas 0.676115*

*Oregon 2.542656*

*dtype: float64*

Many of the most common array statistics (like sum and mean) are DataFrame meth‐ ods, so using apply is not necessary.

The function passed to apply need not return a scalar value; it can also return a Series with multiple values:

def f(x):

.....: return pd.Series([x.min(), x.max()], index=['min', 'max'])

frame.apply(f)

*b d e*

*min -0.555730 0.281746 -1.296221*

*max 1.246435 1.965781 1.393406*

Element-wise Python functions can be used, too. Suppose you wanted to compute a formatted string from each floating-point value in frame. You can do this with apply map:

format = lambda x: '%.2f' % x

frame.applymap(format)

*b d e*

*Utah -0.20 0.48 -0.52*

*Ohio -0.56 1.97 1.39*

*Texas 0.09 0.28 0.77*

*Oregon 1.25 1.01 -1.30*

The reason for the name applymap is that Series has a map method for applying an element-wise function:

frame['e'].map(format)

*Utah -0.52*

*Ohio 1.39*

*Texas 0.77*

*Oregon -1.30*

*Name: e, dtype: object*

#### Sorting and Ranking

Sorting a dataset by some criterion is another important built-in operation. To sort lexicographically by row or column index, use the sort\_index method, which returns a new, sorted object:

obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])

obj.sort\_index()

*a 1*

*b 2*

*c 3*

*d 0*

*dtype: int64*

With a DataFrame, you can sort by index on either axis:

frame = pd.DataFrame(np.arange(8).reshape((2, 4)),

.....: index=['three', 'one'],

.....: columns=['d', 'a', 'b', 'c'])

frame.sort\_index()

*d a b c*

*one 4 5 6 7*

*three 0 1 2 3*

frame.sort\_index(axis=1)

*a b c d*

*three 1 2 3 0*

*one 5 6 7 4*

The data is sorted in ascending order by default, but can be sorted in descending order, too:

frame.sort\_index(axis=1, ascending=False)

d c b a

three 0 3 2 1

one 4 7 6 5

To sort a Series by its values, use its sort\_values method:

obj = pd.Series([4, 7, -3, 2])

obj.sort\_values()

*2 -3*

*3 2*

*0 4*

*1 7*

*dtype: int64*

Any missing values are sorted to the end of the Series by default:

obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])

obj.sort\_values()

*4 -3.0*

*5 2.0*

*0 4.0*

*2 7.0*

*1 NaN*

*3 NaN*

*dtype: float64*

When sorting a DataFrame, you can use the data in one or more columns as the sort keys. To do so, pass one or more column names to the by option of sort\_values:

frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})

frame

*a b*

*0 0 4*

*1 1 7*

*2 0 -3*

*3 1 2*

frame.sort\_values(by='b')

*a b*

*2 0 -3*

*3 1 2*

*0 0 4*

*1 1 7*

To sort by multiple columns, pass a list of names:

frame.sort\_values(by=['a', 'b'])

*a b*

*2 0 -3*

*0 0 4*

*3 1 2*

*1 1 7*

Ranking assigns ranks from one through the number of valid data points in an array. The rank methods for Series and DataFrame are the place to look; by default rank breaks ties by assigning each group the mean rank:

obj = pd.Series([7, -5, 7, 4, 2, 0, 4])

obj.rank()

*0 6.5*

*1 1.0*

*2 6.5*

*3 4.5*

*4 3.0*

*5 2.0*

*6 4.5*

*dtype: float64*

Ranks can also be assigned according to the order in which they’re observed in the data:

obj.rank(method='first')

*0 6.0*

*1 1.0*

*2 7.0*

*3 4.0*

*4 3.0*

*5 2.0*

*6 5.0*

*dtype: float64*

Here, instead of using the average rank 6.5 for the entries 0 and 2, they instead have been set to 6 and 7 because label 0 precedes label 2 in the data.

You can rank in descending order, too:

**# Assign tie values the maximum rank in the group**

obj.rank(ascending=False, method='max')

*0 2.0*

*1 7.0*

*2 2.0*

*3 4.0*

*4 5.0*

*5 6.0*

*6 4.0*

*dtype: float64*

See Table 5-6 for a list of tie-breaking methods available. DataFrame can compute ranks over the rows or the columns:

frame = pd.DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],

.....: 'c': [-2, 5, 8, -2.5]})

frame

*a b c*

*0 0 4.3 -2.0*

*1 1 7.0 5.0*

*2 0 -3.0 8.0*

*3 1 2.0 -2.5*

frame.rank(axis='columns')

*a b c*

*0 2.0 3.0 1.0*

*1 1.0 3.0 2.0*

*2 2.0 1.0 3.0*

*3 2.0 3.0 1.0*

Table 5-6. Tie-breaking methods with rank

**Table Description**

**'average'** Default: assign the average rank to each entry in the equal group

**'min'** Use the minimum rank for the whole group

**'max'** Use the maximum rank for the whole group

**'first'** Assign ranks in the order the values appear in the data

'**dense'** Like method='min', but ranks always increase by 1 in between groups rather than the number

of equal elements in a group

#### Axis Indexes with Duplicate Labels

Up until now all of the examples we’ve looked at have had unique axis labels (index values). While many pandas functions (like reindex) require that the labels be unique, it’s not mandatory. Let’s consider a small Series with duplicate indices:

obj = pd.Series(range(5), index=['a', 'a', 'b', 'b', 'c'])

obj

*a 0*

*a 1*

*b 2*

*b 3*

*c 4*

*dtype: int64*

The index’s is\_unique property can tell you whether its labels are unique or not:

obj.index.is\_unique

*False*

Data selection is one of the main things that behaves differently with duplicates. Indexing a label with multiple entries returns a Series, while single entries return a scalar value:

obj['a']

*a 0*

*a 1*

*dtype: int64*

obj['c']

*4*

This can make your code more complicated, as the output type from indexing can vary based on whether a label is repeated or not.

The same logic extends to indexing rows in a DataFrame:

df = pd.DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])

**df**

0 1 2

a 0.274992 0.228913 1.352917

a 0.886429 -2.001637 -0.371843

b 1.669025 -0.438570 -0.539741

b 0.476985 3.248944 -1.021228

**df.loc['b']**

0 1 2

b 1.669025 -0.438570 -0.539741

b 0.476985 3.248944 -1.021228

### Summarizing and Computing Descriptive Statistics

pandas objects are equipped with a set of common mathematical and statistical meth‐ ods. Most of these fall into the category of reductions or summary statistics, methods that extract a single value (like the sum or mean) from a Series or a Series of values from the rows or columns of a DataFrame. Compared with the similar methods found on NumPy arrays, they have built-in handling for missing data. Consider a small DataFrame:

In [230]: df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5],

.....: [np.nan, np.nan], [0.75, -1.3]],

.....: index=['a', 'b', 'c', 'd'],

.....: columns=['one', 'two'])

**df**

*one two*

*a 1.40 NaN*

*b 7.10 -4.5*

*c NaN NaN*

*d 0.75 -1.3*

Calling DataFrame’s sum method returns a Series containing column sums:

**df.sum()**

*one 9.25*

*two -5.80*

*dtype: float64*

Passing axis='columns' or axis=1 sums across the columns instead:

df.sum(axis='columns')

*a 1.40*

*b 2.60*

*c NaN*

*d -0.55*

*dtype: float64*

NA values are excluded unless the entire slice (row or column in this case) is NA. This can be disabled with the skipna option:

df.mean(axis='columns', skipna=False)

*a NaN*

*b 1.300*

*c NaN*

*d -0.275*

*dtype: float64*

See Table 5-7 for a list of common options for each reduction method.

Table 5-7. Options for reduction methods

**Method Description**

**axis** Axis to reduce over; 0 for DataFrame’s rows and 1 for columns

**skipna** Exclude missing values; True by default

**level** Reduce grouped by level if the axis is hierarchically indexed (MultiIndex)

Some methods, like idxmin and idxmax, return indirect statistics like the index value where the minimum or maximum values are attained:

**df.idxmax()**

*one b*

*two d*

*dtype: object*

Other methods are accumulations:

**df.cumsum()**

one two

a 1.40 NaN

b 8.50 -4.5

c NaN NaN

d 9.25 -5.8

Another type of method is neither a reduction nor an accumulation. describe is one such example, producing multiple summary statistics in one shot:

**df.describe()**

one two

count 3.000000 2.000000

mean 3.083333 -2.900000

std 3.493685 2.262742

min 0.750000 -4.500000

25% 1.075000 -3.700000

50% 1.400000 -2.900000

75% 4.250000 -2.100000

max 7.100000 -1.300000

On non-numeric data, describe produces alternative summary statistics:

obj = pd.**Series**(['a', 'a', 'b', 'c'] \* 4)

obj.**describe()**

*count 16*

*unique 3*

*top a*

*freq 8*

*dtype: object*

See [Table 5-8](#_bookmark19) for a full list of summary statistics and related methods.

Table 5-8. Descriptive and summary statistics

**Method Description**

**count** Number of non-NA values

**describe** Compute set of summary statistics for Series or each DataFrame column

**min**, **max** Compute minimum and maximum values

**argmin**, **argmax** Compute index locations (integers) at which minimum or maximum value obtained, respectively

**idxmin**, **idxmax** Compute index labels at which minimum or maximum value obtained, respectively

**quantile** Compute sample quantile ranging from 0 to 1

**sum** Sum of values

**mean** Mean of values

**median** Arithmetic median (50% quantile) of values

**mad** Mean absolute deviation from mean value

**prod** Product of all values

**var** Sample variance of values

**std** Sample standard deviation of values

**skew** Sample skewness (third moment) of values

**kurt** Sample kurtosis (fourth moment) of values

**cumsum** Cumulative sum of values

**cummin**, **cummax** Cumulative minimum or maximum of values, respectively

**cumprod** Cumulative product of values

**diff** Compute first arithmetic difference (useful for time series)

**pct\_change** Compute percent changes

#### Correlation and Covariance

Some summary statistics, like correlation and covariance, are computed from pairs of arguments. Let’s consider some DataFrames of stock prices and volumes obtained from Yahoo! Finance using the add-on pandas-datareader package. If you don’t have it installed already, it can be obtained via conda or pip:

conda install pandas-datareader

I use the pandas\_datareader module to download some data for a few stock tickers:

import pandas\_datareader.data as web

all\_data = { ticker: web.get\_data\_yahoo(ticker)

for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG'] }

price = pd.DataFrame({ ticker: data['Adj Close']

for ticker, data in all\_data.items()}) volume =

pd.DataFrame({ticker: data['Volume']

for ticker, data in all\_data.items() })

It’s possible by the time you are reading this that Yahoo! Finance no longer exists since Yahoo! was acquired by Verizon in 2017. Refer to the pandas-datareader documentation online for the latest functionality.

I now compute percent changes of the prices, a time series operation which will be explored further in Chapter 11:

returns = price.pct\_change()

returns.tail()

*AAPL GOOG IBM MSFT*

*Date*

*2016-10-17 -0.000680 0.001837 0.002072 -0.003483*

*2016-10-18 -0.000681 0.019616 -0.026168 0.007690*

*2016-10-19 -0.002979 0.007846 0.003583 -0.002255*

*2016-10-20 -0.000512 -0.005652 0.001719 -0.004867*

*2016-10-21 -0.003930 0.003011 -0.012474 0.042096*

The corr method of Series computes the correlation of the overlapping, non-NA, aligned-by-index values in two Series. Relatedly, cov computes the covariance:

returns['MSFT'].corr(returns['IBM'])

*0.49976361144151144*

returns['MSFT'].cov(returns['IBM'])

*8.8706554797035462e-05*

Since MSFT is a valid Python attribute, we can also select these columns using more concise syntax:

returns.MSFT.corr(returns.IBM)

*0.49976361144151144*

DataFrame’s corr and cov methods, on the other hand, return a full correlation or covariance matrix as a DataFrame, respectively:

returns.corr()

*AAPL GOOG IBM MSFT*

*AAPL 1.000000 0.407919 0.386817 0.389695*

*GOOG 0.407919 1.000000 0.405099 0.465919*

*IBM 0.386817 0.405099 1.000000 0.499764*

*MSFT 0.389695 0.465919 0.499764 1.000000*

returns.cov()

*AAPL GOOG IBM MSFT*

*AAPL 0.000277 0.000107 0.000078 0.000095*

*GOOG 0.000107 0.000251 0.000078 0.000108*

*IBM 0.000078 0.000078 0.000146 0.000089*

*MSFT 0.000095 0.000108 0.000089 0.000215*

Using DataFrame’s corrwith method, you can compute pairwise correlations between a DataFrame’s columns or rows with another Series or DataFrame. Passing a Series returns a Series with the correlation value computed for each column:

returns.corrwith(returns.IBM)

*AAPL 0.386817*

*GOOG 0.405099*

*IBM 1.000000*

*MSFT 0.499764*

*dtype: float64*

Passing a DataFrame computes the correlations of matching column names. Here I compute correlations of percent changes with volume:

returns.corrwith(volume)

*AAPL -0.075565*

*GOOG -0.007067*

*IBM -0.204849*

*MSFT -0.092950*

*dtype: float64*

Passing axis='columns' does things row-by-row instead. In all cases, the data points are aligned by label before the correlation is computed.

#### Unique Values, Value Counts, and Membership

Another class of related methods extracts information about the values contained in a one-dimensional Series. To illustrate these, consider this example:

obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])

The first function is unique, which gives you an array of the unique values in a Series:

uniques = obj.unique()

uniques

array(['c', 'a', 'd', 'b'], dtype=object)

The unique values are not necessarily returned in sorted order, but could be sorted after the fact if needed (uniques.sort()). Relatedly, value\_counts computes a Series containing value frequencies:

obj.value\_counts() Out[254]:

*c 3*

*a 3*

*b*

*c 2*

*d*

*d 1*

*dtype: int64*

The Series is sorted by value in descending order as a convenience. value\_counts is also available as a top-level pandas method that can be used with any array or sequence:

pd.value\_counts(obj.values, sort=False)

*a 3*

*b*

*c 2*

*d*

*e 3*

*f*

*g 1*

*h*

*dtype: int64*

isin performs a vectorized set membership check and can be useful in filtering a dataset down to a subset of values in a Series or column in a DataFrame:

obj

*0 c*

*1*

*2 a*

*3*

*4 d*

*5*

*6 a*

*7*

*8 a*

*9*

*10 b*

*11*

*12 b*

*13*

*14 c*

*15*

*16 c*

*17*

*dtype: object*

mask = obj.isin(['b', 'c'])

mask

*0 True*

*1*

*2 False*

*3*

*4 False*

*5*

*6 False*

*7*

*8 False*

*9*

*10 True*

*11*

*12 True*

*13*

*14 True*

*15*

*16 True dtype: bool*

*17*

obj[mask]

*0 c*

*5 b*

*6*

*7 b*

*8*

*9 c*

*10*

*11 c*

*12*

*dtype: object*

Related to isin is the Index.get\_indexer method, which gives you an index array from an array of possibly non-distinct values into another array of distinct values:

to\_match = pd.Series(['c', 'a', 'b', 'b', 'c', 'a'])

unique\_vals = pd.Series(['c', 'b', 'a'])

pd.Index(unique\_vals).get\_indexer(to\_match)

array([0, 2, 1, 1, 0, 2])

See [Table 5-9](#_bookmark20) for a reference on these methods.

Table 5-9. Unique, value counts, and set membership methods

**Method Description**

**isin** Compute boolean array indicating whether each Series value is contained in the passed sequence of values

**match** Compute integer indices for each value in an array into another array of distinct values; helpful for

data alignment and join-type operations

**unique** Compute array of unique values in a Series, returned in the order observed

**value\_counts** Return a Series containing unique values as its index and frequencies as its values, ordered

count in descending order

In some cases, you may want to compute a histogram on multiple related columns in a DataFrame. Here’s an example:

data = pd.DataFrame({'Qu1': [1, 3, 4, 3, 4],

.....: 'Qu2': [2, 3, 1, 2, 3],

.....: 'Qu3': [1, 5, 2, 4, 4]})

data

Qu1 Qu2 Qu3

0 1 2 1

1 3 3 5

2 4 1 2

3 3 2 4

4 4 3 4

Passing pandas.value\_counts to this DataFrame’s apply function gives:

result = data**.apply(**pd.value\_counts**).fillna(0)**

result

Qu1 Qu2 Qu3

1 1.0 1.0 1.0

2 0.0 2.0 1.0

3 2.0 2.0 0.0

4 2.0 0.0 2.0

5 0.0 0.0 1.0

Here, the row labels in the result are the distinct values occurring in all of the col‐ umns. The values are the respective counts of these values in each column.

### Conclusion

In the next chapter, we’ll discuss tools for reading (or loading) and writing datasets with pandas. After that, we’ll dig deeper into data cleaning, wrangling, analysis, and visualization tools using pandas.

## Data Loading, Storage, and File Formats

Accessing data is a necessary first step for using most of the tools in this book. I’m going to be focused on data input and output using pandas, though there are numer‐ ous tools in other libraries to help with reading and writing data in various formats.

Input and output typically falls into a few main categories: reading text files and other more efficient on-disk formats, loading data from databases, and interacting with net‐ work sources like web APIs.

### Reading and Writing Data in Text Format

pandas features a number of functions for reading tabular data as a DataFrame object. [Table 6-1](#_bookmark22) summarizes some of them, though read\_csv and read\_table are likely the ones you’ll use the most.

Table 6-1. Parsing functions in pandas

**Function Description**

**read\_csv** Load delimited data from a file, URL, or file-like object; use comma as default delimiter read\_table

Load delim. data from file, URL, or file-like object; use tab ('\t') as default delimiter read\_fwf

Read data in fixed-width column format (i.e., no delimiters)

**read\_clipboard** Version of read\_table that reads data from clipboard; useful for converting tables from web pages

**read\_excel** Read tabular data from an Excel XLS or XLSX file

**read\_hdf** Read HDF5 files written by pandas

**read\_html** Read all tables found in the given HTML document

**read\_json** Read data from a JSON (JavaScript Object Notation) string representation

**read\_msgpack** Read pandas data encoded using the MessagePack binary format

**read\_pickle**  Read an arbitrary object stored in Python pickle format

**read\_sas** Read a SAS dataset stored in one of the SAS system’s custom storage formats

**read\_sql** Read the results of a SQL query (using SQLAlchemy) as a pandas DataFrame

**read\_stata** Read a dataset from Stata file format

**read\_feather** Read the Feather binary file format

I’ll give an overview of the mechanics of these functions, which are meant to convert text data into a DataFrame. The optional arguments for these functions may fall into a few categories:

Indexing

Can treat one or more columns as the returned DataFrame, and whether to get column names from the file, the user, or not at all.

Type inference and data conversion

This includes the user-defined value conversions and custom list of missing value markers.

Datetime parsing

Includes combining capability, including combining date and time information spread over multiple columns into a single column in the result.

Iterating

Support for iterating over chunks of very large files.

Unclean data issues

Skipping rows or a footer, comments, or other minor things like numeric data with thousands separated by commas.

Because of how messy data in the real world can be, some of the data loading func‐ tions (especially read\_csv) have grown very complex in their options over time. It’s normal to feel overwhelmed by the number of different parameters (read\_csv has over 50 as of this writing). The online pandas documentation has many examples about how each of them works, so if you’re struggling to read a particular file, there might be a similar enough example to help you find the right parameters.

Some of these functions, like pandas.read\_csv, perform type inference, because the column data types are not part of the data format. That means you don’t necessarily have to specify which columns are numeric, integer, boolean, or string. Other data formats, like HDF5, Feather, and msgpack, have the data types stored in the format.

Handling dates and other custom types can require extra effort. Let’s start with a small comma-separated (CSV) text file:

!cat examples/ex1.csv

*a,b,c,d,message*

*1,2,3,4,hello*

*5,6,7,8,world*

*9,10,11,12,foo*

Here I used the Unix cat shell command to print the raw contents of the file to the screen. If you’re on Windows, you can use type instead of cat to achieve the same effect.

Since this is comma-delimited, we can use read\_csv to read it into a DataFrame:

df = pd.read\_csv('examples/ex1.csv')

df

a b c d message

0 1 2 3 4 hello

1 5 6 7 8 world

2 9 10 11 12 foo

We could also have used read\_table and specified the delimiter:

pd.read\_table('examples/ex1.csv', sep=',')

a b c d message

0 1 2 3 4 hello

1 5 6 7 8 world

2 9 10 11 12 foo

A file will not always have a header row. Consider this file:

!cat examples/ex2.csv

1,2,3,4,hello

5,6,7,8,world

9,10,11,12,foo

To read this file, you have a couple of options. You can allow pandas to assign default column names, or you can specify names yourself:

pd.read\_csv('examples/ex2.csv', header=None)

*0 1 2 3 4*

*0 1 2 3 4 hello*

*1 5 6 7 8 world*

*2 9 10 11 12 foo*

pd.read\_csv('examples/ex2.csv', names=['a', 'b', 'c', 'd', 'message'])

*a b c d message*

*1 2 3 4 hello*

*5 6 7 8 world*

*9 10 11 12 foo*

Suppose you wanted the message column to be the index of the returned DataFrame. You can either indicate you want the column at index 4 or named 'message' using the index\_col argument:

names = ['a', 'b', 'c', 'd', 'message']

pd.read\_csv('examples/ex2.csv', names=names, index\_col='message')

*message a b c d*

*hello 1 2 3 4*

*world 5 6 7 8*

*foo 9 10 11 12*

In the event that you want to form a hierarchical index from multiple columns, pass a list of column numbers or names:

!cat examples/csv\_mindex.csv

*key1,key2,value1,value2*

*one,a,1,2*

*one,b,3,4*

*one,c,5,6*

*one,d,7,8*

*two,a,9,10*

*two,b,11,12*

*two,c,13,14*

*two,d,15,16*

parsed = pd.read\_csv('examples/csv\_mindex.csv',

....: index\_col=['key1', 'key2'])

parsed

*value1 value2*

*key1 key2*

*one a 1 2*

*b 3 4*

*c 5 6*

*d 7 8*

*two a 9 10*

*b 11 12*

*c 13 14*

*d 15 16*

In some cases, a table might not have a fixed delimiter, using whitespace or some other pattern to separate fields. Consider a text file that looks like this:

list(open('examples/ex3.txt'))

*[' A B C\n',*

*'aaa -0.264438 -1.026059 -0.619500\n',*

*'bbb 0.927272 0.302904 -0.032399\n',*

*'ccc -0.264273 -0.386314 -0.217601\n',*

*'ddd -0.871858 -0.348382 1.100491\n']*

While you could do some munging by hand, the fields here are separated by a vari‐ able amount of whitespace. In these cases, you can pass a regular expression as a delimiter for read\_table. This can be expressed by the regular expression \s+, so we have then:

result = pd.read\_table('examples/ex3.txt', sep='\s+')

result

*A B C*

*aaa -0.264438 -1.026059 -0.619500*

*bbb 0.927272 0.302904 -0.032399*

*ccc -0.264273 -0.386314 -0.217601*

*ddd -0.871858 -0.348382 1.100491*

Because there was one fewer column name than the number of data rows, read\_table infers that the first column should be the DataFrame’s index in this spe‐ cial case.

The parser functions have many additional arguments to help you handle the wide variety of exception file formats that occur (see a partial listing in [Table 6-2](#_bookmark23)). For example, you can skip the first, third, and fourth rows of a file with skiprows:

!cat examples/ex4.csv

# hey!

a,b,c,d,message

# just wanted to make things more difficult for you # who reads CSV files with computers, anyway?

1,2,3,4,hello

5,6,7,8,world 9,10,11,12,foo

pd.read\_csv('examples/ex4.csv', skiprows=[0, 2, 3])

a b c d message

0 1 2 3 4 hello

1 5 6 7 8 world

2 9 10 11 12 foo

Handling missing values is an important and frequently nuanced part of the file pars‐ ing process. Missing data is usually either not present (empty string) or marked by some sentinel value. By default, pandas uses a set of commonly occurring sentinels, such as NA and NULL:

!cat examples/ex5.csv something,a,b,c,d,message one,1,2,3,4,NA

two,5,6,,8,world three,9,10,11,12,foo

result = pd.read\_csv('examples/ex5.csv')

result

*something a b c d message 0 one 1 2 3.0 4 NaN*

*1 two 5 6 NaN 8 world 2 three 9 10 11.0 12 foo*

pd**.isnull(**result**)**

*something a b c d message*

*0 False False False False False True*

*1 False False False True False False*

*2 False False False False False False*

The na\_values option can take either a list or set of strings to consider missing values:

result = pd.read\_csv('examples/ex5.csv', na\_values=['NULL'])

**result**

*something a b c d message 0 one 1 2 3.0 4 NaN*

*1 two 5 6 NaN 8 world 2 three 9 10 11.0 12 foo*

Different NA sentinels can be specified for each column in a dict:

sentinels = {'message': ['foo', 'NA'], 'something': ['two']}

pd.read\_csv('examples/ex5.csv', na\_values=sentinels)

*something a b c d message 0 one 1 2 3.0 4 NaN*

*1 NaN 5 6 NaN 8 world 2 three 9 10 11.0 12 NaN*

[Table 6-2](#_bookmark23) lists some frequently used options in pandas.read\_csv and pan das.read\_table.

Table 6-2. Some read\_csv/read\_table function arguments

**Argument Description**

**path** String indicating filesystem location, URL, or file-like object

**sep** or **delimiter** Character sequence or regular expression to use to split fields in each row

**header** Row number to use as column names; defaults to 0 (first row), but should be None if there is no header row

**index**\_**col** Column numbers or names to use as the row index in the result; can be a single name/number

or a list of them for a hierarchical index

**names** List of column names for result, combine with header=None

**skiprows** Number of rows at beginning of file to ignore or list of row numbers (starting from 0) to skip.

**na**\_**values** Sequence of values to replace with NA.

**comment** Character(s) to split comments off the end of lines.

parse\_dates Attempt to parse data to datetime; False by default.

If True, will attempt to parse all columns. Otherwise can specify

a list of column numbers or name to parse. If element of list is

tuple or list, will combine multiple columns together and parse to

date (e.g., if date/time split across two columns).

**keep**\_**date**\_**col** If joining columns to parse date, keep the joined columns;

False by default.

**converters** Dict containing column number of name mapping to functions

(e.g., {'foo': f} would apply the function f to all values in

the 'foo' column).

**dayfirst** When parsing potentially ambiguous dates, treat as international format (e.g., 7/6/2012 -> June 7, 2012);

False by default.

**date**\_**parser** Function to use to parse dates.

**nrows** Number of rows to read from beginning of file. iterator

Return a TextParser object for reading file piecemeal.

**chunksize** For iteration, size of file chunks.

**skip**\_**footer** Number of lines to ignore at end of file.

**verbose** Print various parser output information, like the number of missing values placed in non-numeric columns.

**encoding** Text encoding for Unicode (e.g., 'utf-8' for UTF-8 encoded text).squeeze. If the parsed data only

contains one column, return a Series. Thousands Separator for thousands (e.g., ',' or '.').

#### Reading Text Files in Pieces

When processing very large files or figuring out the right set of arguments to cor‐ rectly process a large file, you may only want to read in a small piece of a file or iterate through smaller chunks of the file.

Before we look at a large file, we make the pandas display settings more compact:

pd.options.display.max\_rows = 10

Now we have:

result = pd.read\_csv('examples/ex6.csv')

result

one two three four key

0 0.467976 -0.038649 -0.295344 -1.824726 L

1 -0.358893 1.404453 0.704965 -0.200638 B

2 -0.501840 0.659254 -0.421691 -0.057688 G

3 0.204886 1.074134 1.388361 -0.982404 R

4 0.354628 -0.133116 0.283763 -0.837063 Q

... ... ... ... ... ..

9995 2.311896 -0.417070 -1.409599 -0.515821 L

9996 -0.479893 -0.650419 0.745152 -0.646038 E

9997 0.523331 0.787112 0.486066 1.093156 K

9998 -0.362559 0.598894 -1.843201 0.887292 G

9999 -0.096376 -1.012999 -0.657431 -0.573315 0

[10000 rows x 5 columns]

If you want to only read a small number of rows (avoiding reading the entire file), specify that with nrows:

pd.read\_csv('examples/ex6.csv', nrows=5)

one two three four key

0 0.467976 -0.038649 -0.295344 -1.824726 L

1 -0.358893 1.404453 0.704965 -0.200638 B

2 -0.501840 0.659254 -0.421691 -0.057688 G

3 0.204886 1.074134 1.388361 -0.982404 R

4 0.354628 -0.133116 0.283763 -0.837063 Q

To read a file in pieces, specify a chunksize as a number of rows:

chunker = pd.read\_csv('examples/ex6.csv', chunksize=1000)

chunker

*<pandas.io.parsers.TextFileReader at 0x7f6b1e2672e8>*

The TextParser object returned by read\_csv allows you to iterate over the parts of the file according to the chunksize. For example, we can iterate over ex6.csv, aggre‐ gating the value counts in the 'key' column like so:

chunker = pd.read\_csv('examples/ex6.csv', chunksize=1000)

tot = pd.Series([])

for piece in chunker:

tot = tot.add(piece['key'].value\_counts(), fill\_value=0)

tot = tot.sort\_values(ascending=False)

We have then:

tot[:10]

*E 368.0*

*X 364.0*

*L 346.0*

*O 343.0*

*Q 340.0*

*M 338.0*

*J 337.0*

*F 335.0*

*K 334.0*

*H 330.0*

*dtype: float64*

TextParser is also equipped with a get\_chunk method that enables you to read pieces of an arbitrary size.

#### Writing Data to Text Format

Data can also be exported to a delimited format. Let’s consider one of the CSV files read before:

data = pd.read\_csv('examples/ex5.csv')

data Out[42]:

something a b c d message 0 one 1 2 3.0 4 NaN

1 two 5 6 NaN 8 world 2 three 9 10 11.0 12 foo

Using DataFrame’s to\_csv method, we can write the data out to a comma-separated file:

data.to\_csv('examples/out.csv')

!cat examples/out.csv

,something,a,b,c,d,message 0,one,1,2,3.0,4,

1,two,5,6,,8,world 2,three,9,10,11.0,12,foo

Other delimiters can be used, of course (writing to sys.stdout so it prints the text result to the console):

import sys

data.to\_csv(sys.stdout, sep='|')

|something|a|b|c|d|message 0|one|1|2|3.0|4| 1|two|5|6||8|world 2|three|9|10|11.0|12|foo

Missing values appear as empty strings in the output. You might want to denote them by some other sentinel value:

data.to\_csv(sys.stdout, na\_rep='NULL')

,something,a,b,c,d,message 0,one,1,2,3.0,4,NULL

1,two,5,6,NULL,8,world 2,three,9,10,11.0,12,foo

With no other options specified, both the row and column labels are written. Both of these can be disabled:

In [48]: data.to\_csv(sys.stdout, index=False, header=False)

*one,1,2,3.0,4,*

*two,5,6,,8,world*

*three,9,10,11.0,12,foo*

You can also write only a subset of the columns, and in an order of your choosing:

In [49]: data.to\_csv(sys.stdout, index=False, columns=['a', 'b', 'c'])

*a,b,c*

*1,2,3.0*

*5,6,*

*9,10,11.0*

Series also has a to\_csv method:

dates = pd.date\_range('1/1/2000', periods=7)

ts = pd.Series(np.arange(7), index=dates)

ts.to\_csv('examples/tseries.csv')

!cat examples/tseries.csv 2000-01-01,0

*2000-01-02,1*

*2000-01-03,2*

*2000-01-04,3*

*2000-01-05,4*

*2000-01-06,5*

*2000-01-07,6*

#### Working with Delimited Formats

It’s possible to load most forms of tabular data from disk using functions like pan das.read\_table. In some cases, however, some manual processing may be necessary. It’s not uncommon to receive a file with one or more malformed lines that trip up read\_table. To illustrate the basic tools, consider a small CSV file:

!cat examples/ex7.csv "a","b","c"

"1","2","3"

"1","2","3"

For any file with a single-character delimiter, you can use Python’s built-in csv mod‐ ule. To use it, pass any open file or file-like object to csv.reader:

import csv

f = open('examples/ex7.csv')

reader = csv.reader(f)

Iterating through the reader like a file yields tuples of values with any quote charac‐ ters removed:

In [56]: for line in reader:

....: print(line) ['a', 'b', 'c']

['1', '2', '3']

['1', '2', '3']

From there, it’s up to you to do the wrangling necessary to put the data in the form that you need it. Let’s take this step by step. First, we read the file into a list of lines:

with open('examples/ex7.csv') as f:

....: lines = list(csv.reader(f))

Then, we split the lines into the header line and the data lines:

header, values = lines[0], lines[1:]

Then we can create a dictionary of data columns using a dictionary comprehension and the expression zip(\*values), which transposes rows to columns:

data\_dict = {h: v for h, v in zip(header, zip(\*values))}

data\_dict

*{'a': ('1', '1'), 'b': ('2', '2'), 'c': ('3', '3')}*

CSV files come in many different flavors. To define a new format with a different delimiter, string quoting convention, or line terminator, we define a simple subclass of csv.Dialect:

class my\_dialect(csv.Dialect): lineterminator = '\n' delimiter = ';'

quotechar = '"'

quoting = csv.QUOTE\_MINIMAL

reader = csv.reader(f, dialect=my\_dialect)

We can also give individual CSV dialect parameters as keywords to csv.reader without having to define a subclass:

reader = csv.reader(f, delimiter='|')

The possible options (attributes of csv.Dialect) and what they do can be found in [Table 6-3](#_bookmark24).

Table 6-3. CSV dialect options

**Argument Description**

delimiter One-character string to separate fields; defaults to ','.

lineterminator Line terminator for writing; defaults to '\r\n'. Reader ignores this

and recognizes cross-platform line terminators.

quotechar Quote character for fields with special characters (like a delimiter);

default is '"'.

quoting Quoting convention. Options include csv.QUOTE\_ALL (quote all fields), csv.QUOTE\_MINI MAL (only fields with special characters like the delimiter), csv.QUOTE\_NONNUMERIC, and csv.QUOTE\_NONE (no quoting). See Python’s documentation for full details. Defaults to QUOTE\_MINIMAL.

skipinitialspace Ignore whitespace after each delimiter; default is False.

doublequote How to handle quoting character inside a field; if True, it is doubled (see online documentation

for full detail and behavior).

escapechar String to escape the delimiter if quoting is set to csv.QUOTE\_NONE; disabled by default.

For files with more complicated or fixed multicharacter delimiters, you will not be able to use the csv module. In those cases, you’ll have to do the line splitting and other cleanup using string’s split method or the regular expression method re.split.

To write delimited files manually, you can use csv.writer. It accepts an open, writa‐ ble file object and the same dialect and format options as csv.reader:

with open('mydata.csv', 'w') as f:

writer = csv.writer(f, dialect=my\_dialect)

writer.writerow(('one', 'two', 'three'))

writer.writerow(('1', '2', '3'))

writer.writerow(('4', '5', '6'))

writer.writerow(('7', '8', '9'))

#### JSON Data

JSON (short for JavaScript Object Notation) has become one of the standard formats for sending data by HTTP request between web browsers and other applications. It is a much more free-form data format than a tabular text form like CSV. Here is an example:

obj = """

{"name": "Wes",

"places\_lived": ["United States", "Spain", "Germany"], "pet": null,

"siblings": [{"name": "Scott", "age": 30, "pets": ["Zeus", "Zuko"]},

{"name": "Katie", "age": 38,

"pets": ["Sixes", "Stache", "Cisco"]}]

} """

JSON is very nearly valid Python code with the exception of its null value null and some other nuances (such as disallowing trailing commas at the end of lists). The basic types are objects (dicts), arrays (lists), strings, numbers, booleans, and nulls. All of the keys in an object must be strings. There are several Python libraries for reading   
and writing JSON data. I’ll use json here, as it is built into the Python standard library. To convert a JSON string to Python form, use json.loads:

import json

result = json.loads(obj)

result

*{'name': 'Wes', 'pet': None,*

*'places\_lived': ['United States', 'Spain', 'Germany'],*

*'siblings': [{'age': 30, 'name': 'Scott', 'pets': ['Zeus', 'Zuko']},*

*{'age': 38, 'name': 'Katie', 'pets': ['Sixes', 'Stache', 'Cisco']}]}*

json.dumps, on the other hand, converts a Python object back to JSON:

asjson = json.dumps(result)

How you convert a JSON object or list of objects to a DataFrame or some other data structure for analysis will be up to you. Conveniently, you can pass a list of dicts (which were previously JSON objects) to the DataFrame constructor and select a sub‐ set of the data fields:

siblings = pd.DataFrame(result['siblings'], columns=['name', 'age'])

siblings

*name age*

*Scott 30*

*Katie 38*

The pandas.read\_json can automatically convert JSON datasets in specific arrange‐ ments into a Series or DataFrame. For example:

!cat examples/example.json [{"a": 1, "b": 2, "c": 3},

{"a": 4, "b": 5, "c": 6},

{"a": 7, "b": 8, "c": 9}]

The default options for pandas.read\_json assume that each object in the JSON array is a row in the table:

data = pd.read\_json('examples/example.json')

data

*a b c*

*0 1 2 3*

*1 4 5 6*

*2 7 8 9*

For an extended example of reading and manipulating JSON data (including nested records), see the USDA Food Database example in Chapter 7.

If you need to export data from pandas to JSON, one way is to use the to\_json meth‐ ods on Series and DataFrame:

print(data.to\_json())

*{"a":{"0":1,"1":4,"2":7},"b":{"0":2,"1":5,"2":8},"c":{"0":3,"1":6,"2":9}}*

print( data.to\_json(orient='records') )

*[{"a":1,"b":2,"c":3},{"a":4,"b":5,"c":6},{"a":7,"b":8,"c":9}]*

#### XML and HTML: Web Scraping

Python has many libraries for reading and writing data in the ubiquitous HTML and XML formats. Examples include [lxml](http://lxml.de/), Beautiful Soup, and html5lib. While lxml is comparatively much faster in general, the other libraries can better handle malformed HTML or XML files.

pandas has a built-in function, read\_html, which uses libraries like lxml and Beauti‐ ful Soup to automatically parse tables out of HTML files as DataFrame objects. To show how this works, I downloaded an HTML file (used in the pandas documenta‐ tion) from the United States FDIC government agency showing bank failures.[1](#_bookmark25) First, you must install some additional libraries used by read\_html:

conda install lxml

pip install beautifulsoup4 html5lib

If you are not using conda, pip install lxml will likely also work.

The pandas.read\_html function has a number of options, but by default it searches for and attempts to parse all tabular data contained within <table> tags. The result is a list of DataFrame objects:

tables = pd.read\_html('examples/fdic\_failed\_bank\_list.html')

len(tables)

1

failures = tables[0]

failures.head()

Bank Name City ST CERT \

0 Allied Bank Mulberry AR 91

1 The Woodbury Banking Company Woodbury GA 11297

2 First CornerStone Bank King of Prussia PA 35312

1 For the full list, see https://www.fdic.gov/bank/individual/failed/banklist.html.

3 Trust Company Bank Memphis TN 9956

4

5 North Milwaukee State Bank Milwaukee WI 20364

Acquiring Institution Closing Date Updated Date

0 Today's Bank September 23, 2016 November 17, 2016

1 United Bank August 19, 2016 November 17, 2016

2 First-Citizens Bank & Trust Company May 6, 2016 September 6, 2016

3 The Bank of Fayette County April 29, 2016 September 6, 2016

4 First-Citizens Bank & Trust Company March 11, 2016 June 16, 2016

Because failures has many columns, pandas inserts a line break character \.

As you will learn in later chapters, from here we could proceed to do some data cleaning and analysis, like computing the number of bank failures by year:

close\_timestamps = pd.to\_datetime(failures['Closing Date'])

close\_timestamps.dt.year.value\_counts()

*2010 157*

*2009 140*

*2011 92*

*2012 51*

*2008 25*

*...*

*2004 4*

*2001 4*

*2007 3*

*2003 3*

*2000 2*

Name: Closing Date, Length: 15, dtype: int64

##### Parsing XML with lxml.objectify

XML (eXtensible Markup Language) is another common structured data format sup‐ porting hierarchical, nested data with metadata. The book you are currently reading was actually created from a series of large XML documents.

Earlier, I showed the pandas.read\_html function, which uses either lxml or Beautiful Soup under the hood to parse data from HTML. XML and HTML are structurally similar, but XML is more general. Here, I will show an example of how to use lxml to parse data from a more general XML format.

The New York Metropolitan Transportation Authority (MTA) publishes a number of [data series about its bus and train services](http://www.mta.info/developers/download.html). Here we’ll look at the performance data, which is contained in a set of XML files. Each train or bus service has a different file (like Performance\_MNR.xml for the Metro-North Railroad) containing monthly data as a series of XML records that look like this:

<INDICATOR>

<INDICATOR\_SEQ>373889</INDICATOR\_SEQ>

<PARENT\_SEQ></PARENT\_SEQ>

<AGENCY\_NAME>Metro-North Railroad</AGENCY\_NAME>

<INDICATOR\_NAME>Escalator Availability</INDICATOR\_NAME>

<DESCRIPTION>Percent of the time that escalators are operational

systemwide. The availability rate is based on physical observations performed the morning of regular business days only. This is a new indicator the agency began reporting in 2009.</DESCRIPTION>

<PERIOD\_YEAR>2011</PERIOD\_YEAR>

<PERIOD\_MONTH>12</PERIOD\_MONTH>

<CATEGORY>Service Indicators</CATEGORY>

<FREQUENCY>M</FREQUENCY>

<DESIRED\_CHANGE>U</DESIRED\_CHANGE>

<INDICATOR\_UNIT>%</INDICATOR\_UNIT>

<DECIMAL\_PLACES>1</DECIMAL\_PLACES>

<YTD\_TARGET>97.00</YTD\_TARGET>

<YTD\_ACTUAL></YTD\_ACTUAL>

<MONTHLY\_TARGET>97.00</MONTHLY\_TARGET>

<MONTHLY\_ACTUAL></MONTHLY\_ACTUAL>

</INDICATOR>

Using lxml.objectify, we parse the file and get a reference to the root node of the XML file with getroot:

from lxml import objectify

path = 'examples/mta\_perf/Performance\_MNR.xml'

parsed = objectify.parse(open(path))

root = parsed.getroot()

root.INDICATOR returns a generator yielding each <INDICATOR> XML element. For each record, we can populate a dict of tag names (like YTD\_ACTUAL) to data values (excluding a few tags):

data = []

skip\_fields = ['PARENT\_SEQ', 'INDICATOR\_SEQ', 'DESIRED\_CHANGE', 'DECIMAL\_PLACES']

for elt in root.INDICATOR: el\_data = {}

for child in elt.getchildren():

if child.tag in skip\_fields:

continue

el\_data[child.tag] = child.pyval data.append(el\_data)

Lastly, convert this list of dicts into a DataFrame:

perf = pd.DataFrame(data)

perf.head()

Empty DataFrame

Columns: []

Index: []

XML data can get much more complicated than this example. Each tag can have metadata, too. Consider an HTML link tag, which is also valid XML:

from io import StringIO

tag = '<a [href="http://www.google.com">Google</a>'](http://www.google.com/)

root = objectify.parse(StringIO(tag)).getroot()

You can now access any of the fields (like href) in the tag or the link text:

root

*<Element a at 0x7f6b15817748>*

root.get('href')

*'http://www.google.com'*

root.text

'Google'

### Binary Data Formats

One of the easiest ways to store data (also known as serialization) efficiently in binary format is using Python’s built-in pickle serialization. pandas objects all have a to\_pickle method that writes the data to disk in pickle format:

frame = **pd.read\_csv(**'examples/ex1.csv'**)**

**frame**

a b c d message

0 1 2 3 4 hello

1 5 6 7 8 world

2 9 10 11 12 foo

frame.**to\_pickle**('examples/frame\_pickle')

You can read any “pickled” object stored in a file by using the built-in pickle directly, or even more conveniently using pandas.read\_pickle:

pd.**read\_pickle**('examples/frame\_pickle')

a b c d message

0 1 2 3 4 hello

1 5 6 7 8 world

2 9 10 11 12 foo

pickle is only recommended as a short-term storage format. The problem is that it is hard to guarantee that the format will be stable over time; an object pickled today may not unpickle with a later version of a library. We have tried to maintain backward compati‐ bility when possible, but at some point in the future it may be nec‐ essary to “break” the pickle format.

pandas has built-in support for two more binary data formats: HDF5 and Message‐ Pack. I will give some HDF5 examples in the next section, but I encourage you to explore different file formats to see how fast they are and how well they work for your analysis. Some other storage formats for pandas or NumPy data include:

[*bcolz*](http://bcolz.blosc.org/)

A compressable column-oriented binary format based on the Blosc compression library.

[*Feather*](http://github.com/wesm/feather)

A cross-language column-oriented file format I designed with the R program‐ ming community’s [Hadley Wickham](http://hadley.nz/). Feather uses the [Apache Arrow](http://apache.arrow.org/) columnar memory format.

#### Using HDF5 Format

HDF5 is a well-regarded file format intended for storing large quantities of scientific array data. It is available as a C library, and it has interfaces available in many other languages, including Java, Julia, MATLAB, and Python. The “HDF” in HDF5 stands for hierarchical data format. Each HDF5 file can store multiple datasets and support‐ ing metadata. Compared with simpler formats, HDF5 supports on-the-fly compres‐ sion with a variety of compression modes, enabling data with repeated patterns to be stored more efficiently. HDF5 can be a good choice for working with very large data‐ sets that don’t fit into memory, as you can efficiently read and write small sections of much larger arrays.

While it’s possible to directly access HDF5 files using either the PyTables or h5py libraries, pandas provides a high-level interface that simplifies storing Series and DataFrame object. The HDFStore class works like a dict and handles the low-level details:

frame = pd.DataFrame({'a': np.random.randn(100)})

store = pd.HDFStore('mydata.h5')

store['obj1'] = frame

store['obj1\_col'] = frame['a']

store

<class 'pandas.io.pytables.HDFStore'> File path: mydata.h5

/obj1 frame (shape->[100,1])

/obj1\_col series (shape->[100])

/obj2 frame\_table (typ->appendable,nrows->100,ncols->1,indexers-> [index])

/obj3 frame\_table (typ->appendable,nrows->100,ncols->1,indexers-> [index])

Objects contained in the HDF5 file can then be retrieved with the same dict-like API:

store['obj1']

*a*

*0 -0.204708*

*1 0.478943*

*2 -0.519439*

*3 -0.555730*

*4 1.965781*

*.. ...*

*95 0.795253*

*96 0.118110*

*97 -0.748532*

*98 0.584970*

*99 0.152677*

*[100 rows x 1 columns]*

HDFStore supports two storage schemas, 'fixed' and 'table'. The latter is generally slower, but it supports query operations using a special syntax:

store.put('obj2', frame, format='table')

store.select('obj2', where=['index >= 10 and index <= 15']) Out[99]:

*a*

*10 1.007189*

*11 -1.296221*

*12 0.274992*

*13 0.228913*

*14 1.352917*

*15 0.886429*

store.close()

The put is an explicit version of the store['obj2'] = frame method but allows us to set other options like the storage format.

The pandas.read\_hdf function gives you a shortcut to these tools:

frame.to\_hdf('mydata.h5', 'obj3', format='table')

pd.read\_hdf('mydata.h5', 'obj3', where=['index < 5'])

*a*

*0 -0.204708*

*1 0.478943*

*2 -0.519439*

*3 -0.555730*

*4 1.965781*

If you are processing data that is stored on remote servers, like Amazon S3 or HDFS, using a different binary format designed for distributed storage like [Apache Parquet](http://parquet.apache.org/) may be more suitable. Python for Parquet and other such storage formats is still develop‐ ing, so I do not write about them in this book.

If you work with large quantities of data locally, I would encourage you to explore PyTables and h5py to see how they can suit your needs. Since many data analysis problems are I/O-bound (rather than CPU-bound), using a tool like HDF5 can mas‐ sively accelerate your applications.

HDF5 is not a database. It is best suited for write-once, read-many datasets. While data can be added to a file at any time, if multiple writers do so simultaneously, the file can become corrupted.

#### Reading Microsoft Excel Files

pandas also supports reading tabular data stored in Excel 2003 (and higher) files using either the ExcelFile class or pandas.read\_excel function. Internally these tools use the add-on packages xlrd and openpyxl to read XLS and XLSX files, respec‐ tively. You may need to install these manually with pip or conda.

To use ExcelFile, create an instance by passing a path to an xls or xlsx file:

xlsx = pd.**ExcelFile**('examples/ex1.xlsx')

Data stored in a sheet can then be read into DataFrame with parse:

pd.read\_excel(xlsx, 'Sheet1')

a b c d message

0 1 2 3 4 hello

1 5 6 7 8 world

2 9 10 11 12 foo

If you are reading multiple sheets in a file, then it is faster to create the ExcelFile, but you can also simply pass the filename to pandas.read\_excel:

frame = pd.read\_excel('examples/ex1.xlsx', 'Sheet1')

frame

a b c d message

0 1 2 3 4 hello

1 5 6 7 8 world

2 9 10 11 12 foo

To write pandas data to Excel format, you must first create an ExcelWriter, then write data to it using pandas objects’ to\_excel method:

writer = pd.ExcelWriter('examples/ex2.xlsx')

frame.to\_excel(writer, 'Sheet1')

writer.save()

You can also pass a file path to to\_excel and avoid the ExcelWriter:

frame.to\_excel('examples/ex2.xlsx')

Interacting with Web APIs

Many websites have public APIs providing data feeds via JSON or some other format. There are a number of ways to access these APIs from Python; one easy-to-use method that I recommend is the [requests package](http://docs.python-requests.org/).

To find the last 30 GitHub issues for pandas on GitHub, we can make a GET HTTP request using the add-on requests library:

import requests

url = 'https://api.github.com/repos/pandas-dev/pandas/issues'

resp = requests.get(url)

resp

*<Response [200]>*

The Response object’s json method will return a dictionary containing JSON parsed into native Python objects:

data = resp.json()

data[0]['title']

'Period does not round down for frequencies less that 1 hour'

Each element in data is a dictionary containing all of the data found on a GitHub issue page (except for the comments). We can pass data directly to DataFrame and extract fields of interest:

issues = pd.DataFrame(data, columns=['number', 'title',

.....: 'labels', 'state'])

issues

Number title \

0 17666 Period does not round down for frequencies les...

1

2 17665 DOC: improve docstring of function where

3

4 17664 COMPAT: skip 32-bit test on int repr

5

6 17662 implement Delegator class

7

8 17654 BUG: Fix series rename called with str alterin...

9

.. ... ...

25 17603 BUG: Correctly localize naive datetime strings...

26

27 17599 core.dtypes.generic --> cython

28

29 17596 Merge cdate\_range functionality into bdate\_range

30

31 17587 Time Grouper bug fix when applied for list gro...

32

33 17583 BUG: fix tz-aware DatetimeIndex + TimedeltaInd...

34

labels state

0 [] open

1

2 [{'id': 134699, 'url': 'https://api.github.com... open 2

[{'id': 563047854, 'url': 'https://api.github. open

3

3 [] open

4

5 [{'id': 76811, 'url': 'https://api.github.com/. open

6

.. ... ...

25 [{'id': 76811, 'url': 'https://api.github.com/. open

26 [{'id': 49094459, 'url': 'https://api.github.c. open

27 [{'id': 35818298, 'url': 'https://api.github.c. open

28 [{'id': 233160, 'url': 'https://api.github.com. open

29

30 [{'id': 76811, 'url': 'https://api.github.com/. open

31

[30 rows x 4 columns]

With a bit of elbow grease, you can create some higher-level interfaces to common web APIs that return DataFrame objects for easy analysis.

### Interacting with Databases

In a business setting, most data may not be stored in text or Excel files. SQL-based relational databases (such as SQL Server, PostgreSQL, and MySQL) are in wide use, and many alternative databases have become quite popular. The choice of database is usually dependent on the performance, data integrity, and scalability needs of an application.

Loading data from SQL into a DataFrame is fairly straightforward, and pandas has some functions to simplify the process. As an example, I’ll create a SQLite database using Python’s built-in sqlite3 driver:

import sqlite3

query = """

.....: CREATE TABLE test

.....: (a VARCHAR(20), b VARCHAR(20),

.....: c REAL, d INTEGER

.....: );"""

con = sqlite3.connect('mydata.sqlite')

con.execute(query)

*<sqlite3.Cursor at 0x7f6b12a50f10>*

con.commit()

Then, insert a few rows of data:

data = [('Atlanta', 'Georgia', 1.25, 6),

.....: ('Tallahassee', 'Florida', 2.6, 3),

.....: ('Sacramento', 'California', 1.7, 5)]

stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"

con.executemany(stmt, data)

*<sqlite3.Cursor at 0x7f6b15c66ce0>*

con.commit()

Most Python SQL drivers (PyODBC, psycopg2, MySQLdb, pymssql, etc.) return a list of tuples when selecting data from a table:

cursor = con.execute('select \* from test')

rows = cursor.fetchall()

rows

*[('Atlanta', 'Georgia', 1.25, 6),*

*('Tallahassee', 'Florida', 2.6, 3),*

*('Sacramento', 'California', 1.7, 5)]*

You can pass the list of tuples to the DataFrame constructor, but you also need the column names, contained in the cursor’s description attribute:

cursor.description

(('a', None, None, None, None, None, None),

('b', None, None, None, None, None, None),

('c', None, None, None, None, None, None),

('d', None, None, None, None, None, None))

pd.DataFrame(rows, columns=[x[0] for x in cursor.description])

a b c d

0 Atlanta Georgia 1.25 6

1 Tallahassee Florida 2.60 3

2 Sacramento California 1.70 5

This is quite a bit of munging that you’d rather not repeat each time you query the database. The [SQLAlchemy project](http://www.sqlalchemy.org/) is a popular Python SQL toolkit that abstracts away many of the common differences between SQL databases. pandas has a read\_sql function that enables you to read data easily from a general SQLAlchemy connection. Here, we’ll connect to the same SQLite database with SQLAlchemy and read data from the table created before:

import sqlalchemy as sqla

db = sqla.create\_engine('sqlite:///mydata.sqlite')

pd.read\_sql('select \* from test', db)

a b c d

0 Atlanta Georgia 1.25 6

1 Tallahassee Florida 2.60 3

2 Sacramento California 1.70 5

### Conclusion

Getting access to data is frequently the first step in the data analysis process. We have looked at a number of useful tools in this chapter that should help you get started. In the upcoming chapters we will dig deeper into data wrangling, data visualization, time series analysis, and other topics.

Data Cleaning and Preparation

During the course of doing data analysis and modeling, a significant amount of time is spent on data preparation: loading, cleaning, transforming, and rearranging. Such tasks are often reported to take up 80% or more of an analyst’s time. Sometimes the way that data is stored in files or databases is not in the right format for a particular task. Many researchers choose to do ad hoc processing of data from one form to another using a general-purpose programming language, like Python, Perl, R, or Java, or Unix text-processing tools like sed or awk. Fortunately, pandas, along with the built-in Python language features, provides you with a high-level, flexible, and fast set of tools to enable you to manipulate data into the right form.

If you identify a type of data manipulation that isn’t anywhere in this book or else‐ where in the pandas library, feel free to share your use case on one of the Python mailing lists or on the pandas GitHub site. Indeed, much of the design and imple‐ mentation of pandas has been driven by the needs of real-world applications.

In this chapter I discuss tools for missing data, duplicate data, string manipulation, and some other analytical data transformations. In the next chapter, I focus on com‐ bining and rearranging datasets in various ways.

### Handling Missing Data

Missing data occurs commonly in many data analysis applications. One of the goals of pandas is to make working with missing data as painless as possible. For example, all of the descriptive statistics on pandas objects exclude missing data by default.

The way that missing data is represented in pandas objects is somewhat imperfect, but it is functional for a lot of users. For numeric data, pandas uses the floating-point value NaN (Not a Number) to represent missing data. We call this a sentinel value that can be easily detected:

string\_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado'])

string\_data

0 aardvark

1

2 artichoke

3

4 NaN

5

6 avocado dtype: object

7

string\_data**.isnull()**

0 False

1

2 False

3

4 True

5

6 False dtype: bool

In pandas, we’ve adopted a convention used in the R programming language by refer‐ ring to missing data as NA, which stands for not available. In statistics applications, NA data may either be data that does not exist or that exists but was not observed (through problems with data collection, for example). When cleaning up data for analysis, it is often important to do analysis on the missing data itself to identify data collection problems or potential biases in the data caused by missing data.

The built-in Python None value is also treated as NA in object arrays:

string\_data[0] = None

string\_data.isnull()

0 True

1

2 False

3

4 True

5

6 False dtype: bool

7

There is work ongoing in the pandas project to improve the internal details of how missing data is handled, but the user API functions, like pandas.isnull, abstract away many of the annoying details. See [Table 7-1](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark0) for a list of some functions related to missing data handling.

Table 7-1. NA handling methods

Argument Description

dropna Filter axis labels based on whether values for each label have missing data, with varying thresholds

for how much missing data to tolerate.

fillna Fill in missing data with some value or using an interpolation method such as 'ffill' or 'bfill'. isnull

Return boolean values indicating which values are missing/NA.

notnull Negation of isnull.

#### Filtering Out Missing Data

There are a few ways to filter out missing data. While you always have the option to do it by hand using pandas.isnull and boolean indexing, the dropna can be helpful. On a Series, it returns the Series with only the non-null data and index values:

from numpy import nan as NA

data = pd.Series([1, NA, 3.5, NA, 7])

data.dropna()

*0 1.0*

*2 3.5*

*4 7.0*

*dtype: float64*

This is equivalent to:

data[data.notnull()]

*0 1.0*

*2 3.5*

*4 7.0*

*dtype: float64*

With DataFrame objects, things are a bit more complex. You may want to drop rows or columns that are all NA or only those containing any NAs. dropna by default drops any row containing a missing value:

data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],

....: [NA, NA, NA], [NA, 6.5, 3.]])

cleaned = data.dropna()

data

0 1 2

0 1.0 6.5 3.0

1 1.0 NaN NaN

2 NaN NaN NaN

3 NaN 6.5 3.0

cleaned

*0 1 2*

*0 1.0 6.5 3.0*

Passing how='all' will only drop rows that are all NA:

data.dropna(how='all')

*0 1 2*

*0 1.0 6.5 3.0*

*1 1.0 NaN NaN*

*3 NaN 6.5 3.0*

To drop columns in the same way, pass axis=1:

data[4] = NA

data

*0 1 2 4*

*0 1.0 6.5 3.0 NaN*

*1 1.0 NaN NaN NaN*

*2*

*3 NaN NaN NaN NaN*

*3 NaN 6.5 3.0 NaN*

*4*

data.dropna(axis=1, how='all')

0 1 2

0 1.0 6.5 3.0

1 1.0 NaN NaN

2 NaN NaN NaN

3 NaN 6.5 3.0

A related way to filter out DataFrame rows tends to concern time series data. Suppose you want to keep only rows containing a certain number of observations. You can indicate this with the thresh argument:

df = pd.DataFrame(np.random.randn(7, 3))

df.iloc[:4, 1] = NA

df.iloc[:2, 2] = NA

df

*0 1 2*

*0 -0.204708 NaN NaN*

*1 -0.555730 NaN NaN*

*2 0.092908 NaN 0.769023*

*3 1.246435 NaN -1.296221*

*4 0.274992 0.228913 1.352917*

*5 0.886429 -2.001637 -0.371843*

*6 1.669025 -0.438570 -0.539741*

df.dropna()

*0 1 2*

*4 0.274992 0.228913 1.352917*

*5 0.886429 -2.001637 -0.371843*

*6 1.669025 -0.438570 -0.539741*

df**.dropna(thresh=2)**

*0 1 2*

*2 0.092908 NaN 0.769023*

*3 1.246435 NaN -1.296221*

*4 0.274992 0.228913 1.352917*

*5 0.886429 -2.001637 -0.371843*

*6 1.669025 -0.438570 -0.539741*

#### Filling In Missing Data

Rather than filtering out missing data (and potentially discarding other data along with it), you may want to fill in the “holes” in any number of ways. For most pur‐ poses, the fillna method is the workhorse function to use. Calling fillna with a constant replaces missing values with that value:

df.fillna(0)

0 1 2

0 -0.204708 0.000000 0.000000

1 -0.555730 0.000000 0.000000

2 0.092908 0.000000 0.769023

3 1.246435 0.000000 -1.296221

4 0.274992 0.228913 1.352917

5 0.886429 -2.001637 -0.371843

6 1.669025 -0.438570 -0.539741

Calling fillna with a dict, you can use a different fill value for each column:

df.fillna({1: 0.5, 2: 0})

0 1 2

0 -0.204708 0.500000 0.000000

1 -0.555730 0.500000 0.000000

2 0.092908 0.500000 0.769023

3 1.246435 0.500000 -1.296221

4 0.274992 0.228913 1.352917

5 0.886429 -2.001637 -0.371843

6 1.669025 -0.438570 -0.539741

fillna returns a new object, but you can modify the existing object in-place:

\_ = df.fillna(0, inplace=True)

df

0 1 2

0 -0.204708 0.000000 0.000000

1 -0.555730 0.000000 0.000000

2 0.092908 0.000000 0.769023

3 1.246435 0.000000 -1.296221

4 0.274992 0.228913 1.352917

5 0.886429 -2.001637 -0.371843

6 1.669025 -0.438570 -0.539741

The same interpolation methods available for reindexing can be used with fillna:

df = pd.DataFrame(np.random.randn(6, 3))

df.iloc[2:, 1] = NA

df.iloc[4:, 2] = NA

df

*0 1 2*

*0 0.476985 3.248944 -1.021228*

*1 -0.577087 0.124121 0.302614*

*2 0.523772 NaN 1.343810*

*3 -0.713544 NaN -2.370232*

*4 -1.860761 NaN NaN*

*5 -1.265934 NaN NaN*

df.fillna(method='ffill')

*0 1 2*

*0 0.476985 3.248944 -1.021228*

*1 -0.577087 0.124121 0.302614*

*2 0.523772 0.124121 1.343810*

*3 -0.713544 0.124121 -2.370232*

*4 -1.860761 0.124121 -2.370232*

*5 -1.265934 0.124121 -2.370232*

df.fillna(method='ffill', limit=2)

*0 1 2*

*0 0.476985 3.248944 -1.021228*

*1 -0.577087 0.124121 0.302614*

*2 0.523772 0.124121 1.343810*

*3 -0.713544 0.124121 -2.370232*

*4 -1.860761 NaN -2.370232*

*5 -1.265934 NaN -2.370232*

With fillna you can do lots of other things with a little creativity. For example, you might pass the mean or median value of a Series:

data = pd.Series([1., NA, 3.5, NA, 7])

data.fillna(data.mean())

*0 1.000000*

*1 3.833333*

*2 3.500000*

*3 3.833333*

*4 7.000000*

*dtype: float64*

See Table 7-2 for a reference on fillna.

Table 7-2. fillna function arguments

**Argument Description**

**value** Scalar value or dict-like object to use to fill missing values

**method** Interpolation; by default 'ffill' if function called with no other arguments

**axis** Axis to fill on; default axis=0

**inplace** Modify the calling object without producing a copy

**limit** For forward and backward filling, maximum number of consecutive periods to fill

### Data Transformation

So far in this chapter we’ve been concerned with rearranging data. Filtering, cleaning, and other transformations are another class of important operations.

#### Removing Duplicates

Duplicate rows may be found in a DataFrame for any number of reasons. Here is an example:

data = pd.DataFrame({'k1': ['one', 'two'] \* 3 + ['two'],

....: 'k2': [1, 1, 2, 3, 3, 4, 4]})

data

k1 k2

0 one 1

1 two 1

2 one 2

3 two 3

4 one 3

5 two 4

6 two 4

The DataFrame method duplicated returns a boolean Series indicating whether each row is a duplicate (has been observed in a previous row) or not:

data.duplicated()

0 False

1

2 False

3

4 False

5

6 False

7

8 False

9

10 False

11

12 True dtype: bool

13

Relatedly, drop\_duplicates returns a DataFrame where the duplicated array is False:

data.drop\_duplicates()

k1 k2

0 one 1

1 two 1

2 one 2

3 two 3

4 one 3

5 two 4

Both of these methods by default consider all of the columns; alternatively, you can specify any subset of them to detect duplicates. Suppose we had an additional column of values and wanted to filter duplicates only based on the 'k1' column:

data['v1'] = range(7)

data.drop\_duplicates(['k1'])

k1 k2 v1

0 one 1 0

1 two 1 1

duplicated and drop\_duplicates by default keep the first observed value combina‐ tion. Passing keep='last' will return the last one:

data.drop\_duplicates(['k1', 'k2'], keep='last')

*k1 k2 v1*

*0 one 1 0*

*1*

*2 two 1 1*

*3*

*4 one 2 2*

*5*

*6 two 3 3*

*7*

*8 one 3 4*

*9*

*6 two 4 6*

#### Transforming Data Using a Function or Mapping

For many datasets, you may wish to perform some transformation based on the val‐ ues in an array, Series, or column in a DataFrame. Consider the following hypotheti‐ cal data collected about various kinds of meat:

data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon',

....: 'Pastrami', 'corned beef', 'Bacon',

....: 'pastrami', 'honey ham', 'nova lox'],

....: 'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})

data

food ounces

0 bacon 4.0

1 pulled pork 3.0

2 bacon 12.0

3 Pastrami 6.0

4 corned beef 7.5

5 Bacon 8.0

6 pastrami 3.0

7 honey ham 5.0

8 nova lox 6.0

Suppose you wanted to add a column indicating the type of animal that each food came from. Let’s write down a mapping of each distinct meat type to the kind of animal:

meat\_to\_animal = { 'bacon': 'pig',

'pulled pork': 'pig',

'pastrami': 'cow',

'corned beef': 'cow',

'honey ham': 'pig', 'nova lox': 'salmon'

}

The map method on a Series accepts a function or dict-like object containing a map‐ ping, but here we have a small problem in that some of the meats are capitalized and others are not. Thus, we need to convert each value to lowercase using the str.lower Series method:

lowercased = data['food'].str.lower()

lowercased

*0 bacon*

*1*

*2 pulled pork*

*3*

*4 bacon*

*5*

*6 pastrami*

*7*

*8 corned beef*

*9*

*10 bacon*

*11*

*12 pastrami*

*13*

*14 honey ham*

*15*

*16 nova lox*

*17*

*Name: food, dtype: object*

data['animal'] = lowercased.map(meat\_to\_animal)

data

food ounces animal

0 bacon 4.0 pig

1 pulled pork 3.0 pig

2 bacon 12.0 pig

3 Pastrami 6.0 cow

4 corned beef 7.5 cow

5 Bacon 8.0 pig

6 pastrami 3.0 cow

7 honey ham 5.0 pig

8 nova lox 6.0 salmon

We could also have passed a function that does all the work:

data['food'].map(lambda x: meat\_to\_animal[x.lower()])

*0 pig*

*1*

*2 pig*

*3*

*4 pig*

*5*

*6 cow*

*7*

*8 cow*

*9*

*10 pig*

*11*

*12 cow*

*13*

*14 pig*

*15*

*16 salmon*

*17*

*Name: food, dtype: object*

Using map is a convenient way to perform element-wise transformations and other data cleaning–related operations.

#### Replacing Values

Filling in missing data with the fillna method is a special case of more general value replacement. As you’ve already seen, map can be used to modify a subset of values in an object but replace provides a simpler and more flexible way to do so. Let’s con‐ sider this Series:

data = pd.Series([1., -999., 2., -999., -1000., 3.])

data

*0 1.0*

*1 -999.0*

*2 2.0*

*3 -999.0*

*4 -1000.0*

*5 3.0*

*dtype: float64*

The -999 values might be sentinel values for missing data. To replace these with NA values that pandas understands, we can use replace, producing a new Series (unless you pass inplace=True):

data.replace(-999, np.nan)

*0 1.0*

*1 NaN*

*2 2.0*

*3 NaN*

*4 -1000.0*

*5 3.0*

*dtype: float64*

If you want to replace multiple values at once, you instead pass a list and then the substitute value:

data.replace([-999, -1000], np.nan)

*0 1.0*

*1 NaN*

*2 2.0*

*3 NaN*

*4*

*5 NaN*

*6*

*5 3.0*

dtype: float64

To use a different replacement for each value, pass a list of substitutes:

data.replace([-999, -1000], [np.nan, 0])

*0 1.0*

*1 NaN*

*2 2.0*

*3 NaN*

*4 0.0*

*5 3.0*

*dtype: float64*

The argument passed can also be a dict:

data.replace({-999: np.nan, -1000: 0})

*0 1.0*

*1 NaN*

*2 2.0*

*3 NaN*

*4 0.0*

*5 3.0*

*dtype: float64*

The data.replace method is distinct from data.str.replace, which performs string substitution element-wise. We look at these string methods on Series later in the chapter.

#### Renaming Axis Indexes

Like values in a Series, axis labels can be similarly transformed by a function or map‐ ping of some form to produce new, differently labeled objects. You can also modify the axes in-place without creating a new data structure. Here’s a simple example:

In [66]: data = pd.DataFrame(np.arange(12).reshape((3, 4)),

....: index=['Ohio', 'Colorado', 'New York'],

....: columns=['one', 'two', 'three', 'four'])

Like a Series, the axis indexes have a map method:

transform = lambda x: x[:4].upper()

data.index.map(transform)

Index(['OHIO', 'COLO', 'NEW '], dtype='object')

You can assign to index, modifying the DataFrame in-place:

data.index = data.index.map(transform)

data

one two three four

OHIO 0 1 2 3

COLO 4 5 6 7

NEW 8 9 10 11

If you want to create a transformed version of a dataset without modifying the origi‐ nal, a useful method is rename:

data.rename(index=str.title, columns=str.upper)

ONE TWO THREE FOUR

Ohio 0 1 2 3

Colo 4 5 6 7

New 8 9 10 11

Notably, rename can be used in conjunction with a dict-like object providing new val‐ ues for a subset of the axis labels:

data.rename(index={'OHIO': 'INDIANA'},

....: columns={'three': 'peekaboo'})

one two peekaboo four

INDIANA 0 1 2 3

COLO 4 5 6 7

NEW 8 9 10 11

rename saves you from the chore of copying the DataFrame manually and assigning to its index and columns attributes. Should you wish to modify a dataset in-place, pass inplace=True:

data.rename(index={'OHIO': 'INDIANA'}, inplace=True)

data

one two three four

INDIANA 0 1 2 3

COLO 4 5 6 7

NEW 8 9 10 11

Discretization and Binning

Continuous data is often discretized or otherwise separated into “bins” for analysis. Suppose you have data about a group of people in a study, and you want to group them into discrete age buckets:

ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]

Let’s divide these into bins of 18 to 25, 26 to 35, 36 to 60, and finally 61 and older. To do so, you have to use cut, a function in pandas:

bins = [18, 25, 35, 60, 100]

cats = pd.cut(ages, bins)

cats

*[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100], (35,*

*60], (35, 60], (25, 35]]*

*Length: 12*

*Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]*

The object pandas returns is a special Categorical object. The output you see describes the bins computed by pandas.cut. You can treat it like an array of strings indicating the bin name; internally it contains a categories array specifying the dis‐ tinct category names along with a labeling for the ages data in the codes attribute:

cats.codes

*array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)*

cats.categories

*IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]]*

*closed='right', dtype='interval[int64]')*

pd.value\_counts(cats)

*(18, 25] 5*

*(35, 60] 3*

*(25, 35] 3*

*(60, 100] 1*

*dtype: int64*

Note that pd.value\_counts(cats) are the bin counts for the result of pandas.cut.

Consistent with mathematical notation for intervals, a parenthesis means that the side is open, while the square bracket means it is closed (inclusive). You can change which side is closed by passing right=False:

In [82]: pd.cut(ages, [18, 26, 36, 61, 100], right=False)

*[[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), ..., [26, 36), [61, 100), [36,*

*61), [36, 61), [26, 36)]*

Length: 12

Categories (4, interval[int64]): [[18, 26) < [26, 36) < [36, 61) < [61, 100)]

You can also pass your own bin names by passing a list or array to the labels option:

group\_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']

pd.cut(ages, bins, labels=group\_names)

*[Youth, Youth, Youth, YoungAdult, Youth, ..., YoungAdult, Senior, MiddleAged, Mid dleAged, YoungAdult]*

*Length: 12*

*Categories (4, object): [Youth < YoungAdult < MiddleAged < Senior]*

If you pass an integer number of bins to cut instead of explicit bin edges, it will com‐ pute equal-length bins based on the minimum and maximum values in the data. Consider the case of some uniformly distributed data chopped into fourths:

data = np.random.rand(20)

pd.cut(data, 4, precision=2)

*[(0.34, 0.55], (0.34, 0.55], (0.76, 0.97], (0.76, 0.97], (0.34, 0.55], ..., (0.34*

*, 0.55], (0.34, 0.55], (0.55, 0.76], (0.34, 0.55], (0.12, 0.34]]*

*Length: 20*

*Categories (4, interval[float64]): [(0.12, 0.34] < (0.34, 0.55] < (0.55, 0.76] <*

*(0.76, 0.97]]*

The precision=2 option limits the decimal precision to two digits.

A closely related function, qcut, bins the data based on sample quantiles. Depending on the distribution of the data, using cut will not usually result in each bin having the same number of data points. Since qcut uses sample quantiles instead, by definition you will obtain roughly equal-size bins:

data = np.random.randn(1000) # Normally distributed

cats = pd.qcut(data, 4) # Cut into quartiles

cats

[(-0.0265, 0.62], (0.62, 3.928], (-0.68, -0.0265], (0.62, 3.928], (-0.0265, 0.62]

, ..., (-0.68, -0.0265], (-0.68, -0.0265], (-2.95, -0.68], (0.62, 3.928], (-0.68,

-0.0265]]

Length: 1000

Categories (4, interval[float64]): [(-2.95, -0.68] < (-0.68, -0.0265] < (-0.0265,

0.62] < (0.62, 3.928]]

pd.value\_counts(cats)

*(0.62, 3.928] 250*

*(-0.0265, 0.62] 250*

*(-0.68, -0.0265] 250*

*(-2.95, -0.68] 250*

*dtype: int64*

Similar to cut you can pass your own quantiles (numbers between 0 and 1, inclusive):

pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])

*[(-0.0265, 1.286], (-0.0265, 1.286], (-1.187, -0.0265], (-0.0265, 1.286], (-0.026*

*5, 1.286], ..., (-1.187, -0.0265], (-1.187, -0.0265], (-2.95, -1.187], (-0.0265,*

*1.286], (-1.187, -0.0265]]*

*Length: 1000*

Categories (4, interval[float64]): [(-2.95, -1.187] < (-1.187, -0.0265] < (-0.026

5, 1.286] < (1.286, 3.928]]

We’ll return to cut and qcut later in the chapter during our discussion of aggregation and group operations, as these discretization functions are especially useful for quan‐ tile and group analysis.

#### Detecting and Filtering Outliers

Filtering or transforming outliers is largely a matter of applying array operations. Consider a DataFrame with some normally distributed data:

data = pd.DataFrame(np.random.randn(1000, 4))

data.describe()

0 1 2 3

count 1000.000000 1000.000000 1000.000000 1000.000000

mean 0.049091 0.026112 -0.002544 -0.051827

std 0.996947 1.007458 0.995232 0.998311

min -3.645860 -3.184377 -3.745356 -3.428254

25% -0.599807 -0.612162 -0.687373 -0.747478

50% 0.047101 -0.013609 -0.022158 -0.088274

75% 0.756646 0.695298 0.699046 0.623331

max 2.653656 3.525865 2.735527 3.366626

Suppose you wanted to find values in one of the columns exceeding 3 in absolute value:

col = data[2]

col[np.abs(col) > 3]

41 -3.399312

136 -3.745356

Name: 2, dtype: float64

To select all rows having a value exceeding 3 or –3, you can use the any method on a boolean DataFrame:

data[(np.abs(data) > 3).any(1)]

0 1 2 3

41 0.457246 -0.025907 -3.399312 -0.974657

60 1.951312 3.260383 0.963301 1.201206

136 0.508391 -0.196713 -3.745356 -1.520113

235 -0.242459 -3.056990 1.918403 -0.578828

258 0.682841 0.326045 0.425384 -3.428254

322 1.179227 -3.184377 1.369891 -1.074833

544 -3.548824 1.553205 -2.186301 1.277104

635 -0.578093 0.193299 1.397822 3.366626

782 -0.207434 3.525865 0.283070 0.544635

803 -3.645860 0.255475 -0.549574 -1.907459

Values can be set based on these criteria. Here is code to cap values outside the inter‐ val –3 to 3:

data[np.abs(data) > 3] = np.sign(data) \* 3

data.describe()

0 1 2 3

count 1000.000000 1000.000000 1000.000000 1000.000000

mean 0.050286 0.025567 -0.001399 -0.051765

std 0.992920 1.004214 0.991414 0.995761

min -3.000000 -3.000000 -3.000000 -3.000000

25% -0.599807 -0.612162 -0.687373 -0.747478

50% 0.047101 -0.013609 -0.022158 -0.088274

75% 0.756646 0.695298 0.699046 0.623331

max 2.653656 3.000000 2.735527 3.000000

The statement np.sign(data) produces 1 and –1 values based on whether the values in data are positive or negative:

np.sign(data).head()

0 1 2 3

0 -1.0 1.0 -1.0 1.0

1 1.0 -1.0 1.0 -1.0

2 1.0 1.0 1.0 -1.0

3 -1.0 -1.0 1.0 -1.0

4 -1.0 1.0 -1.0 -1.0

#### Permutation and Random Sampling

Permuting (randomly reordering) a Series or the rows in a DataFrame is easy to do using the numpy.random.permutation function. Calling permutation with the length of the axis you want to permute produces an array of integers indicating the new ordering:

df = pd.DataFrame(np.arange(5 \* 4).reshape((5, 4)))

sampler = np.random.permutation(5)

sampler

*array([3, 1, 4, 2, 0])*

That array can then be used in iloc-based indexing or the equivalent take function:

df

*0 1 2 3*

*0 0 1 2 3*

*1 4 5 6 7*

*2 8 9 10 11*

*3 12 13 14 15*

*4 16 17 18 19*

df.take(sampler)

*0 1 2 3*

*3 12 13 14 15*

*1 4 5 6 7*

*4 16 17 18 19*

*2 8 9 10 11*

*0 0 1 2 3*

To select a random subset without replacement, you can use the sample method on Series and DataFrame:

df.sample(n=3)

0 1 2 3

*3 12 13 14 15*

*4 16 17 18 19*

*2 8 9 10 11*

To generate a sample with replacement (to allow repeat choices), pass replace=True

to sample:

choices = pd.Series([5, 7, -1, 6, 4])

draws = choices.sample(n=10, replace=True)

draws

*4 4*

*1 7*

*4 4*

*2 -1*

*0 5*

*3 6*

*1 7*

*4 4*

*0 5*

*4 4*

*dtype: int64*

#### Computing Indicator/Dummy Variables

Another type of transformation for statistical modeling or machine learning applica‐ tions is converting a categorical variable into a “dummy” or “indicator” matrix. If a column in a DataFrame has k distinct values, you would derive a matrix or Data‐ Frame with k columns containing all 1s and 0s. pandas has a get\_dummies function for doing this, though devising one yourself is not difficult. Let’s return to an earlier example DataFrame:

df = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],

.....: 'data1': range(6)})

pd.get\_dummies(df['key'])

*a b c*

*0 0 1 0*

*1 0 1 0*

*2 1 0 0*

*3 0 0 1*

*4 1 0 0*

*5 0 1 0*

In some cases, you may want to add a prefix to the columns in the indicator Data‐ Frame, which can then be merged with the other data. get\_dummies has a prefix argu‐ ment for doing this:

dummies = pd.get\_dummies(df['key'], prefix='key')

df\_with\_dummy = df[['data1']].join(dummies)

**df\_with\_dummy**

data1 key\_a key\_b key\_c

0 0 0 1 0

1 1 0 1 0

2 2 1 0 0

3 3 0 0 1

4 4 1 0 0

5 5 0 1 0

If a row in a DataFrame belongs to multiple categories, things are a bit more compli‐ cated. Let’s look at the MovieLens 1M dataset, which is investigated in more detail in Chapter 14:

mnames = ['movie\_id', 'title', 'genres']

movies = pd.read\_table('datasets/movielens/movies.dat', sep='::',

.....: header=None, names=mnames)

movies[:10]

movie\_id title genres

0 1 Toy Story (1995) Animation|Children's|Comedy

1

2 2 Jumanji (1995) Adventure|Children's|Fantasy

3

4 3 Grumpier Old Men (1995) Comedy|Romance

5

6 4 Waiting to Exhale (1995) Comedy|Drama

7

8 5 Father of the Bride Part II (1995) Comedy

9

10 6 Heat (1995) Action|Crime|Thriller

11

12 7 Sabrina (1995) Comedy|Romance

13

14 8 Tom and Huck (1995) Adventure|Children's

15

16 9 Sudden Death (1995) Action

17

18 10 GoldenEye (1995) Action|Adventure|Thriller

19

Adding indicator variables for each genre requires a little bit of wrangling. First, we extract the list of unique genres in the dataset:

all\_genres = []

for x in movies.genres:

.....: all\_genres.extend(x.split('|'))

genres = pd.unique(all\_genres)

Now we have:

genres

*array(['Animation', "Children's", 'Comedy', 'Adventure', 'Fantasy', 'Romance', 'Drama', 'Action', 'Crime', 'Thriller', 'Horror',*

*'Sci-Fi', 'Documentary', 'War', 'Musical', 'Mystery', 'Film-Noir', 'Western'], dtype=object)*

One way to construct the indicator DataFrame is to start with a DataFrame of all zeros:

zero\_matrix = np.zeros((len(movies), len(genres)))

dummies = pd.DataFrame(zero\_matrix, columns=genres)

Now, iterate through each movie and set entries in each row of dummies to 1. To do this, we use the dummies.columns to compute the column indices for each genre:

gen = movies.genres[0]

gen.split('|')

*['Animation', "Children's", 'Comedy']*

dummies.columns.get\_indexer(gen.split('|'))

*array([0, 1, 2])*

Then, we can use .iloc to set values based on these indices:

for i, gen in enumerate(movies.genres):

.....: indices = dummies.columns.get\_indexer(gen.split('|'))

.....: dummies.iloc[i, indices] = 1

.....:

Then, as before, you can combine this with movies:

movies\_windic = **movies.join(dummies.add\_prefix('Genre\_'))**

movies\_windic.iloc[0]

movie\_id 1

title Toy Story (1995)

genres Animation|Children's|Comedy

Genre\_Animation 1

Genre\_Children's 1

Genre\_Comedy 1

Genre\_Adventure 0

Genre\_Fantasy 0

Genre\_Romance 0

Genre\_Drama 0

...

Genre\_Crime 0

Genre\_Thriller 0

Genre\_Horror 0

Genre\_Sci-Fi 0

Genre\_Documentary 0

Genre\_War 0

Genre\_Musical 0

Genre\_Mystery 0

Genre\_Film-Noir 0

Genre\_Western 0

Name: 0, Length: 21, dtype: object

For much larger data, this method of constructing indicator vari‐ ables with multiple membership is not especially speedy. It would be better to write a lower-level function that writes directly to a NumPy array, and then wrap the result in a DataFrame.

A useful recipe for statistical applications is to combine get\_dummies with a discreti‐ zation function like cut:

np.random.seed(12345)

values = np.random.rand(10)

values

*array([ 0.9296, 0.3164, 0.1839, 0.2046, 0.5677, 0.5955, 0.9645, 0.6532, 0.7489, 0.6536])*

bins = [0, 0.2, 0.4, 0.6, 0.8, 1]

pd.get\_dummies(pd.cut(values, bins))

*(0.0, 0.2] (0.2, 0.4] (0.4, 0.6] (0.6, 0.8] (0.8, 1.0]*

*0 0 0 0 0 1*

*1 0 1 0 0 0*

*2 1 0 0 0 0*

*3 0 1 0 0 0*

*4 0 0 1 0 0*

*5 0 0 1 0 0*

*6 0 0 0 0 1*

*7 0 0 0 1 0*

*8 0 0 0 1 0*

*9 0 0 0 1 0*

We set the random seed with numpy.random.seed to make the example deterministic. We will look again at pandas.get\_dummies later in the book.

### String Manipulation

Python has long been a popular raw data manipulation language in part due to its ease of use for string and text processing. Most text operations are made simple with the string object’s built-in methods. For more complex pattern matching and text manipulations, regular expressions may be needed. pandas adds to the mix by ena‐ bling you to apply string and regular expressions concisely on whole arrays of data, additionally handling the annoyance of missing data.

#### String Object Methods

In many string munging and scripting applications, built-in string methods are suffi‐ cient. As an example, a comma-separated string can be broken into pieces with split:

val = 'a,b, guido'

val.split(',')

*['a', 'b', ' guido']*

split is often combined with strip to trim whitespace (including line breaks):

pieces = [x.strip() for x in val.split(',')]

pieces

*['a', 'b', 'guido']*

These substrings could be concatenated together with a two-colon delimiter using addition:

first, second, third = pieces

first + '::' + second + '::' + third

*'a::b::guido'*

But this isn’t a practical generic method. A faster and more Pythonic way is to pass a list or tuple to the join method on the string '::':

'::'.join(pieces)

'a::b::guido'

Other methods are concerned with locating substrings. Using Python’s in keyword is the best way to detect a substring, though index and find can also be used:

'guido' in val

*True*

val.index(',')

*1*

val.find(':')

*-1*

Note the difference between find and index is that index raises an exception if the string isn’t found (versus returning –1):

val.index(':')

---------------------------------------------------------------------------

*ValueError Traceback (most recent call last)*

*<ipython-input-144-280f8b2856ce> in <module>()*

*----> 1 val.index(':')*

*ValueError: substring not found*

Relatedly, count returns the number of occurrences of a particular substring:

val.count(',')

*2*

replace will substitute occurrences of one pattern for another. It is commonly used to delete patterns, too, by passing an empty string:

val.replace(',', '::')

*'a::b:: guido'*

val.replace(',', '')

*'ab guido'*

**See** [**Table 7-3**](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark2) **for a listing of some of Python’s string methods.**

Regular expressions can also be used with many of these operations, as you’ll see.

Table 7-3. Python built-in string methods

Argument Description

**count** Return the number of non-overlapping occurrences of substring in the string.

**endswith** Returns True if string ends with suffix.

**startswith** Returns True if string starts with prefix.

**join** Use string as delimiter for concatenating a sequence of other strings.

**index** Return position of first character in substring if found in the string; raises ValueError if not found.

**find** Return position of first character of first occurrence of substring in the string; like index,

but returns –1 if not found.

**rfind** Return position of first character of last occurrence of substring in the string; returns –1 if not found.

**replace** Replace occurrences of string with another string.

**strip**, **rstrip**, Trim whitespace, including newlines; equivalent to x.strip() (and rstrip, lstrip, respectively)

**lstrip** for each element.

**split** Break string into list of substrings using passed delimiter.

**lower** Convert alphabet characters to lowercase.

**upper** Convert alphabet characters to uppercase.

**casefold** Convert characters to lowercase, and convert any region-specific variable character combinations

to a common comparable form.

**ljust, rjust** Left justify or right justify, respectively; pad opposite side of string with spaces (or some other fill

character) to return a string with a minimum width.

#### Regular Expressions

Regular expressions provide a flexible way to search or match (often more complex) string patterns in text. A single expression, commonly called a regex, is a string formed according to the regular expression language. Python’s built-in re module is responsible for applying regular expressions to strings; I’ll give a number of examples of its use here.

The art of writing regular expressions could be a chapter of its own and thus is outside the book’s scope. There are many excellent tuto‐ rials and references available on the internet and in other books.

The re module functions fall into three categories: pattern matching, substitution, and splitting. Naturally these are all related; a regex describes a pattern to locate in the text, which can then be used for many purposes. Let’s look at a simple example:

suppose we wanted to split a string with a variable number of whitespace characters (tabs, spaces, and newlines). The regex describing one or more whitespace characters is \s+:

import re

text = "foo bar\t baz \tqux"

re.split('\s+', text)

*['foo', 'bar', 'baz', 'qux']*

When you call re.split('\s+', text), the regular expression is first compiled, and then its split method is called on the passed text. You can compile the regex yourself with re.compile, forming a reusable regex object:

regex = re.compile('\s+')

regex.split(text)

*['foo', 'bar', 'baz', 'qux']*

If, instead, you wanted to get a list of all patterns matching the regex, you can use the findall method:

regex.findall(text)

*[' ', '\t ', ' \t']*

To avoid unwanted escaping with \ in a regular expression, use raw string literals like r'C:\x' instead of the equivalent 'C:\\x'.

Creating a regex object with re.compile is highly recommended if you intend to apply the same expression to many strings; doing so will save CPU cycles.

match and search are closely related to findall. While findall returns all matches in a string, search returns only the first match. More rigidly, match only matches at the beginning of the string. As a less trivial example, let’s consider a block of text and a regular expression capable of identifying most email addresses:

text = """Dave [dave@google.com](mailto:dave@google.com) Steve [steve@gmail.com](mailto:steve@gmail.com)

Rob [rob@gmail.com](mailto:rob@gmail.com) Ryan [ryan@yahoo.com](mailto:ryan@yahoo.com) """

pattern = **r'**[A-Z0-9.\_%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'

# re.IGNORECASE makes the regex case-insensitive

regex = re.compile(pattern, flags=re.IGNORECASE)

Using findall on the text produces a list of the email addresses:

regex.findall(text)

*['dave@google.com', 'steve@gmail.com', 'rob@gmail.com', 'ryan@yahoo.com']*

search returns a special match object for the first email address in the text. For the preceding regex, the match object can only tell us the start and end position of the pattern in the string:

m = regex.search(text)

m

<\_sre.SRE\_Match object; span=(5, 20), match='dave@google.com'>

text[m.start():m.end()]

*'dave@google.com'*

regex.match returns None, as it only will match if the pattern occurs at the start of the string:

print(regex.match(text))

None

Relatedly, sub will return a new string with occurrences of the pattern replaced by the a new string:

print(regex.sub('REDACTED', text))

Dave REDACTED

Steve REDACTED Rob REDACTED Ryan REDACTED

Suppose you wanted to find email addresses and simultaneously segment each address into its three components: username, domain name, and domain suffix. To do this, put parentheses around the parts of the pattern to segment:

pattern = r'([A-Z0-9.\_%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'

regex = re.compile(pattern, flags=re.IGNORECASE)

A match object produced by this modified regex returns a tuple of the pattern com‐ ponents with its groups method:

m = regex.match('wesm@bright.net')

m.groups()

('wesm', 'bright', 'net')

findall returns a list of tuples when the pattern has groups:

regex.findall(text)

*[('dave', 'google', 'com'),*

*('steve', 'gmail', 'com'),*

*('rob', 'gmail', 'com'),*

*('ryan', 'yahoo', 'com')]*

sub also has access to groups in each match using special symbols like \1 and \2. The symbol \1 corresponds to the first matched group, \2 corresponds to the second, and so forth:

print(regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text))

*Dave Username: dave, Domain: google, Suffix: com*

Steve Username: steve, Domain: gmail, Suffix: com Rob Username: rob, Domain: gmail, Suffix: com Ryan Username: ryan, Domain: yahoo, Suffix: com

There is much more to regular expressions in Python, most of which is outside the book’s scope. Table 7-4 provides a brief summary.

Table 7-4. Regular expression methods

**Argument Description**

findall Return all non-overlapping matching patterns in a string as a list

finditer Like findall, but returns an iterator

match Match pattern at start of string and optionally segment pattern components into groups;

if the pattern matches, returns a match object, and otherwise None

search Scan string for match to pattern; returning a match object if so; unlike match, the match can be

anywhere in the string as opposed to only at the beginning

split Break string into pieces at each occurrence of pattern

sub, subn Replace all (sub) or first n occurrences (subn) of pattern in string with replacement expression; use symbols

\1, \2, ... to refer to match group elements in the replacement string

#### Vectorized String Functions in pandas

Cleaning up a messy dataset for analysis often requires a lot of string munging and regularization. To complicate matters, a column containing strings will sometimes have missing data:

data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com',

.....: 'Rob': 'rob@gmail.com', 'Wes': np.nan**}**

data = pd.Series(data)

data

*Dave dave@google.com*

*Rob rob@gmail.com Steve steve@gmail.com Wes NaN*

*dtype: object*

data.isnull()

*Dave False*

*Rob False*

*Steve False*

*Wes True*

*dtype: bool*

You can apply string and regular expression methods can be applied (passing a lambda or other function) to each value using data.map, but it will fail on the NA (null) values. To cope with this, Series has array-oriented methods for string opera‐ tions that skip NA values. These are accessed through Series’s str attribute; for exam‐ ple, we could check whether each email address has 'gmail' in it with str.contains:

data.str.contains('gmail')

*Dave False*

*Rob True*

*Steve True*

*Wes NaN*

*dtype: object*

Regular expressions can be used, too, along with any re options like IGNORECASE:

pattern

'([A-Z0-9.\_%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,4})'

data.str.findall(pattern, flags=re.IGNORECASE)

*Dave [(dave, google, com)] Rob [(rob, gmail, com)] Steve [(steve, gmail, com)]*

*Wes NaN*

*dtype: object*

There are a couple of ways to do vectorized element retrieval. Either use str.get or index into the str attribute:

matches = data.str.match(pattern, flags=re.IGNORECASE)

matches

*Dave True*

*Rob True*

*Steve True*

*Wes NaN*

*dtype: object*

To access elements in the embedded lists, we can pass an index to either of these functions:

matches**.str.get(1)**

*Dave NaN*

*Rob NaN*

*Steve NaN*

*Wes NaN dtype: float64*

matches**.str[0]**

*Dave NaN*

*Rob NaN*

*Steve NaN*

*Wes NaN*

*dtype: float64*

You can similarly slice strings using this syntax:

data.str[:5]

*Dave dave@*

*Rob rob@g*

*Steve steve*

*Wes NaN*

*dtype: object*

See Table 7-5 for more pandas string methods.

Table 7-5. Partial listing of vectorized string methods

**Method Description**

**cat** Concatenate strings element-wise with optional delimiter contains Return boolean array if each

string contains pattern/regex count Count occurrences of pattern

**extract** Use a regular expression with groups to extract one or more strings from a Series of strings;

the result will be a DataFrame with one column per group

**endswith** Equivalent to x.endswith(pattern) for each element startswith Equivalent to x.startswith(pattern)

for each element findall Compute list of all occurrences of pattern/regex for each string get

Index into each element (retrieve i-th element)

**isalnum** Equivalent to built-in str.alnum

**isalpha** Equivalent to built-in str.isalpha

**isdecimal** Equivalent to built-in str.isdecimal

**isdigit** Equivalent to built-in str.isdigit

**islower** Equivalent to built-in str.islower

**isnumeric** Equivalent to built-in str.isnumeric

**isupper** Equivalent to built-in str.isupper

**join** Join strings in each element of the Series with passed separator

**len** Compute length of each string

**lower**, **upper** Convert cases; equivalent to x.lower() or x.upper() for each element

**match** Use re.match with the passed regular expression on each element, returning matched groups as list

**pad** Add whitespace to left, right, or both sides of strings

**center** Equivalent to pad(side='both')

**repeat** Duplicate values (e.g., s.str.repeat(3) is equivalent to x \* 3 for each string)

**replace** Replace occurrences of pattern/regex with some other string

**slice** Slice each string in the Series

**split** Split strings on delimiter or regular expression strip Trim whitespace from both sides,

including newlines rstrip Trim whitespace on right side

**lstrip** Trim whitespace on left side

### Conclusion

Effective data preparation can significantly improve productive by enabling you to spend more time analyzing data and less time getting it ready for analysis. We have explored a number of tools in this chapter, but the coverage here is by no means com‐ prehensive. In the next chapter, we will explore pandas’s joining and grouping func‐ tionality.

## Data Wrangling: Join, Combine, and Reshape

In many applications, data may be spread across a number of files or databases or be arranged in a form that is not easy to analyze. This chapter focuses on tools to help combine, join, and rearrange data.

First, I introduce the concept of hierarchical indexing in pandas, which is used exten‐ sively in some of these operations. I then dig into the particular data manipulations. You can see various applied usages of these tools in Chapter 14.

### Hierarchical Indexing

Hierarchical indexing is an important feature of pandas that enables you to have mul‐ tiple (two or more) index levels on an axis. Somewhat abstractly, it provides a way for you to work with higher dimensional data in a lower dimensional form. Let’s start with a simple example; create a Series with a list of lists (or arrays) as the index:

data = pd.Series(np.random.randn(9),

...: index=[['a', 'a', 'a', 'b', 'b', 'c', 'c', 'd', 'd'],

...: [1, 2, 3, 1, 3, 1, 2, 2, 3]])

data

*a 1 -0.204708*

*2 0.478943*

*3 -0.519439*

*b 1 -0.555730*

*3 1.965781*

*C 1 1.393406*

*2 0.092908*

*D 2 0.281746*

*3 0.769023*

*dtype: float64*

What you’re seeing is a prettified view of a Series with a MultiIndex as its index. The “gaps” in the index display mean “use the label directly above”:

data.index

*MultiIndex(levels=[['a', 'b', 'c', 'd'], [1, 2, 3]],*

*labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 2, 0, 2, 0, 1, 1, 2]])*

With a hierarchically indexed object, so-called partial indexing is possible, enabling you to concisely select subsets of the data:

data['b']

*1 -0.555730*

*3 1.965781*

*dtype: float64*

data['b':'c']

*b 1 -0.555730*

*3 1.965781*

*c 1 1.393406*

*2 0.092908*

*dtype: float64*

data.loc[['b', 'd']]

b 1 -0.555730

3 1.965781

d 2 0.281746

3 0.769023

dtype: float64

Selection is even possible from an “inner” level:

data.loc[:, 2]

*a 0.478943*

*c 0.092908*

*d 0.281746*

*dtype: float64*

Hierarchical indexing plays an important role in reshaping data and group-based operations like forming a pivot table. For example, you could rearrange the data into a DataFrame using its unstack method:

data.unstack()

1 2 3

a -0.204708 0.478943 -0.519439

b -0.555730 NaN 1.965781

c 1.393406 0.092908 NaN

d NaN 0.281746 0.769023

The inverse operation of unstack is stack:

data.unstack().stack()

*a 1 -0.204708*

*2 0.478943*

*3 -0.519439*

*b 1 -0.555730*

*3 1.965781*

*c 1 1.393406*

*2 0.092908*

*d 2 0.281746*

*3 0.769023*

*dtype: float64*

stack and unstack will be explored in more detail later in this chapter. With a DataFrame, either axis can have a hierarchical index:

frame = pd.DataFrame(np.arange(12).reshape((4, 3)),

....: index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],

....: columns=[['Ohio', 'Ohio', 'Colorado'],

....: ['Green', 'Red', 'Green']])

frame

Ohio

Green

Red Colorado

Green

a 1 0 1 2

2 3 4 5

b 1 6 7 8

2 9 10 11

The hierarchical levels can have names (as strings or any Python objects). If so, these will show up in the console output:

frame.index.names = ['key1', 'key2']

frame.columns.names = ['state', 'color'] In [22]: frame

*state Ohio Colorado*

*color Green Red Green*

*key1 key2*

*a 1 0 1 2*

*2 3 4 5*

*b 1 6 7 8*

*2 9 10 11*

Be careful to distinguish the index names 'state' and 'color'

from the row labels.

With partial column indexing you can similarly select groups of columns:

frame['Ohio']

color Green Red

key1 key2

a 1 0 1

2 3 4

b 1 6 7

2 9 10

A MultiIndex can be created by itself and then reused; the columns in the preceding DataFrame with level names could be created like this:

MultiIndex.from\_arrays([['Ohio', 'Ohio', 'Colorado'], ['Green', 'Red', 'Green']], names=['state', 'color'])

#### Reordering and Sorting Levels

At times you will need to rearrange the order of the levels on an axis or sort the data by the values in one specific level. The swaplevel takes two level numbers or names and returns a new object with the levels interchanged (but the data is otherwise unaltered):

frame.swaplevel('key1', 'key2')

state Ohio Colorado color Green Red Green key2 key1

1 a 0 1 2

2 a 3 4 5

1 b 6 7 8

2 b 9 10 11

sort\_index, on the other hand, sorts the data using only the values in a single level. When swapping levels, it’s not uncommon to also use sort\_index so that the result is lexicographically sorted by the indicated level:

frame.sort\_index(level=1)

*state Ohio Colorado color Green Red Green key1 key2*

*a 1 0 1 2*

*b 1 6 7 8*

*a 2 3 4 5*

*b 2 9 10 11*

frame.swaplevel(0, 1).sort\_index(level=0)

*state Ohio Colorado color Green Red Green key2 key1*

*1 a 0 1 2*

*b 6 7 8*

*2 a 3 4 5*

*b 9 10 11*

Data selection performance is much better on hierarchically indexed objects if the index is lexicographically sorted starting with the outermost level—that is, the result of calling sort\_index(level=0) or sort\_index().

#### Summary Statistics by Level

Many descriptive and summary statistics on DataFrame and Series have a level option in which you can specify the level you want to aggregate by on a particular axis. Consider the above DataFrame; we can aggregate by level on either the rows or columns like so:

frame.sum(level='key2')

*state Ohio Colorado color Green Red Green key2*

*1 6 8 10*

*2 12 14 16*

frame.sum(level='color', axis=1)

*color Green Red*

*key1 key2*

*a 1 2 1*

*2 8 4*

*b 1 14 7*

*2 20 10*

Under the hood, this utilizes pandas’s groupby machinery, which will be discussed in more detail later in the book.

Indexing with a DataFrame’s columns

It’s not unusual to want to use one or more columns from a DataFrame as the row index; alternatively, you may wish to move the row index into the DataFrame’s col‐ umns. Here’s an example DataFrame:

frame = pd.DataFrame({'a': range(7), 'b': range(7, 0, -1),

....: 'c': ['one', 'one', 'one', 'two', 'two',

....: 'two', 'two'],

....: 'd': [0, 1, 2, 0, 1, 2, 3]})

frame

a b c d

0 0 7 one 0

1 1 6 one 1

2 2 5 one 2

3 3 4 two 0

4 4 3 two 1

5 5 2 two 2

6 6 1 two 3

DataFrame’s set\_index function will create a new DataFrame using one or more of its columns as the index:

frame2 = frame.set\_index(['c', 'd'])

frame2

a b

c d

one 0 0 7

1 1 6

2 2 5

two 0 3 4

1 4 3

2 5 2

3 6 1

By default the columns are removed from the DataFrame, though you can leave them in:

frame.set\_index(['c', 'd'], drop=False)

*a b c d*

*c d*

*one 0 0 7 one 0*

*1 1 6 one 1*

*2 2 5 one 2*

*two 0 3 4 two 0*

*1 4 3 two 1*

*2 5 2 two 2*

*3 6 1 two 3*

reset\_index, on the other hand, does the opposite of set\_index; the hierarchical index levels are moved into the columns:

frame2.reset\_index()

*c d a b*

*0 one 0 0 7*

*1 one 1 1 6*

*2 one 2 2 5*

*3 two 0 3 4*

*4 two 1 4 3*

*5 two 2 5 2*

*6 two 3 6 1*

### Combining and Merging Datasets

Data contained in pandas objects can be combined together in a number of ways:

* pandas.merge connects rows in DataFrames based on one or more keys. This will be familiar to users of SQL or other relational databases, as it implements database join operations.
* pandas.concat concatenates or “stacks” together objects along an axis.
* The combine\_first instance method enables splicing together overlapping data to fill in missing values in one object with values from another.

I will address each of these and give a number of examples. They’ll be utilized in examples throughout the rest of the book.

#### Database-Style DataFrame Joins

Merge or join operations combine datasets by linking rows using one or more keys. These operations are central to relational databases (e.g., SQL-based). The merge function in pandas is the main entry point for using these algorithms on your data.

Let’s start with a simple example:

df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],

....: 'data1': range(7)})

df2 = pd.DataFrame({'key': ['a', 'b', 'd'],

....: 'data2': range(3)})

df1

*data1 key*

*0 0 b*

*1 1 b*

*2 2 a*

*3 3 c*

*4 4 a*

*5 5 a*

*6 6 b*

df2

data2 key

0 0 a

1 1 b

2 2 d

This is an example of a many-to-one join; the data in df1 has multiple rows labeled a and b, whereas df2 has only one row for each value in the key column. Calling merge with these objects we obtain:

pd.merge(df1, df2)

data1 key data2

*0 0 b 1*

*1 1 b 1*

*2 6 b 1*

*3 2 a 0*

*4 4 a 0*

*5 5 a 0*

Note that I didn’t specify which column to join on. If that information is not speci‐ fied, merge uses the overlapping column names as the keys. It’s a good practice to specify explicitly, though:

pd.merge(df1, df2, on='key')

*data1 key data2*

*0 0 b 1*

*1 1 b 1*

*2 6 b 1*

*3 2 a 0*

*4 4 a 0*

*5 5 a 0*

If the column names are different in each object, you can specify them separately:

df3 = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],

....: 'data1': range(7)})

df4 = pd.DataFrame({'rkey': ['a', 'b', 'd'],

....: 'data2': range(3)})

pd.merge(df3, df4, left\_on='lkey', right\_on='rkey')

*data1 lkey data2 rkey*

*0 0 b 1 b*

*1 1 b 1 b*

*2*

*3 6 b 1 b*

*4*

*5 2 a 0 a*

*4 4 a 0 a*

*5 5 a 0 a*

You may notice that the 'c' and 'd' values and associated data are missing from the result. By default merge does an 'inner' join; the keys in the result are the intersec‐ tion, or the common set found in both tables. Other possible options are 'left', 'right', and 'outer'. The outer join takes the union of the keys, combining the effect of applying both left and right joins:

pd.merge(df1, df2, how='outer')

*data1 key data2*

*0 0.0 b 1.0*

*1 1.0 b 1.0*

*2 6.0 b 1.0*

*3 2.0 a 0.0*

*4 4.0 a 0.0*

*5 5.0 a 0.0*

*6 3.0 c NaN*

*7*

*8 NaN d 2.0*

*9*

See Table 8-1 for a summary of the options for how.

Table 8-1. Different join types with how argument

**Option Behavior**

**'inner'** Use only the key combinations observed in both tables 'left' Use all key combinations found in the left

table 'right' Use all key combinations found in the right table 'output' Use all key combinations

observed in both tables together

Many-to-many merges have well-defined, though not necessarily intuitive, behavior. Here’s an example:

df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],

....: 'data1': range(6)})

df2 = pd.DataFrame({'key': ['a', 'b', 'a', 'b', 'd'],

....: 'data2': range(5)})

df1

*data1 key*

*0 0 b*

*1 1 b*

*2 2 a*

*3 3 c*

*4 4 a*

*5 5 b*

df2

*data2 key*

*0 0 a*

*1 1 b*

*2 2 a*

*3 3 b*

*4 4 d*

pd.merge(df1, df2, on='key', how='left')

*data1 key data*

*2 0 0 b 1.0*

*1 0 b 3.0*

*2 1 b 1.0*

*3 1 b 3.0*

*4 2 a 0.0*

*5 2 a 2.0*

*6 3 c NaN*

*7 4 a 0.0*

*8 4 a 2.0*

*9 5 b 1.0*

*10 5 b 3.0*

Many-to-many joins form the Cartesian product of the rows. Since there were three 'b' rows in the left DataFrame and two in the right one, there are six 'b' rows in the result. The join method only affects the distinct key values appearing in the result:

pd.merge(df1, df2, how='inner')

*data1 key data2*

*0 0 b 1*

*1 0 b 3*

*2 1 b 1*

*3 1 b 3*

*4 5 b 1*

*5 5 b 3*

*6 2 a 0*

*7 2 a 2*

*8 4 a 0*

*9 4 a 2*

To merge with multiple keys, pass a list of column names:

left = pd.DataFrame({'key1': ['foo', 'foo', 'bar'],

....: 'key2': ['one', 'two', 'one'],

....: 'lval': [1, 2, 3]})

right = pd.DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],

....: 'key2': ['one', 'one', 'one', 'two'],

....: 'rval': [4, 5, 6, 7]})

pd.merge(left, right, on=['key1', 'key2'], how='outer')

key1 key2 lval rval 0 foo one 1.0 4.0

1 foo one 1.0 5.0

2

3 foo two 2.0 NaN 3 bar one 3.0 6.0

4

4 bar two NaN 7.0

To determine which key combinations will appear in the result depending on the choice of merge method, think of the multiple keys as forming an array of tuples to be used as a single join key (even though it’s not actually implemented that way).

When you’re joining columns-on-columns, the indexes on the passed DataFrame objects are discarded.

A last issue to consider in merge operations is the treatment of overlapping column names. While you can address the overlap manually (see the earlier section on renaming axis labels), merge has a suffixes option for specifying strings to append to overlapping names in the left and right DataFrame objects:

pd.merge(left, right, on='key1')

*key1 key2\_x lval key2\_y rval*

*0 foo one 1 one 4*

*1 foo one 1 one 5*

*2 foo two 2 one 4*

*3 foo two 2 one 5*

*4 bar one 3 one 6*

*5 bar one 3 two 7*

pd.merge(left, right, on='key1', suffixes=('\_left', '\_right'))

*key1 key2\_left lval key2\_right rval*

*0 foo one 1 one 4*

*1*

*2 foo one 1 one 5*

*3*

*4 foo two 2 one 4*

*5*

*6 foo two 2 one 5*

*7*

*8 bar one 3 one 6*

*9*

*10 bar one 3 two 7*

*11*

See Table 8-2 for an argument reference on merge. Joining using the DataFrame’s row index is the subject of the next section.

Table 8-2. merge function arguments

**Argument Description**

left DataFrame to be merged on the left side.

right DataFrame to be merged on the right side.

How One of 'inner', 'outer', 'left', or 'right'; defaults to 'inner'.

on Column names to join on. Must be found in both DataFrame objects. If not specified and no other join

keys given, will use the intersection of the column names in left and right as the join keys.

left\_on Columns in left DataFrame to use as join keys.

right\_on Analogous to left\_on for left DataFrame.

left\_index Use row index in left as its join key (or keys, if a MultiIndex).

right\_index Analogous to left\_index.

sort Sort merged data lexicographically by join keys; True by default (disable to get better performance

in some cases on large datasets).

suffixes Tuple of string values to append to column names in case of overlap; defaults to ('\_x', '\_y') (e.g., if

'data' in both DataFrame objects, would appear as 'data\_x' and 'data\_y' in result).

copy If False, avoid copying data into resulting data structure in some exceptional cases; by default

always copies.

indicator Adds a special column \_merge that indicates the source of each row; values will be 'left\_only',

'right\_only', or 'both' based on the origin of the joined data in each row.

#### Merging on Index

In some cases, the merge key(s) in a DataFrame will be found in its index. In this case, you can pass left\_index=True or right\_index=True (or both) to indicate that the index should be used as the merge key:

pd.DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'],

....: 'value': range(6)})

right1 = pd.DataFrame({'group\_val': [3.5, 7]}, index=['a', 'b'])

left1

key value

0 a 0

1 b 1

2 a 2

3 a 3

4 b 4

5 c 5

right1

*group\_val*

*a 3.5*

*b 7.0*

pd.merge(left1, right1, left\_on='key', right\_index=True)

*key value group\_val*

*0 a 0 3.5*

*2 a 2 3.5*

*3 a 3 3.5*

*1 b 1 7.0*

*4 b 4 7.0*

Since the default merge method is to intersect the join keys, you can instead form the union of them with an outer join:

pd.merge(left1, right1, left\_on='key', right\_index=True, how='outer')

key value group\_val

0 a 0 3.5

2 a 2 3.5

3 a 3 3.5

1 b 1 7.0

4 b 4 7.0

5 c 5 NaN

With hierarchically indexed data, things are more complicated, as joining on index is implicitly a multiple-key merge:

lefth = pd.DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio',

....: 'Nevada', 'Nevada'],

....: 'key2': [2000, 2001, 2002, 2001, 2002],

....: 'data': np.arange(5.)})

righth = pd.DataFrame(np.arange(12).reshape((6, 2)),

....: index=[['Nevada', 'Nevada', 'Ohio', 'Ohio',

....: 'Ohio', 'Ohio'],

....: [2001, 2000, 2000, 2000, 2001, 2002]],

....: columns=['event1', 'event2'])

lefth

*data key1 key2*

*0 0.0 Ohio 2000*

*1 1.0 Ohio 2001*

*2 2.0 Ohio 2002*

*3 3.0 Nevada 2001*

*4 4.0 Nevada 2002*

righth

*event1 event2*

*Nevada 2001 0 1*

*2000 2 3*

*Ohio 2000 4 5*

*2000 6 7*

*2001 8 9*

*2002 10 11*

In this case, you have to indicate multiple columns to merge on as a list (note the handling of duplicate index values with how='outer'):

pd.merge(lefth, righth, left\_on=['key1', 'key2'], right\_index=True)

*data key1 key2 event1 event2*

*0 0.0 Ohio 2000 4 5*

*0 0.0 Ohio 2000 6 7*

*1 1.0 Ohio 2001 8 9*

*2 2.0 Ohio 2002 10 11*

*3 3.0 Nevada 2001 0 1*

pd.merge(lefth, righth, left\_on=['key1', 'key2'],

....: right\_index=True, how='outer')

*data key1 key2 event1 event2*

*0 0.0 Ohio 2000 4.0 5.0*

*0 0.0 Ohio 2000 6.0 7.0*

*1 1.0 Ohio 2001 8.0 9.0*

*2 2.0 Ohio 2002 10.0 11.0*

*3 3.0 Nevada 2001 0.0 1.0*

*4 4.0 Nevada 2002 NaN NaN*

*4 NaN Nevada 2000 2.0 3.0*

Using the indexes of both sides of the merge is also possible:

left2 = pd.DataFrame([[1., 2.], [3., 4.], [5., 6.]],

....: index=['a', 'c', 'e'],

....: columns=['Ohio', 'Nevada'])

right2 = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]],

....: index=['b', 'c', 'd', 'e'],

....: columns=['Missouri', 'Alabama'])

left2

*Ohio Nevada*

*a 1.0 2.0*

*c 3.0 4.0*

*e 5.0 6.0*

right2

*Missouri Alabama*

*b 7.0 8.0*

*c 9.0 10.0*

*d 11.0 12.0*

*e 13.0 14.0*

pd.merge(left2, right2, how='outer', left\_index=True, right\_index=True)

*Ohio Nevada Missouri Alabama*

*a 1.0 2.0 NaN NaN*

*b NaN NaN 7.0 8.0*

*c 3.0 4.0 9.0 10.0*

*d NaN NaN 11.0 12.0*

*e 5.0 6.0 13.0 14.0*

DataFrame has a convenient join instance for merging by index. It can also be used to combine together many DataFrame objects having the same or similar indexes but non-overlapping columns. In the prior example, we could have written:

left2.join(right2, how='outer')

*Ohio Nevada Missouri Alabama*

*a 1.0 2.0 NaN NaN*

*b NaN NaN 7.0 8.0*

*c 3.0 4.0 9.0 10.0*

*d NaN NaN 11.0 12.0*

*e 5.0 6.0 13.0 14.0*

In part for legacy reasons (i.e., much earlier versions of pandas), DataFrame’s join method performs a left join on the join keys, exactly preserving the left frame’s row index. It also supports joining the index of the passed DataFrame on one of the col‐ umns of the calling DataFrame:

left1.join(right1, on='key')

*key value group\_val*

*0 a 0 3.5*

*1 b 1 7.0*

*2 a 2 3.5*

*3 a 3 3.5*

*4 b 4 7.0*

*5 c 5 NaN*

Lastly, for simple index-on-index merges, you can pass a list of DataFrames to join as an alternative to using the more general concat function described in the next section:

another = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [16., 17.]],

....: index=['a', 'c', 'e', 'f'],

....: columns=['New York', 'Oregon'])

another

*New York Oregon*

*a 7.0 8.0*

*c 9.0 10.0*

*e 11.0 12.0*

*f 16.0 17.0*

left2.join([right2, another])

*Ohio Nevada Missouri Alabama New York Oregon*

*a 1.0 2.0 NaN NaN 7.0 8.0*

*c 3.0 4.0 9.0 10.0 9.0 10.0*

*e 5.0 6.0 13.0 14.0 11.0 12.0*

left2.join([right2, another], how='outer')

*Ohio Nevada Missouri Alabama New York Oregon*

*a 1.0 2.0 NaN NaN 7.0 8.0*

*b NaN NaN 7.0 8.0 NaN NaN*

*c 3.0 4.0 9.0 10.0 9.0 10.0*

*d NaN NaN 11.0 12.0 NaN NaN*

*e 5.0 6.0 13.0 14.0 11.0 12.0*

*f NaN NaN NaN NaN 16.0 17.0*

#### Concatenating Along an Axis

Another kind of data combination operation is referred to interchangeably as concat‐ enation, binding, or stacking. NumPy’s concatenate function can do this with NumPy arrays:

arr = np.arange(12).reshape((3, 4))

arr

*array([[ 0, 1, 2, 3],*

*[ 4, 5, 6, 7],*

*[ 8, 9, 10, 11]])*

np.concatenate([arr, arr], axis=1)

*array([[ 0, 1, 2, 3, 0, 1, 2, 3],*

*[ 4, 5, 6, 7, 4, 5, 6, 7],*

*[ 8, 9, 10, 11, 8, 9, 10, 11]])*

In the context of pandas objects such as Series and DataFrame, having labeled axes enable you to further generalize array concatenation. In particular, you have a num‐ ber of additional things to think about:

* If the objects are indexed differently on the other axes, should we combine the distinct elements in these axes or use only the shared values (the intersection)?
* Do the concatenated chunks of data need to be identifiable in the resulting object?
* Does the “concatenation axis” contain data that needs to be preserved? In many cases, the default integer labels in a DataFrame are best discarded during concatenation.

The concat function in pandas provides a consistent way to address each of these concerns. I’ll give a number of examples to illustrate how it works. Suppose we have three Series with no index overlap:

s1 = pd.Series([0, 1], index=['a', 'b'])

s2 = pd.Series([2, 3, 4], index=['c', 'd', 'e'])

s3 = pd.Series([5, 6], index=['f', 'g'])

Calling concat with these objects in a list glues together the values and indexes:

pd.concat([s1, s2, s3])

*a 0*

*b*

*c 1*

*d*

*e 2*

*f*

*g 3*

*h*

*i 4*

*j*

*k 5*

*l*

*m 6*

*n*

*dtype: int64*

By default concat works along axis=0, producing another Series. If you pass axis=1, the result will instead be a DataFrame (axis=1 is the columns):

pd.concat([s1, s2, s3], axis=1)

*0 1 2*

*a 0.0 NaN NaN*

*b 1.0 NaN NaN*

*c NaN 2.0 NaN*

*d NaN 3.0 NaN*

*e NaN 4.0 NaN*

*f NaN NaN 5.0*

*g NaN NaN 6.0*

In this case there is no overlap on the other axis, which as you can see is the sorted union (the 'outer' join) of the indexes. You can instead intersect them by passing join='inner':

s4 = pd.concat([s1, s3])

s4

*a 0*

*b*

*c 1*

*d*

*f 5*

*g*

*h 6*

*i*

*dtype: int64*

pd.concat([s1, s4], axis=1)

*0 1*

*a 0.0 0*

*b 1.0 1*

*f NaN 5*

*g NaN 6*

pd.concat([s1, s4], axis=1, join='inner') Out[90]:

0 1

a 0 0

b

c 1 1

d

In this last example, the 'f' and 'g' labels disappeared because of the join='inner' option.

You can even specify the axes to be used on the other axes with join\_axes:

pd.concat([s1, s4], axis=1, join\_axes=[['a', 'c', 'b', 'e']]) Out[91]:

*0 1*

*a 0.0 0.0*

*c NaN NaN*

*b 1.0 1.0*

*e NaN NaN*

A potential issue is that the concatenated pieces are not identifiable in the result. Sup‐ pose instead you wanted to create a hierarchical index on the concatenation axis. To do this, use the keys argument:

result = pd.concat([s1, s1, s3], keys=['one', 'two', 'three'])

result

*one a 0*

*b 1*

*two a 0*

*b 1*

*three f 5*

*g 6*

*dtype: int64*

result.unstack()

a b f g

one 0.0 1.0 NaN NaN

two 0.0 1.0 NaN NaN

three NaN NaN 5.0 6.0

In the case of combining Series along axis=1, the keys become the DataFrame col‐ umn headers:

pd.concat([s1, s2, s3], axis=1, keys=['one', 'two', 'three'])

one two three

a 0.0 NaN NaN

b 1.0 NaN NaN

c NaN 2.0 NaN

d NaN 3.0 NaN

e NaN 4.0 NaN

f NaN NaN 5.0

g NaN NaN 6.0

The same logic extends to DataFrame objects:

df1 = pd.DataFrame(np.arange(6).reshape(3, 2), index=['a', 'b', 'c'],

....: columns=['one', 'two'])

df2 = pd.DataFrame(5 + np.arange(4).reshape(2, 2), index=['a', 'c'],

....: columns=['three', 'four'])

df1

*one two*

*a 0 1*

*b 2 3*

*c 4 5*

df2

three four

a 5 6

c 7 8

pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])

*level1 level2*

*one two three four*

*a 0 1 5.0 6.0*

*b 2 3 NaN NaN*

*c 4 5 7.0 8.0*

If you pass a dict of objects instead of a list, the dict’s keys will be used for the keys

option:

pd.concat({'level1': df1, 'level2': df2}, axis=1)

*level1 level2*

*one two three four*

*a 0 1 5.0 6.0*

*b 2 3 NaN NaN*

*c 4 5 7.0 8.0*

There are additional arguments governing how the hierarchical index is created (see Table 8-3). For example, we can name the created axis levels with the names argument:

pd.concat([df1, df2], axis=1, keys=['level1', 'level2'],

.....: names=['upper', 'lower'])

upper level1 level2 lower one two three four

a 0 1 5.0 6.0

b 2 3 NaN NaN

c 4 5 7.0 8.0

A last consideration concerns DataFrames in which the row index does not contain any relevant data:

In [103]: df1 = pd.DataFrame(np.random.randn(3, 4), columns=['a', 'b', 'c', 'd'])

In [104]: df2 = pd.DataFrame(np.random.randn(2, 3), columns=['b', 'd', 'a'])

df1

*a b c d*

*0 1.246435 1.007189 -1.296221 0.274992*

*1 0.228913 1.352917 0.886429 -2.001637*

*2 -0.371843 1.669025 -0.438570 -0.539741*

df2

b d a

0 0.476985 3.248944 -1.021228

c

1 -0.577087 0.124121 0.302614

In this case, you can pass ignore\_index=True:

pd.concat([df1, df2], ignore\_index=True)

a b c d

0 1.246435 1.007189 -1.296221 0.274992

1 0.228913 1.352917 0.886429 -2.001637

2 -0.371843 1.669025 -0.438570 -0.539741

3 -1.021228 0.476985 NaN 3.248944

4 0.302614 -0.577087 NaN 0.124121

Table 8-3. concat function arguments

**Arguments Description**

objs List or dict of pandas objects to be concatenated; this is the only required argument

axis Axis to concatenate along; defaults to 0 (along rows)

join Either 'inner' or 'outer' ('outer' by default); whether to intersection (inner) or union (outer)

together indexes along the other axes

join\_axes Specific indexes to use for the other n–1 axes instead of performing union/intersection logic

keys Values to associate with objects being concatenated, forming a hierarchical index along the

concatenation axis; can either be a list or array of arbitrary values, an array of tuples, or a list of

arrays (if multiple-level arrays passed in levels)

**Arguments Description**

levels Specific indexes to use as hierarchical index level or levels if keys passed

names Names for created hierarchical levels if keys and/or levels passed

verify\_integrity Check new axis in concatenated object for duplicates and raise exception if so;

by default (False) allows duplicates

ignore\_index Do not preserve indexes along concatenation axis, instead producing a new

range(total\_length) index

#### Combining Data with Overlap

There is another data combination situation that can’t be expressed as either a merge or concatenation operation. You may have two datasets whose indexes overlap in full or part. As a motivating example, consider NumPy’s where function, which performs the array-oriented equivalent of an if-else expression:

a = pd.Series([np.nan, 2.5, np.nan, 3.5, 4.5, np.nan],

.....: index=['f', 'e', 'd', 'c', 'b', 'a'])

b = pd.Series(np.arange(len(a), dtype=np.float64),

.....: index=['f', 'e', 'd', 'c', 'b', 'a'])

b[-1] = np.nan

a

*f NaN*

*e 2.5*

*d NaN*

*c 3.5*

*b 4.5*

*a NaN*

*dtype: float64*

b

*f 0.0*

*e 1.0*

*d 2.0*

*c 3.0*

*b 4.0*

*a NaN*

*dtype: float64*

np.where(pd.isnull(a), b, a)

*array([ 0. , 2.5, 2. , 3.5, 4.5, nan])*

Series has a combine\_first method, which performs the equivalent of this operation along with pandas’s usual data alignment logic:

b[:-2].combine\_first(a[2:])

*a NaN*

*b 4.5*

*c 3.0*

*d 2.0*

*e 1.0*

*f 0.0*

*dtype: float64*

With DataFrames, combine\_first does the same thing column by column, so you can think of it as “patching” missing data in the calling object with data from the object you pass:

df1 = pd.DataFrame({'a': [1., np.nan, 5., np.nan],

.....: 'b': [np.nan, 2., np.nan, 6.],

.....: 'c': range(2, 18, 4)})

df2 = pd.DataFrame({'a': [5., 4., np.nan, 3., 7.],

.....: 'b': [np.nan, 3., 4., 6., 8.]})

**df1**

*a b c*

*0 1.0 NaN 2*

*1 NaN 2.0 6*

*2 5.0 NaN 10*

*3 NaN 6.0 14*

**df2**

*a b*

*b*

*0 5.0 NaN*

*1 4.0 3.0*

*2 NaN 4.0*

*3 3.0 6.0*

*4 7.0 8.0*

df1.combine\_first(df2)

*a b c*

*0 1.0 NaN 2.0*

*1 4.0 2.0 6.0*

*2 5.0 4.0 10.0*

*3 3.0 6.0 14.0*

*4 7.0 8.0 NaN*

### Reshaping and Pivoting

There are a number of basic operations for rearranging tabular data. These are alter‐ natingly referred to as reshape or pivot operations.

Reshaping with Hierarchical Indexing

Hierarchical indexing provides a consistent way to rearrange data in a DataFrame. There are two primary actions:

stack

This “rotates” or pivots from the columns in the data to the rows

unstack

This pivots from the rows into the columns

I’ll illustrate these operations through a series of examples. Consider a small Data‐ Frame with string arrays as row and column indexes:

In [120]: data = pd.DataFrame(np.arange(6).reshape((2, 3)),

.....: index=pd.Index(['Ohio', 'Colorado'], name='state'),

.....: columns=pd.Index(['one', 'two', 'three'],

.....: name='number'))

In [121]: data Out[121]:

|  |  |  |  |
| --- | --- | --- | --- |
| number  state | one | two | three |
| Ohio | 0 | 1 | 2 |
| Colorado | 3 | 4 | 5 |

Using the stack method on this data pivots the columns into the rows, producing a Series:

In [122]: result = data.stack()

In [123]: result Out[123]:

state number

Ohio one 0

two 1

three 2

Colorado one 3

two 4

three 5

dtype: int64

From a hierarchically indexed Series, you can rearrange the data back into a Data‐ Frame with unstack:

In [124]: result.unstack() Out[124]:

|  |  |  |  |
| --- | --- | --- | --- |
| number  state | one | two | three |
| Ohio | 0 | 1 | 2 |
| Colorado | 3 | 4 | 5 |

By default the innermost level is unstacked (same with stack). You can unstack a dif‐ ferent level by passing a level number or name:

In [125]: result.unstack(0) Out[125]:

|  |  |  |
| --- | --- | --- |
| state | Ohio | Colorado |
| number |  |  |
| one | 0 | 3 |
| two | 1 | 4 |
| three | 2 | 5 |

In [126]: result.unstack('state') Out[126]:

|  |  |  |
| --- | --- | --- |
| state | Ohio | Colorado |
| number |  |  |
| one | 0 | 3 |
| two | 1 | 4 |
| three | 2 | 5 |

Unstacking might introduce missing data if all of the values in the level aren’t found in each of the subgroups:

In [127]: s1 = pd.Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])

In [128]: s2 = pd.Series([4, 5, 6], index=['c', 'd', 'e'])

In [129]: data2 = pd.concat([s1, s2], keys=['one', 'two']) In [130]: data2

Out[130]:

one a 0

1. 1
2. 2
3. 3

two c 4

1. 5
2. 6

dtype: int64

In [131]: data2.unstack() Out[131]:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| a | b | c | d | e |
| one 0.0 | 1.0 | 2.0 | 3.0 | NaN |
| two NaN | NaN | 4.0 | 5.0 | 6.0 |

Stacking filters out missing data by default, so the operation is more easily invertible:

In [132]: data2.unstack() Out[132]:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| a | b | c | d | e |
| one 0.0 | 1.0 | 2.0 | 3.0 | NaN |
| two NaN | NaN | 4.0 | 5.0 | 6.0 |

In [133]: data2.unstack().stack() Out[133]:

|  |  |
| --- | --- |
| one a | 0.0 |
| b | 1.0 |
| c | 2.0 |
| d | 3.0 |
| two c | 4.0 |
| d | 5.0 |
| e | 6.0 |

dtype: float64

In [134]: data2.unstack().stack(dropna=False) Out[134]:

one a 0.0

b 1.0

c 2.0

d 3.0

e NaN

two a NaN

b NaN

c 4.0

d 5.0

e 6.0

dtype: float64

When you unstack in a DataFrame, the level unstacked becomes the lowest level in the result:

In [135]: df = pd.DataFrame({'left': result, 'right': result + 5},

.....: columns=pd.Index(['left', 'right'], name='side'))

In [136]: df Out[136]:

side left right

|  |  |  |  |
| --- | --- | --- | --- |
| state | number |  | |
| Ohio | one | 0 | 5 |
|  | two | 1 | 6 |
|  | three | 2 | 7 |
| Colorado | one | 3 | 8 |
|  | two | 4 | 9 |
|  | three | 5 | 10 |

In [137]: df.unstack('state') Out[137]:

side left right

state Ohio Colorado Ohio Colorado number

one 0 3 5 8

two 1 4 6 9

three 2 5 7 10

When calling stack, we can indicate the name of the axis to stack:

In [138]: df.unstack('state').stack('side') Out[138]:

|  |  |  |  |
| --- | --- | --- | --- |
| state |  | Colorado | Ohio |
| number | side |  |  |
| one | left | 3 | 0 |
|  | right | 8 | 5 |
| two | left | 4 | 1 |
|  | right | 9 | 6 |
| three | left | 5 | 2 |
|  | right | 10 | 7 |

#### Pivoting “Long” to “Wide” Format

A common way to store multiple time series in databases and CSV is in so-called long or stacked format. Let’s load some example data and do a small amount of time series wrangling and other data cleaning:

In [139]: data = pd.read\_csv('examples/macrodata.csv')

In [140]: data.head() Out[140]:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| year | quarter | realgdp | | realcons | | realinv | realgovt | realdpi | cpi | \ |
| 0 1959.0 | 1.0 | 2710.349 | | 1707.4 | | 286.898 | 470.045 | 1886.9 | 28.98 |  |
| 1 1959.0 | 2.0 | 2778.801 | | 1733.7 | | 310.859 | 481.301 | 1919.7 | 29.15 |  |
| 2 1959.0 | 3.0 | 2775.488 | | 1751.8 | | 289.226 | 491.260 | 1916.4 | 29.35 |  |
| 3 1959.0 | 4.0 | 2785.204 | | 1753.7 | | 299.356 | 484.052 | 1931.3 | 29.37 |  |
| 4 1960.0 | 1.0 | 2847.699 | | 1770.5 | | 331.722 | 462.199 | 1955.5 | 29.54 |  |
| m1 | tbilrate | unemp | pop | | infl | realint | | | | |
| 0 139.7 | 2.82 | 5.8 | 177.146 | | 0.00 | 0.00 | | | | |
| 1 141.7 | 3.08 | 5.1 | 177.830 | | 2.34 | 0.74 | | | | |
| 2 140.5 | 3.82 | 5.3 | 178.657 | | 2.74 | 1.09 | | | | |
| 3 140.0 | 4.33 | 5.6 | 179.386 | | 0.27 | 4.06 | | | | |
| 4 139.6 | 3.50 | 5.2 | 180.007 | | 2.31 | 1.19 | | | | |

In [141]: periods = pd.PeriodIndex(year=data.year, quarter=data.quarter,

.....: name='date')

In [142]: columns = pd.Index(['realgdp', 'infl', 'unemp'], name='item') In [143]: data = data.reindex(columns=columns)

In [144]: data.index = periods.to\_timestamp('D', 'end')

In [145]: ldata = data.stack().reset\_index().rename(columns={0: 'value'})

We will look at PeriodIndex a bit more closely in Chapter 11. In short, it combines the year and quarter columns to create a kind of time interval type.

Now, ldata looks like:

In [146]: ldata[:10] Out[146]:

|  |  |  |
| --- | --- | --- |
| date | item | value |
| 0 1959-03-31 | realgdp | 2710.349 |
| 1 1959-03-31 | infl | 0.000 |
| 2 1959-03-31 | unemp | 5.800 |
| 3 1959-06-30 | realgdp | 2778.801 |
| 4 1959-06-30 | infl | 2.340 |
| 5 1959-06-30 | unemp | 5.100 |
| 6 1959-09-30 | realgdp | 2775.488 |
| 7 1959-09-30 | infl | 2.740 |
| 8 1959-09-30 | unemp | 5.300 |
| 9 1959-12-31 | realgdp | 2785.204 |

This is the so-called long format for multiple time series, or other observational data with two or more keys (here, our keys are date and item). Each row in the table repre‐ sents a single observation.

Data is frequently stored this way in relational databases like MySQL, as a fixed schema (column names and data types) allows the number of distinct values in the item column to change as data is added to the table. In the previous example, date and item would usually be the primary keys (in relational database parlance), offering both relational integrity and easier joins. In some cases, the data may be more diffi‐ cult to work with in this format; you might prefer to have a DataFrame containing one column per distinct item value indexed by timestamps in the date column. Data‐ Frame’s pivot method performs exactly this transformation:

In [147]: pivoted = ldata.pivot('date', 'item', 'value')

In [148]: pivoted Out[148]:

|  |  |  |  |
| --- | --- | --- | --- |
| item  date | infl | realgdp | unemp |
| 1959-03-31 | 0.00 | 2710.349 | 5.8 |
| 1959-06-30 | 2.34 | 2778.801 | 5.1 |
| 1959-09-30 | 2.74 | 2775.488 | 5.3 |
| 1959-12-31 | 0.27 | 2785.204 | 5.6 |
| 1960-03-31 | 2.31 | 2847.699 | 5.2 |
| 1960-06-30 | 0.14 | 2834.390 | 5.2 |
| 1960-09-30 | 2.70 | 2839.022 | 5.6 |
| 1960-12-31 | 1.21 | 2802.616 | 6.3 |
| 1961-03-31 | -0.40 | 2819.264 | 6.8 |
| 1961-06-30 | 1.47 | 2872.005 | 7.0 |
| ... | ... | ... | ... |
| 2007-06-30 | 2.75 | 13203.977 | 4.5 |
| 2007-09-30 | 3.45 | 13321.109 | 4.7 |
| 2007-12-31 | 6.38 | 13391.249 | 4.8 |
| 2008-03-31 | 2.82 | 13366.865 | 4.9 |
| 2008-06-30 | 8.53 | 13415.266 | 5.4 |
| 2008-09-30 | -3.16 | 13324.600 | 6.0 |
| 2008-12-31 | -8.79 | 13141.920 | 6.9 |
| 2009-03-31 | 0.94 | 12925.410 | 8.1 |
| 2009-06-30 | 3.37 | 12901.504 | 9.2 |

2009-09-30 3.56 12990.341 9.6

[203 rows x 3 columns]

The first two values passed are the columns to be used respectively as the row and column index, then finally an optional value column to fill the DataFrame. Suppose you had two value columns that you wanted to reshape simultaneously:

In [149]: ldata['value2'] = np.random.randn(len(ldata))

In [150]: ldata[:10] Out[150]:

|  |  |  |  |
| --- | --- | --- | --- |
| date | item | value | value2 |
| 0 1959-03-31 | realgdp | 2710.349 | 0.523772 |
| 1 1959-03-31 | infl | 0.000 | 0.000940 |
| 2 1959-03-31 | unemp | 5.800 | 1.343810 |
| 3 1959-06-30 | realgdp | 2778.801 | -0.713544 |
| 4 1959-06-30 | infl | 2.340 | -0.831154 |
| 5 1959-06-30 | unemp | 5.100 | -2.370232 |
| 6 1959-09-30 | realgdp | 2775.488 | -1.860761 |
| 7 1959-09-30 | infl | 2.740 | -0.860757 |
| 8 1959-09-30 | unemp | 5.300 | 0.560145 |
| 9 1959-12-31 | realgdp | 2785.204 | -1.265934 |

By omitting the last argument, you obtain a DataFrame with hierarchical columns:

In [151]: pivoted = ldata.pivot('date', 'item')

In [152]: pivoted[:5] Out[152]:

|  |  |  |  |
| --- | --- | --- | --- |
| item date | value  infl | realgdp | value2  unemp infl realgdp unemp |
| 1959-03-31 | 0.00 | 2710.349 | 5.8 0.000940 0.523772 1.343810 |
| 1959-06-30 | 2.34 | 2778.801 | 5.1 -0.831154 -0.713544 -2.370232 |
| 1959-09-30 | 2.74 | 2775.488 | 5.3 -0.860757 -1.860761 0.560145 |
| 1959-12-31 | 0.27 | 2785.204 | 5.6 0.119827 -1.265934 -1.063512 |
| 1960-03-31 | 2.31 | 2847.699 | 5.2 -2.359419 0.332883 -0.199543 |

In [153]: pivoted['value'][:5] Out[153]:

|  |  |  |  |
| --- | --- | --- | --- |
| item  date | infl | realgdp | unemp |
| 1959-03-31 | 0.00 | 2710.349 | 5.8 |
| 1959-06-30 | 2.34 | 2778.801 | 5.1 |
| 1959-09-30 | 2.74 | 2775.488 | 5.3 |
| 1959-12-31 | 0.27 | 2785.204 | 5.6 |
| 1960-03-31 | 2.31 | 2847.699 | 5.2 |

Note that pivot is equivalent to creating a hierarchical index using set\_index fol‐ lowed by a call to unstack:

In [154]: unstacked = ldata.set\_index(['date', 'item']).unstack('item')

In [155]: unstacked[:7] Out[155]:

|  |  |  |  |
| --- | --- | --- | --- |
| item date | value  infl | realgdp | value2  unemp infl realgdp unemp |
| 1959-03-31 | 0.00 | 2710.349 | 5.8 0.000940 0.523772 1.343810 |
| 1959-06-30 | 2.34 | 2778.801 | 5.1 -0.831154 -0.713544 -2.370232 |
| 1959-09-30 | 2.74 | 2775.488 | 5.3 -0.860757 -1.860761 0.560145 |
| 1959-12-31 | 0.27 | 2785.204 | 5.6 0.119827 -1.265934 -1.063512 |
| 1960-03-31 | 2.31 | 2847.699 | 5.2 -2.359419 0.332883 -0.199543 |
| 1960-06-30 | 0.14 | 2834.390 | 5.2 -0.970736 -1.541996 -1.307030 |
| 1960-09-30 | 2.70 | 2839.022 | 5.6 0.377984 0.286350 -0.753887 |

#### Pivoting “Wide” to “Long” Format

An inverse operation to pivot for DataFrames is pandas.melt. Rather than trans‐ forming one column into many in a new DataFrame, it merges multiple columns into one, producing a DataFrame that is longer than the input. Let’s look at an example:

In [157]: df = pd.DataFrame({'key': ['foo', 'bar', 'baz'],

.....: 'A': [1, 2, 3],

.....: 'B': [4, 5, 6],

.....: 'C': [7, 8, 9]})

In [158]: df Out[158]:

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | C | key |
| 0 1 | 4 | 7 | foo |
| 1 2 | 5 | 8 | bar |
| 2 3 | 6 | 9 | baz |

The 'key' column may be a group indicator, and the other columns are data values. When using pandas.melt, we must indicate which columns (if any) are group indica‐ tors. Let’s use 'key' as the only group indicator here:

In [159]: melted = pd.melt(df, ['key'])

In [160]: melted Out[160]:

|  |  |  |
| --- | --- | --- |
| key | variable | value |
| 0 foo | A | 1 |
| 1 bar | A | 2 |
| 2 baz | A | 3 |
| 3 foo | B | 4 |
| 4 bar | B | 5 |
| 5 baz | B | 6 |
| 6 foo | C | 7 |
| 7 bar | C | 8 |
| 8 baz | C | 9 |

Using pivot, we can reshape back to the original layout:

In [161]: reshaped = melted.pivot('key', 'variable', 'value')

In [162]: reshaped Out[162]:

|  |  |  |  |
| --- | --- | --- | --- |
| variable  key | A | B | C |
| bar | 2 | 5 | 8 |
| baz | 3 | 6 | 9 |
| foo | 1 | 4 | 7 |

Since the result of pivot creates an index from the column used as the row labels, we may want to use reset\_index to move the data back into a column:

In [163]: reshaped.reset\_index() Out[163]:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| variable | key | A | B | C |
| 0 | bar | 2 | 5 | 8 |
| 1 | baz | 3 | 6 | 9 |
| 2 | foo | 1 | 4 | 7 |

You can also specify a subset of columns to use as value columns:

In [164]: pd.melt(df, id\_vars=['key'], value\_vars=['A', 'B']) Out[164]:

|  |  |  |
| --- | --- | --- |
| key | variable | value |
| 0 foo | A | 1 |
| 1 bar | A | 2 |
| 2 baz | A | 3 |
| 3 foo | B | 4 |
| 4 bar | B | 5 |
| 5 baz | B | 6 |

pandas.melt can be used without any group identifiers, too:

In [165]: pd.melt(df, value\_vars=['A', 'B', 'C']) Out[165]:

variable value 0 A 1

1 A 2

2 A 3

3 B 4

4 B 5

5 B 6

6 C 7

7 C 8

8 C 9

In [166]: pd.melt(df, value\_vars=['key', 'A', 'B']) Out[166]:

variable value

1. key foo

|  |  |  |
| --- | --- | --- |
| key bar2 | key | baz |
| 3 | A | 1 |
| 4 | A | 2 |
| 5 | A | 3 |
| 6 | B | 4 |
| 7 | B | 5 |
| 8 | B | 6 |

### Conclusion

Now that you have some pandas basics for data import, cleaning, and reorganization under your belt, we are ready to move on to data visualization with matplotlib. We will return to pandas later in the book when we discuss more advanced analytics.

## Plotting and Visualization

Making informative visualizations (sometimes called plots) is one of the most impor‐ tant tasks in data analysis. It may be a part of the exploratory process—for example, to help identify outliers or needed data transformations, or as a way of generating ideas for models. For others, building an interactive visualization for the web may be the end goal. Python has many add-on libraries for making static or dynamic visuali‐ zations, but I’ll be mainly focused on [matplotlib](http://matplotlib.sourceforge.net/) and libraries that build on top of it.

matplotlib is a desktop plotting package designed for creating (mostly two- dimensional) publication-quality plots. The project was started by John Hunter in 2002 to enable a MATLAB-like plotting interface in Python. The matplotlib and IPy‐ thon communities have collaborated to simplify interactive plotting from the IPython shell (and now, Jupyter notebook). matplotlib supports various GUI backends on all operating systems and additionally can export visualizations to all of the common vector and raster graphics formats (PDF, SVG, JPG, PNG, BMP, GIF, etc.). With the exception of a few diagrams, nearly all of the graphics in this book were produced using matplotlib.

Over time, matplotlib has spawned a number of add-on toolkits for data visualization that use matplotlib for their underlying plotting. One of these is [seaborn](http://seaborn.pydata.org/), which we explore later in this chapter.

The simplest way to follow the code examples in the chapter is to use interactive plot‐ ting in the Jupyter notebook. To set this up, execute the following statement in a Jupyter notebook:

%matplotlib notebook

#### A Brief matplotlib API Primer

With matplotlib, we use the following import convention:

In [11]: **import matplotlib.pyplot as plt**

After running %matplotlib notebook in Jupyter (or simply %matplotlib in IPy‐ thon), we can try creating a simple plot. If everything is set up right, a line plot like [Figure 9-1](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark8) should appear:

In [12]: **import numpy as np**

In [13]: data = np.arange(10) In [14]: data

Out[14]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [15]: plt.plot(data)

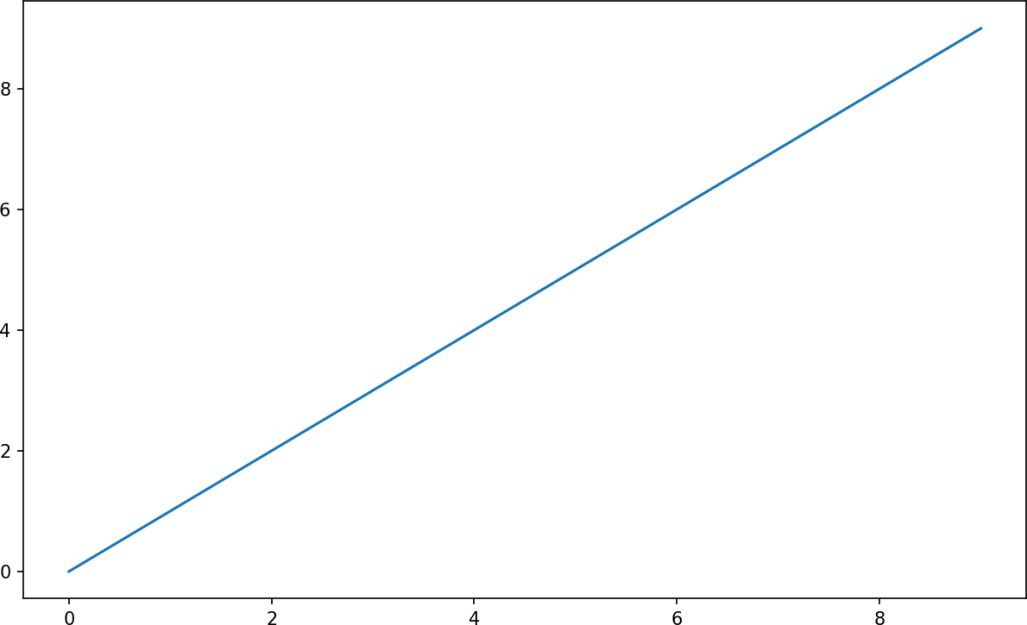


Figure 9-1. Simple line plot

While libraries like seaborn and pandas’s built-in plotting functions will deal with many of the mundane details of making plots, should you wish to customize them beyond the function options provided, you will need to learn a bit about the matplot‐ lib API.

There is not enough room in the book to give a comprehensive treatment to the breadth and depth of functionality in matplotlib. It should be enough to teach you the ropes to get up and running. The matplotlib gallery and documentation are the best resource for learning advanced features.

Figures and Subplots

Plots in matplotlib reside within a Figure object. You can create a new figure with

plt.figure:

In [16]: fig = plt.figure()

In IPython, an empty plot window will appear, but in Jupyter nothing will be shown until we use a few more commands. plt.figure has a number of options; notably, figsize will guarantee the figure has a certain size and aspect ratio if saved to disk.

You can’t make a plot with a blank figure. You have to create one or more subplots

using add\_subplot:

In [17]: ax1 = fig.add\_subplot(2, 2, 1)

This means that the figure should be 2 × 2 (so up to four plots in total), and we’re selecting the first of four subplots (numbered from 1). If you create the next two sub‐ plots, you’ll end up with a visualization that looks like [Figure 9-2](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark9):

In [18]: ax2 = fig.add\_subplot(2, 2, 2)

In [19]: ax3 = fig.add\_subplot(2, 2, 3)

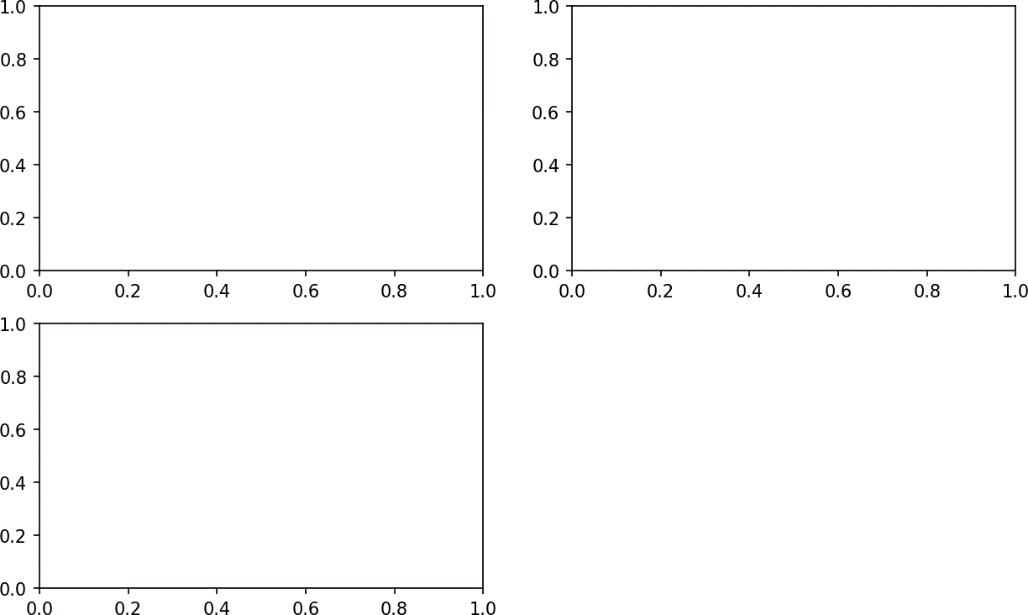


Figure 9-2. An empty matplotlib figure with three subplots

One nuance of using Jupyter notebooks is that plots are reset after each cell is evaluated, so for more complex plots you must put all of the plotting commands in a single notebook cell.

Here we run all of these commands in the same cell:

fig = plt.figure()

ax1 = fig.add\_subplot(2, 2, 1)

ax2 = fig.add\_subplot(2, 2, 2)

ax3 = fig.add\_subplot(2, 2, 3)

When you issue a plotting command like plt.plot([1.5, 3.5, -2, 1.6]), mat‐ plotlib draws on the last figure and subplot used (creating one if necessary), thus hid‐ ing the figure and subplot creation. So if we add the following command, you’ll get something like [Figure 9-3](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark10):

In [20]: plt.plot(np.random.randn(50).cumsum(), 'k--')

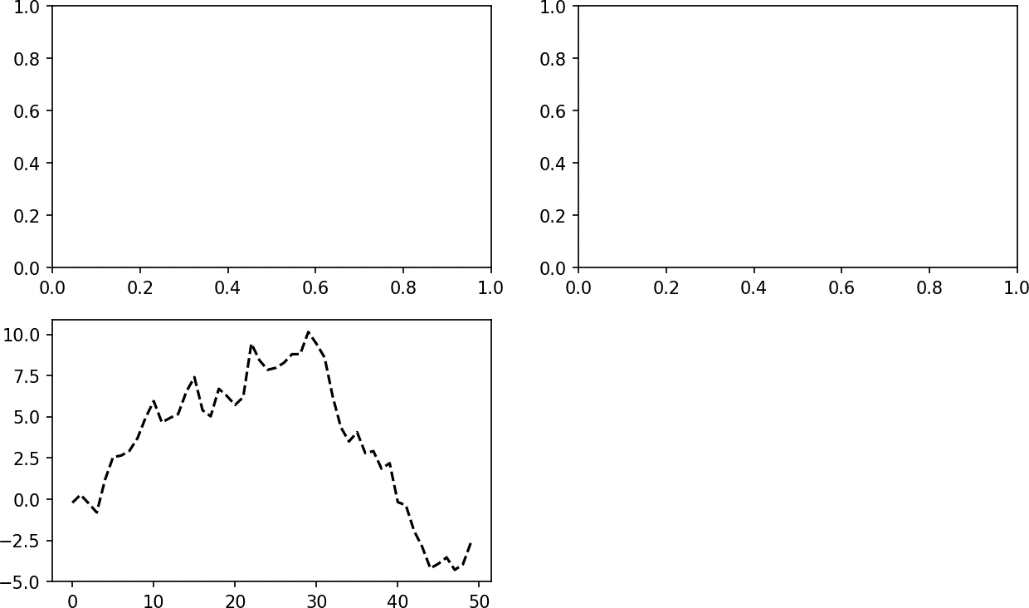


Figure 9-3. Data visualization after single plot

The 'k--' is a style option instructing matplotlib to plot a black dashed line. The objects returned by fig.add\_subplot here are AxesSubplot objects, on which you can directly plot on the other empty subplots by calling each one’s instance method (see [Figure 9-4](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark11)):

In [21]: \_ = ax1.hist(np.random.randn(100), bins=20, color='k', alpha=0.3) In [22]: ax2.scatter(np.arange(30), np.arange(30) + 3 \* np.random.randn(30))

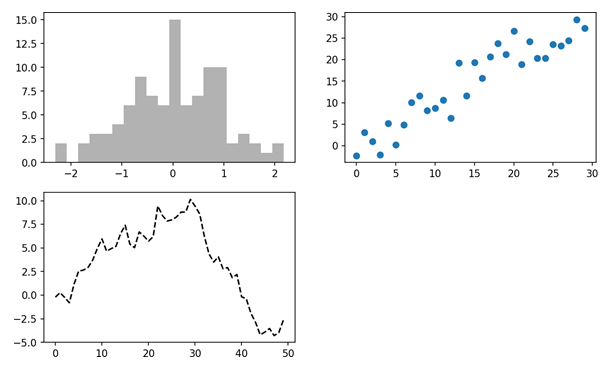


Figure 9-4. Data visualization after additional plots

You can find a comprehensive catalog of plot types in the [matplotlib documentation](http://matplotlib.sourceforge.net/).

Creating a figure with a grid of subplots is a very common task, so matplotlib includes a convenience method, plt.subplots, that creates a new figure and returns a NumPy array containing the created subplot objects:

In [24]: fig, axes = plt.subplots(2, 3)

In [25]: axes Out[25]:

array([[<matplotlib.axes.\_subplots.AxesSubplot object at 0x7fb626374048>,

<matplotlib.axes.\_subplots.AxesSubplot object at 0x7fb62625db00>,

<matplotlib.axes.\_subplots.AxesSubplot object at 0x7fb6262f6c88>], [<matplotlib.axes.\_subplots.AxesSubplot object at 0x7fb6261a36a0>,

<matplotlib.axes.\_subplots.AxesSubplot object at 0x7fb626181860>,

<matplotlib.axes.\_subplots.AxesSubplot object at 0x7fb6260fd4e0>]], dtype

=object)

This is very useful, as the axes array can be easily indexed like a two-dimensional array; for example, axes[0, 1]. You can also indicate that subplots should have the same x- or y-axis using sharex and sharey, respectively. This is especially useful when you’re comparing data on the same scale; otherwise, matplotlib autoscales plot limits independently. See [Table 9-1](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark12) for more on this method.

Table 9-1. pyplot.subplots options

**Argument Description**

nrows Number of rows of subplots

ncols Number of columns of subplots

sharex All subplots should use the same x-axis ticks (adjusting the xlim will affect all subplots) sharey All subplots should use the same y-axis ticks (adjusting the ylim will affect all subplots) subplot\_kw Dict of keywords passed to add\_subplot call used to create each subplot

\*\*fig\_kw Additional keywords to subplots are used when creating the figure, such as plt.subplots(2, 2, figsize=(8, 6))

##### Adjusting the spacing around subplots

By default matplotlib leaves a certain amount of padding around the outside of the subplots and spacing between subplots. This spacing is all specified relative to the height and width of the plot, so that if you resize the plot either programmatically or manually using the GUI window, the plot will dynamically adjust itself. You can change the spacing using the subplots\_adjust method on Figure objects, also avail‐ able as a top-level function:

subplots\_adjust(left=None, bottom=None, right=None, top=None, wspace=None, hspace=None)

wspace and hspace controls the percent of the figure width and figure height, respec‐ tively, to use as spacing between subplots. Here is a small example where I shrink the spacing all the way to zero (see [Figure 9-5](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark13)):

fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)

**for** i **in** range(2):

**for** j **in** range(2):

axes[i, j].hist(np.random.randn(500), bins=50, color='k', alpha=0.5) plt.subplots\_adjust(wspace=0, hspace=0)

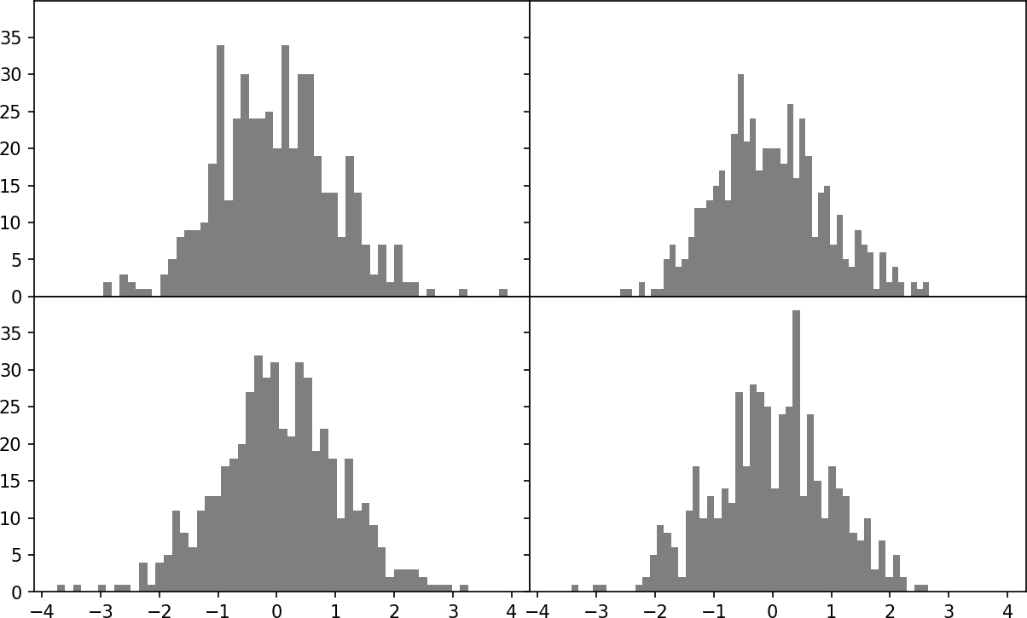


Figure 9-5. Data visualization with no inter-subplot spacing

You may notice that the axis labels overlap. matplotlib doesn’t check whether the labels overlap, so in a case like this you would need to fix the labels yourself by speci‐ fying explicit tick locations and tick labels (we’ll look at how to do this in the follow‐ ing sections).

#### Colors, Markers, and Line Styles

Matplotlib’s main plot function accepts arrays of x and y coordinates and optionally a string abbreviation indicating color and line style. For example, to plot x versus y with green dashes, you would execute:

ax.plot(x, y, 'g--')

This way of specifying both color and line style in a string is provided as a conve‐ nience; in practice if you were creating plots programmatically you might prefer not to have to munge strings together to create plots with the desired style. The same plot could also have been expressed more explicitly as:

ax.plot(x, y, linestyle='--', color='g')

There are a number of color abbreviations provided for commonly used colors, but you can use any color on the spectrum by specifying its hex code (e.g., '#CECECE'). You can see the full set of line styles by looking at the docstring for plot (use plot? in IPython or Jupyter).

Line plots can additionally have markers to highlight the actual data points. Since matplotlib creates a continuous line plot, interpolating between points, it can occa‐ sionally be unclear where the points lie. The marker can be part of the style string, which must have color followed by marker type and line style (see [Figure 9-6](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark14)):

In [30]: **from numpy.random import** randn

In [31]: plt.plot(randn(30).cumsum(), 'ko--')

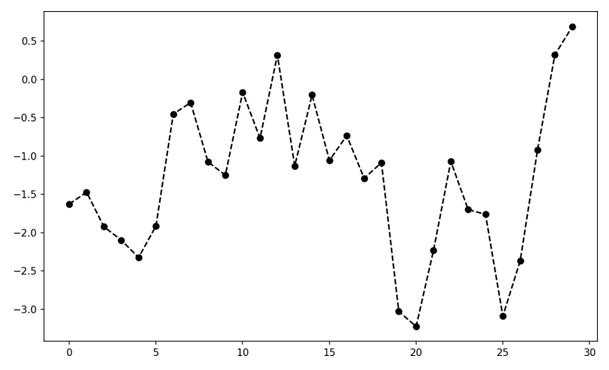


Figure 9-6. Line plot with markers

This could also have been written more explicitly as:

plot(randn(30).cumsum(), color='k', linestyle='dashed', marker='o')

For line plots, you will notice that subsequent points are linearly interpolated by default. This can be altered with the drawstyle option ([Figure 9-7](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark15)):

In [33]: data = np.random.randn(30).cumsum()

In [34]: plt.plot(data, 'k--', label='Default') Out[34]: [<matplotlib.lines.Line2D at 0x7fb624d86160>]

In [35]: plt.plot(data, 'k-', drawstyle='steps-post', label='steps-post') Out[35]: [<matplotlib.lines.Line2D at 0x7fb624d869e8>]

In [36]: plt.legend(loc='best')

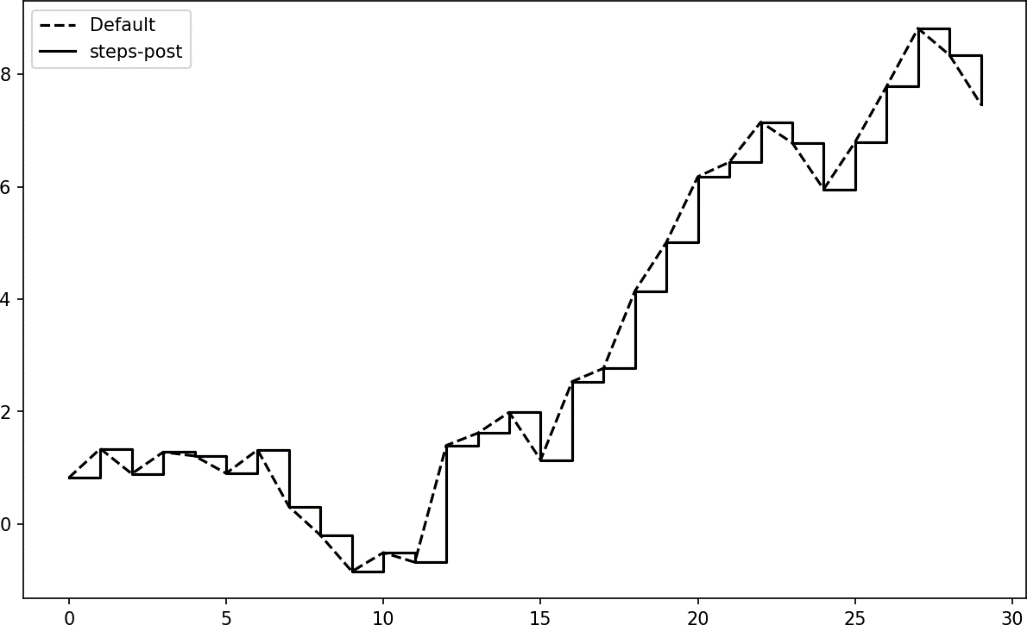


Figure 9-7. Line plot with different drawstyle options

You may notice output like <matplotlib.lines.Line2D at ...> when you run this. matplotlib returns objects that reference the plot subcomponent that was just added. A lot of the time you can safely ignore this output. Here, since we passed the label arguments to plot, we are able to create a plot legend to identify each line using plt.legend.

You must call plt.legend (or ax.legend, if you have a reference to the axes) to create the legend, whether or not you passed the label options when plotting the data.

#### Ticks, Labels, and Legends

For most kinds of plot decorations, there are two main ways to do things: using the procedural pyplot interface (i.e., matplotlib.pyplot) and the more object-oriented native matplotlib API.

The pyplot interface, designed for interactive use, consists of methods like xlim, xticks, and xticklabels. These control the plot range, tick locations, and tick labels, respectively. They can be used in two ways:

Called with no arguments returns the current parameter value (e.g., plt.xlim()

returns the current x-axis plotting range)

Called with parameters sets the parameter value (e.g., plt.xlim([0, 10]), sets the x-axis range to 0 to 10)

All such methods act on the active or most recently created AxesSubplot. Each of them corresponds to two methods on the subplot object itself; in the case of xlim these are ax.get\_xlim and ax.set\_xlim. I prefer to use the subplot instance methods myself in the interest of being explicit (and especially when working with multiple subplots), but you can certainly use whichever you find more convenient.

##### Setting the title, axis labels, ticks, and ticklabels

To illustrate customizing the axes, I’ll create a simple figure and plot of a random walk (see [Figure 9-8](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark16)):

In [37]: fig = plt.figure()

In [38]: ax = fig.add\_subplot(1, 1, 1)

In [39]: ax.plot(np.random.randn(1000).cumsum())

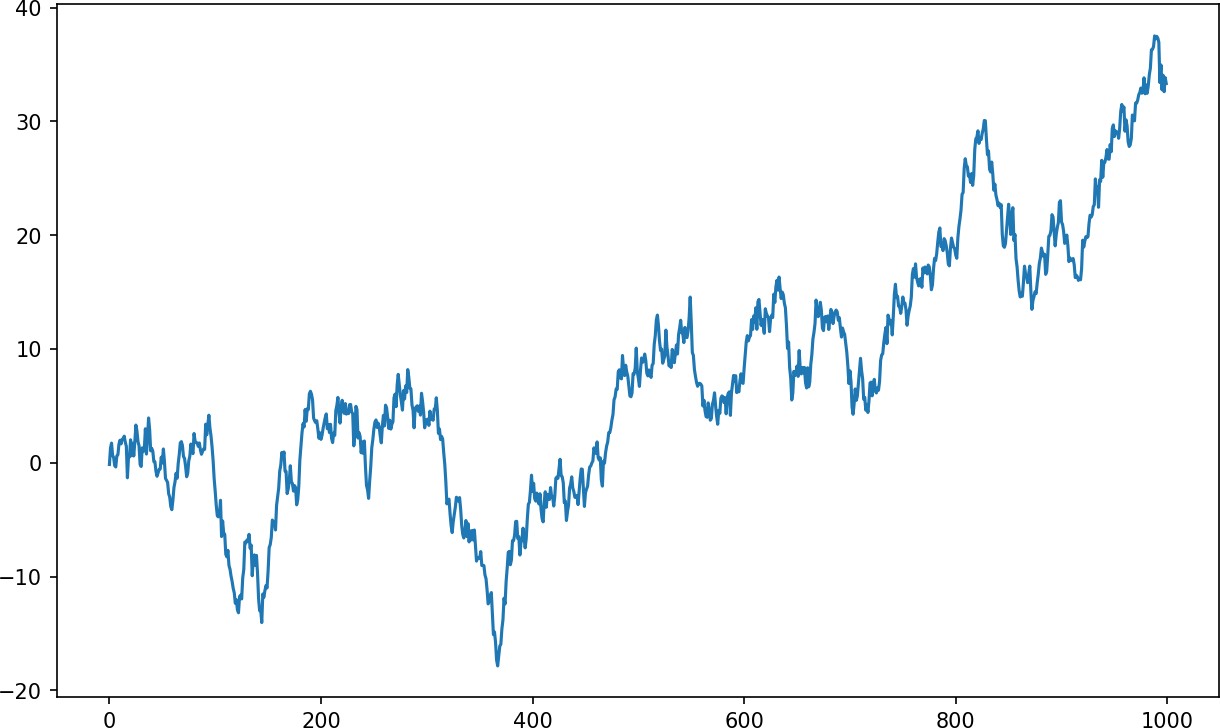


Figure 9-8. Simple plot for illustrating xticks (with label)

To change the x-axis ticks, it’s easiest to use set\_xticks and set\_xticklabels. The former instructs matplotlib where to place the ticks along the data range; by default

these locations will also be the labels. But we can set any other values as the labels using set\_xticklabels:

In [40]: ticks = ax.set\_xticks([0, 250, 500, 750, 1000])

In [41]: labels = ax.set\_xticklabels(['one', 'two', 'three', 'four', 'five'],

....: rotation=30, fontsize='small')

The rotation option sets the x tick labels at a 30-degree rotation. Lastly, set\_xlabel gives a name to the x-axis and set\_title the subplot title (see [Figure 9-9](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark17) for the resulting figure):

In [42]: ax.set\_title('My first matplotlib plot') Out[42]: <matplotlib.text.Text at 0x7fb624d055f8>

In [43]: ax.set\_xlabel('Stages')

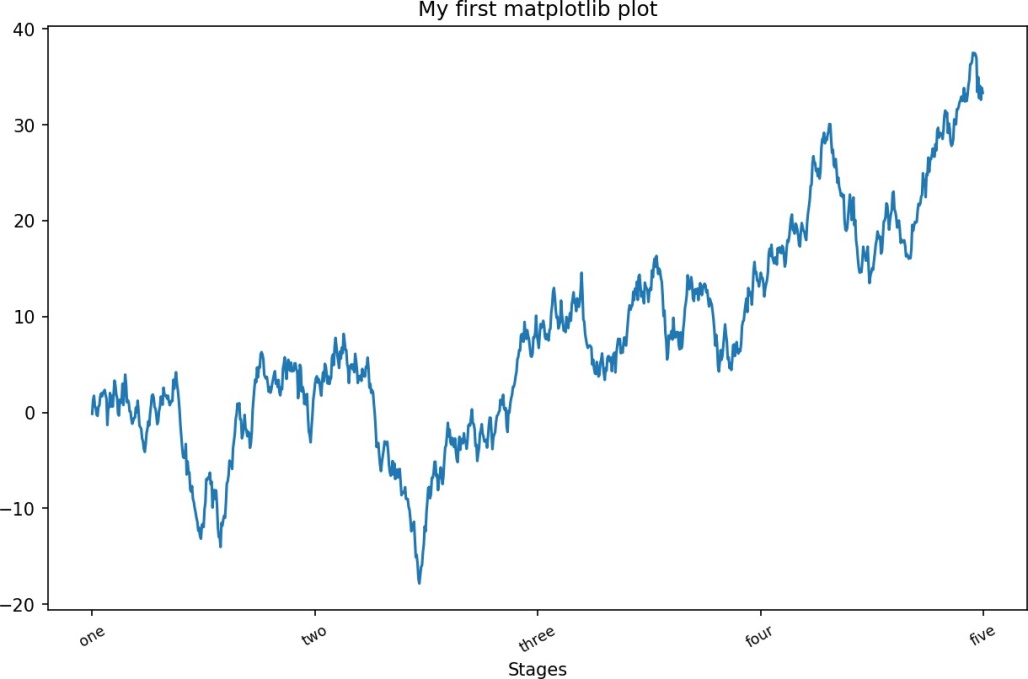


Figure 9-9. Simple plot for illustrating xticks

Modifying the y-axis consists of the same process, substituting y for x in the above. The axes class has a set method that allows batch setting of plot properties. From the prior example, we could also have written:

props = {

'title': 'My first matplotlib plot', 'xlabel': 'Stages'

}

ax.set(\*\*props)

Adding legends

Legends are another critical element for identifying plot elements. There are a couple of ways to add one. The easiest is to pass the label argument when adding each piece of the plot:

In [44]: **from numpy.random import** randn

In [45]: fig = plt.figure(); ax = fig.add\_subplot(1, 1, 1)

In [46]: ax.plot(randn(1000).cumsum(), 'k', label='one') Out[46]: [<matplotlib.lines.Line2D at 0x7fb624bdf860>]

In [47]: ax.plot(randn(1000).cumsum(), 'k--', label='two') Out[47]: [<matplotlib.lines.Line2D at 0x7fb624be90f0>]

In [48]: ax.plot(randn(1000).cumsum(), 'k.', label='three') Out[48]: [<matplotlib.lines.Line2D at 0x7fb624be9160>]

Once you’ve done this, you can either call ax.legend() or plt.legend() to automat‐ ically create a legend. The resulting plot is in [Figure 9-10](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark18):

In [49]: ax.legend(loc='best')



Figure 9-10. Simple plot with three lines and legend

The legend method has several other choices for the location loc argument. See the docstring (with ax.legend?) for more information.

The loc tells matplotlib where to place the plot. If you aren’t picky, 'best' is a good option, as it will choose a location that is most out of the way. To exclude one or more elements from the legend, pass no label or label='\_nolegend\_'.

#### Annotations and Drawing on a Subplot

In addition to the standard plot types, you may wish to draw your own plot annota‐ tions, which could consist of text, arrows, or other shapes. You can add annotations and text using the text, arrow, and annotate functions. text draws text at given coordinates (x, y) on the plot with optional custom styling:

ax.text(x, y, 'Hello world!', family='monospace', fontsize=10)

Annotations can draw both text and arrows arranged appropriately. As an example, let’s plot the closing S&P 500 index price since 2007 (obtained from Yahoo! Finance) and annotate it with some of the important dates from the 2008–2009 financial crisis. You can most easily reproduce this code example in a single cell in a Jupyter note‐ book. See [Figure 9-11](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark19) for the result:

**from datetime import** datetime

fig = plt.figure()

ax = fig.add\_subplot(1, 1, 1)

data = pd.read\_csv('examples/spx.csv', index\_col=0, parse\_dates=True) spx = data['SPX']

spx.plot(ax=ax, style='k-') crisis\_data = [

(datetime(2007, 10, 11), 'Peak of bull market'), (datetime(2008, 3, 12), 'Bear Stearns Fails'),

(datetime(2008, 9, 15), 'Lehman Bankruptcy')

]

**for** date, label **in** crisis\_data:

ax.annotate(label, xy=(date, spx.asof(date) + 75), xytext=(date, spx.asof(date) + 225),

arrowprops=dict(facecolor='black', headwidth=4, width=2, headlength=4),

horizontalalignment='left', verticalalignment='top')

*# Zoom in on 2007-2010* ax.set\_xlim(['1/1/2007', '1/1/2011']) ax.set\_ylim([600, 1800])

ax.set\_title('Important dates in the 2008-2009 financial crisis')



Figure 9-11. Important dates in the 2008–2009 financial crisis

There are a couple of important points to highlight in this plot: the ax.annotate method can draw labels at the indicated x and y coordinates. We use the set\_xlim and set\_ylim methods to manually set the start and end boundaries for the plot rather than using matplotlib’s default. Lastly, ax.set\_title adds a main title to the plot.

See the online matplotlib gallery for many more annotation examples to learn from.

Drawing shapes requires some more care. matplotlib has objects that represent many common shapes, referred to as patches. Some of these, like Rectangle and Circle, are found in matplotlib.pyplot, but the full set is located in matplotlib.patches.

To add a shape to a plot, you create the patch object shp and add it to a subplot by calling ax.add\_patch(shp) (see [Figure 9-12](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark20)):

fig = plt.figure()

ax = fig.add\_subplot(1, 1, 1)

rect = plt.Rectangle((0.2, 0.75), 0.4, 0.15, color='k', alpha=0.3) circ = plt.Circle((0.7, 0.2), 0.15, color='b', alpha=0.3)

pgon = plt.Polygon([[0.15, 0.15], [0.35, 0.4], [0.2, 0.6]],

color='g', alpha=0.5)

ax.add\_patch(rect) ax.add\_patch(circ) ax.add\_patch(pgon)

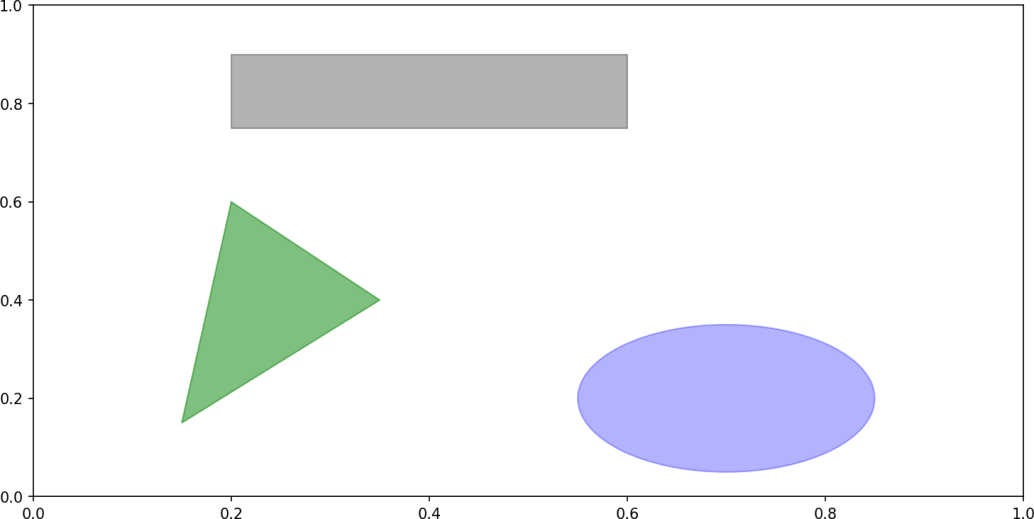


Figure 9-12. Data visualization composed from three different patches

If you look at the implementation of many familiar plot types, you will see that they are assembled from patches.

#### Saving Plots to File

You can save the active figure to file using plt.savefig. This method is equivalent to the figure object’s savefig instance method. For example, to save an SVG version of a figure, you need only type:

plt.savefig('figpath.svg')

The file type is inferred from the file extension. So if you used .pdf instead, you would get a PDF. There are a couple of important options that I use frequently for publishing graphics: dpi, which controls the dots-per-inch resolution, and bbox\_inches, which can trim the whitespace around the actual figure. To get the same plot as a PNG with minimal whitespace around the plot and at 400 DPI, you would do:

plt.savefig('figpath.png', dpi=400, bbox\_inches='tight')

savefig doesn’t have to write to disk; it can also write to any file-like object, such as a

BytesIO:

**from io import** BytesIO buffer = BytesIO() plt.savefig(buffer)

plot\_data = buffer.getvalue()

See [Table 9-2](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark21) for a list of some other options for savefig.

Table 9-2. Figure.savefig options

**Argument Description**

fname String containing a filepath or a Python file-like object. The figure format is inferred from the file

extension (e.g., .pdf for PDF or .png for PNG)

dpi The figure resolution in dots per inch; defaults to 100 out of the box but can be configured  
facecolor, edgecolor. The color of the figure background outside of the subplots; 'w' (white), by default

format The explicit file format to use ('png', 'pdf', 'svg', 'ps', 'eps', ...)

bbox\_inches The portion of the figure to save; if 'tight' is passed, will attempt to trim the empty space around the figure

#### matplotlib Configuration

matplotlib comes configured with color schemes and defaults that are geared primar‐ ily toward preparing figures for publication. Fortunately, nearly all of the default behavior can be customized via an extensive set of global parameters governing figure size, subplot spacing, colors, font sizes, grid styles, and so on. One way to modify the configuration programmatically from Python is to use the rc method; for example, to set the global default figure size to be 10 × 10, you could enter:

plt.rc('figure', figsize=(10, 10))

The first argument to rc is the component you wish to customize, such as 'figure', 'axes', 'xtick', 'ytick', 'grid', 'legend', or many others. After that can follow a sequence of keyword arguments indicating the new parameters. An easy way to write down the options in your program is as a dict:

font\_options = {'family' : 'monospace',

'weight' : 'bold',

'size' : 'small'} plt.rc('font', \*\*font\_options)

For more extensive customization and to see a list of all the options, matplotlib comes with a configuration file matplotlibrc in the matplotlib/mpl-data directory. If you cus‐ tomize this file and place it in your home directory titled .matplotlibrc, it will be loaded each time you use matplotlib.

As we’ll see in the next section, the seaborn package has several built-in plot themes or styles that use matplotlib’s configuration system internally.

### Plotting with pandas and seaborn

matplotlib can be a fairly low-level tool. You assemble a plot from its base compo‐ nents: the data display (i.e., the type of plot: line, bar, box, scatter, contour, etc.), leg‐ end, title, tick labels, and other annotations.

In pandas we may have multiple columns of data, along with row and column labels. pandas itself has built-in methods that simplify creating visualizations from Data‐ Frame and Series objects. Another library is [seaborn](https://seaborn.pydata.org/), a statistical graphics library cre‐ ated by Michael Waskom. Seaborn simplifies creating many common visualization types.



#### Line Plots

Importing seaborn modifies the default matplotlib color schemes and plot styles to improve readability and aesthetics. Even if you do not use the seaborn API, you may prefer to import seaborn as a simple way to improve the visual aesthetics of general matplotlib plots.

Series and DataFrame each have a plot attribute for making some basic plot types. By default, plot() makes line plots (see [Figure 9-13](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark22)):

In [60]: s = pd.Series(np.random.randn(10).cumsum(), index=np.arange(0, 100, 10))

In [61]: s.plot()

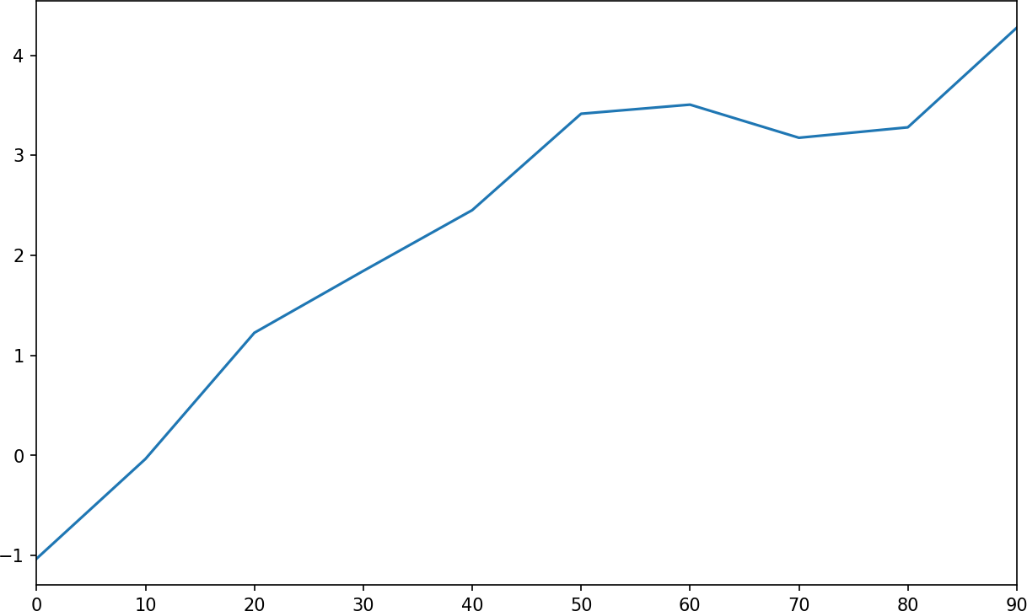


Figure 9-13. Simple Series plot

The Series object’s index is passed to matplotlib for plotting on the x-axis, though you can disable this by passing use\_index=False. The x-axis ticks and limits can be adjusted with the xticks and xlim options, and y-axis respectively with yticks and

ylim. See [Table 9-3](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark24) for a full listing of plot options. I’ll comment on a few more of them throughout this section and leave the rest to you to explore.

Most of pandas’s plotting methods accept an optional ax parameter, which can be a matplotlib subplot object. This gives you more flexible placement of subplots in a grid layout.

DataFrame’s plot method plots each of its columns as a different line on the same subplot, creating a legend automatically (see [Figure 9-14](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark23)):

In [62]: df = pd.DataFrame(np.random.randn(10, 4).cumsum(0),

....: columns=['A', 'B', 'C', 'D'],

....: index=np.arange(0, 100, 10))

In [63]: df.plot()

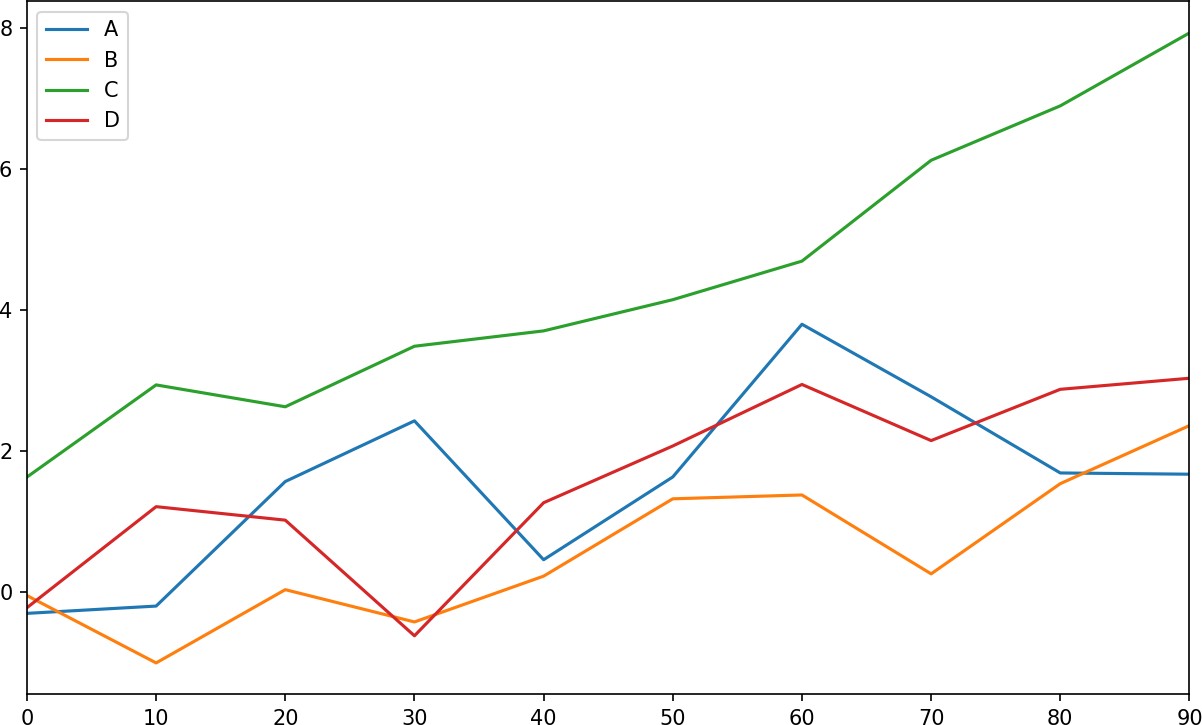


Figure 9-14. Simple DataFrame plot

The plot attribute contains a “family” of methods for different plot types. For exam‐ ple, df.plot() is equivalent to df.plot.line(). We’ll explore some of these methods next.

Additional keyword arguments to plot are passed through to the respective matplotlib plotting function, so you can further custom‐ ize these plots by learning more about the matplotlib API.

Table 9-3. Series.plot method arguments

**Argument Description**

label Label for plot legend

ax matplotlib subplot object to plot on; if nothing passed, uses active matplotlib subplot

style Style string, like 'ko--', to be passed to matplotlib

alpha The plot fill opacity (from 0 to 1)

kind Can be 'area', 'bar', 'barh', 'density', 'hist', 'kde', 'line', 'pie' logy Use logarithmic scaling on the y-axis

use\_index Use the object index for tick labels rot Rotation of tick labels (0 through 360) xticks Values to use for x-axis ticks

yticks Values to use for y-axis ticks

xlim x-axis limits (e.g., [0, 10])

ylim y-axis limits

grid Display axis grid (on by default)

DataFrame has a number of options allowing some flexibility with how the columns are handled; for example, whether to plot them all on the same subplot or to create separate subplots. See [Table 9-4](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark25) for more on these.

Table 9-4. DataFrame-specific plot arguments

**Argument Description**

subplots Plot each DataFrame column in a separate subplot

sharex If subplots=True, share the same x-axis, linking ticks and limits

sharey If subplots=True, share the same y-axis

figsize Size of figure to create as tuple

title Plot title as string

legend Add a subplot legend (True by default)

sort\_columns Plot columns in alphabetical order; by default uses existing column order

For time series plotting, see Chapter 11.

#### Bar Plots

The plot.bar() and plot.barh() make vertical and horizontal bar plots, respec‐ tively. In this case, the Series or DataFrame index will be used as the x (bar) or y (barh) ticks (see [Figure 9-15](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark26)):

In [64]: fig, axes = plt.subplots(2, 1)

In [65]: data = pd.Series(np.random.rand(16), index=list('abcdefghijklmnop')) In [66]: data.plot.bar(ax=axes[0], color='k', alpha=0.7)

Out[66]: <matplotlib.axes.\_subplots.AxesSubplot at 0x7fb62493d470> In [67]: data.plot.barh(ax=axes[1], color='k', alpha=0.7)

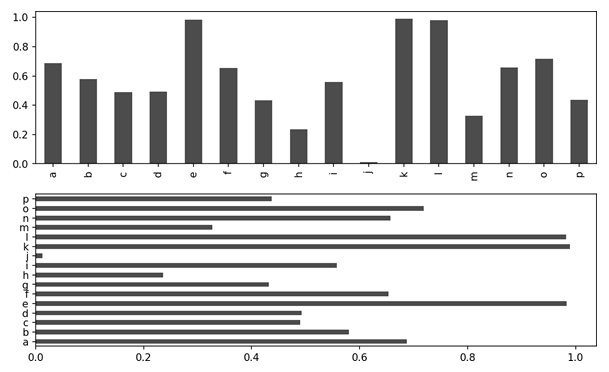


Figure 9-15. Horizonal and vertical bar plot

The options color='k' and alpha=0.7 set the color of the plots to black and use par‐ tial transparency on the filling.

With a DataFrame, bar plots group the values in each row together in a group in bars, side by side, for each value. See [Figure 9-16](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark27):

In [69]: df = pd.DataFrame(np.random.rand(6, 4),

....: index=['one', 'two', 'three', 'four', 'five', 'six'],

....: columns=pd.Index(['A', 'B', 'C', 'D'], name='Genus'))

|  |  |  |  |
| --- | --- | --- | --- |
| In [70]: df |  | | |
| Out[70]: |
| Genus A | B | C | D |
| one 0.370670 | 0.602792 | 0.229159 | 0.486744 |
| two 0.420082 | 0.571653 | 0.049024 | 0.880592 |
| three 0.814568 | 0.277160 | 0.880316 | 0.431326 |
| four 0.374020 | 0.899420 | 0.460304 | 0.100843 |
| five 0.433270 | 0.125107 | 0.494675 | 0.961825 |
| six 0.601648 | 0.478576 | 0.205690 | 0.560547 |

In [71]: df.plot.bar()

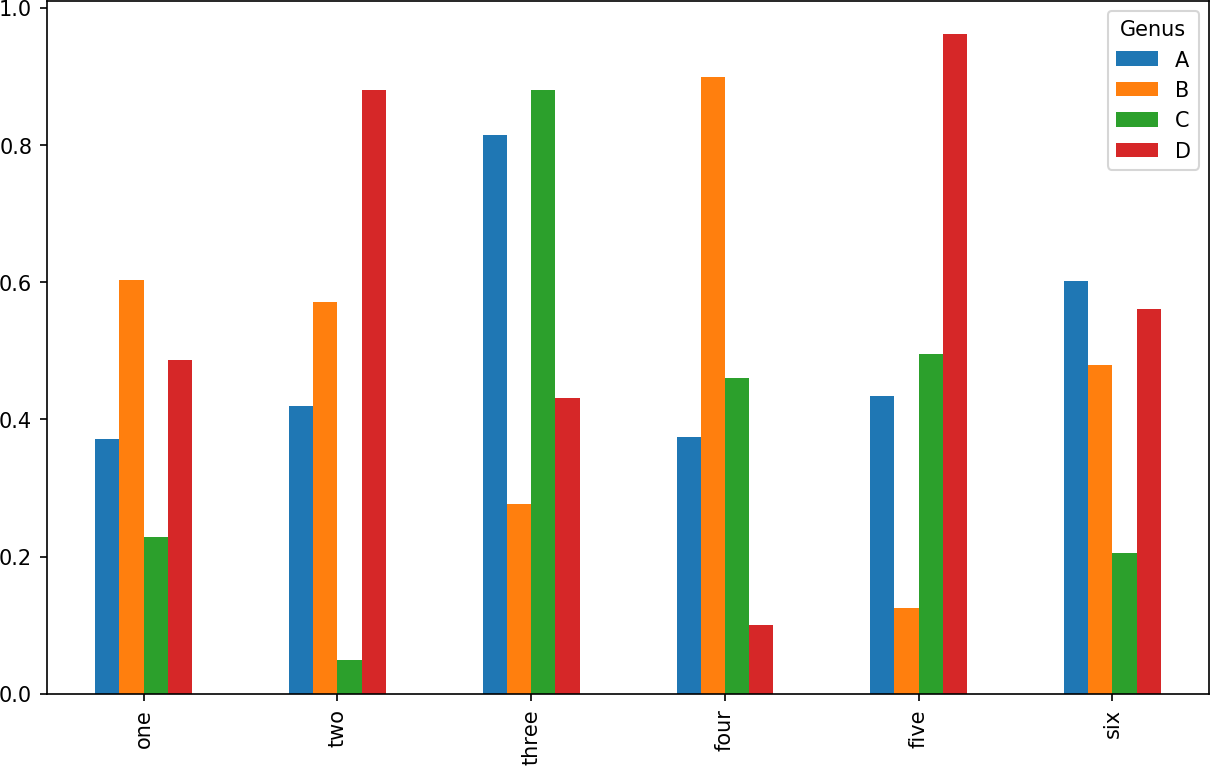


Figure 9-16. DataFrame bar plot

Note that the name “Genus” on the DataFrame’s columns is used to title the legend.

We create stacked bar plots from a DataFrame by passing stacked=True, resulting in the value in each row being stacked together (see [Figure 9-17](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark28)):

In [73]: df.plot.barh(stacked=True, alpha=0.5)

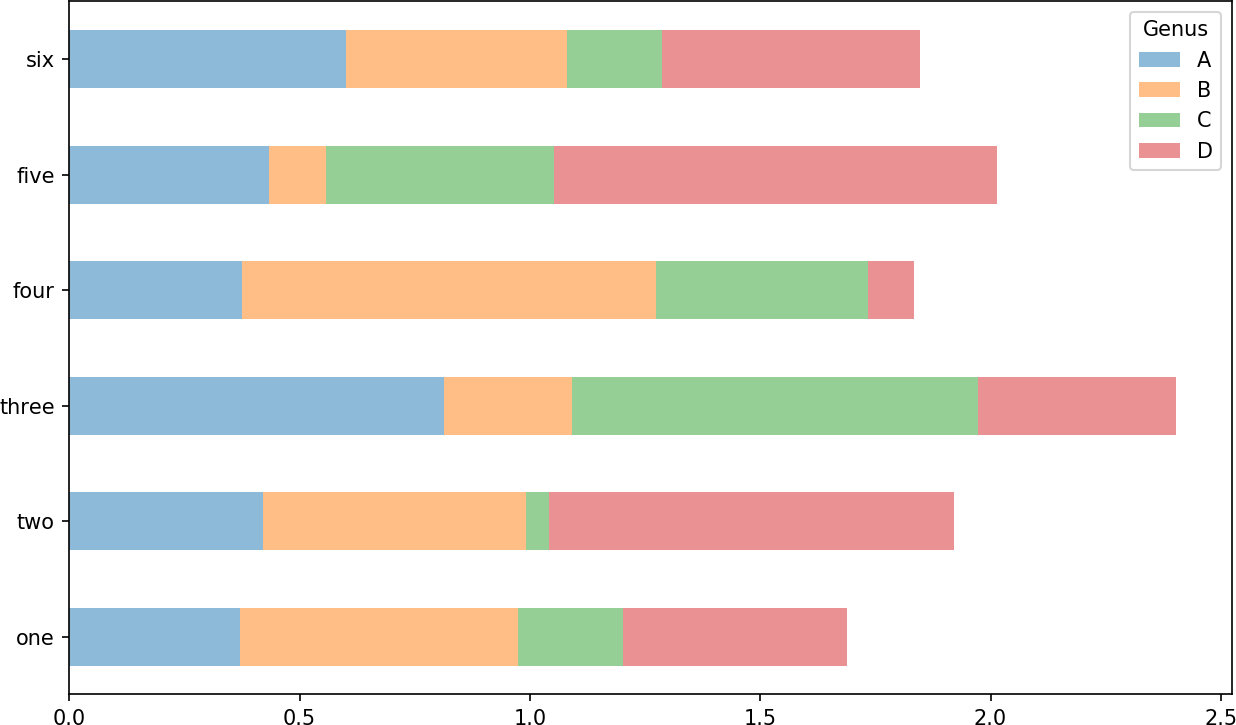


Figure 9-17. DataFrame stacked bar plot

A useful recipe for bar plots is to visualize a Series’s value frequency using value\_counts: s.value\_counts().plot.bar().

Returning to the tipping dataset used earlier in the book, suppose we wanted to make a stacked bar plot showing the percentage of data points for each party size on each day. I load the data using read\_csv and make a cross-tabulation by day and party size:

In [75]: tips = pd.read\_csv('examples/tips.csv')

In [76]: party\_counts = pd.crosstab(tips['day'], tips['size']) In [77]: party\_counts

Out[77]:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| size | 1 | 2 | 3 | 4 | 5 | 6 |
| day |  |  |  |  |  |  |
| Fri | 1 | 16 | 1 | 1 | 0 | 0 |
| Sat | 2 | 53 | 18 | 13 | 1 | 0 |
| Sun | 0 | 39 | 15 | 18 | 3 | 1 |
| Thur | 1 | 48 | 4 | 5 | 1 | 3 |

*# Not many 1- and 6-person parties*

In [78]: party\_counts = party\_counts.loc[:, 2:5]

Then, normalize so that each row sums to 1 and make the plot (see [Figure 9-18](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark29)):

*# Normalize to sum to 1*

In [79]: party\_pcts = party\_counts.div(party\_counts.sum(1), axis=0)

In [80]: party\_pcts Out[80]:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| size | 2 | 3 | 4 | 5 |
| day |  |  |  |  |
| Fri | 0.888889 | 0.055556 | 0.055556 | 0.000000 |
| Sat | 0.623529 | 0.211765 | 0.152941 | 0.011765 |
| Sun | 0.520000 | 0.200000 | 0.240000 | 0.040000 |
| Thur | 0.827586 | 0.068966 | 0.086207 | 0.017241 |

In [81]: party\_pcts.plot.bar()

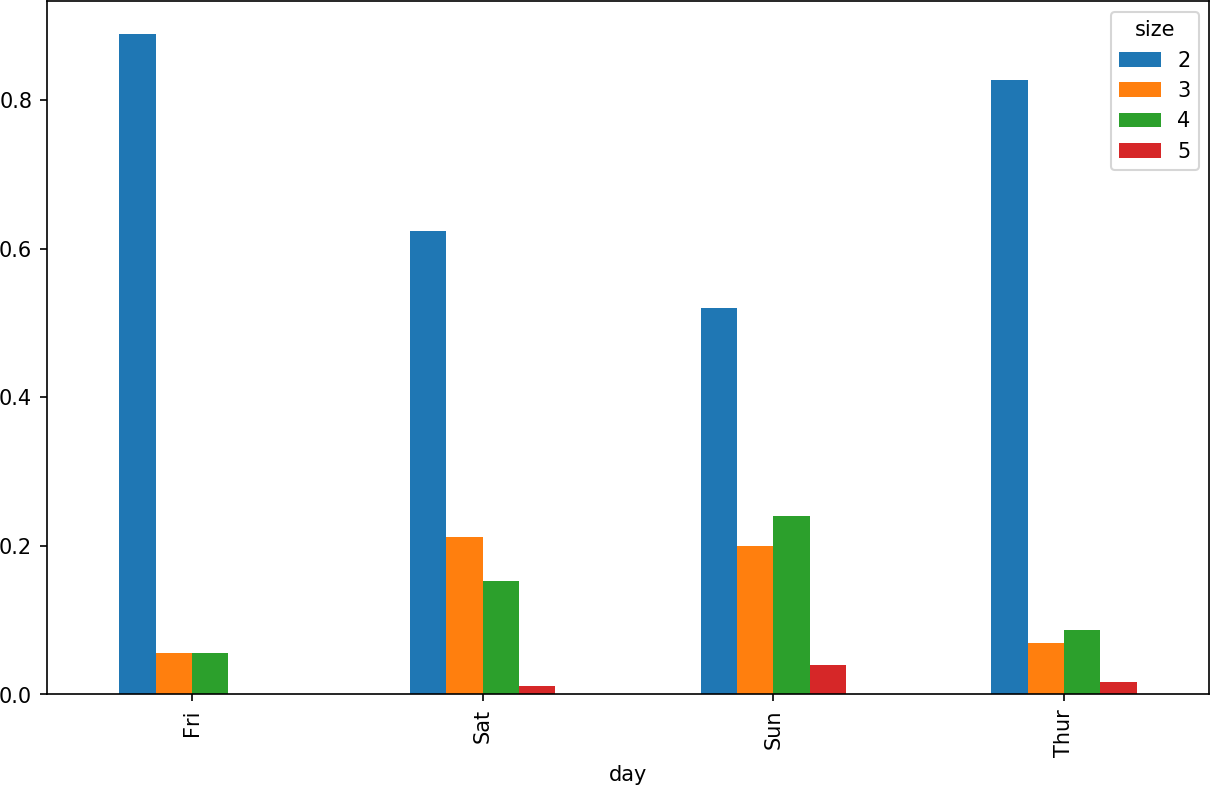


Figure 9-18. Fraction of parties by size on each day

So you can see that party sizes appear to increase on the weekend in this dataset.

With data that requires aggregation or summarization before making a plot, using the seaborn package can make things much simpler. Let’s look now at the tipping per‐ centage by day with seaborn (see [Figure 9-19](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark30) for the resulting plot):

In [83]: **import seaborn as sns**

In [84]: tips['tip\_pct'] = tips['tip'] / (tips['total\_bill'] - tips['tip']) In [85]: tips.head()

Out[85]:

total\_bill tip smoker day time size tip\_pct 0 16.99 1.01 No Sun Dinner 2 0.063204

1 10.34 1.66 No Sun Dinner 3 0.191244

2 21.01 3.50 No Sun Dinner 3 0.199886

3 23.68 3.31 No Sun Dinner 2 0.162494

4 24.59 3.61 No Sun Dinner 4 0.172069

In [86]: sns.barplot(x='tip\_pct', y='day', data=tips, orient='h')

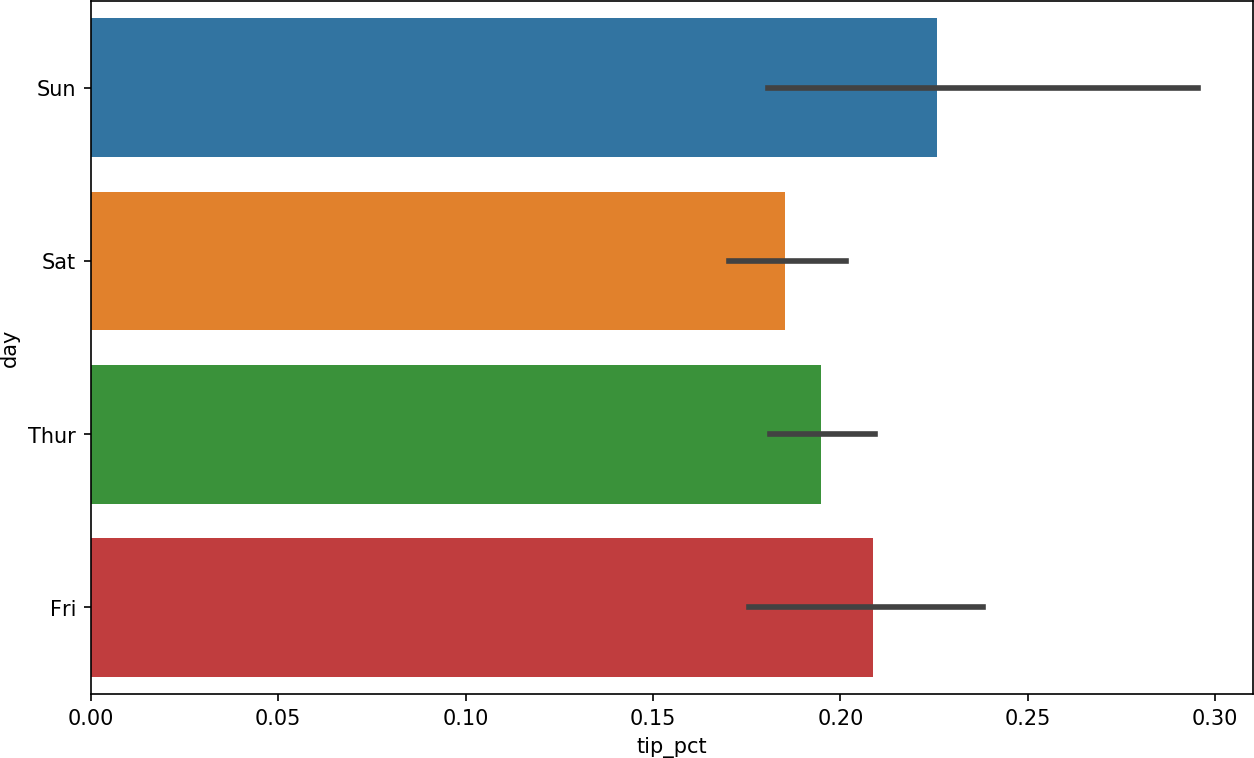


Figure 9-19. Tipping percentage by day with error bars

Plotting functions in seaborn take a data argument, which can be a pandas Data‐ Frame. The other arguments refer to column names. Because there are multiple observations for each value in the day, the bars are the average value of tip\_pct. The black lines drawn on the bars represent the 95% confidence interval (this can be con‐ figured through optional arguments).

seaborn.barplot has a hue option that enables us to split by an additional categorical value ([Figure 9-20](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark31)):

In [88]: sns.barplot(x='tip\_pct', y='day', hue='time', data=tips, orient='h')

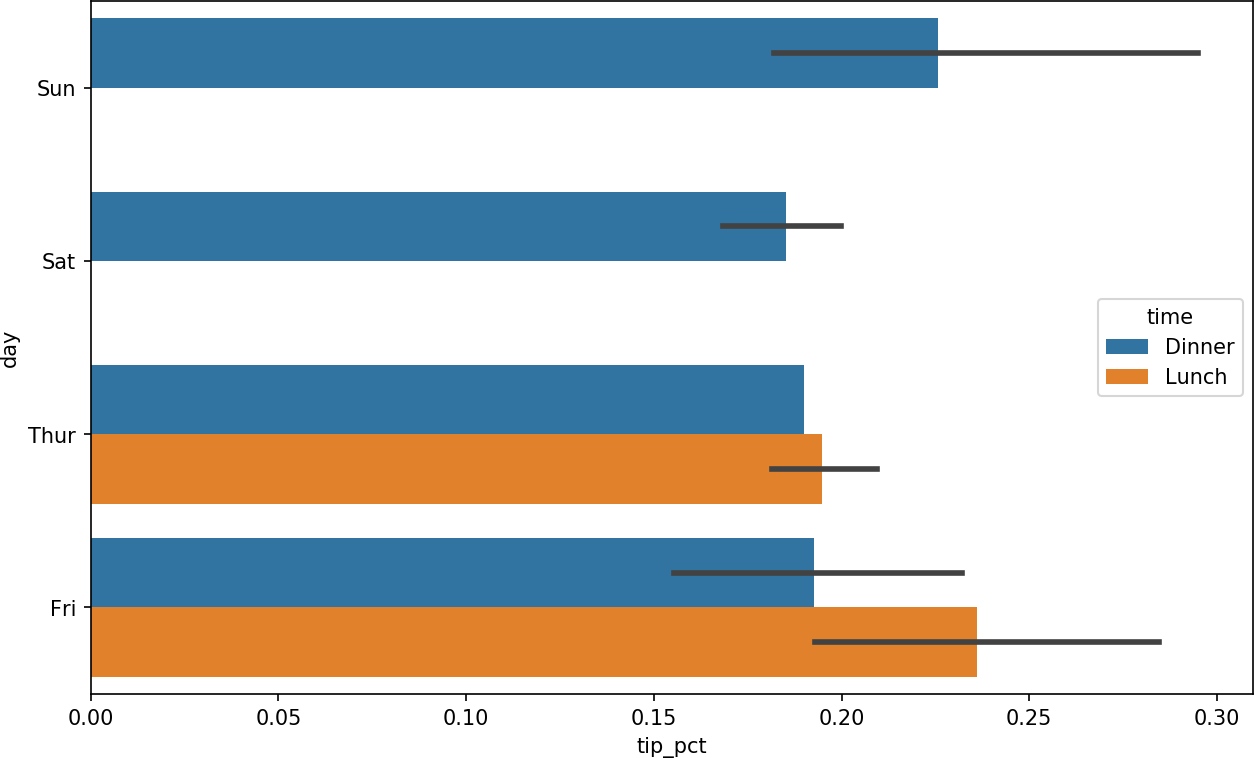


Figure 9-20. Tipping percentage by day and time

Notice that seaborn has automatically changed the aesthetics of plots: the default color palette, plot background, and grid line colors. You can switch between different plot appearances using seaborn.set:

In [90]: sns.set(style="whitegrid")

#### Histograms and Density Plots

A histogram is a kind of bar plot that gives a discretized display of value frequency. The data points are split into discrete, evenly spaced bins, and the number of data points in each bin is plotted. Using the tipping data from before, we can make a histo‐ gram of tip percentages of the total bill using the plot.hist method on the Series (see [Figure 9-21](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark32)):

In [92]: tips['tip\_pct'].plot.hist(bins=50)

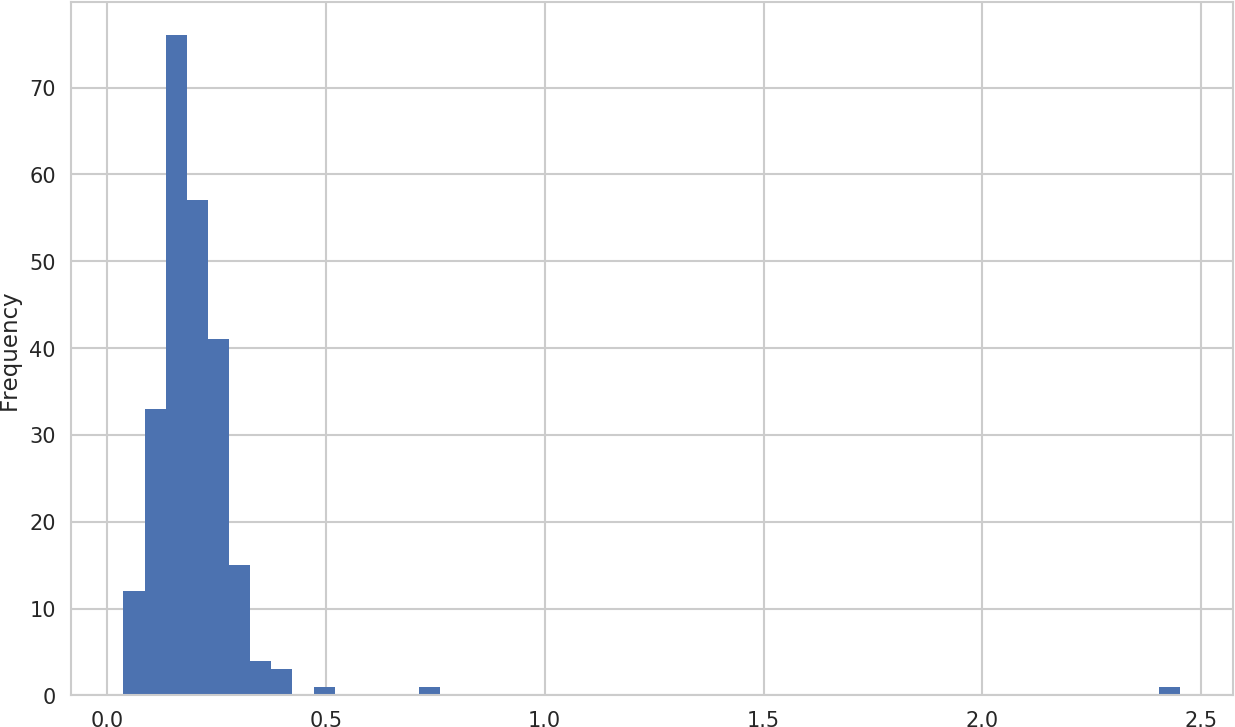


Figure 9-21. Histogram of tip percentages

A related plot type is a density plot, which is formed by computing an estimate of a continuous probability distribution that might have generated the observed data. The usual procedure is to approximate this distribution as a mixture of “kernels”—that is, simpler distributions like the normal distribution. Thus, density plots are also known as kernel density estimate (KDE) plots. Using plot.kde makes a density plot using the conventional mixture-of-normals estimate (see [Figure 9-22](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark33)):

In [94]: tips['tip\_pct'].plot.density()

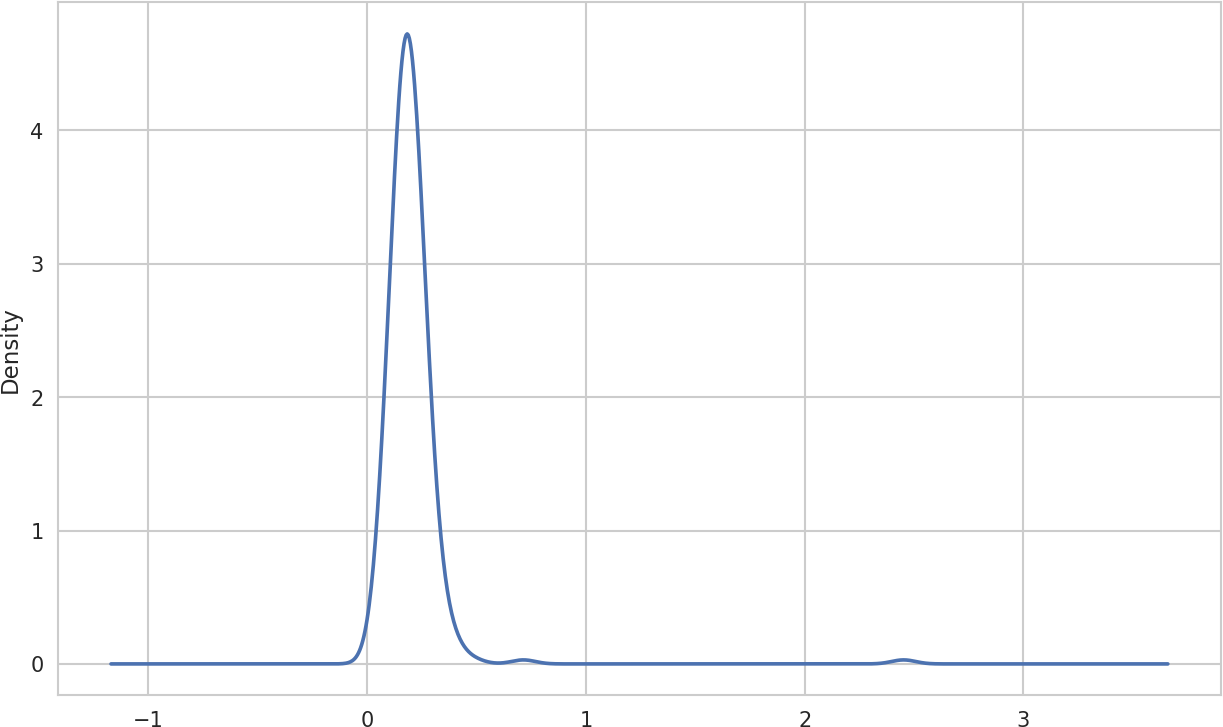


Figure 9-22. Density plot of tip percentages

Seaborn makes histograms and density plots even easier through its distplot method, which can plot both a histogram and a continuous density estimate simulta‐ neously. As an example, consider a bimodal distribution consisting of draws from two different standard normal distributions (see [Figure 9-23](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark34)):

In [96]: comp1 = np.random.normal(0, 1, size=200) In [97]: comp2 = np.random.normal(10, 2, size=200)

In [98]: values = pd.Series(np.concatenate([comp1, comp2])) In [99]: sns.distplot(values, bins=100, color='k')

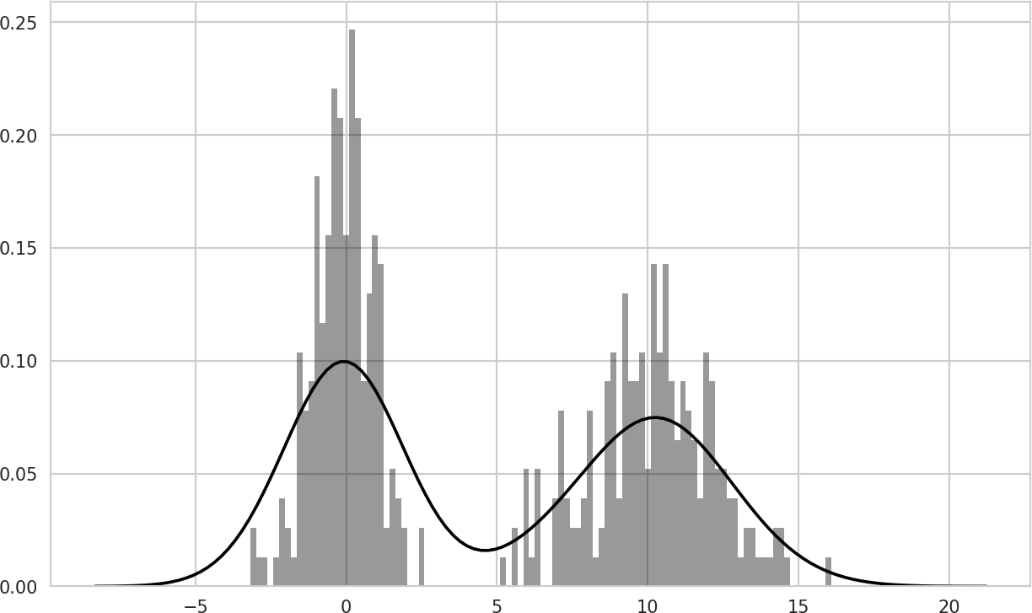


Figure 9-23. Normalized histogram of normal mixture with density estimate

#### Scatter or Point Plots

Point plots or scatter plots can be a useful way of examining the relationship between two one-dimensional data series. For example, here we load the macrodata dataset from the statsmodels project, select a few variables, then compute log differences:

In [100]: macro = pd.read\_csv('examples/macrodata.csv')

In [101]: data = macro[['cpi', 'm1', 'tbilrate', 'unemp']] In [102]: trans\_data = np.log(data).diff().dropna()

In [103]: trans\_data[-5:] Out[103]:

|  |  |  |
| --- | --- | --- |
| cpi | m1 tbilrate | unemp |
| 198 -0.007904 | 0.045361 -0.396881 | 0.105361 |
| 199 -0.021979 | 0.066753 -2.277267 | 0.139762 |
| 200 0.002340 | 0.010286 0.606136 | 0.160343 |
| 201 0.008419 | 0.037461 -0.200671 | 0.127339 |
| 202 0.008894 | 0.012202 -0.405465 | 0.042560 |

We can then use seaborn’s regplot method, which makes a scatter plot and fits a lin‐ ear regression line (see [Figure 9-24](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark35)):

In [105]: sns.regplot('m1', 'unemp', data=trans\_data)

Out[105]: <matplotlib.axes.\_subplots.AxesSubplot at 0x7fb613720be0>

In [106]: plt.title('Changes in log %s versus log %s' % ('m1', 'unemp'))

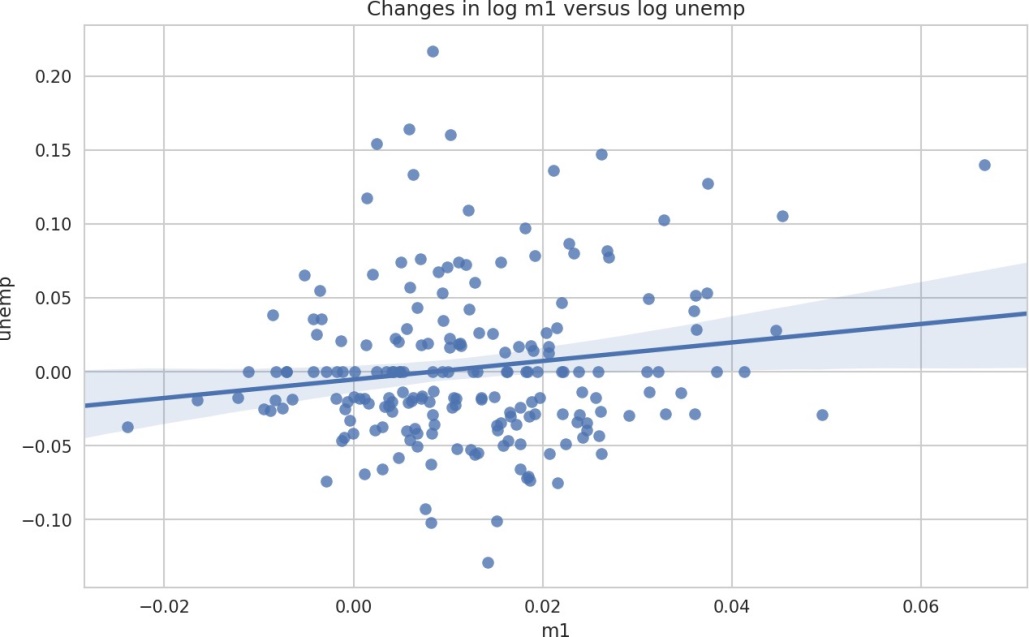


Figure 9-24. A seaborn regression/scatter plot

In exploratory data analysis it’s helpful to be able to look at all the scatter plots among a group of variables; this is known as a pairs plot or scatter plot matrix. Making such a plot from scratch is a bit of work, so seaborn has a convenient pairplot function, which supports placing histograms or density estimates of each variable along the diagonal (see [Figure 9-25](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark36) for the resulting plot):

In [107]: sns.pairplot(trans\_data, diag\_kind='kde', plot\_kws={'alpha': 0.2})



Figure 9-25. Pair plot matrix of statsmodels macro data

You may notice the plot\_kws argument. This enables us to pass down configuration options to the individual plotting calls on the off-diagonal elements. Check out the seaborn.pairplot docstring for more granular configuration options.

Facet Grids and Categorical Data

What about datasets where we have additional grouping dimensions? One way to vis‐ ualize data with many categorical variables is to use a facet grid. Seaborn has a useful built-in function factorplot that simplifies making many kinds of faceted plots (see [Figure 9-26](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark37) for the resulting plot):

In [108]: sns.factorplot(x='day', y='tip\_pct', hue='time', col='smoker',

.....: kind='bar', data=tips[tips.tip\_pct < 1])

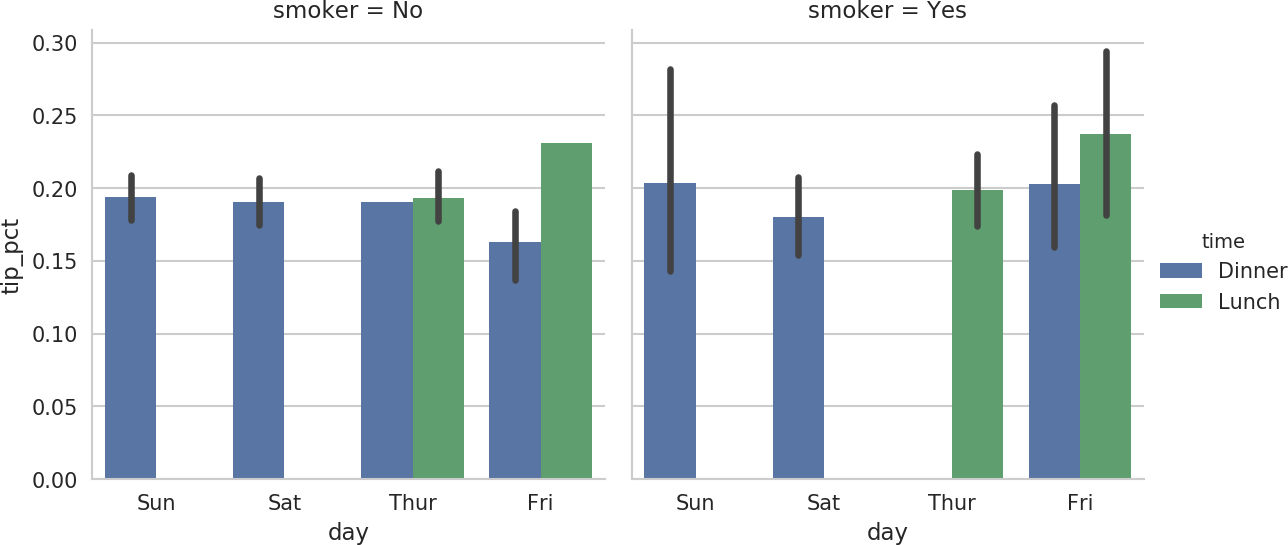


Figure 9-26. Tipping percentage by day/time/smoker

Instead of grouping by 'time' by different bar colors within a facet, we can also expand the facet grid by adding one row per time value ([Figure 9-27](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark38)):

In [109]: sns.factorplot(x='day', y='tip\_pct', row='time',

.....: col='smoker',

.....: kind='bar', data=tips[tips.tip\_pct < 1])

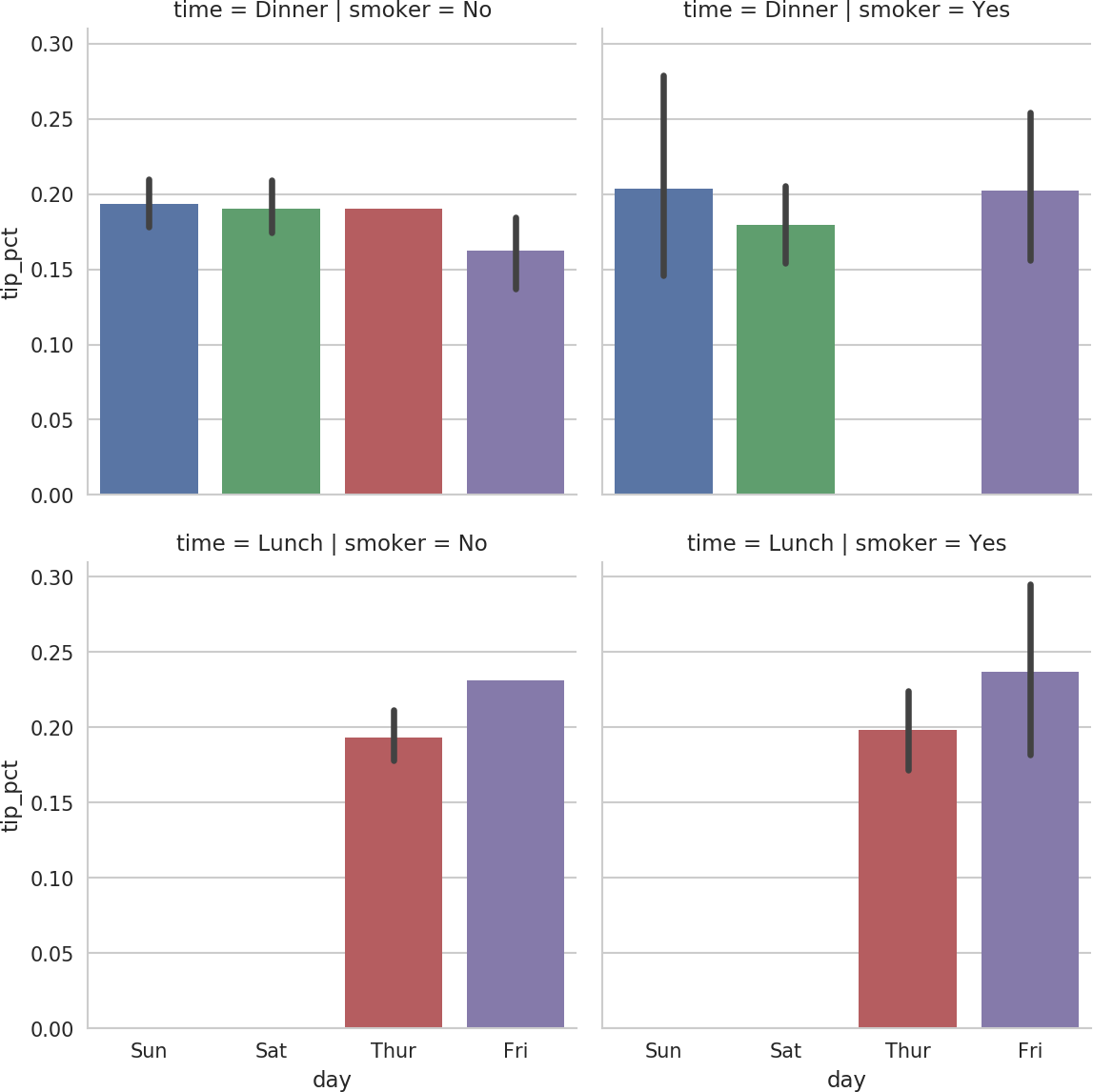


Figure 9-27. tip\_pct by day; facet by time/smoker

factorplot supports other plot types that may be useful depending on what you are trying to display. For example, box plots (which show the median, quartiles, and out‐ liers) can be an effective visualization type ([Figure 9-28](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(1).docx#_bookmark39)):

In [110]: sns.factorplot(x='tip\_pct', y='day', kind='box',

.....: data=tips[tips.tip\_pct < 0.5])

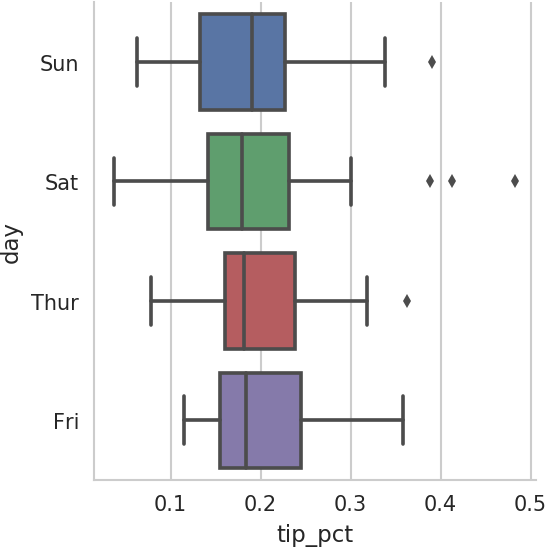


Figure 9-28. Box plot of tip\_pct by day

You can create your own facet grid plots using the more general seaborn.FacetGrid

class. See the [seaborn documentation](https://seaborn.pydata.org/) for more.

### Other Python Visualization Tools

As is common with open source, there are a plethora of options for creating graphics in Python (too many to list). Since 2010, much development effort has been focused on creating interactive graphics for publication on the web. With tools like [Bokeh](http://bokeh.pydata.org/) and [Plotly](https://github.com/plotly/plotly.py), it’s now possible to specify dynamic, interactive graphics in Python that are destined for a web browser.

For creating static graphics for print or web, I recommend defaulting to matplotlib and add-on libraries like pandas and seaborn for your needs. For other data visualiza‐ tion requirements, it may be useful to learn one of the other available tools out there. I encourage you to explore the ecosystem as it continues to involve and innovate into the future.

### Conclusion

The goal of this chapter was to get your feet wet with some basic data visualization using pandas, matplotlib, and seaborn. If visually communicating the results of data analysis is important in your work, I encourage you to seek out resources to learn more about effective data visualization. It is an active field of research and you can practice with many excellent learning resources available online and in print form.

In the next chapter, we turn our attention to data aggregation and group operations with pandas.

## Data Aggregation and Group Operations

Categorizing a dataset and applying a function to each group, whether an aggregation or transformation, is often a critical component of a data analysis workflow. After loading, merging, and preparing a dataset, you may need to compute group statistics or possibly pivot tables for reporting or visualization purposes. pandas provides a flexible groupby interface, enabling you to slice, dice, and summarize datasets in a natural way.

One reason for the popularity of relational databases and SQL (which stands for “structured query language”) is the ease with which data can be joined, filtered, trans‐ formed, and aggregated. However, query languages like SQL are somewhat con‐ strained in the kinds of group operations that can be performed. As you will see, with the expressiveness of Python and pandas, we can perform quite complex group oper‐ ations by utilizing any function that accepts a pandas object or NumPy array. In this chapter, you will learn how to:

* Split a pandas object into pieces using one or more keys (in the form of func‐ tions, arrays, or DataFrame column names)
* Calculate group summary statistics, like count, mean, or standard deviation, or a user-defined function
* Apply within-group transformations or other manipulations, like normalization, linear regression, rank, or subset selection
* Compute pivot tables and cross-tabulations
* Perform quantile analysis and other statistical group analyses

Aggregation of time series data, a special use case of groupby, is referred to as resampling in this book and will receive separate treatment in [Chapter 11](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark6).

### GroupBy Mechanics

Hadley Wickham, an author of many popular packages for the R programming lan‐ guage, coined the term split-apply-combine for describing group operations. In the first stage of the process, data contained in a pandas object, whether a Series, Data‐ Frame, or otherwise, is split into groups based on one or more keys that you provide. The splitting is performed on a particular axis of an object. For example, a DataFrame can be grouped on its rows (axis=0) or its columns (axis=1). Once this is done, a function is applied to each group, producing a new value. Finally, the results of all those function applications are combined into a result object. The form of the result‐ ing object will usually depend on what’s being done to the data. See [Figure 10-1](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark1) for a mockup of a simple group aggregation.

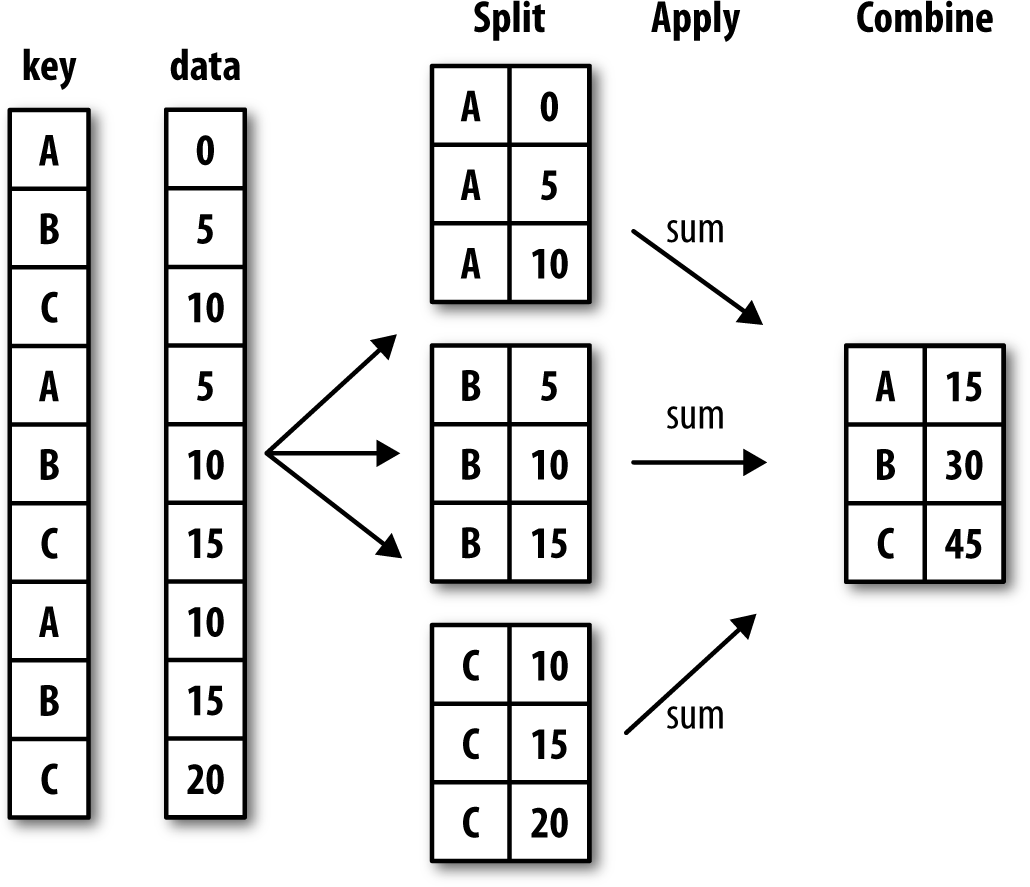


Figure 10-1. Illustration of a group aggregation

Each grouping key can take many forms, and the keys do not have to be all of the same type:

A list or array of values that is the same length as the axis being grouped

A value indicating a column name in a DataFrame

A dict or Series giving a correspondence between the values on the axis being grouped and the group names

A function to be invoked on the axis index or the individual labels in the index

Note that the latter three methods are shortcuts for producing an array of values to be used to split up the object. Don’t worry if this all seems abstract. Throughout this chapter, I will give many examples of all these methods. To get started, here is a small tabular dataset as a DataFrame:

df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],

....: 'key2' : ['one', 'two', 'one', 'two', 'one'],

....: 'data1' : np.random.randn(5),

....: 'data2' : np.random.randn(5)})

df

*data1 data2 key1 key2*

*0 -0.204708 1.393406 a one*

*1 0.478943 0.092908 a two*

*2 -0.519439 0.281746 b one*

*3 -0.555730 0.769023 b two*

*4 1.965781 1.246435 a one*

Suppose you wanted to compute the mean of the data1 column using the labels from key1. There are a number of ways to do this. One is to access data1 and call groupby with the column (a Series) at key1:

grouped = df['data1'].groupby(df['key1'])

grouped

*<pandas.core.groupby.SeriesGroupBy object at 0x7faa31537390>*

This grouped variable is now a GroupBy object. It has not actually computed anything yet except for some intermediate data about the group key df['key1']. The idea is that this object has all of the information needed to then apply some operation to each of the groups. For example, to compute group means we can call the GroupBy’s mean method:

grouped.mean()

key1

*a 0.746672*

*b -0.537585*

*Name: data1, dtype: float64*

Later, I’ll explain more about what happens when you call .mean(). The important thing here is that the data (a Series) has been aggregated according to the group key, producing a new Series that is now indexed by the unique values in the key1 column.

The result index has the name 'key1' because the DataFrame column df['key1'] did.

If instead we had passed multiple arrays as a list, we’d get something different:

means = df['data1'].groupby([df['key1'], df['key2']]).mean()

means

*key1 key2*

*a one 0.880536*

*two 0.478943*

*b one -0.519439*

*two -0.555730*

*Name: data1, dtype: float64*

Here we grouped the data using two keys, and the resulting Series now has a hier‐ archical index consisting of the unique pairs of keys observed:

means.unstack()

*key2 one two key1*

*a 0.880536 0.478943*

*b -0.519439 -0.555730*

In this example, the group keys are all Series, though they could be any arrays of the right length:

states = np.array(['Ohio', 'California', 'California', 'Ohio', 'Ohio'])

years = np.array([2005, 2005, 2006, 2005, 2006])

df['data1'].groupby([states, years]).mean()

*California 2005 0.478943*

*2006 -0.519439*

*Ohio 2005 -0.380219*

*2006 1.965781*

Name: data1, dtype: float64

Frequently the grouping information is found in the same DataFrame as the data you want to work on. In that case, you can pass column names (whether those are strings, numbers, or other Python objects) as the group keys:

df.groupby('key1').mean()

*data1 data2*

*key1*

*a 0.746672 0.910916*

*b -0.537585 0.525384*

df.groupby(['key1', 'key2']).mean()

*data1 data2*

*key1 key2*

*a one 0.880536 1.319920*

*two 0.478943 0.092908*

*b one -0.519439 0.281746*

*two -0.555730 0.769023*

You may have noticed in the first case df.groupby('key1').mean() that there is no key2 column in the result. Because df['key2'] is not numeric data, it is said to be a nuisance column, which is therefore excluded from the result. By default, all of the numeric columns are aggregated, though it is possible to filter down to a subset, as you’ll see soon.

Regardless of the objective in using groupby, a generally useful GroupBy method is

size, which returns a Series containing group sizes:

df.groupby(['key1', 'key2']).size()

key1 key2

a one 2

b

two 1

c one 1

d

two 1

dtype: int64

Take note that any missing values in a group key will be excluded from the result.

#### Iterating Over Groups

The GroupBy object supports iteration, generating a sequence of 2-tuples containing the group name along with the chunk of data. Consider the following:

for name, group in df.groupby('key1'):

....: print(name)

....: print(group)

....:

a

*data1 data2 key1 key2*

*0 -0.204708 1.393406 a one*

*1 0.478943 0.092908 a two*

*4 1.965781 1.246435 a one*

*b*

*data1 data2 key1 key2*

*2 -0.519439 0.281746 b one*

*3 -0.555730 0.769023 b two*

In the case of multiple keys, the first element in the tuple will be a tuple of key values:

for (k1, k2), group in df.groupby(['key1', 'key2']):

....: print((k1, k2))

....: print(group)

....:

*('a', 'one')*

*data1 data2 key1 key2 0 -0.204708 1.393406 a one*

*4 1.965781 1.246435 a one*

*('a', 'two')*

*data1 data2 key1 key2 1 0.478943 0.092908 a two*

*('b', 'one')*

*data1 data2 key1 key2 2 -0.519439 0.281746 b one*

*('b', 'two')*

*data1 data2 key1 key2 3 -0.55573 0.769023 b two*

Of course, you can choose to do whatever you want with the pieces of data. A recipe you may find useful is computing a dict of the data pieces as a one-liner:

pieces = dict(list(df.groupby('key1')))

pieces['b']

*data1 data2 key1 key2 2 -0.519439 0.281746 b one*

*3 -0.555730 0.769023 b two*

By default groupby groups on axis=0, but you can group on any of the other axes. For example, we could group the columns of our example df here by dtype like so:

df.dtypes

*data1 float64*

*data2 float64*

*key1 object*

*key2 object dtype: object*

grouped = df.groupby(df.dtypes, axis=1)

We can print out the groups like so:

for dtype, group in grouped:

....: print(dtype)

....: print(group)

....:

float64

data1 data2 0 -0.204708 1.393406

1 0.478943 0.092908

2 -0.519439 0.281746

3 -0.555730 0.769023

4 1.965781 1.246435

object

key1 key2

0

a one

1

2 a two

3

4 b one

5

6 b two

7

8 a one

9

#### Selecting a Column or Subset of Columns

Indexing a GroupBy object created from a DataFrame with a column name or array of column names has the effect of column subsetting for aggregation. This means that:

df.groupby('key1')['data1']

df.groupby('key1')[['data2']]

are syntactic sugar for:

df['data1'].groupby(df['key1'])

df[['data2']].groupby(df['key1'])

Especially for large datasets, it may be desirable to aggregate only a few columns. For example, in the preceding dataset, to compute means for just the data2 column and get the result as a DataFrame, we could write:

In [31]: **df.groupby([**'key1', 'key2'**])[['data2']].mean() Out[31]:**

*data2*

*key1 key2*

*a one 1.319920*

*two 0.092908*

*b one 0.281746*

*two 0.769023*

The object returned by this indexing operation is a grouped DataFrame if a list or array is passed or a grouped Series if only a single column name is passed as a scalar:

s\_grouped **= df.groupby([**'key1', 'key2'**])['data2']**

**s\_grouped**

*<pandas.core.groupby.SeriesGroupBy object at 0x7faa30c78da0>*

**s\_grouped.mean()** Out[34]:

key1 key2

*a one 1.319920*

*two 0.092908*

*b one 0.281746*

*two 0.769023*

*Name: data2, dtype: float64*

#### Grouping with Dicts and Series

Grouping information may exist in a form other than an array. Let’s consider another example DataFrame:

In [35]: people = pd.DataFrame(np.random.randn(5, 5),

....: columns=['a', 'b', 'c', 'd', 'e'],

....: index=['Joe', 'Steve', 'Wes', 'Jim', 'Travis'])

people.iloc[2:3, [1, 2]] = np.nan # Add a few NA values

people

*a b c d e*

*Joe 1.007189 -1.296221 0.274992 0.228913 1.352917*

*Steve 0.886429 -2.001637 -0.371843 1.669025 -0.438570*

*Wes -0.539741 NaN NaN -1.021228 -0.577087*

*Jim 0.124121 0.302614 0.523772 0.000940 1.343810*

*Travis -0.713544 -0.831154 -2.370232 -1.860761 -0.860757*

Now, suppose I have a group correspondence for the columns and want to sum together the columns by group:

mapping = {'a': 'red', 'b': 'red', 'c': 'blue',

....: 'd': 'blue', 'e': 'red', 'f' : 'orange'}

Now, you could construct an array from this dict to pass to groupby, but instead we can just pass the dict (I included the key 'f' to highlight that unused grouping keys are OK):

by\_column = people.groupby(mapping, axis=1)

by\_column.sum()

*blue red Joe 0.503905 1.063885*

*Steve 1.297183 -1.553778*

*Wes -1.021228 -1.116829*

*Jim 0.524712 1.770545*

*Travis -4.230992 -2.405455*

The same functionality holds for Series, which can be viewed as a fixed-size mapping:

map\_series = pd.Series(mapping)

map\_series

*a red*

*b*

*c red*

*d*

*e blue*

*f*

*g blue*

*h*

*i red*

*j*

*k orange*

*l*

dtype: object

people.groupby(map\_series, axis=1).count()

blue red

Joe 2 3

Steve 2 3

Wes 1 2

Jim 2 3

Travis 2 3

#### Grouping with Functions

Using Python functions is a more generic way of defining a group mapping compared with a dict or Series. Any function passed as a group key will be called once per index value, with the return values being used as the group names. More concretely, con‐ sider the example DataFrame from the previous section, which has people’s first names as index values. Suppose you wanted to group by the length of the names; while you could compute an array of string lengths, it’s simpler to just pass the len function:

people.groupby(len).sum()

*a b c d e*

*3 0.591569 -0.993608 0.798764 -0.791374 2.119639*

*5 0.886429 -2.001637 -0.371843 1.669025 -0.438570*

*6 -0.713544 -0.831154 -2.370232 -1.860761 -0.860757*

Mixing functions with arrays, dicts, or Series is not a problem as everything gets con‐ verted to arrays internally:

key\_list = ['one', 'one', 'one', 'two', 'two']

people.groupby([len, key\_list]).min()

*a b c d e 3*

*one -0.539741 -1.296221 0.274992 -1.021228 -0.577087*

*two 0.124121 0.302614 0.523772 0.000940 1.343810*

*5 one 0.886429 -2.001637 -0.371843 1.669025 -0.438570*

*6 two -0.713544 -0.831154 -2.370232 -1.860761 -0.860757*

#### Grouping by Index Levels

A final convenience for hierarchically indexed datasets is the ability to aggregate using one of the levels of an axis index. Let’s look at an example:

columns = pd.MultiIndex.from\_arrays([['US', 'US', 'US', 'JP', 'JP'],

....: [1, 3, 5, 1, 3]],

....: names=['cty', 'tenor'])

hier\_df = pd.DataFrame(np.random.randn(4, 5), columns=columns)

**hier\_df**

cty US JP

tenor 1 3 5 1 3

0 0.560145 -1.265934 0.119827 -1.063512 0.332883

1 -2.359419 -0.199543 -1.541996 -0.970736 -1.307030

2 0.286350 0.377984 -0.753887 0.331286 1.349742

3 0.069877 0.246674 -0.011862 1.004812 1.327195

To group by level, pass the level number or name using the level keyword:

hier\_df.groupby(level='cty', axis=1).count()

cty JP US

0 2 3

1 2 3

2 2 3

3 2 3

### Data Aggregation

Aggregations refer to any data transformation that produces scalar values from arrays. The preceding examples have used several of them, including mean, count, min, and sum. You may wonder what is going on when you invoke mean() on a GroupBy object. Many common aggregations, such as those found in [Table 10-1](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark2), have optimized implementations. However, you are not limited to only this set of methods.

Table 10-1. Optimized groupby methods

*Function Name Description*

count Number of non-NA values in the group

sum Sum of non-NA values

mean Mean of non-NA values

median Arithmetic median of non-NA values

std, var Unbiased (n – 1 denominator) standard deviation and variance

min, max Minimum and maximum of non-NA values

prod Product of non-NA values

first, last First and last non-NA values

You can use aggregations of your own devising and additionally call any method that is also defined on the grouped object. For example, you might recall that quantile computes sample quantiles of a Series or a DataFrame’s columns.

While quantile is not explicitly implemented for GroupBy, it is a Series method and thus available for use. Internally, GroupBy efficiently slices up the Series, calls

piece.quantile(0.9) for each piece, and then assembles those results together into the result object:

df

*data1 data2 key1 key2*

*0 -0.204708 1.393406 a one*

*1 0.478943 0.092908 a two*

*2 -0.519439 0.281746 b one*

*3 -0.555730 0.769023 b two*

*4 1.965781 1.246435 a one*

grouped = df.groupby('key1')

grouped['data1'].quantile(0.9)

*key1*

*a 1.668413*

*b -0.523068*

*Name: data1, dtype: float64*

To use your own aggregation functions, pass any function that aggregates an array to the aggregate or agg method:

def peak\_to\_peak(arr):

....: return arr.max() - arr.min()

grouped.agg(peak\_to\_peak)

*key1 data1 data2*

*a 2.170488 1.300498*

*b 0.036292 0.487276*

You may notice that some methods like describe also work, even though they are not aggregations, strictly speaking:

grouped.describe()

*data1 \*

*count mean std min 25% 50% 75%*

*key1*

*a 3.0 0.746672 1.109736 -0.204708 0.137118 0.478943 1.222362*

*b 2.0 -0.537585 0.025662 -0.555730 -0.546657 -0.537585 -0.528512*

*data2 \*

*max count mean std min 25% 50%*

*key1*

*a 1.965781 3.0 0.910916 0.712217 0.092908 0.669671 1.246435*

*b -0.519439 2.0 0.525384 0.344556 0.281746 0.403565 0.525384*

*75% max*

*key1*

*a 1.319920 1.393406*

*b 0.647203 0.769023*

I will explain in more detail what has happened here in Section 10.3, “Apply: General split-apply-combine,” on page 302.

Custom aggregation functions are generally much slower than the optimized functions found in Table 10-1. This is because there is some extra overhead (function calls, data rearrangement) in con‐ structing the intermediate group data chunks.

#### Column-Wise and Multiple Function Application

Let’s return to the tipping dataset from earlier examples. After loading it with

read\_csv, we add a tipping percentage column tip\_pct:

tips = pd.read\_csv('examples/tips.csv')

# Add tip percentage of total bill

tips['tip\_pct'] = tips['tip'] / tips['total\_bill']

tips[:6]

total\_bill tip smoker day time size tip\_pct

*0 16.99 1.01 No Sun Dinner 2 0.059447*

*1 10.34 1.66 No Sun Dinner 3 0.160542*

*2 21.01 3.50 No Sun Dinner 3 0.166587*

*3 23.68 3.31 No Sun Dinner 2 0.139780*

*4 24.59 3.61 No Sun Dinner 4 0.146808*

*5 25.29 4.71 No Sun Dinner 4 0.186240*

As you’ve already seen, aggregating a Series or all of the columns of a DataFrame is a matter of using aggregate with the desired function or calling a method like mean or std. However, you may want to aggregate using a different function depending on the column, or multiple functions at once. Fortunately, this is possible to do, which I’ll illustrate through a number of examples. First, I’ll group the tips by day and smoker:

grouped = tips.groupby(['day', 'smoker'])

Note that for descriptive statistics like those in Table 10-1, you can pass the name of the function as a string:

grouped\_pct = grouped['tip\_pct']

grouped\_pct.agg('mean')

*day smoker*

*Fri No 0.151650*

*Yes 0.174783*

*Sat No 0.158048*

*Yes 0.147906*

*Sun No 0.160113*

*Yes 0.187250*

*Thur No 0.160298*

*Yes 0.163863*

*Name: tip\_pct, dtype: float64*

If you pass a list of functions or function names instead, you get back a DataFrame with column names taken from the functions:

grouped\_pct.agg(['mean', 'std', peak\_to\_peak])

*day smoker mean std peak\_to\_peak*

*Fri No 0.151650 0.028123 0.067349*

*Yes 0.174783 0.051293 0.159925*

*Sat No 0.158048 0.039767 0.235193*

*Yes 0.147906 0.061375 0.290095*

*Sun No 0.160113 0.042347 0.193226*

*Yes 0.187250 0.154134 0.644685*

*Thur No 0.160298 0.038774 0.193350*

*Yes 0.163863 0.039389 0.151240*

Here we passed a list of aggregation functions to agg to evaluate indepedently on the data groups.

You don’t need to accept the names that GroupBy gives to the columns; notably, lambda functions have the name '<lambda>', which makes them hard to identify (you can see for yourself by looking at a function’s name attribute). Thus, if you pass a list of (name, function) tuples, the first element of each tuple will be used as the DataFrame column names (you can think of a list of 2-tuples as an ordered mapping):

grouped\_pct.agg([('foo', 'mean'), ('bar', np.std)])

*day smoker foo bar*

*Fri No 0.151650 0.028123*

*Yes 0.174783 0.051293*

*Sat No 0.158048 0.039767*

*Yes 0.147906 0.061375*

*Sun No 0.160113 0.042347*

*Yes 0.187250 0.154134*

*Thur No 0.160298 0.038774*

*Yes 0.163863 0.039389*

With a DataFrame you have more options, as you can specify a list of functions to apply to all of the columns or different functions per column. To start, suppose we wanted to compute the same three statistics for the tip\_pct and total\_bill columns:

functions = ['count', 'mean', 'max']

result = grouped['tip\_pct', 'total\_bill'].agg(functions) In [67]: result

*day smoker tip\_pct count mean max total\_bill count mean max*

*Fri No 4 0.151650 0.187735 4 18.420000 22.75*

*Yes 15 0.174783 0.263480 15 16.813333 40.17*

*Sat No 45 0.158048 0.291990 45 19.661778 48.33*

*Yes 42 0.147906 0.325733 42 21.276667 50.81*

*Sun No 57 0.160113 0.252672 57 20.506667 48.17*

*Yes 19 0.187250 0.710345 19 24.120000 45.35*

*Thur No 45 0.160298 0.266312 45 17.113111 41.19*

*Yes 17 0.163863 0.241255 17 19.190588 43.11*

As you can see, the resulting DataFrame has hierarchical columns, the same as you would get aggregating each column separately and using concat to glue the results together using the column names as the keys argument:

result['tip\_pct']

*day smoker count mean max*

*Fri No 4 0.151650 0.187735*

*Yes 15 0.174783 0.263480*

*Sat No 45 0.158048 0.291990*

*Yes 42 0.147906 0.325733*

*Sun No 57 0.160113 0.252672*

*Yes 19 0.187250 0.710345*

*Thur No 45 0.160298 0.266312*

*Yes 17 0.163863 0.241255*

As before, a list of tuples with custom names can be passed:

ftuples = [('Durchschnitt', 'mean'), ('Abweichung', np.var)]

grouped['tip\_pct', 'total\_bill'].agg(ftuples)

*day smoker tip\_pct Durchschnitt Abweichung total\_bill*

*Durchschnitt Abweichung*

*Fri No 0.151650 0.000791 18.420000 25.596333*

*Yes 0.174783 0.002631 16.813333 82.562438*

*Sat No 0.158048 0.001581 19.661778 79.908965*

*Yes 0.147906 0.003767 21.276667 101.387535*

*Sun No 0.160113 0.001793 20.506667 66.099980*

*Yes 0.187250 0.023757 24.120000 109.046044*

*Thur No 0.160298 0.001503 17.113111 59.625081*

*Yes 0.163863 0.001551 19.190588 69.808518*

Now, suppose you wanted to apply potentially different functions to one or more of the columns. To do this, pass a dict to agg that contains a mapping of column names to any of the function specifications listed so far:

grouped.agg({'tip' : np.max, 'size' : 'sum'})

day smoker tip size

Fri No 3.50 9

Yes 4.73 31

Sat No 9.00 115

Yes 10.00 104

Sun No 6.00 167

Yes 6.50 49

Thur No 6.70 112

Yes 5.00 40

grouped.agg({'tip\_pct' : ['min', 'max', 'mean', 'std'], 'size' : 'sum'})

*day smoker tip\_pct min max mean std size sum*

*Fri No 0.120385 0.187735 0.151650 0.028123 9*

*Yes 0.103555 0.263480 0.174783 0.051293 31*

*Sat No 0.056797 0.291990 0.158048 0.039767 115*

*Yes 0.035638 0.325733 0.147906 0.061375 104*

*Sun No 0.059447 0.252672 0.160113 0.042347 167*

*Yes 0.065660 0.710345 0.187250 0.154134 49*

*Thur No 0.072961 0.266312 0.160298 0.038774 112*

*Yes 0.090014 0.241255 0.163863 0.039389 40*

A DataFrame will have hierarchical columns only if multiple functions are applied to at least one column.

#### Returning Aggregated Data Without Row Indexes

In all of the examples up until now, the aggregated data comes back with an index, potentially hierarchical, composed from the unique group key combinations. Since this isn’t always desirable, you can disable this behavior in most cases by passing as\_index=False to groupby:

tips.groupby(['day', 'smoker'], as\_index=False).mean()

day smoker total\_bill tip size tip\_pct

*0 Fri No 18.420000 2.812500 2.250000 0.151650*

*1 Fri Yes 16.813333 2.714000 2.066667 0.174783*

*2 Sat No 19.661778 3.102889 2.555556 0.158048*

*3 Sat Yes 21.276667 2.875476 2.476190 0.147906*

*4 Sun No 20.506667 3.167895 2.929825 0.160113*

*5 Sun Yes 24.120000 3.516842 2.578947 0.187250*

*6 Thur No 17.113111 2.673778 2.488889 0.160298*

*7 Thur Yes 19.190588 3.030000 2.352941 0.163863*

Of course, it’s always possible to obtain the result in this format by calling reset\_index on the result. Using the as\_index=False method avoids some unneces‐ sary computations.

### Apply: General split-apply-combine

The most general-purpose GroupBy method is apply, which is the subject of the rest of this section. As illustrated in [Figure 10-2](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark4), apply splits the object being manipulated into pieces, invokes the passed function on each piece, and then attempts to concate‐ nate the pieces together.

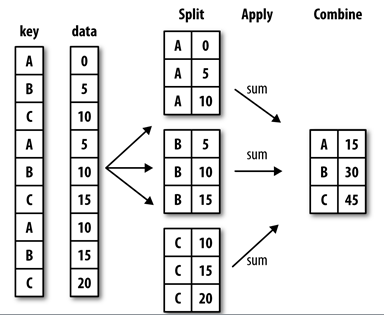


Figure 10-2. Illustration of a group aggregation

Returning to the tipping dataset from before, suppose you wanted to select the top five tip\_pct values by group. First, write a function that selects the rows with the largest values in a particular column:

def top(df, n=5, column='tip\_pct'):

....: return df.sort\_values(by=column)[-n:]

top(tips, n=6) Out[75]:

total\_bill tip smoker day time size tip\_pct

*109 14.31 4.00 Yes Sat Dinner 2 0.279525*

*183 23.17 6.50 Yes Sun Dinner 4 0.280535*

*232 11.61 3.39 No Sat Dinner 2 0.291990*

*67 3.07 1.00 Yes Sat Dinner 1 0.325733*

*178 9.60 4.00 Yes Sun Dinner 2 0.416667*

*172 7.25 5.15 Yes Sun Dinner 2 0.710345*

Now, if we group by smoker, say, and call apply with this function, we get the following:

tips.groupby('smoker').apply(top)

*smoker total\_bill tip smoker day time size tip\_pct*

*No 88 24.71 5.85 No Thur Lunch 2 0.236746*

*185 20.69 5.00 No Sun Dinner 5 0.241663*

*51 10.29 2.60 No Sun Dinner 2 0.252672*

*149 7.51 2.00 No Thur Lunch 2 0.266312*

*232 11.61 3.39 No Sat Dinner 2 0.291990*

*Yes 109 14.31 4.00 Yes Sat Dinner 2 0.279525*

*183 23.17 6.50 Yes Sun Dinner 4 0.280535*

*67 3.07 1.00 Yes Sat Dinner 1 0.325733*

*178 9.60 4.00 Yes Sun Dinner 2 0.416667*

*172 7.25 5.15 Yes Sun Dinner 2 0.710345*

What has happened here? The top function is called on each row group from the DataFrame, and then the results are glued together using pandas.concat, labeling the pieces with the group names. The result therefore has a hierarchical index whose inner level contains index values from the original DataFrame.

If you pass a function to apply that takes other arguments or keywords, you can pass these after the function:

tips.groupby(['smoker', 'day']).apply(top, n=1, column='total\_bill')

smoker day total\_bill tip smoker day time size tip\_pct

No Fri 94 22.75 3.25 No Fri Dinner 2 0.142857

Sat 212 48.33 9.00 No Sat Dinner 4 0.186220

Sun 156 48.17 5.00 No Sun Dinner 6 0.103799

Thur 142 41.19 5.00 No Thur Lunch 5 0.121389

Yes Fri 95 40.17 4.73 Yes Fri Dinner 4 0.117750

Sat 170 50.81 10.00 Yes Sat Dinner 3 0.196812

Sun 182 45.35 3.50 Yes Sun Dinner 3 0.077178

Thur 197 43.11 5.00 Yes Thur Lunch 4 0.115982

Beyond these basic usage mechanics, getting the most out of apply may require some creativity. What occurs inside the function passed is up to you; it only needs to return a pandas object or a scalar value. The rest of this chapter will mainly consist of examples showing you how to solve various problems using groupby.

You may recall that I earlier called describe on a GroupBy object:

result = tips.groupby('smoker')['tip\_pct'].describe()

result

*count mean std min 25% 50% 75% \*

*smoker*

*No 151.0 0.159328 0.039910 0.056797 0.136906 0.155625 0.185014*

*Yes 93.0 0.163196 0.085119 0.035638 0.106771 0.153846 0.195059*

*max*

*smoker*

*No 0.291990*

*Yes 0.710345*

result.unstack('smoker')

*smoker*

*count No 151.000000*

*Yes 93.000000*

*mean No 0.159328*

*Yes 0.163196*

*std No 0.039910*

*Yes 0.085119*

*min No 0.056797*

*Yes 0.035638*

*25% No 0.136906*

*Yes 0.106771*

*50% No 0.155625*

*Yes 0.153846*

*75% No 0.185014*

*Yes 0.195059*

*max No 0.291990*

*Yes 0.710345*

*dtype: float64*

Inside GroupBy, when you invoke a method like describe, it is actually just a short‐ cut for:

f = lambda x: x.describe() grouped.apply(f)

#### Suppressing the Group Keys

In the preceding examples, you see that the resulting object has a hierarchical index formed from the group keys along with the indexes of each piece of the original object. You can disable this by passing group\_keys=False to groupby:

tips.groupby('smoker', group\_keys=False).apply(top)

*total\_bill tip smoker day time size tip\_pct*

*88 24.71 5.85 No Thur Lunch 2 0.236746*

*185 20.69 5.00 No Sun Dinner 5 0.241663*

*51 10.29 2.60 No Sun Dinner 2 0.252672*

*149 7.51 2.00 No Thur Lunch 2 0.266312*

*232 11.61 3.39 No Sat Dinner 2 0.291990*

*109 14.31 4.00 Yes Sat Dinner 2 0.279525*

*183 23.17 6.50 Yes Sun Dinner 4 0.280535*

*67 3.07 1.00 Yes Sat Dinner 1 0.325733*

*178 9.60 4.00 Yes Sun Dinner 2 0.416667*

*172 7.25 5.15 Yes Sun Dinner 2 0.710345*

#### Quantile and Bucket Analysis

As you may recall from Chapter 8, pandas has some tools, in particular cut and qcut, for slicing data up into buckets with bins of your choosing or by sample quantiles. Combining these functions with groupby makes it convenient to perform bucket or quantile analysis on a dataset. Consider a simple random dataset and an equal-length bucket categorization using cut:

frame = pd.DataFrame({'data1': np.random.randn(1000),

....: 'data2': np.random.randn(1000)})

quartiles = pd.cut(frame.data1, 4)

quartiles[:10]

*0 (-1.23, 0.489]*

*1 (-2.956, -1.23]*

*2 (-1.23, 0.489]*

*3 (0.489, 2.208]*

*4 (-1.23, 0.489]*

*5 (0.489, 2.208]*

*6 (-1.23, 0.489]*

*7 (-1.23, 0.489]*

*8 (0.489, 2.208]*

*9 (0.489, 2.208]*

*Name: data1, dtype: category*

Categories (4, interval[float64]): [(-2.956, -1.23] < (-1.23, 0.489] < (0.489, 2.

208] < (2.208, 3.928]]

The Categorical object returned by cut can be passed directly to groupby. So we could compute a set of statistics for the data2 column like so:

def get\_stats(group):

....: return {'min': group.min(), 'max': group.max(),

....: 'count': group.count(), 'mean': group.mean()}

grouped = frame.data2.groupby(quartiles)

grouped.apply(get\_stats).unstack()

count max mean min

*data1*

*(-2.956, -1.23] 95.0 1.670835 -0.039521 -3.399312*

*(-1.23, 0.489] 598.0 3.260383 -0.002051 -2.989741*

*(0.489, 2.208] 297.0 2.954439 0.081822 -3.745356*

*(2.208, 3.928] 10.0 1.765640 0.024750 -1.929776*

These were equal-length buckets; to compute equal-size buckets based on sample quantiles, use qcut. I’ll pass labels=False to just get quantile numbers:

# Return quantile numbers

grouping = pd.qcut(frame.data1, 10, labels=False)

grouped = frame.data2.groupby(grouping)

grouped.apply(get\_stats).unstack()

*data1 count max mean min*

*0 100.0 1.670835 -0.049902 -3.399312*

*1 100.0 2.628441 0.030989 -1.950098*

*2 100.0 2.527939 -0.067179 -2.925113*

*3 100.0 3.260383 0.065713 -2.315555*

*4 100.0 2.074345 -0.111653 -2.047939*

*5 100.0 2.184810 0.052130 -2.989741*

*6 100.0 2.458842 -0.021489 -2.223506*

*7 100.0 2.954439 -0.026459 -3.056990*

*8 100.0 2.735527 0.103406 -3.745356*

*9 100.0 2.377020 0.220122 -2.064111*

We will take a closer look at pandas’s Categorical type in Chapter 12.

#### Example: Filling Missing Values with Group-Specific Values

When cleaning up missing data, in some cases you will replace data observations using dropna, but in others you may want to impute (fill in) the null (NA) values using a fixed value or some value derived from the data. fillna is the right tool to use; for example, here I fill in NA values with the mean:

s = pd.Series(np.random.randn(6))

s[::2] = np.nan

s

*0 NaN*

*1 -0.125921*

*2 NaN*

*3 -0.884475*

*4 NaN*

*5 0.227290*

*dtype: float64*

s.fillna(s.mean())

*0 -0.261035*

*1 -0.125921*

*2 -0.261035*

*3 -0.884475*

*4 -0.261035*

*5 0.227290*

*dtype: float64*

Suppose you need the fill value to vary by group. One way to do this is to group the data and use apply with a function that calls fillna on each data chunk. Here is some sample data on US states divided into eastern and western regions:

states = ['Ohio', 'New York', 'Vermont', 'Florida',

....: 'Oregon', 'Nevada', 'California', 'Idaho']

group\_key = ['East'] \* 4 + ['West'] \* 4

data = pd.Series(np.random.randn(8), index=states)

data

*Ohio 0.922264*

*New York -2.153545*

*Vermont -0.365757*

*Florida -0.375842*

*Oregon 0.329939*

*Nevada 0.981994*

*California 1.105913*

*Idaho -1.613716*

*dtype: float64*

Note that the syntax ['East'] \* 4 produces a list containing four copies of the ele‐ ments in ['East']. Adding lists together concatenates them.

Let’s set some values in the data to be missing:

data[['Vermont', 'Nevada', 'Idaho']] = np.nan

data

*Ohio 0.922264*

*New York -2.153545*

*Vermont NaN*

*Florida -0.375842*

*Oregon 0.329939*

*Nevada NaN*

*California 1.105913*

*Idaho NaN*

*dtype: float64*

data.groupby(group\_key).mean()

East -0.535707

West 0.717926

dtype: float64

We can fill the NA values using the group means like so:

fill\_mean = lambda g: g.fillna(g.mean())

data.groupby(group\_key).apply(fill\_mean)

*Ohio 0.922264*

*New York -2.153545*

*Vermont -0.535707*

*Florida -0.375842*

*Oregon 0.329939*

*Nevada 0.717926*

*California 1.105913*

*Idaho 0.717926*

*dtype: float64*

In another case, you might have predefined fill values in your code that vary by group. Since the groups have a name attribute set internally, we can use that:

fill\_values = {'East': 0.5, 'West': -1}

fill\_func = lambda g: g.fillna(fill\_values[g.name])

data.groupby(group\_key).apply(fill\_func)

Out[106]:

Ohio 0.922264

New York -2.153545

Vermont 0.500000

Florida -0.375842

Oregon 0.329939

Nevada -1.000000

California 1.105913

Idaho -1.000000

dtype: float64

#### Example: Random Sampling and Permutation

Suppose you wanted to draw a random sample (with or without replacement) from a large dataset for Monte Carlo simulation purposes or some other application. There are a number of ways to perform the “draws”; here we use the sample method for Series.

To demonstrate, here’s a way to construct a deck of English-style playing cards:

# Hearts, Spades, Clubs, Diamonds

suits = ['H', 'S', 'C', 'D']

card\_val = (list(range(1, 11)) + [10] \* 3) \* 4

base\_names = ['A'] + list(range(2, 11)) + ['J', 'K', 'Q'] cards = []

for suit in ['H', 'S', 'C', 'D']:

cards.extend(str(num) + suit for num in base\_names)

deck = pd.Series(card\_val, index=cards)

So now we have a Series of length 52 whose index contains card names and values are the ones used in Blackjack and other games (to keep things simple, I just let the ace 'A' be 1):

deck[:13]

AH 1

2H 2

3H 3

4H 4

5H 5

6H 6

7H 7

8H 8

9H 9

10H 10

JH 10

KH 10

QH 10

dtype: int64

Now, based on what I said before, drawing a hand of five cards from the deck could be written as:

def draw(deck, n=5):

.....: return deck.sample(n)

draw(deck)

AD 1

8C 8

5H 5

KC 10

2C 2

dtype: int64

Suppose you wanted two random cards from each suit. Because the suit is the last character of each card name, we can group based on this and use apply:

get\_suit = lambda card: card[-1] # last letter is suit

deck.groupby(get\_suit).apply(draw, n=2)

C 2C 2

3C 3

D KD 10

8D 8

H KH 10

3H 3

S 2S 2

4S 4

dtype: int64

Alternatively, we could write:

deck.groupby(get\_suit, group\_keys=False).apply(draw, n=2)

*KC 10*

*JC 10*

*AD 1*

*5D 5*

*5H 5*

*6H 6*

*7S 7*

*KS 10*

*dtype: int64*

#### Example: Group Weighted Average and Correlation

Under the split-apply-combine paradigm of groupby, operations between columns in a DataFrame or two Series, such as a group weighted average, are possible. As an example, take this dataset containing group keys, values, and some weights:

df = pd.DataFrame({'category': ['a', 'a', 'a', 'a', 'b', 'b', 'b', 'b'],

.....: 'data': np.random.randn(8),

.....: 'weights': np.random.rand(8)})

df

*category data weights*

*0 a 1.561587 0.957515*

*1 a 1.219984 0.347267*

*2 a -0.482239 0.581362*

*3 a 0.315667 0.217091*

*4 b -0.047852 0.894406*

*5 b -0.454145 0.918564*

*6 b -0.556774 0.277825*

*7 b 0.253321 0.955905*

The group weighted average by category would then be:

grouped = df.groupby('category')

get\_wavg = lambda g: np.average(g['data'], weights=g['weights'])

grouped.apply(get\_wavg)

*category*

*a 0.811643*

*b -0.122262*

*dtype: float64*

As another example, consider a financial dataset originally obtained from Yahoo! Finance containing end-of-day prices for a few stocks and the S&P 500 index (the SPX symbol):

close\_px = pd.read\_csv('examples/stock\_px\_2.csv', parse\_dates=True, index\_col=0)

close\_px.info()

*<class 'pandas.core.frame.DataFrame'>*

*DatetimeIndex: 2214 entries, 2003-01-02 to 2011-10-14 Data columns (total 4 columns):*

*AAPL 2214 non-null float64 MSFT 2214 non-null float64 XOM 2214 non-null float64 SPX 2214 non-null float64 dtypes: float64(4)*

*memory usage: 86.5 KB*

close\_px[-4:]

AAPL MSFT XOM SPX

2011-10-11 400.29 27.00 76.27 1195.54

2011-10-12 402.19 26.96 77.16 1207.25

2011-10-13 408.43 27.18 76.37 1203.66

2011-10-14 422.00 27.27 78.11 1224.58

One task of interest might be to compute a DataFrame consisting of the yearly corre‐ lations of daily returns (computed from percent changes) with SPX. As one way to do this, we first create a function that computes the pairwise correlation of each column with the 'SPX' column:

spx\_corr = lambda x: x.corrwith(x['SPX'])

Next, we compute percent change on close\_px using pct\_change:

rets = close\_px.pct\_change().dropna()

Lastly, we group these percent changes by year, which can be extracted from each row label with a one-line function that returns the year attribute of each datetime label:

get\_year = lambda x: x.year

by\_year = rets.groupby(get\_year)

by\_year.apply(spx\_corr) Out[126]:

*AAPL MSFT XOM SPX*

*2003 0.541124 0.745174 0.661265 1.0*

*2004 0.374283 0.588531 0.557742 1.0*

*2005 0.467540 0.562374 0.631010 1.0*

*2006 0.428267 0.406126 0.518514 1.0*

*2007 0.508118 0.658770 0.786264 1.0*

*2008 0.681434 0.804626 0.828303 1.0*

*2009 0.707103 0.654902 0.797921 1.0*

*2010 0.710105 0.730118 0.839057 1.0*

*2011 0.691931 0.800996 0.859975 1.0*

You could also compute inter-column correlations. Here we compute the annual cor‐ relation between Apple and Microsoft:

by\_year.apply(lambda g: g['AAPL'].corr(g['MSFT']))

*2003 0.480868*

*2004 0.259024*

*2005 0.300093*

*2006 0.161735*

*2007 0.417738*

*2008 0.611901*

*2009 0.432738*

*2010 0.571946*

*2011 0.581987*

*dtype: float64*

#### Example: Group-Wise Linear Regression

In the same theme as the previous example, you can use groupby to perform more complex group-wise statistical analysis, as long as the function returns a pandas object or scalar value. For example, I can define the following regress function (using the statsmodels econometrics library), which executes an ordinary least squares (OLS) regression on each chunk of data:

import statsmodels.api as sm def regress(data, yvar, xvars):

Y = data[yvar] X = data[xvars]

X['intercept'] = 1.

result = sm.OLS(Y, X).fit()

return result.params

Now, to run a yearly linear regression of AAPL on SPX returns, execute:

by\_year.apply(regress, 'AAPL', ['SPX'])

*SPX intercept*

*2003 1.195406 0.000710*

*2004 1.363463 0.004201*

*2005 1.766415 0.003246*

*2006 1.645496 0.000080*

*2007 1.198761 0.003438*

*2008 0.968016 -0.001110*

*2009 0.879103 0.002954*

*2010 1.052608 0.001261*

*2011 0.806605 0.001514*

### Pivot Tables and Cross-Tabulation

A pivot table is a data summarization tool frequently found in spreadsheet programs and other data analysis software. It aggregates a table of data by one or more keys, arranging the data in a rectangle with some of the group keys along the rows and some along the columns. Pivot tables in Python with pandas are made possible through the groupby facility described in this chapter combined with reshape opera‐ tions utilizing hierarchical indexing. DataFrame has a pivot\_table method, and there is also a top-level pandas.pivot\_table function. In addition to providing a convenience interface to groupby, pivot\_table can add partial totals, also known as margins.

Returning to the tipping dataset, suppose you wanted to compute a table of group means (the default pivot\_table aggregation type) arranged by day and smoker on the rows:

tips.pivot\_table(index=['day', 'smoker'])

day smoker size tip tip\_pct total\_bill

Fri No 2.250000 2.812500 0.151650 18.420000

Yes 2.066667 2.714000 0.174783 16.813333

Sat No 2.555556 3.102889 0.158048 19.661778

Yes 2.476190 2.875476 0.147906 21.276667

Sun No 2.929825 3.167895 0.160113 20.506667

Yes 2.578947 3.516842 0.187250 24.120000

Thur No 2.488889 2.673778 0.160298 17.113111

Yes 2.352941 3.030000 0.163863 19.190588

This could have been produced with groupby directly. Now, suppose we want to aggregate only tip\_pct and size, and additionally group by time. I’ll put smoker in the table columns and day in the rows:

tips.pivot\_table(['tip\_pct', 'size'], index=['time', 'day'],

.....: columns='smoker')

*size tip\_pct*

*smoker No Yes No Yes time day*

*Dinner Fri 2.000000 2.222222 0.139622 0.165347*

*Sat 2.555556 2.476190 0.158048 0.147906*

*Sun 2.929825 2.578947 0.160113 0.187250*

*Thur 2.000000 NaN 0.159744 NaN*

*Lunch Fri 3.000000 1.833333 0.187735 0.188937*

*Thur 2.500000 2.352941 0.160311 0.163863*

We could augment this table to include partial totals by passing margins=True. This has the effect of adding All row and column labels, with corresponding values being the group statistics for all the data within a single tier:

tips.pivot\_table(['tip\_pct', 'size'], index=['time', 'day'],

.....: columns='smoker', margins=True)

*size tip\_pct*

*smoker No Yes All No Yes All time day*

*Dinner Fri 2.000000 2.222222 2.166667 0.139622 0.165347 0.158916*

*Sat 2.555556 2.476190 2.517241 0.158048 0.147906 0.153152*

*Sun 2.929825 2.578947 2.842105 0.160113 0.187250 0.166897*

*Thur 2.000000 NaN 2.000000 0.159744 NaN 0.159744*

*Lunch Fri 3.000000 1.833333 2.000000 0.187735 0.188937 0.188765*

*Thur 2.500000 2.352941 2.459016 0.160311 0.163863 0.161301*

*All 2.668874 2.408602 2.569672 0.159328 0.163196 0.160803*

Here, the All values are means without taking into account smoker versus non- smoker (the All columns) or any of the two levels of grouping on the rows (the All row).

To use a different aggregation function, pass it to aggfunc. For example, 'count' or

len will give you a cross-tabulation (count or frequency) of group sizes:

tips.pivot\_table('tip\_pct', index=['time', 'smoker'], columns='day',

.....:

*day*

*time smoker*

*Fri*

*Sat aggfunc=len, margins=True)*

*Sun Thur All*

*Dinner No 3.0 45.0 57.0 1.0 106.0*

*Yes 9.0 42.0 19.0 NaN 70.0*

*Lunch No 1.0 NaN NaN 44.0 45.0*

*Yes 6.0 NaN NaN 17.0 23.0*

*All 19.0 87.0 76.0 62.0 244.0*

If some combinations are empty (or otherwise NA), you may wish to pass a fill\_value:

tips.pivot\_table('tip\_pct', index=['time', 'size', 'smoker'],

.....: columns='day', aggfunc='mean', fill\_value=0) Out[134]:

day

time size

smoker Fri Sat Sun Thur

Dinner 1 No 0.000000 0.137931 0.000000 0.000000

Yes 0.000000 0.325733 0.000000 0.000000

2 No 0.139622 0.162705 0.168859 0.159744

Yes 0.171297 0.148668 0.207893 0.000000

3 No 0.000000 0.154661 0.152663 0.000000

Yes 0.000000 0.144995 0.152660 0.000000

4 No 0.000000 0.150096 0.148143 0.000000

Yes 0.117750 0.124515 0.193370 0.000000

5 No 0.000000 0.000000 0.206928 0.000000

Yes 0.000000 0.106572 0.065660 0.000000

... ... ... ... ...

Lunch 1 No 0.000000 0.000000 0.000000 0.181728

Yes 0.223776 0.000000 0.000000 0.000000

2 No 0.000000 0.000000 0.000000 0.166005

Yes 0.181969 0.000000 0.000000 0.158843

3 No 0.187735 0.000000 0.000000 0.084246

Yes 0.000000 0.000000 0.000000 0.204952

4 No 0.000000 0.000000 0.000000 0.138919

Yes 0.000000 0.000000 0.000000 0.155410

5 No 0.000000 0.000000 0.000000 0.121389

6 No 0.000000 0.000000 0.000000 0.173706

[21 rows x 4 columns]

See [Table 10-2](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark5) for a summary of pivot\_table methods.

Table 10-2. pivot\_table options

**Function Name Description**

values Column name or names to aggregate; by default aggregates all numeric columns index Column names or other group keys to group on the rows of the resulting pivot table columns Column names or other group keys to group on the columns of the resulting pivot table

aggfunc Aggregation function or list of functions ('mean' by default); can be any function valid in a groupby context

fill\_value Replace missing values in result table

dropna If True, do not include columns whose entries are all NA margins

Add row/column subtotals and grand total (False by default)

#### Cross-Tabulations: Crosstab

A cross-tabulation (or crosstab for short) is a special case of a pivot table that com‐ putes group frequencies. Here is an example:

data

*Sample Nationality Handedness*

*0 1 USA Right-handed*

*1 2 Japan Left-handed*

*2 3 USA Right-handed*

*3 4 Japan Right-handed*

*4 5 Japan Left-handed*

*5 6 Japan Right-handed*

*6 7 USA Right-handed*

*7 8 USA Left-handed*

*8 9 Japan Right-handed*

*9 10 USA Right-handed*

As part of some survey analysis, we might want to summarize this data by nationality and handedness. You could use pivot\_table to do this, but the pandas.crosstab function can be more convenient:

pd.crosstab(data.Nationality, data.Handedness, margins=True)

*Handedness*

*Nationality Left-handed Right-handed All*

*Japan 2 3 5*

*USA 1 4 5*

*All 3 7 10*

The first two arguments to crosstab can each either be an array or Series or a list of arrays. As in the tips data:

pd.crosstab([tips.time, tips.day], tips.smoker, margins=True)

*smoker*

*time*

*day No Yes All*

*Dinner Fri 3 9 12*

*Sat 45 42 87*

*Sun 57 19 76*

*Thur 1 0 1*

*Lunch Fri 1 6 7*

*Thur 44 17 61*

*All 151 93 244*

### Conclusion

Mastering pandas’s data grouping tools can help both with data cleaning as well as modeling or statistical analysis work. In Chapter 14 we will look at several more example use cases for groupby on real data.

In the next chapter, we turn our attention to time series data.

### Time Series

Time series data is an important form of structured data in many different fields, such as finance, economics, ecology, neuroscience, and physics. Anything that is observed or measured at many points in time forms a time series. Many time series are fixed frequency, which is to say that data points occur at regular intervals according to some rule, such as every 15 seconds, every 5 minutes, or once per month. Time series can also be irregular without a fixed unit of time or offset between units. How you mark and refer to time series data depends on the application, and you may have one of the following:

Timestamps, specific instants in time

Fixed periods, such as the month January 2007 or the full year 2010

Intervals of time, indicated by a start and end timestamp. Periods can be thought of as special cases of intervals

Experiment or elapsed time; each timestamp is a measure of time relative to a particular start time (e.g., the diameter of a cookie baking each second since being placed in the oven)

In this chapter, I am mainly concerned with time series in the first three categories, though many of the techniques can be applied to experimental time series where the index may be an integer or floating-point number indicating elapsed time from the start of the experiment. The simplest and most widely used kind of time series are those indexed by timestamp.

pandas also supports indexes based on timedeltas, which can be a useful way of representing experiment or elapsed time. We do not explore timedelta indexes in this book, but you can learn more in the [pandas documentation](http://pandas.pydata.org/).

pandas provides many built-in time series tools and data algorithms. You can effi‐ ciently work with very large time series and easily slice and dice, aggregate, and resample irregular- and fixed-frequency time series. Some of these tools are especially useful for financial and economics applications, but you could certainly use them to analyze server log data, too.

### Date and Time Data Types and Tools

The Python standard library includes data types for date and time data, as well as calendar-related functionality. The datetime, time, and calendar modules are the main places to start. The datetime.datetime type, or simply datetime, is widely used:

from datetime import datetime In [11]: now = datetime.now()

now

datetime.datetime(2017, 9, 25, 14, 5, 52, 72973)

now.year, now.month, now.day Out[13]: (2017, 9, 25)

datetime stores both the date and time down to the microsecond. timedelta repre‐ sents the temporal difference between two datetime objects:

delta = datetime(2011, 1, 7) - datetime(2008, 6, 24, 8, 15)

delta

datetime.timedelta(926, 56700)

delta.days

*926*

delta.seconds

*56700*

You can add (or subtract) a timedelta or multiple thereof to a datetime object to yield a new shifted object:

from datetime import timedelta

start = datetime(2011, 1, 7)

start + timedelta(12)

datetime.datetime(2011, 1, 19, 0, 0)

start - 2 \* timedelta(12)

datetime.datetime(2010, 12, 14, 0, 0)

[Table 11-1](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark7) summarizes the data types in the datetime module. While this chapter is mainly concerned with the data types in pandas and higher-level time series manipu‐ lation, you may encounter the datetime-based types in many other places in Python in the wild.

Table 11-1. Types in datetime module

**Function Description**

date Store calendar date (year, month, day) using the Gregorian calendar time

Store time of day as hours, minutes, seconds, and microseconds datetime

Stores both date and time

timedelta Represents the difference between two datetime values (as days, seconds, and microseconds)

tzinfo Base type for storing time zone information

Converting Between String and Datetime

You can format datetime objects and pandas Timestamp objects, which I’ll introduce later, as strings using str or the strftime method, passing a format specification:

stamp = datetime(2011, 1, 3)

str(stamp)

'2011-01-03 00:00:00'

stamp.strftime('%Y-%m-%d')

'2011-01-03'

See [Table 11-2](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark8) for a complete list of the format codes (reproduced from Chapter 2).

Table 11-2. Datetime format specification (ISO C89 compatible)

**Type Description**

%Y Four-digit year

%y Two-digit year

%m Two-digit month [01, 12]

%d Two-digit day [01, 31]

%H Hour (24-hour clock) [00, 23]

%I Hour (12-hour clock) [01, 12]

%M Two-digit minute [00, 59]

%S Second [00, 61] (seconds 60, 61 account for leap seconds)

%w Weekday as integer [0 (Sunday), 6]

%U Week number of the year [00, 53]; Sunday is considered the first day of the week, and days before the first Sunday of the year are “week 0”

%W Week number of the year [00, 53]; Monday is considered the first day of the week, and days before the first Monday of the year are “week 0”

%z UTC time zone offset as +HHMM or -HHMM; empty if time zone naive

%F Shortcut for %Y-%m-%d (e.g., 2012-4-18)

%D Shortcut for %m/%d/%y (e.g., 04/18/12)

You can use these same format codes to convert strings to dates using date time.strptime:

In [25]: value = '2011-01-03'

In [26]: datetime.strptime(value, '%Y-%m-%d') Out[26]: datetime.datetime(2011, 1, 3, 0, 0)

In [27]: datestrs = ['7/6/2011', '8/6/2011']

In [28]: [datetime.strptime(x, '%m/%d/%Y') for x in datestrs] Out[28]:

[datetime.datetime(2011, 7, 6, 0, 0),

datetime.datetime(2011, 8, 6, 0, 0)]

datetime.strptime is a good way to parse a date with a known format. However, it can be a bit annoying to have to write a format spec each time, especially for common date formats. In this case, you can use the parser.parse method in the third-party dateutil package (this is installed automatically when you install pandas):

In [29]: from dateutil.parser import parse

In [30]: parse('2011-01-03')

Out[30]: datetime.datetime(2011, 1, 3, 0, 0)

dateutil is capable of parsing most human-intelligible date representations:

In [31]: parse('Jan 31, 1997 10:45 PM')

Out[31]: datetime.datetime(1997, 1, 31, 22, 45)

In international locales, day appearing before month is very common, so you can pass

dayfirst=True to indicate this:

In [32]: parse('6/12/2011', dayfirst=True) Out[32]: datetime.datetime(2011, 12, 6, 0, 0)

pandas is generally oriented toward working with arrays of dates, whether used as an axis index or a column in a DataFrame. The to\_datetime method parses many dif‐ ferent kinds of date representations. Standard date formats like ISO 8601 can be parsed very quickly:

In [33]: datestrs = ['2011-07-06 12:00:00', '2011-08-06 00:00:00']

In [34]: pd.to\_datetime(datestrs)

Out[34]: DatetimeIndex(['2011-07-06 12:00:00', '2011-08-06 00:00:00'], dtype='dat

etime64[ns]', freq=None)

It also handles values that should be considered missing (None, empty string, etc.):

In [35]: idx = pd.to\_datetime(datestrs + [None])

In [36]: idx

Out[36]: DatetimeIndex(['2011-07-06 12:00:00', '2011-08-06 00:00:00', 'NaT'], dty

pe='datetime64[ns]', freq=None)

In [37]: idx[2] Out[37]: NaT

In [38]: pd.isnull(idx)

Out[38]: array([False, False, True], dtype=bool)

NaT (Not a Time) is pandas’s null value for timestamp data.

dateutil.parser is a useful but imperfect tool. Notably, it will rec‐ ognize some strings as dates that you might prefer that it didn’t— for example, '42' will be parsed as the year 2042 with today’s cal‐ endar date.

datetime objects also have a number of locale-specific formatting options for systems in other countries or languages. For example, the abbreviated month names will be different on German or French systems compared with English systems. See [Table 11-3](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark9) for a listing.

Table 11-3. Locale-specific date formatting

**Type Description**

%a Abbreviated weekday name

%A Full weekday name

%b Abbreviated month name

%B Full month name

%c Full date and time (e.g., ‘Tue 01 May 2012 04:20:57 PM’)

%p Locale equivalent of AM or PM

%x Locale-appropriate formatted date (e.g., in the United States, May 1, 2012 yields ’05/01/2012’)

%X Locale-appropriate time (e.g., ’04:24:12 PM’)

## Time Series Basics

A basic kind of time series object in pandas is a Series indexed by timestamps, which is often represented external to pandas as Python strings or datetime objects:

In [39]: **from datetime import** datetime

In [40]: dates = [datetime(2011, 1, 2), datetime(2011, 1, 5),

....: datetime(2011, 1, 7), datetime(2011, 1, 8),

....: datetime(2011, 1, 10), datetime(2011, 1, 12)] In [41]: ts = pd.Series(np.random.randn(6), index=dates)

|  |  |
| --- | --- |
| In [42]: ts |  |
| Out[42]: |
| 2011-01-02 | -0.204708 |
| 2011-01-05 | 0.478943 |
| 2011-01-07 | -0.519439 |
| 2011-01-08 | -0.555730 |
| 2011-01-10 | 1.965781 |
| 2011-01-12 | 1.393406 |

dtype: float64

Under the hood, these datetime objects have been put in a DatetimeIndex:

In [43]: ts.index Out[43]:

DatetimeIndex(['2011-01-02', '2011-01-05', '2011-01-07', '2011-01-08',

'2011-01-10', '2011-01-12'],

dtype='datetime64[ns]', freq=None)

Like other Series, arithmetic operations between differently indexed time series auto‐ matically align on the dates:

In [44]: ts + ts[::2] Out[44]:

|  |  |
| --- | --- |
| 2011-01-02 | -0.409415 |
| 2011-01-05 | NaN |
| 2011-01-07 | -1.038877 |
| 2011-01-08 | NaN |
| 2011-01-10 | 3.931561 |
| 2011-01-12 | NaN |

dtype: float64

Recall that ts[::2] selects every second element in ts.

pandas stores timestamps using NumPy’s datetime64 data type at the nanosecond resolution:

In [45]: ts.index.dtype Out[45]: dtype('<M8[ns]')

Scalar values from a DatetimeIndex are pandas Timestamp objects:

In [46]: stamp = ts.index[0]

In [47]: stamp

Out[47]: Timestamp('2011-01-02 00:00:00')

A Timestamp can be substituted anywhere you would use a datetime object. Addi‐ tionally, it can store frequency information (if any) and understands how to do time zone conversions and other kinds of manipulations. More on both of these things later.

#### Indexing, Selection, Subsetting

Time series behaves like any other pandas.Series when you are indexing and select‐ ing data based on label:

In [48]: stamp = ts.index[2]

In [49]: ts[stamp]

Out[49]: -0.51943871505673811

As a convenience, you can also pass a string that is interpretable as a date:

In [50]: ts['1/10/2011']

Out[50]: 1.9657805725027142

In [51]: ts['20110110']

Out[51]: 1.9657805725027142

For longer time series, a year or only a year and month can be passed to easily select slices of data:

In [52]: longer\_ts = pd.Series(np.random.randn(1000),

....: index=pd.date\_range('1/1/2000', periods=1000))

In [53]: longer\_ts Out[53]:

|  |  |
| --- | --- |
| 2000-01-01 | 0.092908 |
| 2000-01-02 | 0.281746 |
| 2000-01-03 | 0.769023 |
| 2000-01-04 | 1.246435 |
| 2000-01-05 | 1.007189 |
| 2000-01-06 | -1.296221 |
| 2000-01-07 | 0.274992 |
| 2000-01-08 | 0.228913 |
| 2000-01-09 | 1.352917 |
| 2000-01-10 | 0.886429 |
|  | ... |
| 2002-09-17 | -0.139298 |
| 2002-09-18 | -1.159926 |
| 2002-09-19 | 0.618965 |
| 2002-09-20 | 1.373890 |
| 2002-09-21 | -0.983505 |
| 2002-09-22 | 0.930944 |
| 2002-09-23 | -0.811676 |
| 2002-09-24 | -1.830156 |
| 2002-09-25 | -0.138730 |
| 2002-09-26 | 0.334088 |

Freq: D, Length: 1000, dtype: float64

In [54]: longer\_ts['2001'] Out[54]:

|  |  |
| --- | --- |
| 2001-01-01 | 1.599534 |
| 2001-01-02 | 0.474071 |
| 2001-01-03 | 0.151326 |
| 2001-01-04 | -0.542173 |
| 2001-01-05 | -0.475496 |
| 2001-01-06 | 0.106403 |
| 2001-01-07 | -1.308228 |
| 2001-01-08 | 2.173185 |
| 2001-01-09 | 0.564561 |
| 2001-01-10 | -0.190481 |
|  | ... |
| 2001-12-22 | 0.000369 |
| 2001-12-23 | 0.900885 |
| 2001-12-24 | -0.454869 |
| 2001-12-25 | -0.864547 |
| 2001-12-26 | 1.129120 |
| 2001-12-27 | 0.057874 |
| 2001-12-28 | -0.433739 |
| 2001-12-29 | 0.092698 |
| 2001-12-30 | -1.397820 |
| 2001-12-31 | 1.457823 |

Freq: D, Length: 365, dtype: float64

Here, the string '2001' is interpreted as a year and selects that time period. This also works if you specify the month:

In [55]: longer\_ts['2001-05'] Out[55]:

|  |  |
| --- | --- |
| 2001-05-01 | -0.622547 |
| 2001-05-02 | 0.936289 |
| 2001-05-03 | 0.750018 |
| 2001-05-04 | -0.056715 |
| 2001-05-05 | 2.300675 |
| 2001-05-06 | 0.569497 |
| 2001-05-07 | 1.489410 |
| 2001-05-08 | 1.264250 |
| 2001-05-09 | -0.761837 |
| 2001-05-10 | -0.331617 |
|  | ... |
| 2001-05-22 | 0.503699 |
| 2001-05-23 | -1.387874 |
| 2001-05-24 | 0.204851 |
| 2001-05-25 | 0.603705 |
| 2001-05-26 | 0.545680 |
| 2001-05-27 | 0.235477 |
| 2001-05-28 | 0.111835 |
| 2001-05-29 | -1.251504 |
| 2001-05-30 | -2.949343 |
| 2001-05-31 | 0.634634 |

Freq: D, Length: 31, dtype: float64

Slicing with datetime objects works as well:

In [56]: ts[datetime(2011, 1, 7):] Out[56]:

|  |  |
| --- | --- |
| 2011-01-07 | -0.519439 |
| 2011-01-08 | -0.555730 |
| 2011-01-10 | 1.965781 |
| 2011-01-12 | 1.393406 |

dtype: float64

Because most time series data is ordered chronologically, you can slice with time‐ stamps not contained in a time series to perform a range query:

In [57]: ts

Out[57]:

|  |  |
| --- | --- |
| 2011-01-02 | -0.204708 |
| 2011-01-05 | 0.478943 |
| 2011-01-07 | -0.519439 |
| 2011-01-08 | -0.555730 |
| 2011-01-10 | 1.965781 |
| 2011-01-12 | 1.393406 |

dtype: float64

In [58]: ts['1/6/2011':'1/11/2011'] Out[58]:

|  |  |
| --- | --- |
| 2011-01-07 | -0.519439 |
| 2011-01-08 | -0.555730 |
| 2011-01-10 | 1.965781 |

dtype: float64

As before, you can pass either a string date, datetime, or timestamp. Remember that slicing in this manner produces views on the source time series like slicing NumPy arrays. This means that no data is copied and modifications on the slice will be reflec‐ ted in the original data.

There is an equivalent instance method, truncate, that slices a Series between two dates:

In [59]: ts.truncate(after='1/9/2011') Out[59]:

|  |  |
| --- | --- |
| 2011-01-02 | -0.204708 |
| 2011-01-05 | 0.478943 |
| 2011-01-07 | -0.519439 |
| 2011-01-08 | -0.555730 |

dtype: float64

All of this holds true for DataFrame as well, indexing on its rows:

In [60]: dates = pd.date\_range('1/1/2000', periods=100, freq='W-WED')

In [61]: long\_df = pd.DataFrame(np.random.randn(100, 4),

....: index=dates,

....: columns=['Colorado', 'Texas',

....: 'New York', 'Ohio'])

In [62]: long\_df.loc['5-2001'] Out[62]:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Colorado | Texas | New York Ohio |
| 2001-05-02 | -0.006045 | 0.490094 | -0.277186 -0.707213 |
| 2001-05-09 | -0.560107 | 2.735527 | 0.927335 1.513906 |
| 2001-05-16 | 0.538600 | 1.273768 | 0.667876 -0.969206 |
| 2001-05-23 | 1.676091 | -0.817649 | 0.050188 1.951312 |
| 2001-05-30 | 3.260383 | 0.963301 | 1.201206 -1.852001 |

#### Time Series with Duplicate Indices

In some applications, there may be multiple data observations falling on a particular timestamp. Here is an example:

In [63]: dates = pd.DatetimeIndex(['1/1/2000', '1/2/2000', '1/2/2000',

....: '1/2/2000', '1/3/2000'])

In [64]: dup\_ts = pd.Series(np.arange(5), index=dates) In [65]: dup\_ts

Out[65]:

2000-01-01 0

2000-01-02 1

2000-01-02 2

2000-01-02 3

2000-01-03 4

dtype: int64

We can tell that the index is not unique by checking its is\_unique property:

In [66]: dup\_ts.index.is\_unique Out[66]: False

Indexing into this time series will now either produce scalar values or slices depend‐ ing on whether a timestamp is duplicated:

In [67]: dup\_ts['1/3/2000'] *# not duplicated*

Out[67]: 4

In [68]: dup\_ts['1/2/2000'] *# duplicated*

Out[68]:

2000-01-02 1

2000-01-02 2

2000-01-02 3

dtype: int64

Suppose you wanted to aggregate the data having non-unique timestamps. One way to do this is to use groupby and pass level=0:

In [69]: grouped = dup\_ts.groupby(level=0)

In [70]: grouped.mean() Out[70]:

2000-01-01 0

2000-01-02 2

2000-01-03 4

dtype: int64

In [71]: grouped.count() Out[71]:

|  |  |
| --- | --- |
| 2000-01-01 | 1 |
| 2000-01-02 | 3 |
| 2000-01-03 | 1 |
| dtype: int64 |  |

### Date Ranges, Frequencies, and Shifting

Generic time series in pandas are assumed to be irregular; that is, they have no fixed frequency. For many applications this is sufficient. However, it’s often desirable to work relative to a fixed frequency, such as daily, monthly, or every 15 minutes, even if that means introducing missing values into a time series. Fortunately pandas has a full suite of standard time series frequencies and tools for resampling, inferring fre‐ quencies, and generating fixed-frequency date ranges. For example, you can convert the sample time series to be fixed daily frequency by calling resample:

In [72]: ts

Out[72]:

|  |  |
| --- | --- |
| 2011-01-02 | -0.204708 |
| 2011-01-05 | 0.478943 |
| 2011-01-07 | -0.519439 |
| 2011-01-08 | -0.555730 |
| 2011-01-10 | 1.965781 |
| 2011-01-12 | 1.393406 |

dtype: float64

In [73]: resampler = ts.resample('D')

The string 'D' is interpreted as daily frequency.

Conversion between frequencies or resampling is a big enough topic to have its own section later ([Section 11.6, “Resampling and Frequency Conversion,” on page 348](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark13)). Here I’ll show you how to use the base frequencies and multiples thereof.

Generating Date Ranges

While I used it previously without explanation, pandas.date\_range is responsible for generating a DatetimeIndex with an indicated length according to a particular frequency:

In [74]: index = pd.date\_range('2012-04-01', '2012-06-01')

In [75]: index Out[75]:

DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-04-04',

'2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08',

'2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12',

'2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16',

'2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20',

'2012-04-21', '2012-04-22', '2012-04-23', '2012-04-24',

'2012-04-25', '2012-04-26', '2012-04-27', '2012-04-28',

'2012-04-29', '2012-04-30', '2012-05-01', '2012-05-02',

'2012-05-03', '2012-05-04', '2012-05-05', '2012-05-06',

'2012-05-07', '2012-05-08', '2012-05-09', '2012-05-10',

'2012-05-11', '2012-05-12', '2012-05-13', '2012-05-14',

'2012-05-15', '2012-05-16', '2012-05-17', '2012-05-18',

'2012-05-19', '2012-05-20', '2012-05-21', '2012-05-22',

'2012-05-23', '2012-05-24', '2012-05-25', '2012-05-26',

'2012-05-27', '2012-05-28', '2012-05-29', '2012-05-30',

'2012-05-31', '2012-06-01'],

dtype='datetime64[ns]', freq='D')

By default, date\_range generates daily timestamps. If you pass only a start or end date, you must pass a number of periods to generate:

In [76]: pd.date\_range(start='2012-04-01', periods=20) Out[76]:

DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-04-04',

'2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08',

'2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12',

'2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16',

'2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20'],

dtype='datetime64[ns]', freq='D')

In [77]: pd.date\_range(end='2012-06-01', periods=20) Out[77]:

DatetimeIndex(['2012-05-13', '2012-05-14', '2012-05-15', '2012-05-16',

'2012-05-17', '2012-05-18', '2012-05-19', '2012-05-20',

'2012-05-21', '2012-05-22', '2012-05-23', '2012-05-24',

'2012-05-25', '2012-05-26', '2012-05-27', '2012-05-28',

'2012-05-29', '2012-05-30', '2012-05-31', '2012-06-01'],

dtype='datetime64[ns]', freq='D')

The start and end dates define strict boundaries for the generated date index. For example, if you wanted a date index containing the last business day of each month, you would pass the 'BM' frequency (business end of month; see more complete listing

of frequencies in [Table 11-4](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark10)) and only dates falling on or inside the date interval will be included:

In [78]: pd.date\_range('2000-01-01', '2000-12-01', freq='BM') Out[78]:

DatetimeIndex(['2000-01-31', '2000-02-29', '2000-03-31', '2000-04-28',

'2000-05-31', '2000-06-30', '2000-07-31', '2000-08-31',

'2000-09-29', '2000-10-31', '2000-11-30'],

dtype='datetime64[ns]', freq='BM')

Table 11-4. Base time series frequencies (not comprehensive)

**Alias Offset type Description**

D Day Calendar daily

B BusinessDay Business daily

H Hour Hourly

T or min Minute Minutely

S Second Secondly

L or ms Milli Millisecond (1/1,000 of 1 second)

U Micro Microsecond (1/1,000,000 of 1 second)

M MonthEnd Last calendar day of month

BM BusinessMonthEnd Last business day (weekday) of month

MS MonthBegin First calendar day of month

BMS BusinessMonthBegin First weekday of month

W-MON, W-TUE, ... Week Weekly on given day of week (MON, TUE, WED, THU, FRI, SAT, or SUN)

WOM-1MON, WOM-2MON, ... WeekOfMonth Generate weekly dates in the first, second, third, or

fourth week of the month (e.g., WOM-3FRI for the third Friday of each month)

Q-JAN, Q-FEB, ... QuarterEnd Quarterly dates anchored on last calendar day of each

month, for year ending in indicated month (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, or DEC)

BQ-JAN, BQ-FEB, ... BusinessQuarterEnd Quarterly dates anchored on last weekday day of each

month, for year ending in indicated month

QS-JAN, QS-FEB, ... QuarterBegin Quarterly dates anchored on first calendar day of each

month, for year ending in indicated month

BQS-JAN, BQS-FEB, ... BusinessQuarterBegin Quarterly dates anchored on first weekday day of each

month, for year ending in indicated month

A-JAN, A-FEB, ... YearEnd Annual dates anchored on last calendar day of given

month (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, or DEC)

BA-JAN, BA-FEB, ... BusinessYearEnd Annual dates anchored on last weekday of given

month

AS-JAN, AS-FEB, ... YearBegin Annual dates anchored on first day of given month

BAS-JAN, BAS-FEB, ... BusinessYearBegin Annual dates anchored on first weekday of given

month

date\_range by default preserves the time (if any) of the start or end timestamp:

In [79]: pd.date\_range('2012-05-02 12:56:31', periods=5) Out[79]:

DatetimeIndex(['2012-05-02 12:56:31', '2012-05-03 12:56:31',

'2012-05-04 12:56:31', '2012-05-05 12:56:31',

'2012-05-06 12:56:31'],

dtype='datetime64[ns]', freq='D')

Sometimes you will have start or end dates with time information but want to gener‐ ate a set of timestamps normalized to midnight as a convention. To do this, there is a normalize option:

In [80]: pd.date\_range('2012-05-02 12:56:31', periods=5, normalize=True) Out[80]:

DatetimeIndex(['2012-05-02', '2012-05-03', '2012-05-04', '2012-05-05',

'2012-05-06'],

dtype='datetime64[ns]', freq='D')

### Frequencies and Date Offsets

Frequencies in pandas are composed of a base frequency and a multiplier. Base fre‐ quencies are typically referred to by a string alias, like 'M' for monthly or 'H' for hourly. For each base frequency, there is an object defined generally referred to as a date offset. For example, hourly frequency can be represented with the Hour class:

In [81]: **from pandas.tseries.offsets import** Hour, Minute In [82]: hour = Hour()

In [83]: hour Out[83]: <Hour>

You can define a multiple of an offset by passing an integer:

In [84]: four\_hours = Hour(4)

In [85]: four\_hours Out[85]: <4 \* Hours>

In most applications, you would never need to explicitly create one of these objects, instead using a string alias like 'H' or '4H'. Putting an integer before the base fre‐ quency creates a multiple:

In [86]: pd.date\_range('2000-01-01', '2000-01-03 23:59', freq='4h') Out[86]:

DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 04:00:00',

'2000-01-01 08:00:00', '2000-01-01 12:00:00',

'2000-01-01 16:00:00', '2000-01-01 20:00:00',

'2000-01-02 00:00:00', '2000-01-02 04:00:00',

'2000-01-02 08:00:00', '2000-01-02 12:00:00',

'2000-01-02 16:00:00', '2000-01-02 20:00:00',

'2000-01-03 00:00:00', '2000-01-03 04:00:00',

'2000-01-03 08:00:00', '2000-01-03 12:00:00',

'2000-01-03 16:00:00', '2000-01-03 20:00:00'],

dtype='datetime64[ns]', freq='4H')

Many offsets can be combined together by addition:

In [87]: Hour(2) + Minute(30) Out[87]: <150 \* Minutes>

Similarly, you can pass frequency strings, like '1h30min', that will effectively be parsed to the same expression:

In [88]: pd.date\_range('2000-01-01', periods=10, freq='1h30min') Out[88]:

DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 01:30:00',

'2000-01-01 03:00:00', '2000-01-01 04:30:00',

'2000-01-01 06:00:00', '2000-01-01 07:30:00',

'2000-01-01 09:00:00', '2000-01-01 10:30:00',

'2000-01-01 12:00:00', '2000-01-01 13:30:00'],

dtype='datetime64[ns]', freq='90T')

Some frequencies describe points in time that are not evenly spaced. For example, 'M' (calendar month end) and 'BM' (last business/weekday of month) depend on the number of days in a month and, in the latter case, whether the month ends on a weekend or not. We refer to these as anchored offsets.

Refer back to [Table 11-4](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark10) for a listing of frequency codes and date offset classes avail‐ able in pandas.

Users can define their own custom frequency classes to provide date logic not available in pandas, though the full details of that are outside the scope of this book.

**Week of month dates**

One useful frequency class is “week of month,” starting with WOM. This enables you to get dates like the third Friday of each month:

In [89]: rng = pd.date\_range('2012-01-01', '2012-09-01', freq='WOM-3FRI')

In [90]: list(rng) Out[90]:

[Timestamp('2012-01-20 00:00:00', freq='WOM-3FRI'),

Timestamp('2012-02-17 00:00:00', freq='WOM-3FRI'),

Timestamp('2012-03-16 00:00:00', freq='WOM-3FRI'),

Timestamp('2012-04-20 00:00:00', freq='WOM-3FRI'),

Timestamp('2012-05-18 00:00:00', freq='WOM-3FRI'),

Timestamp('2012-06-15 00:00:00', freq='WOM-3FRI'),

Timestamp('2012-07-20 00:00:00', freq='WOM-3FRI'),

Timestamp('2012-08-17 00:00:00', freq='WOM-3FRI')]

### Shifting (Leading and Lagging) Data

“Shifting” refers to moving data backward and forward through time. Both Series and DataFrame have a shift method for doing naive shifts forward or backward, leaving the index unmodified:

In [91]: ts = pd.Series(np.random.randn(4),

....: index=pd.date\_range('1/1/2000', periods=4, freq='M'))

|  |  |
| --- | --- |
| In [92]: ts |  |
| Out[92]: |
| 2000-01-31 | -0.066748 |
| 2000-02-29 | 0.838639 |
| 2000-03-31 | -0.117388 |
| 2000-04-30 | -0.517795 |

Freq: M, dtype: float64

In [93]: ts.shift(2) Out[93]:

2000-01-31 NaN

2000-02-29 NaN

2000-03-31 -0.066748

2000-04-30 0.838639

Freq: M, dtype: float64

In [94]: ts.shift(-2) Out[94]:

2000-01-31 -0.117388

2000-02-29 -0.517795

2000-03-31 NaN

2000-04-30 NaN

Freq: M, dtype: float64

When we shift like this, missing data is introduced either at the start or the end of the time series.

A common use of shift is computing percent changes in a time series or multiple time series as DataFrame columns. This is expressed as:

ts / ts.shift(1) - 1

Because naive shifts leave the index unmodified, some data is discarded. Thus if the frequency is known, it can be passed to shift to advance the timestamps instead of simply the data:

In [95]: ts.shift(2, freq='M') Out[95]:

2000-03-31 -0.066748

2000-04-30 0.838639

2000-05-31 -0.117388

2000-06-30 -0.517795

Freq: M, dtype: float64

Other frequencies can be passed, too, giving you some flexibility in how to lead and lag the data:

In [96]: ts.shift(3, freq='D') Out[96]:

|  |  |
| --- | --- |
| 2000-02-03 | -0.066748 |
| 2000-03-03 | 0.838639 |
| 2000-04-03 | -0.117388 |
| 2000-05-03 | -0.517795 |

dtype: float64

In [97]: ts.shift(1, freq='90T') Out[97]:

2000-01-31 01:30:00 -0.066748

2000-02-29 01:30:00 0.838639

2000-03-31 01:30:00 -0.117388

2000-04-30 01:30:00 -0.517795

Freq: M, dtype: float64

The T here stands for minutes.

#### Shifting dates with offsets

The pandas date offsets can also be used with datetime or Timestamp objects:

In [98]: **from pandas.tseries.offsets import** Day, MonthEnd In [99]: now = datetime(2011, 11, 17)

In [100]: now + 3 \* Day()

Out[100]: Timestamp('2011-11-20 00:00:00')

If you add an anchored offset like MonthEnd, the first increment will “roll forward” a date to the next date according to the frequency rule:

In [101]: now + MonthEnd()

Out[101]: Timestamp('2011-11-30 00:00:00')

In [102]: now + MonthEnd(2)

Out[102]: Timestamp('2011-12-31 00:00:00')

Anchored offsets can explicitly “roll” dates forward or backward by simply using their

rollforward and rollback methods, respectively:

In [103]: offset = MonthEnd()

In [104]: offset.rollforward(now)

Out[104]: Timestamp('2011-11-30 00:00:00')

In [105]: offset.rollback(now)

Out[105]: Timestamp('2011-10-31 00:00:00')

A creative use of date offsets is to use these methods with groupby:

In [106]: ts = pd.Series(np.random.randn(20),

.....: index=pd.date\_range('1/15/2000', periods=20, freq='4d'))

|  |  |
| --- | --- |
| In [107]: ts |  |
| Out[107]: |  |
| 2000-01-15 | -0.116696 |
| 2000-01-19 | 2.389645 |
| 2000-01-23 | -0.932454 |
| 2000-01-27 | -0.229331 |
| 2000-01-31 | -1.140330 |
| 2000-02-04 | 0.439920 |
| 2000-02-08 | -0.823758 |
| 2000-02-12 | -0.520930 |
| 2000-02-16 | 0.350282 |
| 2000-02-20 | 0.204395 |
| 2000-02-24 | 0.133445 |
| 2000-02-28 | 0.327905 |
| 2000-03-03 | 0.072153 |
| 2000-03-07 | 0.131678 |
| 2000-03-11 | -1.297459 |
| 2000-03-15 | 0.997747 |
| 2000-03-19 | 0.870955 |
| 2000-03-23 | -0.991253 |
| 2000-03-27 | 0.151699 |
| 2000-03-31 | 1.266151 |

Freq: 4D, dtype: float64

In [108]: ts.groupby(offset.rollforward).mean() Out[108]:

|  |  |
| --- | --- |
| 2000-01-31 | -0.005833 |
| 2000-02-29 | 0.015894 |
| 2000-03-31 | 0.150209 |

dtype: float64

Of course, an easier and faster way to do this is using resample (we’ll discuss this in much more depth in [Section 11.6, “Resampling and Frequency Conversion,” on page](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark13) [348](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark13)):

In [109]: ts.resample('M').mean() Out[109]:

|  |  |
| --- | --- |
| 2000-01-31 | -0.005833 |
| 2000-02-29 | 0.015894 |
| 2000-03-31 | 0.150209 |

Freq: M, dtype: float64

### Time Zone Handling

Working with time zones is generally considered one of the most unpleasant parts of time series manipulation. As a result, many time series users choose to work with time series in coordinated universal time or UTC, which is the successor to Greenwich Mean Time and is the current international standard. Time zones are expressed as offsets from UTC; for example, New York is four hours behind UTC during daylight saving time and five hours behind the rest of the year.

In Python, time zone information comes from the third-party pytz library (installa‐ ble with pip or conda), which exposes the Olson database, a compilation of world time zone information. This is especially important for historical data because the daylight saving time (DST) transition dates (and even UTC offsets) have been changed numerous times depending on the whims of local governments. In the Uni‐ ted States, the DST transition times have been changed many times since 1900!

For detailed information about the pytz library, you’ll need to look at that library’s documentation. As far as this book is concerned, pandas wraps pytz’s functionality so you can ignore its API outside of the time zone names. Time zone names can be found interactively and in the docs:

In [110]: import pytz

In [111]: pytz.common\_timezones[-5:]

Out[111]: ['US/Eastern', 'US/Hawaii', 'US/Mountain', 'US/Pacific', 'UTC']

To get a time zone object from pytz, use pytz.timezone:

In [112]: tz = pytz.timezone('America/New\_York')

In [113]: tz

Out[113]: <DstTzInfo 'America/New\_York' LMT-1 day, 19:04:00 STD>

Methods in pandas will accept either time zone names or these objects.

#### Time Zone Localization and Conversion

By default, time series in pandas are time zone naive. For example, consider the fol‐ lowing time series:

In [114]: rng = pd.date\_range('3/9/2012 9:30', periods=6, freq='D') In [115]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

|  |  |
| --- | --- |
| In [116]: ts |  |
| Out[116]: |
| 2012-03-09 09:30:00 | -0.202469 |
| 2012-03-10 09:30:00 | 0.050718 |
| 2012-03-11 09:30:00 | 0.639869 |
| 2012-03-12 09:30:00 | 0.597594 |

2012-03-13 09:30:00 -0.797246

2012-03-14 09:30:00 0.472879

Freq: D, dtype: float64

The index’s tz field is None:

In [117]: **print**(ts.index.tz) None

Date ranges can be generated with a time zone set:

In [118]: pd.date\_range('3/9/2012 9:30', periods=10, freq='D', tz='UTC') Out[118]:

DatetimeIndex(['2012-03-09 09:30:00+00:00', '2012-03-10 09:30:00+00:00',

'2012-03-11 09:30:00+00:00', '2012-03-12 09:30:00+00:00',

'2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00',

'2012-03-15 09:30:00+00:00', '2012-03-16 09:30:00+00:00',

'2012-03-17 09:30:00+00:00', '2012-03-18 09:30:00+00:00'],

dtype='datetime64[ns, UTC]', freq='D')

Conversion from naive to localized is handled by the tz\_localize method:

In [119]: ts Out[119]:

|  |  |
| --- | --- |
| 2012-03-09 09:30:00 | -0.202469 |
| 2012-03-10 09:30:00 | 0.050718 |
| 2012-03-11 09:30:00 | 0.639869 |
| 2012-03-12 09:30:00 | 0.597594 |
| 2012-03-13 09:30:00 | -0.797246 |
| 2012-03-14 09:30:00 | 0.472879 |

Freq: D, dtype: float64

In [120]: ts\_utc = ts.tz\_localize('UTC')

|  |  |
| --- | --- |
| In [121]: ts\_utc  Out[121]:  2012-03-09 09:30:00+00:00 | -0.202469 |
| 2012-03-10 09:30:00+00:00 | 0.050718 |
| 2012-03-11 09:30:00+00:00 | 0.639869 |
| 2012-03-12 09:30:00+00:00 | 0.597594 |
| 2012-03-13 09:30:00+00:00 | -0.797246 |
| 2012-03-14 09:30:00+00:00  Freq: D, dtype: float64 | 0.472879 |
| In [122]: ts\_utc.index Out[122]:  DatetimeIndex(['2012-03-09 | 09:30:00+00:00', '2012-03-10 09:30:00+00:00', |

'2012-03-11 09:30:00+00:00', '2012-03-12 09:30:00+00:00',

'2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00'],

dtype='datetime64[ns, UTC]', freq='D')

Once a time series has been localized to a particular time zone, it can be converted to another time zone with tz\_convert:

In [123]: ts\_utc.tz\_convert('America/New\_York') Out[123]:

|  |  |
| --- | --- |
| 2012-03-09 04:30:00-05:00 | -0.202469 |
| 2012-03-10 04:30:00-05:00 | 0.050718 |
| 2012-03-11 05:30:00-04:00 | 0.639869 |
| 2012-03-12 05:30:00-04:00 | 0.597594 |
| 2012-03-13 05:30:00-04:00 | -0.797246 |
| 2012-03-14 05:30:00-04:00 | 0.472879 |
| Freq: D, dtype: float64 |  |

In the case of the preceding time series, which straddles a DST transition in the Amer ica/New\_York time zone, we could localize to EST and convert to, say, UTC or Berlin time:

In [124]: ts\_eastern = ts.tz\_localize('America/New\_York')

In [125]: ts\_eastern.tz\_convert('UTC') Out[125]:

|  |  |
| --- | --- |
| 2012-03-09 14:30:00+00:00 | -0.202469 |
| 2012-03-10 14:30:00+00:00 | 0.050718 |
| 2012-03-11 13:30:00+00:00 | 0.639869 |
| 2012-03-12 13:30:00+00:00 | 0.597594 |
| 2012-03-13 13:30:00+00:00 | -0.797246 |
| 2012-03-14 13:30:00+00:00 | 0.472879 |
| Freq: D, dtype: float64 |  |

In [126]: ts\_eastern.tz\_convert('Europe/Berlin') Out[126]:

|  |  |
| --- | --- |
| 2012-03-09 15:30:00+01:00 | -0.202469 |
| 2012-03-10 15:30:00+01:00 | 0.050718 |
| 2012-03-11 14:30:00+01:00 | 0.639869 |
| 2012-03-12 14:30:00+01:00 | 0.597594 |
| 2012-03-13 14:30:00+01:00 | -0.797246 |
| 2012-03-14 14:30:00+01:00 | 0.472879 |
| Freq: D, dtype: float64 |  |

tz\_localize and tz\_convert are also instance methods on DatetimeIndex:

In [127]: ts.index.tz\_localize('Asia/Shanghai') Out[127]:

DatetimeIndex(['2012-03-09 09:30:00+08:00', '2012-03-10 09:30:00+08:00',

'2012-03-11 09:30:00+08:00', '2012-03-12 09:30:00+08:00',

'2012-03-13 09:30:00+08:00', '2012-03-14 09:30:00+08:00'],

dtype='datetime64[ns, Asia/Shanghai]', freq='D')

Localizing naive timestamps also checks for ambiguous or non- existent times around daylight saving time transitions.

Operations with Time Zone−Aware Timestamp Objects

Similar to time series and date ranges, individual Timestamp objects similarly can be localized from naive to time zone–aware and converted from one time zone to another:

In [128]: stamp = pd.Timestamp('2011-03-12 04:00') In [129]: stamp\_utc = stamp.tz\_localize('utc')

In [130]: stamp\_utc.tz\_convert('America/New\_York')

Out[130]: Timestamp('2011-03-11 23:00:00-0500', tz='America/New\_York')

You can also pass a time zone when creating the Timestamp:

In [131]: stamp\_moscow = pd.Timestamp('2011-03-12 04:00', tz='Europe/Moscow')

In [132]: stamp\_moscow

Out[132]: Timestamp('2011-03-12 04:00:00+0300', tz='Europe/Moscow')

Time zone–aware Timestamp objects internally store a UTC timestamp value as nano‐ seconds since the Unix epoch (January 1, 1970); this UTC value is invariant between time zone conversions:

In [133]: stamp\_utc.value

Out[133]: 1299902400000000000

In [134]: stamp\_utc.tz\_convert('America/New\_York').value

Out[134]: 1299902400000000000

When performing time arithmetic using pandas’s DateOffset objects, pandas respects daylight saving time transitions where possible. Here we construct time‐ stamps that occur right before DST transitions (forward and backward). First, 30 minutes before transitioning to DST:

In [135]: **from pandas.tseries.offsets import** Hour

In [136]: stamp = pd.Timestamp('2012-03-12 01:30', tz='US/Eastern') In [137]: stamp

Out[137]: Timestamp('2012-03-12 01:30:00-0400', tz='US/Eastern')

In [138]: stamp + Hour()

Out[138]: Timestamp('2012-03-12 02:30:00-0400', tz='US/Eastern')

Then, 90 minutes before transitioning out of DST:

In [139]: stamp = pd.Timestamp('2012-11-04 00:30', tz='US/Eastern')

In [140]: stamp

Out[140]: Timestamp('2012-11-04 00:30:00-0400', tz='US/Eastern')

In [141]: stamp + 2 \* Hour()

Out[141]: Timestamp('2012-11-04 01:30:00-0500', tz='US/Eastern')

#### Operations Between Different Time Zones

If two time series with different time zones are combined, the result will be UTC. Since the timestamps are stored under the hood in UTC, this is a straightforward operation and requires no conversion to happen:

In [142]: rng = pd.date\_range('3/7/2012 9:30', periods=10, freq='B') In [143]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [144]: ts Out[144]:

|  |  |  |
| --- | --- | --- |
| 2012-03-07 | 09:30:00 | 0.522356 |
| 2012-03-08 | 09:30:00 | -0.546348 |
| 2012-03-09 | 09:30:00 | -0.733537 |
| 2012-03-12 | 09:30:00 | 1.302736 |
| 2012-03-13 | 09:30:00 | 0.022199 |
| 2012-03-14 | 09:30:00 | 0.364287 |
| 2012-03-15 | 09:30:00 | -0.922839 |
| 2012-03-16 | 09:30:00 | 0.312656 |
| 2012-03-19 | 09:30:00 | -1.128497 |
| 2012-03-20 | 09:30:00 | -0.333488 |

Freq: B, dtype: float64

In [145]: ts1 = ts[:7].tz\_localize('Europe/London') In [146]: ts2 = ts1[2:].tz\_convert('Europe/Moscow') In [147]: result = ts1 + ts2

In [148]: result.index Out[148]:

DatetimeIndex(['2012-03-07 09:30:00+00:00', '2012-03-08 09:30:00+00:00',

'2012-03-09 09:30:00+00:00', '2012-03-12 09:30:00+00:00',

'2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00',

'2012-03-15 09:30:00+00:00'],

dtype='datetime64[ns, UTC]', freq='B')

### Periods and Period Arithmetic

Periods represent timespans, like days, months, quarters, or years. The Period class represents this data type, requiring a string or integer and a frequency from [Table 11-4](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark10):

In [149]: p = pd.Period(2007, freq='A-DEC')

In [150]: p

Out[150]: Period('2007', 'A-DEC')

In this case, the Period object represents the full timespan from January 1, 2007, to December 31, 2007, inclusive. Conveniently, adding and subtracting integers from periods has the effect of shifting by their frequency:

In [151]: p + 5

Out[151]: Period('2012', 'A-DEC')

In [152]: p - 2

Out[152]: Period('2005', 'A-DEC')

If two periods have the same frequency, their difference is the number of units between them:

In [153]: pd.Period('2014', freq='A-DEC') - p Out[153]: 7

Regular ranges of periods can be constructed with the period\_range function:

In [154]: rng = pd.period\_range('2000-01-01', '2000-06-30', freq='M')

In [155]: rng

Out[155]: PeriodIndex(['2000-01', '2000-02', '2000-03', '2000-04', '2000-05', '20

00-06'], dtype='period[M]', freq='M')

The PeriodIndex class stores a sequence of periods and can serve as an axis index in any pandas data structure:

In [156]: pd.Series(np.random.randn(6), index=rng) Out[156]:

|  |  |
| --- | --- |
| 2000-01 | -0.514551 |
| 2000-02 | -0.559782 |
| 2000-03 | -0.783408 |
| 2000-04 | -1.797685 |
| 2000-05 | -0.172670 |
| 2000-06 | 0.680215 |
| Freq: M, | dtype: float64 |

If you have an array of strings, you can also use the PeriodIndex class:

In [157]: values = ['2001Q3', '2002Q2', '2003Q1']

In [158]: index = pd.PeriodIndex(values, freq='Q-DEC') In [159]: index

Out[159]: PeriodIndex(['2001Q3', '2002Q2', '2003Q1'], dtype='period[Q-DEC]', freq

='Q-DEC')

#### Period Frequency Conversion

Periods and PeriodIndex objects can be converted to another frequency with their

asfreq method. As an example, suppose we had an annual period and wanted to

convert it into a monthly period either at the start or end of the year. This is fairly straightforward:

In [160]: p = pd.Period('2007', freq='A-DEC')

In [161]: p

Out[161]: Period('2007', 'A-DEC')

In [162]: p.asfreq('M', how='start')

Out[162]: Period('2007-01', 'M')

In [163]: p.asfreq('M', how='end')

Out[163]: Period('2007-12', 'M')

You can think of Period('2007', 'A-DEC') as being a sort of cursor pointing to a span of time, subdivided by monthly periods. See [Figure 11-1](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark11) for an illustration of this. For a fiscal year ending on a month other than December, the corresponding monthly subperiods are different:

In [164]: p = pd.Period('2007', freq='A-JUN')

In [165]: p

Out[165]: Period('2007', 'A-JUN')

In [166]: p.asfreq('M', 'start')

Out[166]: Period('2006-07', 'M')

In [167]: p.asfreq('M', 'end')

Out[167]: Period('2007-06', 'M')

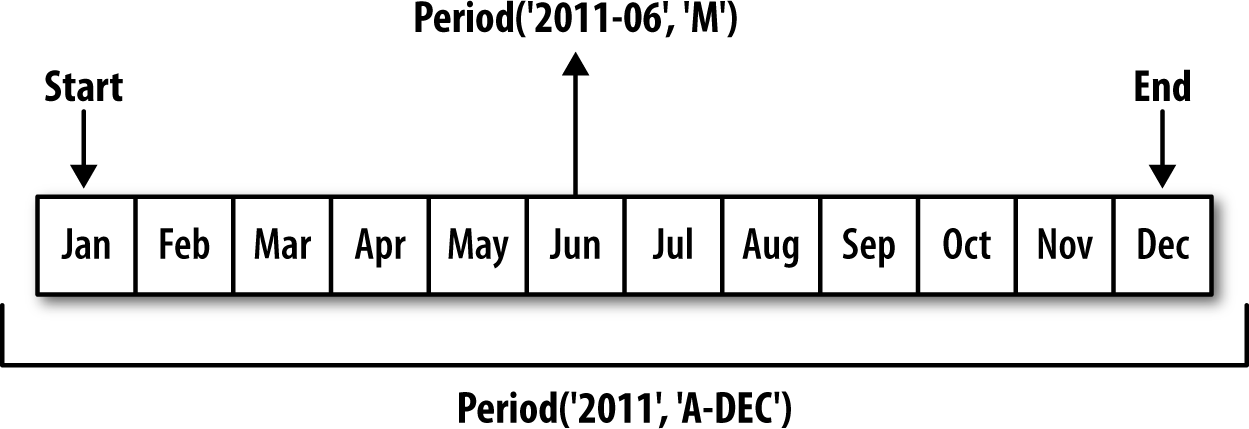


Figure 11-1. Period frequency conversion illustration

When you are converting from high to low frequency, pandas determines the super‐ period depending on where the subperiod “belongs.” For example, in A-JUN fre‐ quency, the month Aug-2007 is actually part of the 2008 period:

In [168]: p = pd.Period('Aug-2007', 'M')

In [169]: p.asfreq('A-JUN')

Out[169]: Period('2008', 'A-JUN')

Whole PeriodIndex objects or time series can be similarly converted with the same semantics:

In [170]: rng = pd.period\_range('2006', '2009', freq='A-DEC')

In [171]: ts = pd.Series(np.random.randn(len(rng)), index=rng) In [172]: ts

Out[172]:

|  |  |
| --- | --- |
| 2006 | 1.607578 |
| 2007 | 0.200381 |
| 2008 | -0.834068 |
| 2009 | -0.302988 |

Freq: A-DEC, dtype: float64

In [173]: ts.asfreq('M', how='start') Out[173]:

|  |  |
| --- | --- |
| 2006-01 | 1.607578 |
| 2007-01 | 0.200381 |
| 2008-01 | -0.834068 |
| 2009-01 | -0.302988 |

Freq: M, dtype: float64

Here, the annual periods are replaced with monthly periods corresponding to the first month falling within each annual period. If we instead wanted the last business day of each year, we can use the 'B' frequency and indicate that we want the end of the period:

In [174]: ts.asfreq('B', how='end') Out[174]:

|  |  |
| --- | --- |
| 2006-12-29 | 1.607578 |
| 2007-12-31 | 0.200381 |
| 2008-12-31 | -0.834068 |
| 2009-12-31 | -0.302988 |

Freq: B, dtype: float64

#### Quarterly Period Frequencies

Quarterly data is standard in accounting, finance, and other fields. Much quarterly data is reported relative to a fiscal year end, typically the last calendar or business day of one of the 12 months of the year. Thus, the period 2012Q4 has a different meaning depending on fiscal year end. pandas supports all 12 possible quarterly frequencies as Q-JAN through Q-DEC:

In [175]: p = pd.Period('2012Q4', freq='Q-JAN')

In [176]: p

Out[176]: Period('2012Q4', 'Q-JAN')

In the case of fiscal year ending in January, 2012Q4 runs from November through Jan‐ uary, which you can check by converting to daily frequency. See [Figure 11-2](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark12) for an illustration.



Figure 11-2. Different quarterly frequency conventions

In [177]: p.asfreq('D', 'start')

Out[177]: Period('2011-11-01', 'D')

In [178]: p.asfreq('D', 'end')

Out[178]: Period('2012-01-31', 'D')

Thus, it’s possible to do easy period arithmetic; for example, to get the timestamp at 4 PM on the second-to-last business day of the quarter, you could do:

In [179]: p4pm = (p.asfreq('B', 'e') - 1).asfreq('T', 's') + 16 \* 60

In [180]: p4pm

Out[180]: Period('2012-01-30 16:00', 'T')

In [181]: p4pm.to\_timestamp()

Out[181]: Timestamp('2012-01-30 16:00:00')

You can generate quarterly ranges using period\_range. Arithmetic is identical, too:

In [182]: rng = pd.period\_range('2011Q3', '2012Q4', freq='Q-JAN') In [183]: ts = pd.Series(np.arange(len(rng)), index=rng)

In [184]: ts Out[184]:

2011Q3 0

2011Q4 1

2012Q1 2

2012Q2 3

2012Q3 4

2012Q4 5

Freq: Q-JAN, dtype: int64

In [185]: new\_rng = (rng.asfreq('B', 'e') - 1).asfreq('T', 's') + 16 \* 60 In [186]: ts.index = new\_rng.to\_timestamp()

In [187]: ts Out[187]:

2010-10-28 16:00:00 0

2011-01-28 16:00:00 1

2011-04-28 16:00:00 2

2011-07-28 16:00:00 3

2011-10-28 16:00:00 4

2012-01-30 16:00:00 5

dtype: int64

#### Converting Timestamps to Periods (and Back)

Series and DataFrame objects indexed by timestamps can be converted to periods with the to\_period method:

In [188]: rng = pd.date\_range('2000-01-01', periods=3, freq='M') In [189]: ts = pd.Series(np.random.randn(3), index=rng)

|  |  |
| --- | --- |
| In [190]: ts |  |
| Out[190]: |
| 2000-01-31 | 1.663261 |
| 2000-02-29 | -0.996206 |
| 2000-03-31 | 1.521760 |

Freq: M, dtype: float64

In [191]: pts = ts.to\_period() In [192]: pts

Out[192]:

|  |  |
| --- | --- |
| 2000-01 | 1.663261 |
| 2000-02 | -0.996206 |
| 2000-03 | 1.521760 |

Freq: M, dtype: float64

Since periods refer to non-overlapping timespans, a timestamp can only belong to a single period for a given frequency. While the frequency of the new PeriodIndex is inferred from the timestamps by default, you can specify any frequency you want. There is also no problem with having duplicate periods in the result:

In [193]: rng = pd.date\_range('1/29/2000', periods=6, freq='D') In [194]: ts2 = pd.Series(np.random.randn(6), index=rng)

|  |  |
| --- | --- |
| In [195]: ts2 |  |
| Out[195]: |
| 2000-01-29 | 0.244175 |
| 2000-01-30 | 0.423331 |
| 2000-01-31 | -0.654040 |
| 2000-02-01 | 2.089154 |
| 2000-02-02 | -0.060220 |

2000-02-03 -0.167933

Freq: D, dtype: float64

In [196]: ts2.to\_period('M') Out[196]:

|  |  |
| --- | --- |
| 2000-01 | 0.244175 |
| 2000-01 | 0.423331 |
| 2000-01 | -0.654040 |
| 2000-02 | 2.089154 |
| 2000-02 | -0.060220 |
| 2000-02 | -0.167933 |

Freq: M, dtype: float64

To convert back to timestamps, use to\_timestamp:

In [197]: pts = ts2.to\_period()

|  |  |
| --- | --- |
| In [198]: pts |  |
| Out[198]: |
| 2000-01-29 | 0.244175 |
| 2000-01-30 | 0.423331 |
| 2000-01-31 | -0.654040 |
| 2000-02-01 | 2.089154 |
| 2000-02-02 | -0.060220 |
| 2000-02-03 | -0.167933 |

Freq: D, dtype: float64

In [199]: pts.to\_timestamp(how='end') Out[199]:

|  |  |
| --- | --- |
| 2000-01-29 | 0.244175 |
| 2000-01-30 | 0.423331 |
| 2000-01-31 | -0.654040 |
| 2000-02-01 | 2.089154 |
| 2000-02-02 | -0.060220 |
| 2000-02-03 | -0.167933 |

Freq: D, dtype: float64

#### Creating a PeriodIndex from Arrays

Fixed frequency datasets are sometimes stored with timespan information spread across multiple columns. For example, in this macroeconomic dataset, the year and quarter are in different columns:

In [200]: data = pd.read\_csv('examples/macrodata.csv')

In [201]: data.head(5) Out[201]:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| year | quarter | realgdp | realcons | realinv | realgovt | realdpi | cpi \ |
| 0 1959.0 | 1.0 | 2710.349 | 1707.4 | 286.898 | 470.045 | 1886.9 | 28.98 |
| 1 1959.0 | 2.0 | 2778.801 | 1733.7 | 310.859 | 481.301 | 1919.7 | 29.15 |
| 2 1959.0 | 3.0 | 2775.488 | 1751.8 | 289.226 | 491.260 | 1916.4 | 29.35 |
| 3 1959.0 | 4.0 | 2785.204 | 1753.7 | 299.356 | 484.052 | 1931.3 | 29.37 |

4 1960.0 1.0 2847.699 1770.5 331.722 462.199 1955.5 29.54

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| m1 | tbilrate | unemp | pop | infl | realint |
| 0 139.7 | 2.82 | 5.8 | 177.146 | 0.00 | 0.00 |
| 1 141.7 | 3.08 | 5.1 | 177.830 | 2.34 | 0.74 |
| 2 140.5 | 3.82 | 5.3 | 178.657 | 2.74 | 1.09 |
| 3 140.0 | 4.33 | 5.6 | 179.386 | 0.27 | 4.06 |
| 4 139.6 | 3.50 | 5.2 | 180.007 | 2.31 | 1.19 |

In [202]: data.year Out[202]:

|  |  |
| --- | --- |
| 0 | 1959.0 |
| 1 | 1959.0 |
| 2 | 1959.0 |
| 3 | 1959.0 |
| 4 | 1960.0 |
| 5 | 1960.0 |
| 6 | 1960.0 |
| 7 | 1960.0 |
| 8 | 1961.0 |
| 9 | 1961.0 |
|  | ... |
| 193 | 2007.0 |
| 194 | 2007.0 |
| 195 | 2007.0 |
| 196 | 2008.0 |
| 197 | 2008.0 |
| 198 | 2008.0 |
| 199 | 2008.0 |
| 200 | 2009.0 |
| 201 | 2009.0 |
| 202 | 2009.0 |

Name: year, Length: 203, dtype: float64 In [203]: data.quarter

Out[203]:

|  |  |
| --- | --- |
| 0 | 1.0 |
| 1 | 2.0 |
| 2 | 3.0 |
| 3 | 4.0 |
| 4 | 1.0 |
| 5 | 2.0 |
| 6 | 3.0 |
| 7 | 4.0 |
| 8 | 1.0 |
| 9 | 2.0 |

...

193 2.0

194 3.0

195 4.0

196 1.0

197 2.0

198 3.0

199 4.0

200 1.0

201 2.0

202 3.0

Name: quarter, Length: 203, dtype: float64

By passing these arrays to PeriodIndex with a frequency, you can combine them to form an index for the DataFrame:

In [204]: index = pd.PeriodIndex(year=data.year, quarter=data.quarter,

.....: freq='Q-DEC')

In [205]: index Out[205]:

PeriodIndex(['1959Q1', '1959Q2', '1959Q3', '1959Q4', '1960Q1', '1960Q2', '1960Q3', '1960Q4', '1961Q1', '1961Q2',

...

'2007Q2', '2007Q3', '2007Q4', '2008Q1', '2008Q2', '2008Q3',

'2008Q4', '2009Q1', '2009Q2', '2009Q3'],

dtype='period[Q-DEC]', length=203, freq='Q-DEC') In [206]: data.index = index

In [207]: data.infl Out[207]:

1959Q1 0.00

1959Q2 2.34

1959Q3 2.74

1959Q4 0.27

1960Q1 2.31

1960Q2 0.14

1960Q3 2.70

1960Q4 1.21

1961Q1 -0.40

1961Q2 1.47

...

2007Q2 2.75

2007Q3 3.45

2007Q4 6.38

2008Q1 2.82

2008Q2 8.53

2008Q3 -3.16

2008Q4 -8.79

2009Q1 0.94

2009Q2 3.37

2009Q3 3.56

Freq: Q-DEC, Name: infl, Length: 203, dtype: float64

### Resampling and Frequency Conversion

Resampling refers to the process of converting a time series from one frequency to another. Aggregating higher frequency data to lower frequency is called downsam‐ pling, while converting lower frequency to higher frequency is called upsampling. Not all resampling falls into either of these categories; for example, converting W-WED (weekly on Wednesday) to W-FRI is neither upsampling nor downsampling.

pandas objects are equipped with a resample method, which is the workhorse func‐ tion for all frequency conversion. resample has a similar API to groupby; you call resample to group the data, then call an aggregation function:

In [208]: rng = pd.date\_range('2000-01-01', periods=100, freq='D')

|  |  |
| --- | --- |
| In [209]: ts | = pd.Series(np.random.randn(len(rng)), index=rng) |
| In [210]: ts |  |
| Out[210]: |  |
| 2000-01-01 | 0.631634 |
| 2000-01-02 | -1.594313 |
| 2000-01-03 | -1.519937 |
| 2000-01-04 | 1.108752 |
| 2000-01-05 | 1.255853 |
| 2000-01-06 | -0.024330 |
| 2000-01-07 | -2.047939 |
| 2000-01-08 | -0.272657 |
| 2000-01-09 | -1.692615 |
| 2000-01-10 | 1.423830 |
|  | ... |
| 2000-03-31 | -0.007852 |
| 2000-04-01 | -1.638806 |
| 2000-04-02 | 1.401227 |
| 2000-04-03 | 1.758539 |
| 2000-04-04 | 0.628932 |
| 2000-04-05 | -0.423776 |
| 2000-04-06 | 0.789740 |
| 2000-04-07 | 0.937568 |
| 2000-04-08 | -2.253294 |
| 2000-04-09 | -1.772919 |

Freq: D, Length: 100, dtype: float64

In [211]: ts.resample('M').mean() Out[211]:

|  |  |
| --- | --- |
| 2000-01-31 | -0.165893 |
| 2000-02-29 | 0.078606 |
| 2000-03-31 | 0.223811 |
| 2000-04-30 | -0.063643 |

Freq: M, dtype: float64

In [212]: ts.resample('M', kind='period').mean() Out[212]:

|  |  |
| --- | --- |
| 2000-01 | -0.165893 |
| 2000-02 | 0.078606 |
| 2000-03 | 0.223811 |
| 2000-04 | -0.063643 |

Freq: M, dtype: float64

resample is a flexible and high-performance method that can be used to process very large time series. The examples in the following sections illustrate its semantics and use. [Table 11-5](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark14) summarizes some of its options.

Table 11-5. Resample method arguments

**Argument Description**

freq String or DateOffset indicating desired resampled frequency (e.g., ‘M', ’5min', or Second(15)) axis Axis to resample on; default axis=0

fill\_method How to interpolate when upsampling, as in 'ffill' or 'bfill'; by default does no interpolation

closed In downsampling, which end of each interval is closed (inclusive), 'right' or 'left'

label In downsampling, how to label the aggregated result, with the 'right' or 'left' bin edge (e.g., the 9:30 to 9:35 five-minute interval could be labeled 9:30 or 9:35)

loffset Time adjustment to the bin labels, such as '-1s' / Second(-1) to shift the aggregate labels one second earlier

limit When forward or backward filling, the maximum number of periods to fill

kind Aggregate to periods ('period') or timestamps ('timestamp'); defaults to the type of index the time series has

convention When resampling periods, the convention ('start' or 'end') for converting the low-frequency period to high frequency; defaults to 'end'

#### Downsampling

Aggregating data to a regular, lower frequency is a pretty normal time series task. The data you’re aggregating doesn’t need to be fixed frequently; the desired frequency defines bin edges that are used to slice the time series into pieces to aggregate. For example, to convert to monthly, 'M' or 'BM', you need to chop up the data into one- month intervals. Each interval is said to be half-open; a data point can only belong to one interval, and the union of the intervals must make up the whole time frame. There are a couple things to think about when using resample to downsample data:

Which side of each interval is closed

How to label each aggregated bin, either with the start of the interval or the end To illustrate, let’s look at some one-minute data:

In [213]: rng = pd.date\_range('2000-01-01', periods=12, freq='T')

In [214]: ts = pd.Series(np.arange(12), index=rng)

In [215]: ts Out[215]:

|  |  |  |
| --- | --- | --- |
| 2000-01-01 | 00:00:00 | 0 |
| 2000-01-01 | 00:01:00 | 1 |
| 2000-01-01 | 00:02:00 | 2 |
| 2000-01-01 | 00:03:00 | 3 |
| 2000-01-01 | 00:04:00 | 4 |
| 2000-01-01 | 00:05:00 | 5 |
| 2000-01-01 | 00:06:00 | 6 |
| 2000-01-01 | 00:07:00 | 7 |
| 2000-01-01 | 00:08:00 | 8 |
| 2000-01-01 | 00:09:00 | 9 |
| 2000-01-01 | 00:10:00 | 10 |
| 2000-01-01 | 00:11:00 | 11 |

Freq: T, dtype: int64

Suppose you wanted to aggregate this data into five-minute chunks or bars by taking the sum of each group:

In [216]: ts.resample('5min', closed='right').sum() Out[216]:

1999-12-31 23:55:00 0

2000-01-01 00:00:00 15

2000-01-01 00:05:00 40

2000-01-01 00:10:00 11

Freq: 5T, dtype: int64

The frequency you pass defines bin edges in five-minute increments. By default, the left bin edge is inclusive, so the 00:00 value is included in the 00:00 to 00:05 interval.[1](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark15) Passing closed='right' changes the interval to be closed on the right:

In [217]: ts.resample('5min', closed='right').sum() Out[217]:

1999-12-31 23:55:00 0

2000-01-01 00:00:00 15

2000-01-01 00:05:00 40

2000-01-01 00:10:00 11

Freq: 5T, dtype: int64

The resulting time series is labeled by the timestamps from the left side of each bin. By passing label='right' you can label them with the right bin edge:

In [218]: ts.resample('5min', closed='right', label='right').sum() Out[218]:

2000-01-01 00:00:00 0

2000-01-01 00:05:00 15

1 The choice of the default values for closed and label might seem a bit odd to some users. In practice the choice is somewhat arbitrary; for some target frequencies, closed='left' is preferable, while for others closed='right' makes more sense. The important thing is that you keep in mind exactly how you are seg‐ menting the data.

2000-01-01 00:10:00 40

2000-01-01 00:15:00 11

Freq: 5T, dtype: int64

See [Figure 11-3](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark16) for an illustration of minute frequency data being resampled to five- minute frequency.

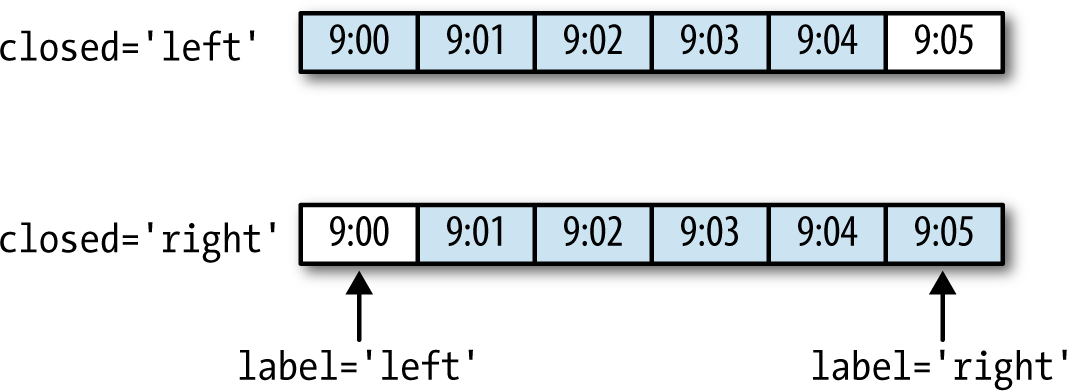


Figure 11-3. Five-minute resampling illustration of closed, label conventions

Lastly, you might want to shift the result index by some amount, say subtracting one second from the right edge to make it more clear which interval the timestamp refers to. To do this, pass a string or date offset to loffset:

In [219]: ts.resample('5min', closed='right',

|  |  |
| --- | --- |
| .....: | label='right', loffset='-1s').sum() |
| Out[219]: |  |
| 1999-12-31 23:59:59 | 0 |
| 2000-01-01 00:04:59 | 15 |
| 2000-01-01 00:09:59 | 40 |
| 2000-01-01 00:14:59 | 11 |

Freq: 5T, dtype: int64

You also could have accomplished the effect of loffset by calling the shift method on the result without the loffset.

##### Open-High-Low-Close (OHLC) resampling

In finance, a popular way to aggregate a time series is to compute four values for each bucket: the first (open), last (close), maximum (high), and minimal (low) values. By using the ohlc aggregate function you will obtain a DataFrame having columns con‐ taining these four aggregates, which are efficiently computed in a single sweep of the data:

In [220]: ts.resample('5min').ohlc() Out[220]:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | open | high | low | close |
| 2000-01-01 00:00:00 | 0 | 4 | 0 | 4 |
| 2000-01-01 00:05:00 | 5 | 9 | 5 | 9 |
| 2000-01-01 00:10:00 | 10 | 11 | 10 | 11 |

#### Upsampling and Interpolation

When converting from a low frequency to a higher frequency, no aggregation is needed. Let’s consider a DataFrame with some weekly data:

In [221]: frame = pd.DataFrame(np.random.randn(2, 4),

.....: index=pd.date\_range('1/1/2000', periods=2,

.....: freq='W-WED'),

.....: columns=['Colorado', 'Texas', 'New York', 'Ohio'])

In [222]: frame Out[222]:

Colorado Texas New York Ohio 2000-01-05 -0.896431 0.677263 0.036503 0.087102

2000-01-12 -0.046662 0.927238 0.482284 -0.867130

When you are using an aggregation function with this data, there is only one value per group, and missing values result in the gaps. We use the asfreq method to con‐ vert to the higher frequency without any aggregation:

In [223]: df\_daily = frame.resample('D').asfreq()

In [224]: df\_daily Out[224]:

Colorado Texas New York Ohio 2000-01-05 -0.896431 0.677263 0.036503 0.087102

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2000-01-06 | NaN | NaN | NaN | NaN |
| 2000-01-07 | NaN | NaN | NaN | NaN |
| 2000-01-08 | NaN | NaN | NaN | NaN |
| 2000-01-09 | NaN | NaN | NaN | NaN |
| 2000-01-10 | NaN | NaN | NaN | NaN |
| 2000-01-11 | NaN | NaN | NaN | NaN |
| 2000-01-12 | -0.046662 | 0.927238 | 0.482284 | -0.867130 |

Suppose you wanted to fill forward each weekly value on the non-Wednesdays. The same filling or interpolation methods available in the fillna and reindex methods are available for resampling:

In [225]: frame.resample('D').ffill() Out[225]:

|  |  |  |  |
| --- | --- | --- | --- |
| Colorado | Texas | New York | Ohio |
| 2000-01-05 -0.896431 | 0.677263 | 0.036503 | 0.087102 |
| 2000-01-06 -0.896431 | 0.677263 | 0.036503 | 0.087102 |
| 2000-01-07 -0.896431 | 0.677263 | 0.036503 | 0.087102 |
| 2000-01-08 -0.896431 | 0.677263 | 0.036503 | 0.087102 |
| 2000-01-09 -0.896431 | 0.677263 | 0.036503 | 0.087102 |
| 2000-01-10 -0.896431 | 0.677263 | 0.036503 | 0.087102 |
| 2000-01-11 -0.896431 | 0.677263 | 0.036503 | 0.087102 |
| 2000-01-12 -0.046662 | 0.927238 | 0.482284 | -0.867130 |

You can similarly choose to only fill a certain number of periods forward to limit how far to continue using an observed value:

In [226]: frame.resample('D').ffill(limit=2) Out[226]:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Colorado | Texas | New York | Ohio |
| 2000-01-05 | -0.896431 | 0.677263 | 0.036503 | 0.087102 |
| 2000-01-06 | -0.896431 | 0.677263 | 0.036503 | 0.087102 |
| 2000-01-07 | -0.896431 | 0.677263 | 0.036503 | 0.087102 |
| 2000-01-08 | NaN | NaN | NaN | NaN |
| 2000-01-09 | NaN | NaN | NaN | NaN |
| 2000-01-10 | NaN | NaN | NaN | NaN |
| 2000-01-11 | NaN | NaN | NaN | NaN |
| 2000-01-12 | -0.046662 | 0.927238 | 0.482284 | -0.867130 |

Notably, the new date index need not overlap with the old one at all:

In [227]: frame.resample('W-THU').ffill() Out[227]:

Colorado Texas New York Ohio 2000-01-06 -0.896431 0.677263 0.036503 0.087102

2000-01-13 -0.046662 0.927238 0.482284 -0.867130

#### Resampling with Periods

Resampling data indexed by periods is similar to timestamps:

In [228]: frame = pd.DataFrame(np.random.randn(24, 4),

.....: index=pd.period\_range('1-2000', '12-2001',

.....: freq='M'),

.....: columns=['Colorado', 'Texas', 'New York', 'Ohio'])

In [229]: frame[:5] Out[229]:

|  |  |  |  |
| --- | --- | --- | --- |
| Colorado | Texas | New York | Ohio |
| 2000-01 0.493841 | -0.155434 | 1.397286 | 1.507055 |
| 2000-02 -1.179442 | 0.443171 | 1.395676 | -0.529658 |
| 2000-03 0.787358 | 0.248845 | 0.743239 | 1.267746 |

2000-04 1.302395 -0.272154 -0.051532 -0.467740

2000-05 -1.040816 0.426419 0.312945 -1.115689

In [230]: annual\_frame = frame.resample('A-DEC').mean() In [231]: annual\_frame

Out[231]:

Colorado Texas New York Ohio 2000 0.556703 0.016631 0.111873 -0.027445

2001 0.046303 0.163344 0.251503 -0.157276

Upsampling is more nuanced, as you must make a decision about which end of the timespan in the new frequency to place the values before resampling, just like the asfreq method. The convention argument defaults to 'start' but can also be 'end':

*# Q-DEC: Quarterly, year ending in December*

In [232]: annual\_frame.resample('Q-DEC').ffill() Out[232]:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Colorado | Texas | New York | Ohio |
| 2000Q1 | 0.556703 | 0.016631 | 0.111873 | -0.027445 |
| 2000Q2 | 0.556703 | 0.016631 | 0.111873 | -0.027445 |
| 2000Q3 | 0.556703 | 0.016631 | 0.111873 | -0.027445 |
| 2000Q4 | 0.556703 | 0.016631 | 0.111873 | -0.027445 |
| 2001Q1 | 0.046303 | 0.163344 | 0.251503 | -0.157276 |
| 2001Q2 | 0.046303 | 0.163344 | 0.251503 | -0.157276 |
| 2001Q3 | 0.046303 | 0.163344 | 0.251503 | -0.157276 |
| 2001Q4 | 0.046303 | 0.163344 | 0.251503 | -0.157276 |

In [233]: annual\_frame.resample('Q-DEC', convention='end').ffill() Out[233]:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Colorado | Texas | New York | Ohio |
| 2000Q4 | 0.556703 | 0.016631 | 0.111873 | -0.027445 |
| 2001Q1 | 0.556703 | 0.016631 | 0.111873 | -0.027445 |
| 2001Q2 | 0.556703 | 0.016631 | 0.111873 | -0.027445 |
| 2001Q3 | 0.556703 | 0.016631 | 0.111873 | -0.027445 |
| 2001Q4 | 0.046303 | 0.163344 | 0.251503 | -0.157276 |

Since periods refer to timespans, the rules about upsampling and downsampling are more rigid:

In downsampling, the target frequency must be a subperiod of the source frequency.

In upsampling, the target frequency must be a superperiod of the source frequency.

If these rules are not satisfied, an exception will be raised. This mainly affects the quarterly, annual, and weekly frequencies; for example, the timespans defined by Q- MAR only line up with A-MAR, A-JUN, A-SEP, and A-DEC:

In [234]: annual\_frame.resample('Q-MAR').ffill() Out[234]:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Colorado | Texas | New York | Ohio |
| 2000Q4 | 0.556703 | 0.016631 | 0.111873 | -0.027445 |
| 2001Q1 | 0.556703 | 0.016631 | 0.111873 | -0.027445 |
| 2001Q2 | 0.556703 | 0.016631 | 0.111873 | -0.027445 |
| 2001Q3 | 0.556703 | 0.016631 | 0.111873 | -0.027445 |
| 2001Q4 | 0.046303 | 0.163344 | 0.251503 | -0.157276 |
| 2002Q1 | 0.046303 | 0.163344 | 0.251503 | -0.157276 |
| 2002Q2 | 0.046303 | 0.163344 | 0.251503 | -0.157276 |
| 2002Q3 | 0.046303 | 0.163344 | 0.251503 | -0.157276 |

### Moving Window Functions

An important class of array transformations used for time series operations are statis‐ tics and other functions evaluated over a sliding window or with exponentially decay‐ ing weights. This can be useful for smoothing noisy or gappy data. I call these moving window functions, even though it includes functions without a fixed-length window

like exponentially weighted moving average. Like other statistical functions, these also automatically exclude missing data.

Before digging in, we can load up some time series data and resample it to business day frequency:

In [235]: close\_px\_all = pd.read\_csv('examples/stock\_px\_2.csv',

.....: parse\_dates=True, index\_col=0) In [236]: close\_px = close\_px\_all[['AAPL', 'MSFT', 'XOM']]

In [237]: close\_px = close\_px.resample('B').ffill()

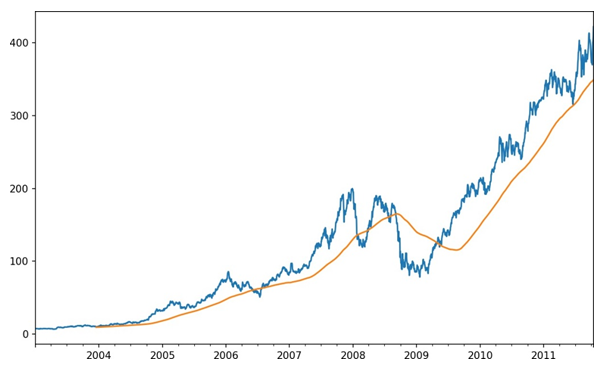
I now introduce the rolling operator, which behaves similarly to resample and groupby. It can be called on a Series or DataFrame along with a window (expressed as a number of periods; see [Figure 11-4](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark17) for the plot created):

In [238]: close\_px.AAPL.plot()

Out[238]: <matplotlib.axes.\_subplots.AxesSubplot at 0x7f2f2570cf98> In [239]: close\_px.AAPL.rolling(250).mean().plot()

Figure 11-4. Apple Price with 250-day MA

The expression rolling(250) is similar in behavior to groupby, but instead of group‐ ing it creates an object that enables grouping over a 250-day sliding window. So here we have the 250-day moving window average of Apple’s stock price.



By default rolling functions require all of the values in the window to be non-NA. This behavior can be changed to account for missing data and, in particular, the fact that you will have fewer than window periods of data at the beginning of the time series (see [Figure 11-5](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark18)):

In [241]: appl\_std250 = close\_px.AAPL.rolling(250, min\_periods=10).std()

In [242]: appl\_std250[5:12] Out[242]:

|  |  |
| --- | --- |
| 2003-01-09 | NaN |
| 2003-01-10 | NaN |
| 2003-01-13 | NaN |
| 2003-01-14 | NaN |
| 2003-01-15 | 0.077496 |
| 2003-01-16 | 0.074760 |
| 2003-01-17 | 0.112368 |

Freq: B, Name: AAPL, dtype: float64 In [243]: appl\_std250.plot()

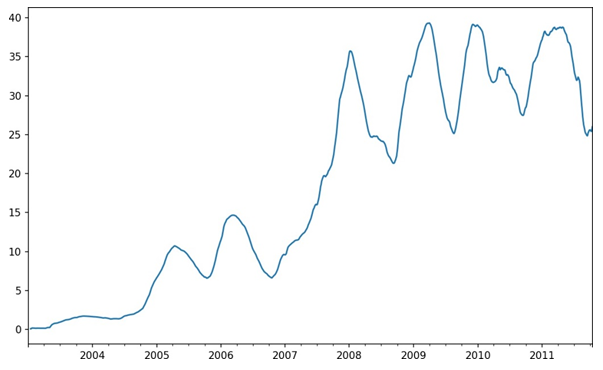


Figure 11-5. Apple 250-day daily return standard deviation

In order to compute an expanding window mean, use the expanding operator instead of rolling. The expanding mean starts the time window from the beginning of the time series and increases the size of the window until it encompasses the whole series. An expanding window mean on the apple\_std250 time series looks like this:

In [244]: expanding\_mean = appl\_std250.expanding().mean()

Calling a moving window function on a DataFrame applies the transformation to each column (see [Figure 11-6](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark19)):

In [246]: close\_px.rolling(60).mean().plot(logy=True)

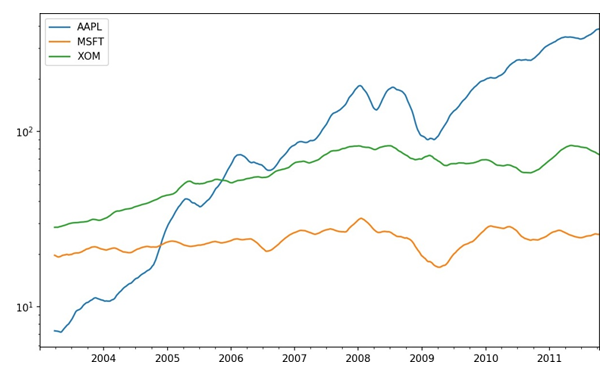


Figure 11-6. Stocks prices 60-day MA (log Y-axis)

The rolling function also accepts a string indicating a fixed-size time offset rather than a set number of periods. Using this notation can be useful for irregular time ser‐ ies. These are the same strings that you can pass to resample. For example, we could compute a 20-day rolling mean like so:

In [247]: close\_px.rolling('20D').mean() Out[247]:

|  |  |  |  |
| --- | --- | --- | --- |
|  | AAPL | MSFT | XOM |
| 2003-01-02 | 7.400000 | 21.110000 | 29.220000 |
| 2003-01-03 | 7.425000 | 21.125000 | 29.230000 |
| 2003-01-06 | 7.433333 | 21.256667 | 29.473333 |
| 2003-01-07 | 7.432500 | 21.425000 | 29.342500 |
| 2003-01-08 | 7.402000 | 21.402000 | 29.240000 |
| 2003-01-09 | 7.391667 | 21.490000 | 29.273333 |
| 2003-01-10 | 7.387143 | 21.558571 | 29.238571 |
| 2003-01-13 | 7.378750 | 21.633750 | 29.197500 |
| 2003-01-14 | 7.370000 | 21.717778 | 29.194444 |
| 2003-01-15 | 7.355000 | 21.757000 | 29.152000 |
| ... | ... | ... | ... |
| 2011-10-03 | 398.002143 | 25.890714 | 72.413571 |
| 2011-10-04 | 396.802143 | 25.807857 | 72.427143 |
| 2011-10-05 | 395.751429 | 25.729286 | 72.422857 |

|  |  |  |  |
| --- | --- | --- | --- |
| 2011-10-06 | 394.099286 | 25.673571 | 72.375714 |
| 2011-10-07 | 392.479333 | 25.712000 | 72.454667 |
| 2011-10-10 | 389.351429 | 25.602143 | 72.527857 |
| 2011-10-11 | 388.505000 | 25.674286 | 72.835000 |
| 2011-10-12 | 388.531429 | 25.810000 | 73.400714 |
| 2011-10-13 | 388.826429 | 25.961429 | 73.905000 |
| 2011-10-14 | 391.038000 | 26.048667 | 74.185333 |
| [2292 rows | x 3 columns] |  |  |

#### Exponentially Weighted Functions

An alternative to using a static window size with equally weighted observations is to specify a constant decay factor to give more weight to more recent observations. There are a couple of ways to specify the decay factor. A popular one is using a span, which makes the result comparable to a simple moving window function with win‐ dow size equal to the span.

Since an exponentially weighted statistic places more weight on more recent observa‐ tions, it “adapts” faster to changes compared with the equal-weighted version.

pandas has the ewm operator to go along with rolling and expanding. Here’s an example comparing a 60-day moving average of Apple’s stock price with an EW mov‐ ing average with span=60 (see [Figure 11-7](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark20)):

In [249]: aapl\_px = close\_px.AAPL['2006':'2007']

In [250]: ma60 = aapl\_px.rolling(30, min\_periods=20).mean() In [251]: ewma60 = aapl\_px.ewm(span=30).mean()

In [252]: ma60.plot(style='k--', label='Simple MA')

Out[252]: <matplotlib.axes.\_subplots.AxesSubplot at 0x7f2f252161d0>

In [253]: ewma60.plot(style='k-', label='EW MA')

Out[253]: <matplotlib.axes.\_subplots.AxesSubplot at 0x7f2f252161d0> In [254]: plt.legend()

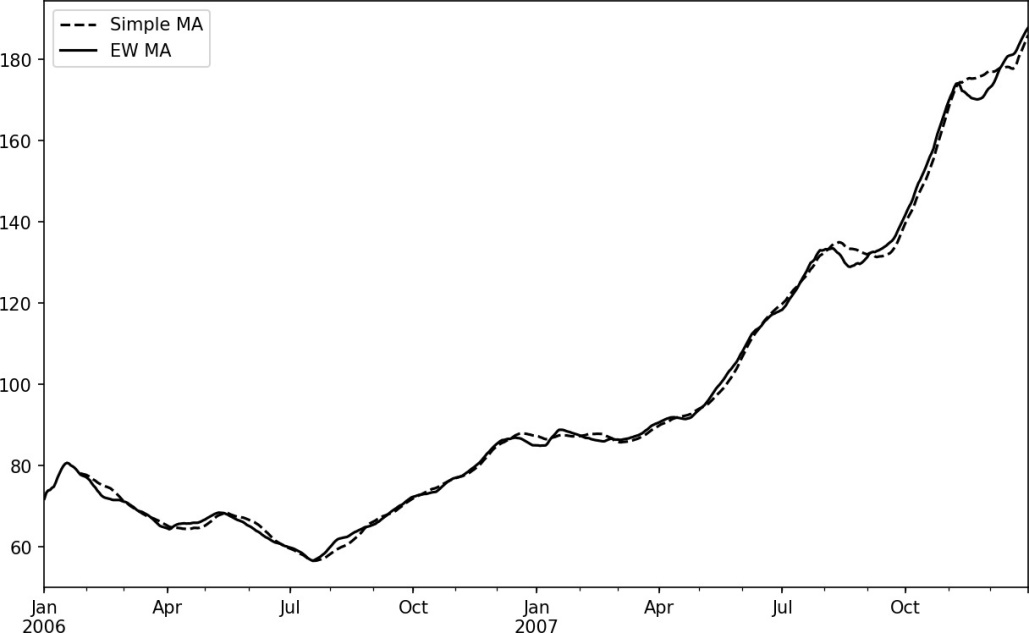


Figure 11-7. Simple moving average versus exponentially weighted

#### Binary Moving Window Functions

Some statistical operators, like correlation and covariance, need to operate on two time series. As an example, financial analysts are often interested in a stock’s correla‐ tion to a benchmark index like the S&P 500. To have a look at this, we first compute the percent change for all of our time series of interest:

In [256]: spx\_px = close\_px\_all['SPX'] In [257]: spx\_rets = spx\_px.pct\_change()

In [258]: returns = close\_px.pct\_change()

The corr aggregation function after we call rolling can then compute the rolling correlation with spx\_rets (see [Figure 11-8](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark21) for the resulting plot):

In [259]: corr = returns.AAPL.rolling(125, min\_periods=100).corr(spx\_rets) In [260]: corr.plot()

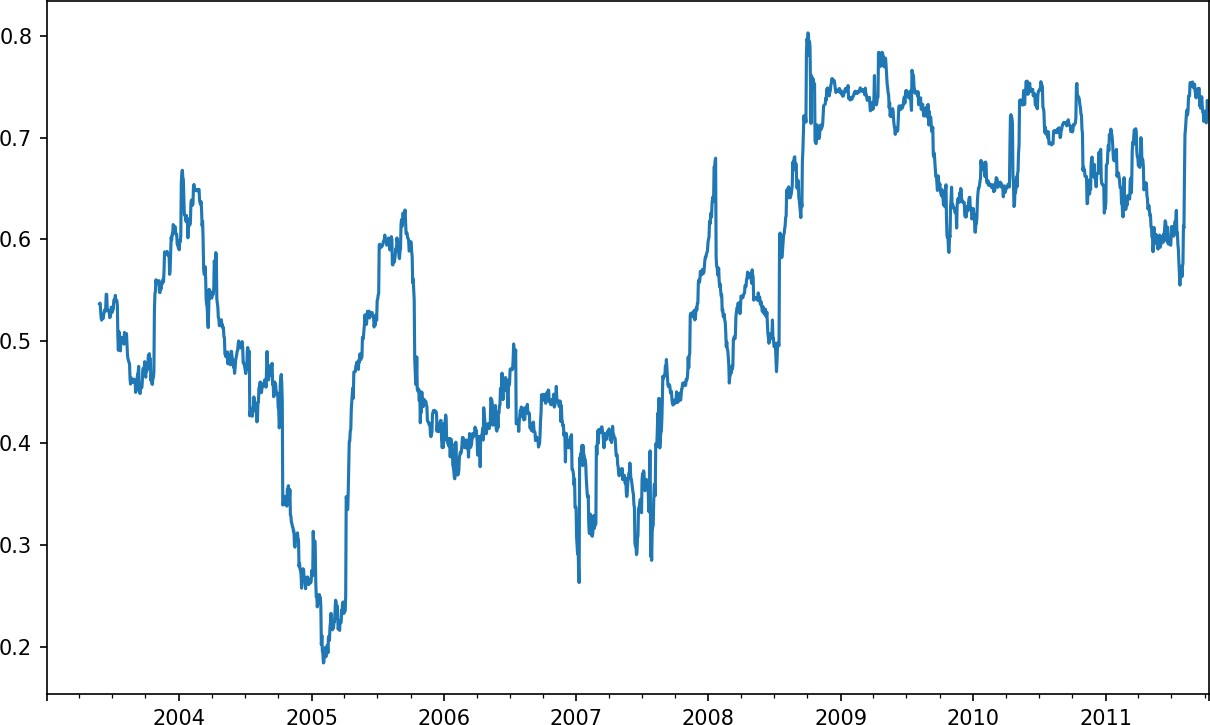


Figure 11-8. Six-month AAPL return correlation to S&P 500

Suppose you wanted to compute the correlation of the S&P 500 index with many stocks at once. Writing a loop and creating a new DataFrame would be easy but might get repetitive, so if you pass a Series and a DataFrame, a function like rolling\_corr will compute the correlation of the Series (spx\_rets, in this case) with each column in the DataFrame (see [Figure 11-9](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark22) for the plot of the result):

In [262]: corr = returns.rolling(125, min\_periods=100).corr(spx\_rets) In [263]: corr.plot()



*Figure 11-9. Six-month return correlations to S&P 500*

#### User-Defined Moving Window Functions

The apply method on rolling and related methods provides a means to apply an array function of your own devising over a moving window. The only requirement is that the function produce a single value (a reduction) from each piece of the array. For example, while we can compute sample quantiles using rolling(...).quan tile(q), we might be interested in the percentile rank of a particular value over the sample. The scipy.stats.percentileofscore function does just this (see [Figure 11-10](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark23) for the resulting plot):

In [265]: from scipy.stats import percentileofscore

In [266]: score\_at\_2percent = lambda x: percentileofscore(x, 0.02)

In [267]: result = returns.AAPL.rolling(250).apply(score\_at\_2percent) In [268]: result.plot()

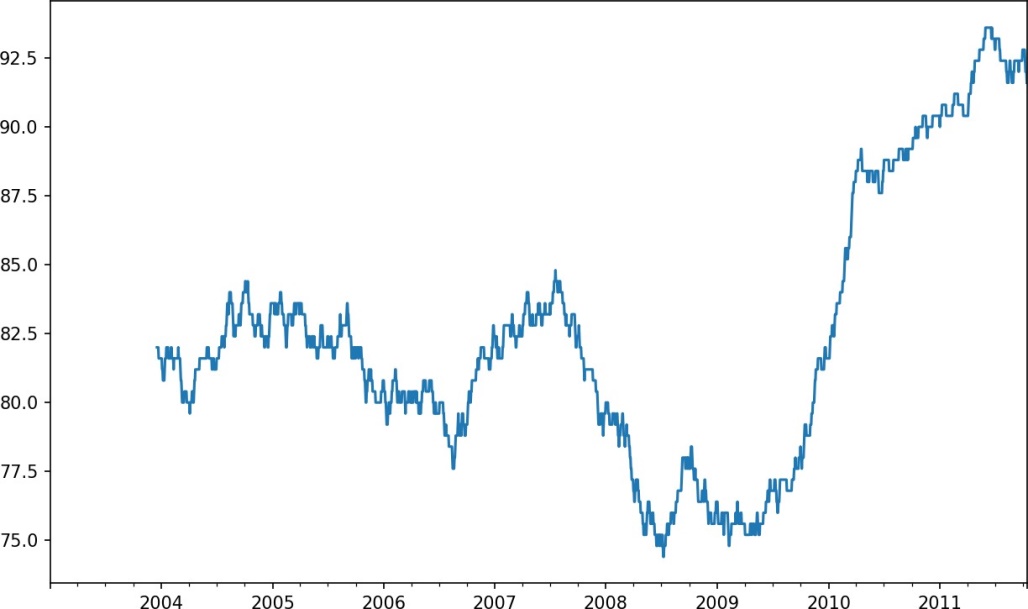


Figure 11-10. Percentile rank of 2% AAPL return over one-year window

If you don’t have SciPy installed already, you can install it with conda or pip.

### Conclusion

Time series data calls for different types of analysis and data transformation tools than the other types of data we have explored in previous chapters.

In the following chapters, we will move on to some advanced pandas methods and show how to start using modeling libraries like statsmodels and scikit-learn.

Advanced pandas

The preceding chapters have focused on introducing different types of data wrangling workflows and features of NumPy, pandas, and other libraries. Over time, pandas has developed a depth of features for power users. This chapter digs into a few more advanced feature areas to help you deepen your expertise as a pandas user.

#### Categorical Data

This section introduces the pandas Categorical type. I will show how you can ach‐ ieve better performance and memory use in some pandas operations by using it. I also introduce some tools for using categorical data in statistics and machine learning applications.

##### Background and Motivation

Frequently, a column in a table may contain repeated instances of a smaller set of dis‐ tinct values. We have already seen functions like unique and value\_counts, which enable us to extract the distinct values from an array and compute their frequencies, respectively:

import numpy as np; import pandas as pd

In [11]: values = pd.Series(['apple', 'orange', 'apple',

....: 'apple'] \* 2)

values Out[12]:

apple

orange

apple

apple

apple

orange

apple

apple dtype: object

pd.unique(values)

Out[13]: array(['apple', 'orange'], dtype=object)

pd.value\_counts(values) Out[14]:

apple 6

orange 2

dtype: int64

Many data systems (for data warehousing, statistical computing, or other uses) have developed specialized approaches for representing data with repeated values for more efficient storage and computation. In data warehousing, a best practice is to use so- called dimension tables containing the distinct values and storing the primary obser‐ vations as integer keys referencing the dimension table:

values = pd.Series([0, 1, 0, 0] \* 2)

dim = pd.Series(['apple', 'orange']) values

Out[17]:

0 0

1 1

2 0

3 0

4 0

5 1

6 0

7 0

dtype: int64

In [18]: dim Out[18]:

apple

orange dtype: object

We can use the take method to restore the original Series of strings:

In [19]: dim.take(values) Out[19]:

apple

orange

0 apple

0 apple

apple

orange

0 apple

0 apple dtype: object

This representation as integers is called the categorical or dictionary-encoded repre‐ sentation. The array of distinct values can be called the categories, dictionary, or levels of the data. In this book we will use the terms categorical and categories. The integer values that reference the categories are called the category codes or simply codes.

The categorical representation can yield significant performance improvements when you are doing analytics. You can also perform transformations on the categories while leaving the codes unmodified. Some example transformations that can be made at relatively low cost are:

Renaming categories

Appending a new category without changing the order or position of the existing categories

##### Categorical Type in pandas

pandas has a special Categorical type for holding data that uses the integer-based categorical representation or encoding. Let’s consider the example Series from before:

In [20]: fruits = ['apple', 'orange', 'apple', 'apple'] \* 2 In [21]: N = len(fruits)

In [22]: df = pd.DataFrame({'fruit': fruits,

....: 'basket\_id': np.arange(N),

....: 'count': np.random.randint(3, 15, size=N),

....: 'weight': np.random.uniform(0, 4, size=N)},

....: columns=['basket\_id', 'fruit', 'count', 'weight'])

In [23]: df

Out[23]:

basket\_id fruit count weight 0 0 apple 5 3.858058

1 1 orange 8 2.612708

2 2 apple 4 2.995627

3 3 apple 7 2.614279

4 4 apple 12 2.990859

5 5 orange 8 3.845227

6 6 apple 5 0.033553

7 7 apple 4 0.425778

Here, df['fruit'] is an array of Python string objects. We can convert it to categori‐ cal by calling:

In [24]: fruit\_cat = df['fruit'].astype('category')

In [25]: fruit\_cat Out[25]:

apple

orange

apple

apple

apple

orange

apple

apple

Name: fruit, dtype: category

Categories (2, object): [apple, orange]

The values for fruit\_cat are not a NumPy array, but an instance of pandas.Catego rical:

In [26]: c = fruit\_cat.values

In [27]: type(c)

Out[27]: pandas.core.categorical.Categorical

The Categorical object has categories and codes attributes:

In [28]: c.categories

Out[28]: Index(['apple', 'orange'], dtype='object')

In [29]: c.codes

Out[29]: array([0, 1, 0, 0, 0, 1, 0, 0], dtype=int8)

You can convert a DataFrame column to categorical by assigning the converted result:

In [30]: df['fruit'] = df['fruit'].astype('category')

In [31]: df.fruit Out[31]:

apple

orange

apple

apple

apple

orange

apple

apple

Name: fruit, dtype: category

Categories (2, object): [apple, orange]

You can also create pandas.Categorical directly from other types of Python sequences:

In [32]: my\_categories = pd.Categorical(['foo', 'bar', 'baz', 'foo', 'bar']) In [33]: my\_categories

Out[33]:

[foo, bar, baz, foo, bar]

Categories (3, object): [bar, baz, foo]

If you have obtained categorical encoded data from another source, you can use the alternative from\_codes constructor:

In [34]: categories = ['foo', 'bar', 'baz']

In [35]: codes = [0, 1, 2, 0, 0, 1]

In [36]: my\_cats\_2 = pd.Categorical.from\_codes(codes, categories) In [37]: my\_cats\_2

Out[37]:

[foo, bar, baz, foo, foo, bar] Categories (3, object): [foo, bar, baz]

Unless explicitly specified, categorical conversions assume no specific ordering of the categories. So the categories array may be in a different order depending on the ordering of the input data. When using from\_codes or any of the other constructors, you can indicate that the categories have a meaningful ordering:

In [38]: ordered\_cat = pd.Categorical.from\_codes(codes, categories,

....: ordered=True)

In [39]: ordered\_cat Out[39]:

[foo, bar, baz, foo, foo, bar]

Categories (3, object): [foo < bar < baz]

The output [foo < bar < baz] indicates that 'foo' precedes 'bar' in the ordering, and so on. An unordered categorical instance can be made ordered with as\_ordered:

In [40]: my\_cats\_2.as\_ordered() Out[40]:

[foo, bar, baz, foo, foo, bar]

Categories (3, object): [foo < bar < baz]

As a last note, categorical data need not be strings, even though I have only showed string examples. A categorical array can consist of any immutable value types.

#### Computations with Categoricals

Using Categorical in pandas compared with the non-encoded version (like an array of strings) generally behaves the same way. Some parts of pandas, like the groupby function, perform better when working with categoricals. There are also some func‐ tions that can utilize the ordered flag.

Let’s consider some random numeric data, and use the pandas.qcut binning func‐ tion. This return pandas.Categorical; we used pandas.cut earlier in the book but glossed over the details of how categoricals work:

In [41]: np.random.seed(12345)

In [42]: draws = np.random.randn(1000) In [43]: draws[:5]

Out[43]: array([-0.2047, 0.4789, -0.5194, -0.5557, 1.9658])

Let’s compute a quartile binning of this data and extract some statistics:

In [44]: bins = pd.qcut(draws, 4)

In [45]: bins Out[45]:

[(-0.684, -0.0101], (-0.0101, 0.63], (-0.684, -0.0101], (-0.684, -0.0101], (0.63,

3.928], ..., (-0.0101, 0.63], (-0.684, -0.0101], (-2.95, -0.684], (-0.0101, 0.63

], (0.63, 3.928]]

Length: 1000

Categories (4, interval[float64]): [(-2.95, -0.684] < (-0.684, -0.0101] < (-0.010

1, 0.63] <

(0.63, 3.928]]

While useful, the exact sample quartiles may be less useful for producing a report than quartile names. We can achieve this with the labels argument to qcut:

In [46]: bins = pd.qcut(draws, 4, labels=['Q1', 'Q2', 'Q3', 'Q4'])

In [47]: bins Out[47]:

[Q2, Q3, Q2, Q2, Q4, ..., Q3, Q2, Q1, Q3, Q4]

Length: 1000

Categories (4, object): [Q1 < Q2 < Q3 < Q4]

In [48]: bins.codes[:10]

Out[48]: array([1, 2, 1, 1, 3, 3, 2, 2, 3, 3], dtype=int8)

The labeled bins categorical does not contain information about the bin edges in the data, so we can use groupby to extract some summary statistics:

In [49]: bins = pd.Series(bins, name='quartile')

In [50]: results = (pd.Series(draws)

....: .groupby(bins)

....: .agg(['count', 'min', 'max'])

....: .reset\_index())

In [51]: results Out[51]:

quartile count min max 0 Q1 250 -2.949343 -0.685484

|  |  |  |
| --- | --- | --- |
| 1 | Q2 | 250 -0.683066 -0.010115 |
| 2 | Q3 | 250 -0.010032 0.628894 |
| 3 | Q4 | 250 0.634238 3.927528 |

The 'quartile' column in the result retains the original categorical information, including ordering, from bins:

In [52]: results['quartile'] Out[52]:

Q1

Q2

Q3

Q4

Name: quartile, dtype: category

Categories (4, object): [Q1 < Q2 < Q3 < Q4]

##### Better performance with categoricals

If you do a lot of analytics on a particular dataset, converting to categorical can yield substantial overall performance gains. A categorical version of a DataFrame column will often use significantly less memory, too. Let’s consider some Series with 10 mil‐ lion elements and a small number of distinct categories:

In [53]: N = 10000000

In [54]: draws = pd.Series(np.random.randn(N))

In [55]: labels = pd.Series(['foo', 'bar', 'baz', 'qux'] \* (N // 4))

Now we convert labels to categorical:

In [56]: categories = labels.astype('category')

Now we note that labels uses significantly more memory than categories:

In [57]: labels.memory\_usage()

Out[57]: 80000080

In [58]: categories.memory\_usage()

Out[58]: 10000272

The conversion to category is not free, of course, but it is a one-time cost:

In [59]: %time \_ = labels.astype('category')

CPU times: user 490 ms, sys: 240 ms, total: 730 ms Wall time: 726 ms

GroupBy operations can be significantly faster with categoricals because the underly‐ ing algorithms use the integer-based codes array instead of an array of strings.

Categorical Methods

Series containing categorical data have several special methods similar to the Ser ies.str specialized string methods. This also provides convenient access to the cate‐ gories and codes. Consider the Series:

In [60]: s = pd.Series(['a', 'b', 'c', 'd'] \* 2) In [61]: cat\_s = s.astype('category')

In [62]: cat\_s Out[62]:

a

b

c

d

a

b

c

d

dtype: category

Categories (4, object): [a, b, c, d]

The special attribute cat provides access to categorical methods:

In [63]: cat\_s.cat.codes Out[63]:

0 0

1 1

2 2

3 3

4 0

5 1

6 2

7 3

dtype: int8

In [64]: cat\_s.cat.categories

Out[64]: Index(['a', 'b', 'c', 'd'], dtype='object')

Suppose that we know the actual set of categories for this data extends beyond the four values observed in the data. We can use the set\_categories method to change them:

In [65]: actual\_categories = ['a', 'b', 'c', 'd', 'e']

In [66]: cat\_s2 = cat\_s.cat.set\_categories(actual\_categories) In [67]: cat\_s2

Out[67]:

a

b

c

d

a

b

c

d

dtype: category

Categories (5, object): [a, b, c, d, e]

While it appears that the data is unchanged, the new categories will be reflected in operations that use them. For example, value\_counts respects the categories, if present:

In [68]: cat\_s.value\_counts() Out[68]:

d 2

c 2

b 2

a 2

dtype: int64

In [69]: cat\_s2.value\_counts() Out[69]:

|  |  |
| --- | --- |
| d | 2 |
| c | 2 |
| b | 2 |
| a | 2 |
| e | 0 |

dtype: int64

In large datasets, categoricals are often used as a convenient tool for memory savings and better performance. After you filter a large DataFrame or Series, many of the categories may not appear in the data. To help with this, we can use the remove\_unused\_categories method to trim unobserved categories:

In [70]: cat\_s3 = cat\_s[cat\_s.isin(['a', 'b'])]

In [71]: cat\_s3 Out[71]:

a

b

a

b

dtype: category

Categories (4, object): [a, b, c, d]

In [72]: cat\_s3.cat.remove\_unused\_categories() Out[72]:

a

b

a

b

dtype: category

Categories (2, object): [a, b]

See [Table 12-1](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark25) for a listing of available categorical methods.

Table 12-1. Categorical methods for Series in pandas

**Method Description**

add\_categories Append new (unused) categories at end of existing categories

as\_ordered Make categories ordered

as\_unordered Make categories unordered

remove\_categories Remove categories, setting any removed values to null remove\_unused\_categories Remove any category values which do not appear in the data rename\_categories Replace categories with indicated set of new category names; cannot change the

number of categories

reorder\_categories Behaves like rename\_categories, but can also change the result to have ordered

categories

set\_categories Replace the categories with the indicated set of new categories; can add or remove

categories

##### Creating dummy variables for modeling

When you’re using statistics or machine learning tools, you’ll often transform catego‐ rical data into dummy variables, also known as one-hot encoding. This involves creat‐ ing a DataFrame with a column for each distinct category; these columns contain 1s for occurrences of a given category and 0 otherwise.

Consider the previous example:

In [73]: cat\_s = pd.Series(['a', 'b', 'c', 'd'] \* 2, dtype='category')

As mentioned previously in Chapter 7, the pandas.get\_dummies function converts this one-dimensional categorical data into a DataFrame containing the dummy variable:

In [74]: pd.get\_dummies(cat\_s) Out[74]:

|  |  |  |  |
| --- | --- | --- | --- |
| a | b | c | d |
| 0 1 | 0 | 0 | 0 |
| 1 0 | 1 | 0 | 0 |
| 2 0 | 0 | 1 | 0 |
| 3 0 | 0 | 0 | 1 |
| 4 1 | 0 | 0 | 0 |
| 5 0 | 1 | 0 | 0 |
| 6 0 | 0 | 1 | 0 |
| 7 0 | 0 | 0 | 1 |

#### Advanced GroupBy Use

While we’ve already discussed using the groupby method for Series and DataFrame in depth in [Chapter 10](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark0), there are some additional techniques that you may find of use.

##### Group Transforms and “Unwrapped” GroupBys

In [Chapter 10](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark0) we looked at the apply method in grouped operations for performing transformations. There is another built-in method called transform, which is similar to apply but imposes more constraints on the kind of function you can use:

It can produce a scalar value to be broadcast to the shape of the group

It can produce an object of the same shape as the input group

It must not mutate its input

Let’s consider a simple example for illustration:

In [75]: df = pd.DataFrame({'key': ['a', 'b', 'c'] \* 4,

....: 'value': np.arange(12.)})

In [76]: df

Out[76]:

key value 0 a 0.0

1 b 1.0

2 c 2.0

3 a 3.0

4 b 4.0

5 c 5.0

6 a 6.0

7 b 7.0

8 c 8.0

9 a 9.0

10 b 10.0

11 c 11.0

Here are the group means by key:

In [77]: g = df.groupby('key').value

In [78]: g.mean() Out[78]:

key

a 4.5

b 5.5

c 6.5

Name: value, dtype: float64

Suppose instead we wanted to produce a Series of the same shape as df['value'] but with values replaced by the average grouped by 'key'. We can pass the function lambda x: x.mean() to transform:

In [79]: g.transform(**lambda** x: x.mean()) Out[79]:

|  |  |
| --- | --- |
| 0 | 4.5 |
| 1 | 5.5 |
| 2 | 6.5 |
| 3 | 4.5 |
| 4 | 5.5 |
| 5 | 6.5 |
| 6 | 4.5 |
| 7 | 5.5 |
| 8 | 6.5 |
| 9 | 4.5 |
| 10 | 5.5 |
| 11 | 6.5 |
| Name: | value, dtype: float64 |

For built-in aggregation functions, we can pass a string alias as with the GroupBy agg

method:

In [80]: g.transform('mean') Out[80]:

|  |  |
| --- | --- |
| 0 | 4.5 |
| 1 | 5.5 |
| 2 | 6.5 |
| 3 | 4.5 |
| 4 | 5.5 |
| 5 | 6.5 |
| 6 | 4.5 |
| 7 | 5.5 |
| 8 | 6.5 |
| 9 | 4.5 |
| 10 | 5.5 |
| 11 | 6.5 |
| Name: | value, dtype: float64 |

Like apply, transform works with functions that return Series, but the result must be the same size as the input. For example, we can multiply each group by 2 using a lambda function:

In [81]: g.transform(lambda x: x \* 2) Out[81]:

|  |  |
| --- | --- |
| 0 | 0.0 |
| 1 | 2.0 |
| 2 | 4.0 |
| 3 | 6.0 |
| 4 | 8.0 |
| 5 | 10.0 |
| 6 | 12.0 |

7 14.0

8 16.0

9 18.0

10 20.0

11 22.0

Name: value, dtype: float64

As a more complicated example, we can compute the ranks in descending order for each group:

In [82]: g.transform(**lambda** x: x.rank(ascending=False)) Out[82]:

|  |  |
| --- | --- |
| 0 | 4.0 |
| 1 | 4.0 |
| 2 | 4.0 |
| 3 | 3.0 |
| 4 | 3.0 |
| 5 | 3.0 |
| 6 | 2.0 |
| 7 | 2.0 |
| 8 | 2.0 |
| 9 | 1.0 |
| 10 | 1.0 |
| 11 | 1.0 |
| Name: | value, dtype: float64 |

Consider a group transformation function composed from simple aggregations:

**def** normalize(x):

**return** (x - x.mean()) / x.std()

We can obtain equivalent results in this case either using transform or apply:

In [84]: g.transform(normalize) Out[84]:

0 -1.161895

1 -1.161895

2 -1.161895

3 -0.387298

4 -0.387298

5 -0.387298

6 0.387298

7 0.387298

8 0.387298

9 1.161895

10 1.161895

11 1.161895

Name: value, dtype: float64

In [85]: g.apply(normalize) Out[85]:

0 -1.161895

1 -1.161895

2 -1.161895

3 -0.387298

4 -0.387298

5 -0.387298

6 0.387298

7 0.387298

8 0.387298

9 1.161895

10 1.161895

11 1.161895

Name: value, dtype: float64

Built-in aggregate functions like 'mean' or 'sum' are often much faster than a general apply function. These also have a “fast past” when used with transform. This allows us to perform a so-called unwrapped group operation:

In [86]: g.transform('mean') Out[86]:

|  |  |
| --- | --- |
| 0 | 4.5 |
| 1 | 5.5 |
| 2 | 6.5 |
| 3 | 4.5 |
| 4 | 5.5 |
| 5 | 6.5 |
| 6 | 4.5 |
| 7 | 5.5 |
| 8 | 6.5 |
| 9 | 4.5 |
| 10 | 5.5 |
| 11 | 6.5 |
| Name: | value, dtype: float64 |

In [87]: normalized = (df['value'] - g.transform('mean')) / g.transform('std') In [88]: normalized

Out[88]:

0 -1.161895

1 -1.161895

2 -1.161895

3 -0.387298

4 -0.387298

5 -0.387298

6 0.387298

7 0.387298

8 0.387298

9 1.161895

10 1.161895

11 1.161895

Name: value, dtype: float64

While an unwrapped group operation may involve multiple group aggregations, the overall benefit of vectorized operations often outweighs this.

Grouped Time Resampling

For time series data, the resample method is semantically a group operation based on a time intervalization. Here’s a small example table:

In [89]: N = 15

In [90]: times = pd.date\_range('2017-05-20 00:00', freq='1min', periods=N) In [91]: df = pd.DataFrame({'time': times,

....: 'value': np.arange(N)})

In [92]: df

Out[92]:

|  |  |  |  |
| --- | --- | --- | --- |
|  | | time | value |
| 0 | 2017-05-20 | 00:00:00 | 0 |
| 1 | 2017-05-20 | 00:01:00 | 1 |
| 2 | 2017-05-20 | 00:02:00 | 2 |
| 3 | 2017-05-20 | 00:03:00 | 3 |
| 4 | 2017-05-20 | 00:04:00 | 4 |
| 5 | 2017-05-20 | 00:05:00 | 5 |
| 6 | 2017-05-20 | 00:06:00 | 6 |
| 7 | 2017-05-20 | 00:07:00 | 7 |
| 8 | 2017-05-20 | 00:08:00 | 8 |
| 9 | 2017-05-20 | 00:09:00 | 9 |
| 10 | 2017-05-20 | 00:10:00 | 10 |
| 11 | 2017-05-20 | 00:11:00 | 11 |
| 12 | 2017-05-20 | 00:12:00 | 12 |
| 13 | 2017-05-20 | 00:13:00 | 13 |
| 14 | 2017-05-20 | 00:14:00 | 14 |

Here, we can index by 'time' and then resample:

In [93]: df.set\_index('time').resample('5min').count() Out[93]:

value

time

2017-05-20 00:00:00 5

2017-05-20 00:05:00 5

2017-05-20 00:10:00 5

Suppose that a DataFrame contains multiple time series, marked by an additional group key column:

In [94]: df2 = pd.DataFrame({'time': times.repeat(3),

....: 'key': np.tile(['a', 'b', 'c'], N),

....: 'value': np.arange(N \* 3.)})

In [95]: df2[:7] Out[95]:

key time value

0 a 2017-05-20 00:00:00 0.0

1 b 2017-05-20 00:00:00 1.0

2 c 2017-05-20 00:00:00 2.0

3 a 2017-05-20 00:01:00 3.0

4 b 2017-05-20 00:01:00 4.0

5 c 2017-05-20 00:01:00 5.0

6 a 2017-05-20 00:02:00 6.0

To do the same resampling for each value of 'key', we introduce the pandas.Time Grouper object:

In [96]: time\_key = pd.TimeGrouper('5min')

We can then set the time index, group by 'key' and time\_key, and aggregate:

In [97]: resampled = (df2.set\_index('time')

....: .groupby(['key', time\_key])

....: .sum())

In [98]: resampled Out[98]:

key time

value

a 2017-05-20 00:00:00 30.0

2017-05-20 00:05:00 105.0

2017-05-20 00:10:00 180.0

b 2017-05-20 00:00:00 35.0

2017-05-20 00:05:00 110.0

2017-05-20 00:10:00 185.0

c 2017-05-20 00:00:00 40.0

2017-05-20 00:05:00 115.0

2017-05-20 00:10:00 190.0

In [99]: resampled.reset\_index() Out[99]:

|  |  |  |
| --- | --- | --- |
| key | time | value |
| 0 a | 2017-05-20 00:00:00 | 30.0 |
| 1 a | 2017-05-20 00:05:00 | 105.0 |
| 2 a | 2017-05-20 00:10:00 | 180.0 |
| 3 b | 2017-05-20 00:00:00 | 35.0 |
| 4 b | 2017-05-20 00:05:00 | 110.0 |
| 5 b | 2017-05-20 00:10:00 | 185.0 |
| 6 c | 2017-05-20 00:00:00 | 40.0 |
| 7 c | 2017-05-20 00:05:00 | 115.0 |
| 8 c | 2017-05-20 00:10:00 | 190.0 |

One constraint with using TimeGrouper is that the time must be the index of the Ser‐ ies or DataFrame.

#### Techniques for Method Chaining

When applying a sequence of transformations to a dataset, you may find yourself cre‐ ating numerous temporary variables that are never used in your analysis. Consider this example, for instance:

df = load\_data()

df2 = df[df['col2'] < 0]

df2['col1\_demeaned'] = df2['col1'] - df2['col1'].mean() result = df2.groupby('key').col1\_demeaned.std()

While we’re not using any real data here, this example highlights some new methods. First, the DataFrame.assign method is a functional alternative to column assign‐ ments of the form df[k] = v. Rather than modifying the object in-place, it returns a new DataFrame with the indicated modifications. So these statements are equivalent:

# Usual non-functional way

df2 = df.copy() df2['k'] = v

# Functional assign way

df2 = df.assign(k=v)

Assigning in-place may execute faster than using assign, but assign enables easier method chaining:

result = (df2.assign(col1\_demeaned=df2.col1 - df2.col2.mean())

.groupby('key')

.col1\_demeaned.std())

I used the outer parentheses to make it more convenient to add line breaks.

One thing to keep in mind when doing method chaining is that you may need to refer to temporary objects. In the preceding example, we cannot refer to the result of load\_data until it has been assigned to the temporary variable df. To help with this, assign and many other pandas functions accept function-like arguments, also known as callables.

To show callables in action, consider a fragment of the example from before:

df = load\_data()

df2 = df[df['col2'] < 0]

This can be rewritten as:

df = (load\_data()

[lambda x: x['col2'] < 0])

Here, the result of load\_data is not assigned to a variable, so the function passed into

[] is then bound to the object at that stage of the method chain.

We can continue, then, and write the entire sequence as a single chained expression:

result = (load\_data()

[lambda x: x.col2 < 0]

.assign(col1\_demeaned=lambda x: x.col1 - x.col1.mean())

.groupby('key')

.col1\_demeaned.std())

Whether you prefer to write code in this style is a matter of taste, and splitting up the expression into multiple steps may make your code more readable.

##### The pipe Method

You can accomplish a lot with built-in pandas functions and the approaches to method chaining with callables that we just looked at. However, sometimes you need to use your own functions or functions from third-party libraries. This is where the pipe method comes in.

Consider a sequence of function calls:

a = f(df, arg1=v1)

b = g(a, v2, arg3=v3) c = h(b, arg4=v4)

When using functions that accept and return Series or DataFrame objects, you can rewrite this using calls to pipe:

result = (df.pipe(f, arg1=v1)

.pipe(g, v2, arg3=v3)

.pipe(h, arg4=v4))

The statement f(df) and df.pipe(f) are equivalent, but pipe makes chained invoca‐ tion easier.

A potentially useful pattern for pipe is to generalize sequences of operations into reusable functions. As an example, let’s consider substracting group means from a column:

g = df.groupby(['key1', 'key2'])

df['col1'] = df['col1'] - g.transform('mean')

Suppose that you wanted to be able to demean more than one column and easily change the group keys. Additionally, you might want to perform this transformation in a method chain. Here is an example implementation:

def group\_demean(df, by, cols): result = df.copy()

g = df.groupby(by)

for c in cols:

result[c] = df[c] - g[c].transform('mean')

return result

Then it is possible to write:

result = (df[df.col1 < 0]

.pipe(group\_demean, ['key1', 'key2'], ['col1']))

### Conclusion

pandas, like many open source software projects, is still changing and acquiring new and improved functionality. As elsewhere in this book, the focus here has been on the most stable functionality that is less likely to change over the next several years.

To deepen your expertise as a pandas user, I encourage you to explore the [documen‐](http://pandas.pydata.org/) [tation](http://pandas.pydata.org/) and read the release notes as the development team makes new open source releases. We also invite you to join in on pandas development: fixing bugs, building new features, and improving the documentation.

### Introduction to Modeling Libraries in Python

In this book, I have focused on providing a programming foundation for doing data analysis in Python. Since data analysts and scientists often report spending a dispro‐ portionate amount of time with data wrangling and preparation, the book’s structure reflects the importance of mastering these techniques.

Which library you use for developing models will depend on the application. Many statistical problems can be solved by simpler techniques like ordinary least squares regression, while other problems may call for more advanced machine learning methods. Fortunately, Python has become one of the languages of choice for imple‐ menting analytical methods, so there are many tools you can explore after completing this book.

In this chapter, I will review some features of pandas that may be helpful when you’re crossing back and forth between data wrangling with pandas and model fitting and scoring. I will then give short introductions to two popular modeling toolkits, [stats‐](http://statsmodels.org/) [models](http://statsmodels.org/) and [scikit-learn](http://scikit-learn.org/). Since each of these projects is large enough to warrant its own dedicated book, I make no effort to be comprehensive and instead direct you to both projects’ online documentation along with some other Python-based books on data science, statistics, and machine learning.

##### Interfacing Between pandas and Model Code

A common workflow for model development is to use pandas for data loading and cleaning before switching over to a modeling library to build the model itself. An important part of the model development process is called feature engineering in machine learning. This can describe any data transformation or analytics that extract

information from a raw dataset that may be useful in a modeling context. The data aggregation and GroupBy tools we have explored in this book are used often in a fea‐ ture engineering context.

While details of “good” feature engineering are out of scope for this book, I will show some methods to make switching between data manipulation with pandas and mod‐ eling as painless as possible.

The point of contact between pandas and other analysis libraries is usually NumPy arrays. To turn a DataFrame into a NumPy array, use the .values property:

import pandas as pd

In [11]: import numpy as np

data = pd.DataFrame({

....: 'x0': [1, 2, 3, 4, 5],

....: 'x1': [0.01, -0.01, 0.25, -4.1, 0.],

....: 'y': [-1.5, 0., 3.6, 1.3, -2.]})

data Out[13]:

x0 x1 y 0 1 0.01 -1.5

1 2 -0.01 0.0

2 3 0.25 3.6

3 4 -4.10 1.3

4 5 0.00 -2.0

data.columns

Out[14]: Index(['x0', 'x1', 'y'], dtype='object')

data.values Out[15]:

|  |  |  |
| --- | --- | --- |
| array([[ 1. | , 0.01, | -1.5 ], |
| [ 2. | , -0.01, | 0. ], |
| [ 3. | , 0.25, | 3.6 ], |
| [ 4. | , -4.1 , | 1.3 ], |
| [ 5. | , 0. , | -2. ]]) |

To convert back to a DataFrame, as you may recall from earlier chapters, you can pass a two-dimensional ndarray with optional column names:

df2 = pd.DataFrame(data.values, columns=['one', 'two', 'three'])

df2 Out[17]:

|  |  |
| --- | --- |
| one two | three |
| 0 1.0 0.01 | -1.5 |
| 1 2.0 -0.01 | 0.0 |
| 2 3.0 0.25 | 3.6 |

3 4.0 -4.10 1.3

4 5.0 0.00 -2.0

The .values attribute is intended to be used when your data is homogeneous—for example, all numeric types. If you have hetero‐ geneous data, the result will be an ndarray of Python objects:

In [18]: df3 = data.copy()

In [19]: df3['strings'] = ['a', 'b', 'c', 'd', 'e'] In [20]: df3

Out[20]:

x0 x1 y strings 0 1 0.01 -1.5 a

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 -0.01 | 0.0 | b |
| 2 | 3 0.25 | 3.6 | c |
| 3 | 4 -4.10 | 1.3 | d |
| 4 | 5 0.00 | -2.0 | e |

In [21]: df3.values Out[21]:

array([[1, 0.01, -1.5, 'a'],

[2, -0.01, 0.0, 'b'],

[3, 0.25, 3.6, 'c'],

[4, -4.1, 1.3, 'd'],

[5, 0.0, -2.0, 'e']], dtype=object)

For some models, you may only wish to use a subset of the columns. I recommend using loc indexing with values:

In [22]: model\_cols = ['x0', 'x1']

In [23]: data.loc[:, model\_cols].values Out[23]:

|  |  |
| --- | --- |
| array([[ 1. | , 0.01], |
| [ 2. | , -0.01], |
| [ 3. | , 0.25], |
| [ 4. | , -4.1 ], |
| [ 5. | , 0. ]]) |

Some libraries have native support for pandas and do some of this work for you auto‐ matically: converting to NumPy from DataFrame and attaching model parameter names to the columns of output tables or Series. In other cases, you will have to per‐ form this “metadata management” manually.

In [Chapter 12](file:///C:\Users\Dimpu\AppData\Roaming\Microsoft\Downloads\Wes%20McKinney%20-%20Python%20for%20Data%20Analysis.%20Data%20Wrangling%20with%20Pandas,%20NumPy,%20and%20IPython%20(2017,%20O’Reilly)%20-%20libgen.lc%20(2).docx#_bookmark24) we looked at pandas’s Categorical type and the pandas.get\_dummies

function. Suppose we had a non-numeric column in our example dataset:

In [24]: data['category'] = pd.Categorical(['a', 'b', 'a', 'a', 'b'],

....: categories=['a', 'b'])

In [25]: data Out[25]:

x0 x1 y category 0 1 0.01 -1.5 a

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 -0.01 | 0.0 | b |
| 2 | 3 0.25 | 3.6 | a |
| 3 | 4 -4.10 | 1.3 | a |
| 4 | 5 0.00 | -2.0 | b |

If we wanted to replace the 'category' column with dummy variables, we create dummy variables, drop the 'category' column, and then join the result:

In [26]: dummies = pd.get\_dummies(data.category, prefix='category')

In [27]: data\_with\_dummies = data.drop('category', axis=1).join(dummies) In [28]: data\_with\_dummies

Out[28]:

x0 x1 y category\_a category\_b

0 1 0.01 -1.5 1 0

1 2 -0.01 0.0 0 1

2 3 0.25 3.6 1 0

3 4 -4.10 1.3 1 0

4 5 0.00 -2.0 0 1

There are some nuances to fitting certain statistical models with dummy variables. It may be simpler and less error-prone to use Patsy (the subject of the next section) when you have more than simple numeric columns.

##### Creating Model Descriptions with Patsy

[Patsy](https://patsy.readthedocs.io/) is a Python library for describing statistical models (especially linear models) with a small string-based “formula syntax,” which is inspired by (but not exactly the same as) the formula syntax used by the R and S statistical programming languages.

Patsy is well supported for specifying linear models in statsmodels, so I will focus on some of the main features to help you get up and running. Patsy’s formulas are a spe‐ cial string syntax that looks like:

y ~ x0 + x1

The syntax a + b does not mean to add a to b, but rather that these are terms in the design matrix created for the model. The patsy.dmatrices function takes a formula string along with a dataset (which can be a DataFrame or a dict of arrays) and pro‐ duces design matrices for a linear model:

In [29]: data = pd.DataFrame({

....: 'x0': [1, 2, 3, 4, 5],