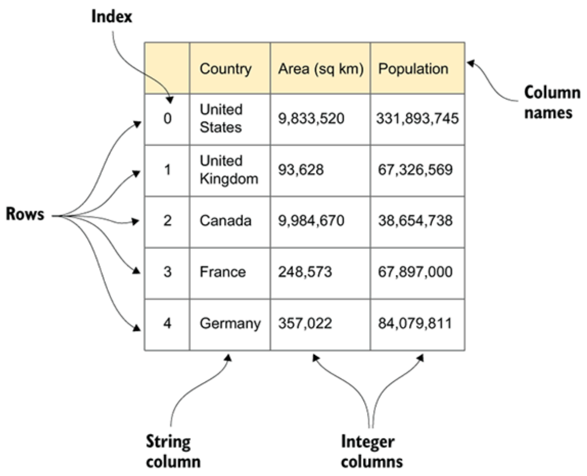
# 1 Series

If you have any experience with pandas, you know that we typically work with data in two-dimensional tables known as *data frames*, with rows and columns. But each column in a data frame is built from a *series*, a one-dimensional data structure (figure 1.1), which means you can think of a data frame as a collection of series.



| | Country | Area (sq km) | Population |
|---|---|---|---|
| 0 | United States | 9,833,520 | 331,893,745 |
| 1 | United Kingdom | 93,628 | 67,326,569 |
| 2 | Canada | 9,984,670 | 38,654,738 |
| 3 | France | 248,573 | 67,897,000 |
| 4 | Germany | 357,022 | 84,079,811 |

Figure 1.1 Each of a data frame's columns is a series.

This perspective is particularly useful once you learn what methods are available on a series, because most of those methods are also available on data frames—but instead of getting a single result, we get one result for each column in the data frame. For example, when applied to a series, the `mean` method returns the mean of the values in the series (figure 1.2). If you invoke `mean` on a data frame, pandas invokes the `mean` method on each column, returning a collection of mean values. Moreover, those values are themselves returned as a series on which you can invoke further methods.

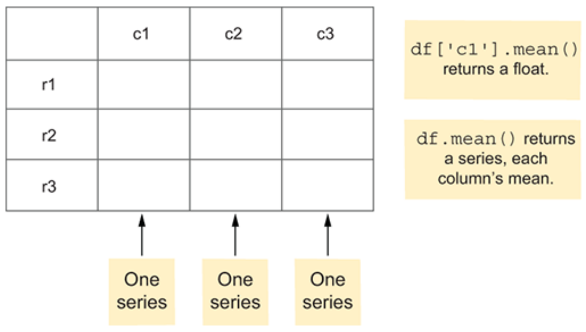

c1
c2
c3
r1
r2
r3
df['c1'].mean() returns a float.
df.mean() returns a series, each column's mean.
One series
One series
One series

Figure 1.2 Invoking a series method (such as `mean`) on a data frame often returns one value for each column.

A deep understanding of series can be useful in other ways, too. In particular, with a *boolean index* (also known as a *mask index*), we can retrieve selected rows and columns of a data frame. (If you aren't familiar with boolean indexes, see the sidebar "Selecting values with booleans," later in this chapter.)

One of the most important and powerful tools we have as pandas users is the index, used to retrieve values from both series and data frames. We'll look at indexes in greater depth in later chapters, but knowing how to set and modify an index, as well as retrieve values using unique and nonunique values, comes in handy just about every time you use pandas. This chapter will help you better understand how to use indexes effectively.

Naming conventions in this book

I use several variable names throughout this book:

- `s` refers to a series.
- `df` refers to a data frame.
- `pd` is an alias to the pandas library, loaded with `import pandas as pd`.

Although I’m a big fan of using semantically powerful variable names, I use `s` and `df` quite a bit when teaching pandas. Given that we normally work with only one series or data frame at a time, I’ll assume its meaning is clear. In the rare cases when I use more than one series or data frame, I’ll normally add numbers to `s` and `df`.

I also like to refer to the `Series` and `DataFrame` classes without an initial `pd` before their names. My code thus usually starts with

```
from pandas import Series, DataFrame
```

## Jseful references

Table 1.1 What you need to know

| Concept | What is it? | Example | To learn more |
|---|---|---|---|
| Jupyter | Web-based system for programming in Python and data science | `jupyter notebook` | http://mng.bz/BmYq |
| f-strings | Strings into which expressions can be interpolated | `f'It is currently {datetime.datetime.now()}'` | http://mng.bz/lWoz and http://mng.bz/a1dJ |
| data types (aka `dtype`) | Data types allowed in series | `np.int64` | http://mng.bz/gBVR |
| `pd.Series.astype` | Returns a new series with the same contents, converted to the target `dtype` | `s.astype(np.int32)` | http://mng.bz/xjVB |
| `pd.Series.mean` | Returns the arithmetic mean of the series contents | `s.mean()` | http://mng.bz/e1DJ |
| `pd.Series.max` | Returns the highest value in a series | `s.max()` | http://mng.bz/A8pW |

|  |  |  |  |
|---|---|---|---|
| `pd.Series.idxmin` | Returns the index of the lowest value in a series | `s.idxmin()` | http://mng.bz/ZR6Z |
| `pd.Series.idxmax` | Returns the index of the highest value in a series | `s.idxmax()` | http://mng.bz/RmrP |
| `np.random.default_rng` | Returns a NumPy random-number generator with an optional seed | `np.random.default_rng(0)` | http://mng.bz/27RX |
| `g.integers` | Returns a NumPy array of randomly selected integers via the generator | `g.integers(0, 10, 100)` | http://mng.bz/1JZg |
| `g.random` | Returns a NumPy array of randomly selected floats between 0 and 1 via the generator | `np.random.rand(10)` | http://mng.bz/PRBP |
| `s.std()` | Returns the standard deviation of a series | `s.std()` | http://mng.bz/Gy4N |
| `s.loc` | Accesses elements of a series by | `s.loc['a']` | http://mng.bz/zXlZ |

| | | | |
|---|---|---|---|
| | labels or a boolean array | | |
| `s.iloc` | Accesses elements of a series by position | `s.iloc[0]` | http://mng.bz/0K7z |
| `s.value_counts` | Returns a sorted (descending frequency) series counting how many times each value appears in `s` | `s.value_counts()` | http://mng.bz/WzOX |
| `s.round` | Returns a new series based on `s` in which the values are rounded to the specified number of decimals | `s.round(2)` | http://mng.bz/8rzg |
| `s.diff` | Returns a new series based on `s` whose values contain the differences between each value in `s` and a previous row | `s.diff(1)` | http://mng.bz/jP59 |
| `s.describe` | Returns a series summarizing | `s.describe()` | http://mng.bz/EQ1r |

| | | | |
|---|---|---|---|
| | all major descriptive statistics in `s` | | |
| `pd.cut` | Returns a series with the same index as `s` but with categorized values based on cut points | `pd.cut(s, bins=[0, 10, 20], labels=['a', 'b', 'c'])` | http://mng.bz/N2eX |
| `pd.read_csv` with `squeeze` | Returns a new series based on a single-column file | `s = pd.read_csv('filename.csv').squeeze()` | http://mng.bz/D4N0 |
| `str.split` | Breaks strings apart, returning a list | `'abc def ghi' .split() # returns ['abc', 'def', 'ghi']` | http://mng.bz/aR4z |
| `str.get` | Retrieves a character from a series | `s.str.get(0)` | http://mng.bz/JdWv |
| `s.fillna` | Replaces `NaN` values with a specified value | `s.fillna(5)` | http://mng.bz/wjrQ |

## xercise 1 • Test scores

Create a series of 10 elements, random integers from 70 to 100, representing scores on a monthly exam. Set the index to be the month names, starting in September and ending in June. (If these months don’t match the school year in your location, feel free to make them more realistic.)

With this series, write code to answer the following questions:

- What is the student's average test score for the entire year?
- What is the student's average test score during the first half of the year (i.e., the first five months)?
- What is the student's average test score during the second half of the year?
- Did the student improve their performance in the second half? If so, by how much?

## Working it out

In this first exercise, I asked you to create a series of 10 elements with random integers from 70 to 100. This raises several questions:

- How do we define a series?
- How can we create 10 random integers from 70 to 100?
- How can we set the index of the series to month names?

To define a pandas series, we call `Series`, passing it an iterable—typically a Python list or NumPy array, for example:

```
s = Series([10, 20, 30, 40, 50])
```

Here, I asked you to define the series such that it contains 10 random integers. There are certain areas in which pandas defers to NumPy, including when generating random numbers. We can get a NumPy array of random integers by creating a random-number generator with `np.default_rng` and then invoking `integers` on the object we get back.

Predictable random numbers

The Python standard library's `random` module has a `randint` method that returns a random integer:

```
random.randint(0, 100)
```

In the case of `random.randint`, the returned values range from 0 to 100, *including* 100.

In the world of NumPy, though, we do things differently. First, we create a new random-number generator object:

```
g = np.random.default_rng()
```

With this generator in hand, we can create an array of random integers by invoking `g.integers`:

```
g.integers(0, 100, 10)
```

This method differs from Python's `random.randint` function in the following ways:

- It returns a NumPy array rather than a single integer.
- It takes three arguments: the minimum value, the maximum value, and the length of the array that should be returned.
- The second argument is one more than the highest value we can get back.

Because this is a book of exercises, you will likely want to compare your solutions with mine. How can we do that, though, if we're both generating random numbers? We can pass a *random seed*: a number that kicks off the random-number generator when we invoke `np.random.default_rng`. If you and I pass the same argument to `default_rng`, we will see the same sequence of random numbers.

Here's an example of how we can create an array of random integers from 0 to 100:

```
g = np.random.default_rng(0)        ①
a = g.integers(0, 100, 10)          ②

g = np.random.default_rng(0)        ③
b = g.integers(0, 100, 10)          ④

a == b                              ⑤
```

① Seeds the random-number generator with 0

② Gets 10 random integers from 0 to 100

③ Seeds the random-number generator with 0

④ Gets another 10 random integers from 0 to 100

⑤ Because the seeds were the same, `a` and `b` will be, too.

If you're a NumPy old-timer like me, you may wonder about the `np.random.seed` function, which operated similarly to the argument passed to `default_rng`. That function still exists, but the core NumPy developers discourage its use in favor of generator objects.

We can thus get 10 random integers between 70 and 100 with

```
g = np.random.default_rng(0)
g.integers(70, 101, 10)        ①
```

① Upper bound of 101 allows for a result of 100.

We can use them to create a series:

```
g = np.random.default_rng(0)
s = Series(g.integers(70, 101, 10))
```

We now have a series of random integers between 70 and 100. But the index contains integers from 0 through 9—much as would be the case in a NumPy array or a Python list. There's nothing inherently wrong with a numeric index, but pandas gives us much more power and flexibility, letting us use a wide variety of data types, including strings.

We can change the index by assigning to the `index` attribute:

```
g = np.random.default_rng(0)
s = Series(g.integers(70, 101, 10))
s.index = 'Sep Oct Nov Dec Jan Feb Mar Apr May Jun'.split()
```

Sure enough, printing the contents of `s` shows the same values, but with our index:

```
Sep    96
Oct    89
Nov    85
Dec    78
Jan    79
```

```
Feb    71
Mar    72
Apr    70
May    75
Jun    95
dtype: int64
```

**NOTE** You can assign a list, NumPy array, or pandas series as an index. However, the data structure you pass must be the same length as the series. If it isn't, you'll get a `ValueError` exception, and the assignment will fail.

If we know what index we want when we create the series, we can assign it to the `index` keyword parameter:

```
g = np.random.default_rng(0)
months = 'Sep Oct Nov Dec Jan Feb Mar Apr May Jun'.split()
s = Series(g.integers(70, 101, 10),
           index=months)
```

This is my preferred method for creating a series, and I use this style for most of the book. That said, if and when I want to change the index, I can assign a new value to `s.index`.

Now that we've created our series, how can we perform the calculations I asked for? We first want to find the student's average test score for the entire year. We can calculate that with the `mean` method, which runs on any numeric series (Even if the series contains only integers, `mean` will always return a float. That's because in Python, division always returns a float.):

```
print(f'Yearly average: {s.mean()}')
```

Note that we put the call to `s.mean()` inside curly braces in a Python f-string. F-strings (short for *format strings*, although I like to call them *fancy strings*) allow any Python expression inside the curly braces. The result is a string suitable for assigning, printing, or passing as an argument to a function or method.

Next, we want to find the averages for the first and second halves of the school year. To do that, we need to retrieve the first five elements in the series and then the second five elements. There are a few different ways to accomplish this.

If we were using a standard Python sequence, we could use a *slice* by using square brackets along with indications of where we wanted to start and end. For example, given a string `s`, we can get the first five elements with the slice `s[:5]`. That returns a new series with the elements of `s` starting with index 0 (the start) up to and not including index 5. Generally, whenever you provide a range in Python—be it in a slice or the `range` built-in—the maximum is always “up to and not including.”

It’s thus not a surprise that we can retrieve the first five elements from our sequence using this same syntax: `s[:5]`. Because a slice always returns an object of the same type, this slice returns a five-element series. Because it’s a series, we can then run the `mean` method on it, getting the mean score for the first semester:

```
s[:5].mean()     ①
```

① Mean scores for the first half

What about the second semester? We can get those scores in a similar way, creating a slice from index 5 until the end of the series with `s[5:]` (figure 1.3). It's important not to provide an ending index here because the max index is always one more than we want. If we were to explicitly state `s[5:9]` or `s[5:-1]`, we would miss the final value. And yes, we can say `s[5:10]`, even though there is no index 10, because slices tend to be forgiving in Python:

```
s[5:].mean()     ①
```

① Mean scores for the second half

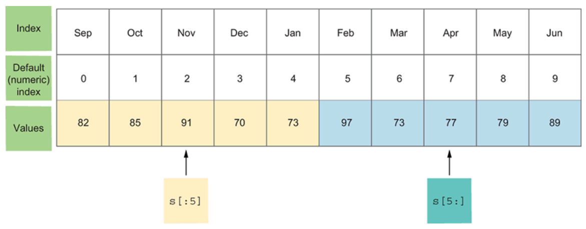


Figure 1.3 Retrieving slices from our series

I would argue that it's even better to use the `.loc` and `.iloc` accessors. Whereas `.loc` retrieves one or more elements based on the index, `.iloc` retrieves based on the numeric position—the default index. Let's start with `.iloc` because its usage is similar to what we've already written:

```
s.iloc[:5].mean()
```

"But wait," you may be saying. "Why are we using the positional, numeric index? Didn't we set an index with the names of the months?" Indeed we did. Moreover, we can use those to get our answers, instead.

Once again, we want to get a slice. And once again, we can do that—pandas is smart enough to let us use the textual index with a slice. We can use the `loc` accessor if we want, which is normally a good idea when working with series and mandatory when working with data frames. It's not mandatory with a series, but it is definitely a good idea to keep your code more readable.

If we want to get the scores from the first five months (September, October, November, December, and January), we can use the following slice:

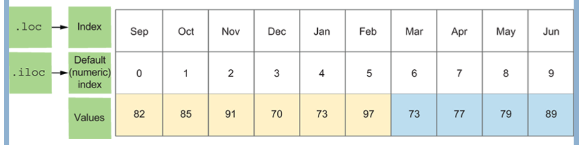
```
first_half_average = s.loc['Sep':'Jan'].mean()
```

The endpoint of a slice is normally "up to and not including," but in this case, the slice endpoint is "up to and including." That is, our `'Sep':'Jan'` slice *includes* the value for January. What gives?

Simply put, when you use `.loc`, the slice end is no longer "up to and not including" but rather "up to *and* including." This makes logical sense because it's not always obvious what "up to and not including" a custom index would be. But it's often surprising for people with Python experience who are starting to use pandas. It's also different from the behavior we saw on the same series with `.iloc`, using positional indexes.

loc vs. iloc vs. head

Most of the time, I prefer to use `.loc`, which is more readable and easier to understand. Plus, `.loc` offers a great deal of flexibility and power when working with data frames. But there is a cost: in some simple benchmarking I performed, pandas took twice as long to get the text-based slice (with `.loc`) as the number-based slice (with `.iloc`).



Retrieve via the index using `.loc` and via the position using `.iloc`.

I should add that there's another way to get the first and second halves of the year: the `head` and `tail` methods. The `head` method takes an integer argument and returns that many elements from the start of `s`. (If you don't pass a value, it returns the first 5, which is convenient for our purposes.) We can thus get the mean for the first five months of the year with

```
s.head().mean()
```

If you prefer to be explicit, you can say

```
s.head(5).mean()
```

We can similarly use the `tail` method to get the final five elements from `s`:

```
s.tail().mean()
```

Again, the default argument value is 5, but we can make it explicit with

```
s.tail(5).mean()
```

Finally, we can check the improvement by subtracting the first half's average from the second half. We assign each half's mean to a variable and then calculate the difference in an f-string:

```
first_half_average = s['Sep':'Jan'].mean()
second_half_average = s['Feb':'Jun'].mean()

print(f'First half average: {first_half_average}')
print(f'Second half average: {second_half_average}')

print(f'Improvement: {second_half_average - first_half_average}')
```

## Solution

```
g = np.random.default_rng(0)
months = 'Sep Oct Nov Dec Jan Feb Mar Apr May Jun'.split()
s = Series(g.integers(70, 101, 10),
          index=months)

print(f'Yearly average: {s.mean()}')

first_half_average = s['Sep':'Jan'].mean()
second_half_average = s['Feb':'Jun'].mean()

print(f'First half average: {first_half_average}')
print(f'Second half average: {second_half_average}')

print(f'Improvement: {second_half_average - first_half_average}')
```

You can explore a version of this in the Pandas Tutor at **http://mng.bz/27ld**.

## Beyond the exercise

Here are three additional exercises to help you better understand using `.loc` and `.iloc` to retrieve data from `s`, the series used in this exercise:

- In which month did this student get their highest score? Note that there are at least three ways to accomplish this: you can sort the values, taking the largest one, using a boolean (*mask*) index to find rows that match the value of `s.max()`, the highest value, or invoking `s.idxmax()`, which returns the index of the highest value.
- What were this student's five highest scores?
- Round the student's scores to the nearest 10. (A score of 82 would be rounded down to 80, but a score of 87 would be rounded up to 90.) Be sure to read the documentation for the `round` method (**http://mng.bz/8rzg**) to understand its arguments and how it handles numbers like 15 and 75.

Understanding mean and standard deviation

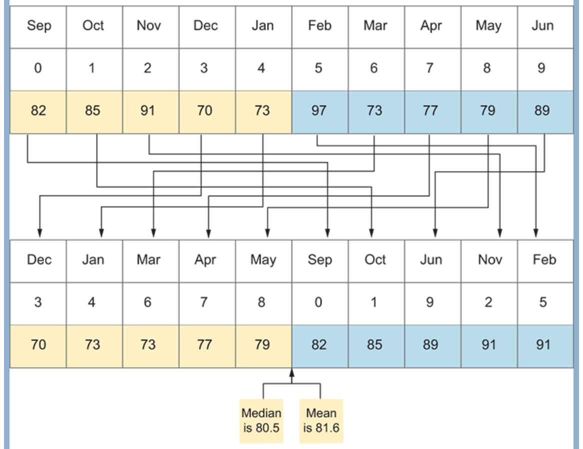
Two of the most common and important calculations we can make on a data set are the mean and the standard deviation. Pandas lets us calculate the mean on a series `s` with `s.mean()` and the standard deviation with `s.std()`.

What are these calculations? And why do we care about them so much?

The mean describes the center of a data set. (In a moment, I'll describe where this description can be flawed.) We add all the values and then divide them by the number of values we have. In pandas syntax, we can say that `s.mean()` is the same as `s.sum() / s.count()` because `s.sum()` adds the values and `s.count()` tells us how many non-`NaN` values are in the series.

Is the mean a truly good measurement of the “middle” of our data? The answer is, it depends. On many occasions, it’s useful because it gives us a central point on which we can focus. For example, we can talk about mean height, mean weight, mean age, or mean income in a population, and it will give us a single number that represents the entire population under discussion.

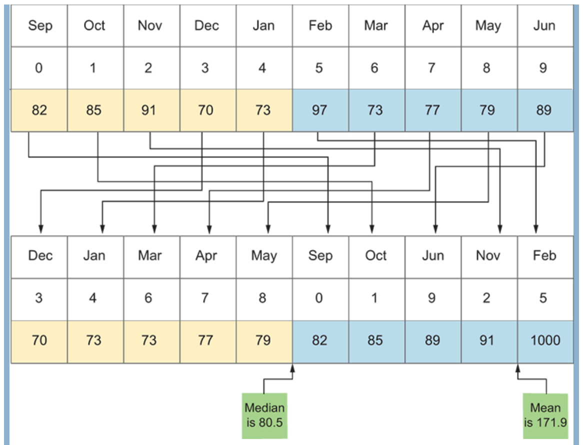
But the mean is flawed because a single large value can skew it. An old statistical joke is that when Bill Gates enters a bar, everyone in the bar is now, on average, a millionaire. For this reason, the mean isn’t the only way we can calculate the “middle” of our values. A common alternative is the *median*, which is the value precisely halfway from the smallest to the largest value. (If there is an even number of values, we take the average of the two innermost ones.) In the Bill Gates example, the median income of everyone in the bar will shift slightly when he enters, but it won’t change any assumptions we’ve made about the population.



| Sep | Oct | Nov | Dec | Jan | Feb | Mar | Apr | May | Jun |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 82 | 85 | 91 | 70 | 73 | 97 | 73 | 77 | 79 | 89 |

| Dec | Jan | Mar | Apr | May | Sep | Oct | Jun | Nov | Feb |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 4 | 6 | 7 | 8 | 0 | 1 | 9 | 2 | 5 |
| 70 | 73 | 73 | 77 | 79 | 82 | 85 | 89 | 91 | 91 |

To calculate the median, we first sort the values and then take the middle one.



By changing one value, we can see that the mean is more easily affected by outliers than the median.

Whether we're using the mean or the median to find the central point in our data set, we will almost certainly want to know the *standard deviation*: a measurement of how much the values in our data set vary from one another. In a data set with 0 standard deviation, the values are all identical. By contrast, a data set with a very large standard deviation has values that vary greatly from the mean value. The higher the standard deviation, the more the values in the data set vary from the mean.

To calculate the standard deviation on series `s`, we do the following:

- Calculate the difference between each value in `s` and its mean.
- Square each of these values.
- Sum the squares.
- Divide by the number of elements in `s`. This is known as the *variance*.
- Take the square root of the variance, which gives us the standard deviation.

Expressed in pandas, we say

```
import math
math.sqrt(((s - s.mean()) ** 2).sum() / s.count())
```

Given our values of `s` from before, this results in a value of `8.380930735902785`. If we then calculate `s.std()`, we get . . . uh, oh. We get a different value, `8.83427667918797`. What's going on?

By default, pandas assumes that we don't want to divide by `s.count()` but rather by `s.count() - 1`. This is known as the *sample standard deviation* and is typically used on a sample of the data rather than the entire population. The pandas authors decided to default to this calculation. (NumPy's `std` calculation doesn't do this.)

If you want to get the same result that we calculated and that NumPy provides, you can pass a value of `0` to the `ddof` (*delta degrees of freedom*) parameter:

```
s.std(ddof=0)
```

This tells pandas to subtract 0 (rather than 1) from `s.count()` and thus match our calculation for standard deviation. In this book, I do not pass this parameter to `std`, and I use the default value of `1` for the `ddof` parameter.

In a normal distribution used for many statistical assumptions, we expect that 68% of a data set's values will be within 1 standard deviation of the mean, 95% within 2 standard deviations, and 99.7 within 3 standard deviations. If you invoke `np.random.randint` (for integers) or `np.random.rand` (for floats), you'll get a truly random distribution. If you prefer to get a normal distribution in which the randomly selected numbers are centered around a mean and within a particular standard deviation, you can instead use `g.normal`. This method takes three arguments: the mean, the standard deviation, and the number of values to generate. It returns a NumPy array with a `dtype` of `np.float64`, which you can use to create a new series.

In this section, we used several so-called *aggregation methods*, which run on a series and return a single number—for example, `sum`, `mean`, `median`, and `std`. We'll use these throughout the book, and you can use them in any data-analysis projects you work on.

When sums go wrong

The `sum` method is useful, as you can imagine. You will likely want to use it on numeric series to combine the values. But it turns out that if you run `s.sum()` when `s` is a series of strings, the result is the strings concatenated together:

```
s = Series('abcd efgh ijkl'.split())
s.sum()                                    ①
```

① Returns 'abcdefghijkl'

Things get even weirder when your series contains strings, but those strings are numeric:

```
s = Series('1234 5678 9012'.split())
s.mean()                                   ①
```

① Returns 41152263004.0

Where does this number come from? The values of `s` are added together as strings, resulting in `'123456789012'`. But then `s.mean()` converts this string into an integer and divides it by 3 (the length of the series).

This is one of those cases where the behavior makes logical sense but is almost certainly not what you want. It also appears to have been fixed in Python 3.12, returning a `TypeError` exception.

Understanding dtype

In Python, we constantly use the built-in core data types: `int`, `float`, `str`, `list`, `tuple`, and `dict`. Pandas is a bit different in that we don't use those types much. Rather, we use the types we get from NumPy, which provide a thin, Python-compatible layer over values defined in C.

Every series has a `dtype` attribute, and you can always read from that to know the type of data it contains. Every value in a series is that type; unlike a Python list or tuple, you cannot mix different types in a series. That said, pandas does allow us to define the `dtype` as `object`, meaning a series contains Python objects. When the `dtype` is `object`, we can usually assume that the series contains Python strings; more on that in chapter 9. Storing nonstring objects is rare and should generally be avoided, but there are sometimes good reasons to do so. You'll also have a `dtype` of `object` if there are multiple types in the series.

Several standard types of `dtype` values are defined by NumPy and used by pandas. There are also special pandas-specific types, some of which we'll discuss later in the book. The core NumPy `dtype` values to know are as follows:

- Integers of different sizes—`np.int8`, `np.int16`, `np.int32`, and `np.int64`.
- Unsigned integers of different sizes—`np.uint8`, `np.uint16`, `np.uint32`, and `np.uint64`.
- Floats of different sizes—`np.float16`, `np.float32`, and `np.float64`. (On some computers, you also have `np.float128`.)
- Python objects—`object`.

When you create a series, pandas normally assigns the `dtype` based on the argument you pass to `Series`:

- If all values are integers, the `dtype` is set to `np.int64`.
- If at least one of the values is a float (including `NaN`), the `dtype` is set to `np.float64`.
- Otherwise, the `dtype` is set to `object`.

You can override these choices by passing a value to the `dtype` parameter when you create a series. For example:

```
s = Series([10, 20, 30], dtype=np.float16)
```

If you try to pass a value that's incompatible with the `dtype` you've specified, pandas will raise a `ValueError` exception.

Why should you care about the `dtype`? Because getting the type right, especially if you're working with large data sets, allows you to balance memory usage and accuracy. These are problems we normally don’t think about in standard Python, but they are front and center when working with pandas.

For example, The `np.int8` type handles 8-bit signed numbers (i.e., both positive and negative), which means it handles numbers from –128 through 127. What happens if you go beyond those boundaries?

```
s = Series([127], dtype=np.int8)
s+1                                  ①
```

① Returns a one-element series with a value of –128.

That’s right: in the world of 8-bit signed integers, 127+1 is –128. It’s like the odometer of your car rolling over back to 0 when you’ve driven it for many years, except you won’t have any warning and thus won’t know whether your calculations are accurate.

Yes, this is a problem. So, you need to ensure that the `dtype` you use on your series is big enough to store whatever data you’re working with, including the results of any calculations you perform. If you’re planning to multiply your data by 10, for example, you need to ensure that the `dtype` is large enough to handle that, even if you won’t be displaying or directly using such values.

Given this problem, why not go for broke and use 64-bit integers for everything? After all, those are likely to handle just about any value you may have.

Perhaps, but those will also use a lot of memory. Remember that 64 bits is 8 bytes, which doesn’t sound like much for a modern computer. But if you’re dealing with 1 billion numbers, using 64 bits means the data will consume 8 gigabytes of memory without considering any overhead that Python, your operating system, and the rest of pandas may need. And, of course, you’re unlikely to have just only numbers in memory.

As a result, you need to consider how many bits you want and need to use for your data. There’s no magic answer; each case must be evaluated on its own merits.

What if you want to change the `dtype` of a series after you’ve already created it? You can’t set the `dtype` attribute; it’s read-only. Instead, you have to create a new series based on the existing one by invoking the `astype` method:

```
s = Series('10 20 30'.split())
s.dtype                                  ①

s = s.astype(np.int64)
s.dtype                                  ②
```

① Returns "object"

② Returns "np.int64"

If you try to invoke `astype` with a type that isn’t appropriate for the data, you’ll get (as we saw when constructing a series) a `ValueError` exception.

## xercise 2 • Scaling test scores

When I was in high school and college, our instructors sometimes gave extremely hard tests. Rather than fail most of the class, they would *scale* the test scores, known in some places as *grading on a curve*. They would assume that the average test score should be 80, calculate the difference between our actual mean and 80, and then add that difference to each of our scores.

For this exercise, I want you to generate 10 test scores between 40 and 60, again using an index starting with September and ending with June. Find the mean of the

scores and add the difference between the mean and 80 to each of the scores.

## Working it out

One of the most important ideas in pandas (and in NumPy) is that of vectorized operations. When we perform an operation on two different series, the indexes are matched, and the operation is performed via the indexes (figure 1.4). For example, consider

```
s1 = Series([10, 20, 30, 40])
s2 = Series([100, 200, 300, 400])

s1 + s2
```

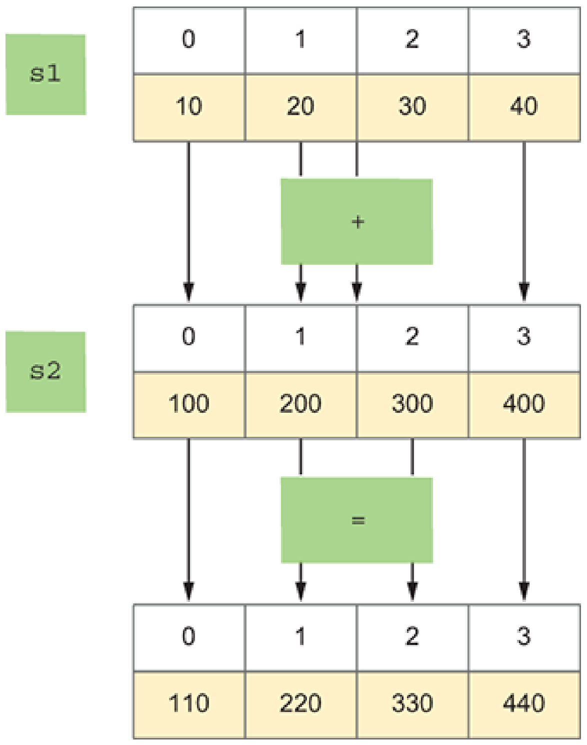


Figure 1.4 When we add two series together, the result is a new series—the result of adding elements at the same index.

The result is

```
0    110
1    220
2    330
3    440
dtype: int64
```

What happens if we set an explicit index rather than rely on the default positional index?

```
s1 = Series([10, 20, 30, 40],
    index=list('abcd'))
s2 = Series([100, 200, 300, 400],
    index=list('dcba'))

s1+s2
```

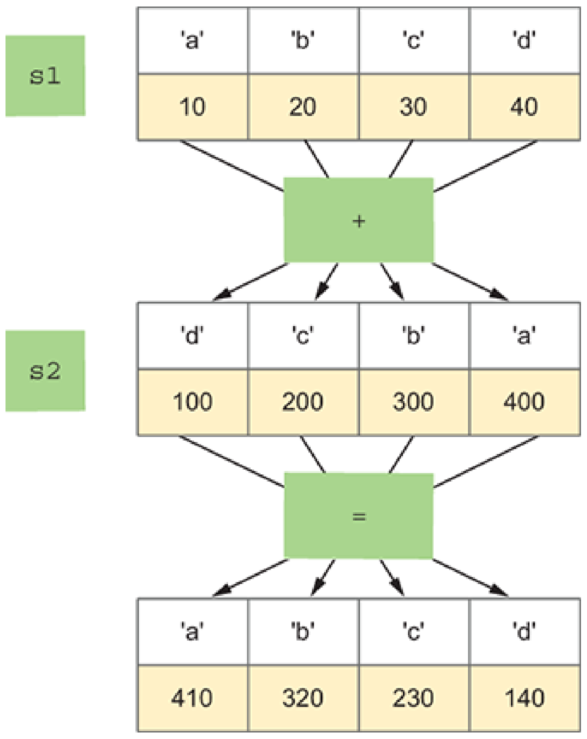


Figure 1.5 Vectorized operations work using the index, not the position.

The result is

```
a    410
b    320
c    230
d    140
dtype: int64
```

Again, pandas added the values together according to the index. Notice that this happened even though the index in `s1` was forward (`abcd`), whereas the index in `s2` was backward (`dcba`). The index values determine the value match-ups, not their position (figure 1.5).

But what happens when we try to add not one series and another series but rather a

series and a scalar value? Pandas does something known as *broadcasting*—it applies the operator and that scalar value to each value in the series, returning a new series. For example:

```
s = Series([10, 20, 30, 40],
    index=list('abcd'))

s + 3
```

The result is

```
a    13
b    23
c    33
d    43
dtype: int64
```

Notice that we get back from the operation a new series whose index matches those of `s` and whose values are the result of adding each element of `s` and the broadcast integer `3` (figure 1.6). We can do this with any operator, including comparison operators such as `==` and `<`. (The result of the latter is a boolean series, which we can then use as a *mask index* to keep only the rows we want.)

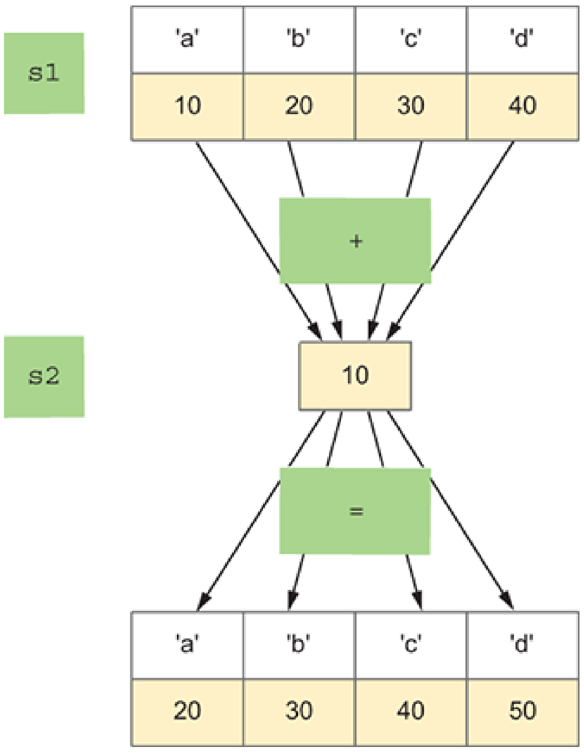


Figure 1.6 Operations involving a series and a scalar value result in *broadcasting* the operation, resulting in a new series.

So, if we want to generate 10 test scores between 40 and 60 and then add 10 points to them, we can do the following:

```
g = np.random.default_rng(0)

months = 'Sep Oct Nov Dec Jan Feb Mar Apr May Jun'.split()

s = Series(g.integers(40, 60, 10),
          index=months)

s+10
```

And sure enough, we'll get the following:

```
Sep    62
Oct    65
Nov    50
Dec    53
Jan    53
Feb    57
Mar    59
Apr    69
May    68
Jun    54
dtype: int64
```

That's nice, but the code still doesn't quite do what we want. That's because we don't know how many points we need to add to each score. We must first find the mean of `s` and then determine how far that is from 80. We can do that by invoking `s.mean()` and then subtracting the result from 80. Whatever we get back is the scale factor we need to add.

In other words, we can say

```
s + (80-s.mean())
```

And the result?

```
Sep    83.0
Oct    86.0
Nov    71.0
Dec    74.0
Jan    74.0
Feb    78.0
Mar    80.0
Apr    90.0
May    89.0
Jun    75.0
dtype: float64
```

Notice how this solution moves back and forth between scalar values and series, which is common in pandas calculations: The call to `s.mean()` returns a scalar value. We then calculate 80 - `s.mean()`, resulting in a scalar value. But then we add `s` and that number, adding (using broadcast) our series to that scalar value.

**NOTE** The final series has a `dtype` of `float64`, whereas `s` had a `dtype` of `int64`. Why the change? Because whenever we perform an operation on an int and a float, we get back a float, even if there's no need for it, as with addition. And division in Python 3 always returns a float. So the call to `s.mean()`, because it invokes division, always returns a float. And then when we add (via broadcast) the integer values in `s` to the floating-point mean, we get a series of floats.

## Solution

```
s + (80 - s.mean())
```

You can explore a version of this in the Pandas Tutor at **http://mng.bz/1JDV**.

### Beyond the exercise

Whether you’re performing an operation on two series or using broadcasts to combine a series and a scalar, the index is one of the most important ideas in pandas. It dictates how vectorized operations are performed as well as the index of the new series created by the operation. Here are some more exercises having to do with these topics:

- There’s at least one other way to scale test scores: by looking at both the mean of the scores and their standard deviation. Anyone who scored within one standard deviation of the mean got a C (below the mean) or a B (above the mean). Anyone who scored more than one standard deviation above the mean got an A, and anyone who got more than one standard deviation below the mean got a D. During which months did our student get an A, B, C, and D?
- Were there any test scores more than two standard deviations above or below the mean? If so, in which months?
- How close are the mean and median to each other? What does it mean if they are close? What would it mean if they were far apart?

## xercise 3 • Counting tens digits

In this exercise, I want you to generate 10 random integers in the range 0 to 100. (Remember that the `np.random.randint` function returns numbers that include the lower bound but exclude the upper bound.) Create a series containing those numbers’ tens digits. Thus, if our series contains `10, 25, 32`, we want the series `1, 2, 3`.

## Working it out

Given that we have created our series with `np.random.randint(0, 100, 10)`, we know that the 10 integers will all range from 0 (at the low end) to 99 (at the high end). We know that each number will contain either one or two digits.

To get the tens digit, we can do this:

1. Divide our series by 10, turning the `dtype` into a floating type and moving the decimal point one position to the left.
2. Turn our series back into `np.int8`, removing the fractional part of the number.

If the original number had two digits, we now have the tens digit. And if the original number had one digit, we are left with 0.

Sure enough, this works just fine, resulting in

```
0    4
1    4
2    6
3    6
4    6
5    0
6    8
7    2
8    3
9    8
dtype: int8
```

Notice that the `dtype` here is `int8` (figure 1.7).

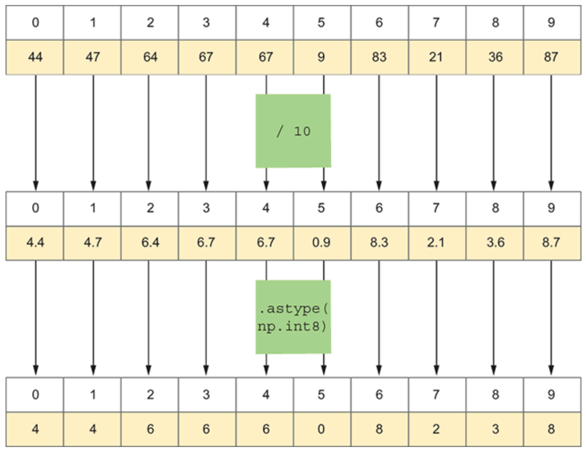


Figure 1.7 Graphical depiction of dividing the series by 10, converting to `np.int8`

But we can do even better than this: Python's `//` ("floordiv") operator performs integer division. If we divide the series by 10 using `//`, we'll still get our `dtype` of `int8`. That's the approach I'll go with because it reduces the number of operations we need to perform.

There is another way to do this, which involves more type conversions. This time, we convert our series not into floats but rather into *strings*. Why? Because when we turn integers into strings, we can retrieve particular elements from them, such as the second-to-last digit.

To do this, we convert our series of integers (`dtype` of `int8`) into a series of strings (`dtype` of `str`) using the `astype` method:

```
s.astype(str)
```

But then what? We'll talk about this more in chapter 8, which discusses strings in depth, but the key is the `str` accessor that lets us apply a string method to every element in the series. The `get` method on that accessor works like square brackets on a traditional Python string—so if we say `s.astype(str).str.get(0)`, we get the first character in each integer; and if we say `s.astype(str).str.get(-1)`, we get the final character in each string. (In Python, negative string indexes count from the end.) We can thus get the second-to-last digit, aka the tens digit, with `s.astype(str).str .get(-2)`.

But of course, that's not enough: if we have a one-digit number, what will `get(-2)` return? It won't give us an error or an empty string, but rather `NaN`. Fortunately, we can use the `fillna` method to replace `NaN` with any other value—for example, `'0'`. We then get back a series containing one-character strings: the tens digits from our original series. Our code looks like this:

```
s.astype(str).str.get(-2).fillna('0')
```

And the result is

```
0    4
1    4
2    6
3    6
4    6
5    0
6    8
7    2
8    3
9    8
dtype: object
```

As you can see from the `dtype`, the result is `object`, which typically means Python strings. Can we turn it back into a series of integers? Yes, by calling `astype` with an integer argument (figure 1.8). We'll use `np.int8` because all of our numbers are small:

```
s.astype(str).str.get(-2).fillna('0').astype(np.int8)
```

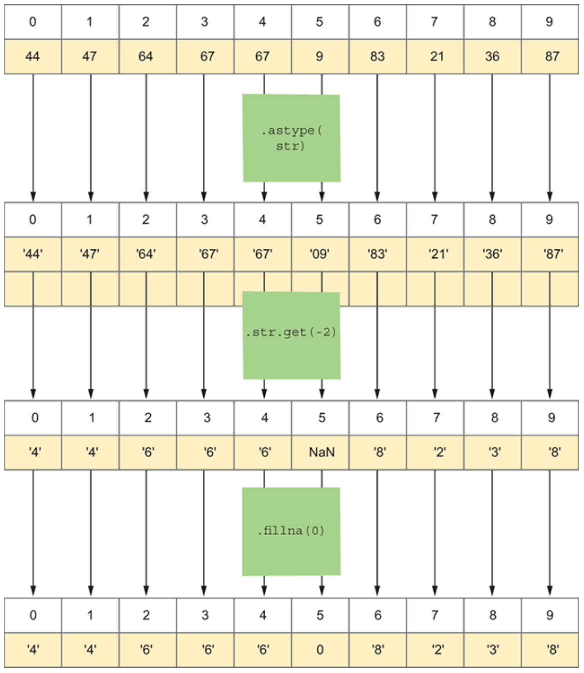


Figure 1.8 Graphical depiction of turning the series into strings, retrieving the item at index –2, and replacing `NaN` with 0

And the result is

```
0    4
1    4
2    6
3    6
4    6
5    0
6    8
7    2
8    3
9    8
dtype: int8
```

I think this is a cleaner way to do things than the int-to-float technique I showed earlier. But it is also more complex, and if you know you'll only have two-digit data, it may be overkill.

If you think the previous code puts too much on a single line, you can use a sneaky trick popularized by my friend Matt

Harrison (www.metasnake.com): Python allows us to split code across lines if we're still inside open parentheses. We can thus open parentheses on purpose and put each method call on a separate line. This can make things easier to read and follow and lets us put comments on each line:

```
(
    s
    .astype(str)     # get a series based on s, with dtype str
    .str.get(-2)     # retrieve the second-to-last character
    .fillna('0')     # replace NaN with '0'
    .astype(np.int8) # get a new series back dtype int8
)
```

This style gives the same result as the initial one-liner but is often more readable, especially as the code gets more complex.

**NOTE** Pandas has traditionally used Python strings, and that's what I assume in this book. As of this writing, however, there is an experimental new type, `pd.StringDType`, which aims to replace `str`. This is part of a larger movement in pandas to create new data types, partly so `NaN` will no longer always be a float and can represent a missing value from any type. I wouldn't be surprised if `pd.StringDtype` is a standard, recommended part of pandas in the coming years. There is also increasing support for Apache Arrow, including its string types. For now, though, Python strings are the best-supported version of strings in pandas, and we use them in this book.

## Solution

```
g = np.random.default_rng(0)
s = Series(g.integers(0, 100, 10))
```

```
s // 10
```

You can explore a version of this in the Pandas Tutor at **http://mng.bz/PRY9**.

## Beyond the exercise

- What if the range were from 0 to 10,000? How would that change your strategy, if at all?
- Given a range from 0 to 10,000, what's the smallest `dtype` you should use for integers?
- Create a new series with 10 floating-point values between 0 and 1,000. Find the numbers whose integer component (i.e., ignoring any fractional part) are even.

Selecting values with booleans

In Python and other traditional programming languages, we select elements from a sequence using a combination of `for` loops and `if` statements. Although you *can* do that in pandas, you almost certainly don't want to. Instead, you want to select items using a combination of techniques known as a *boolean index* or a *mask index*.

Mask indexes are useful and powerful, but their syntax can take some getting used to. First, consider that we can retrieve any element of a series via square brackets and an index:

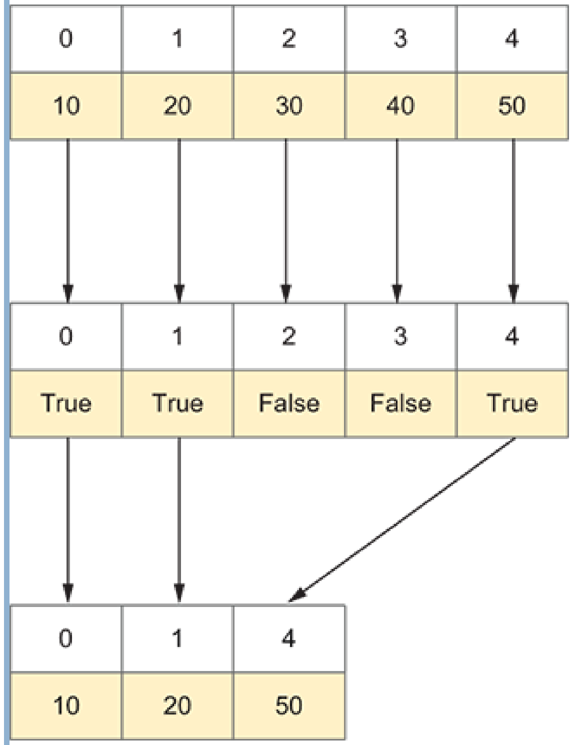
```
s = Series([10, 20, 30, 40, 50])
s.loc[3]
```
(1)

(1) Returns 40

Instead of passing a single integer, we can also pass a list (or NumPy array, or series) of boolean values (i.e., `True` and `False`):

```
s = Series([10, 20, 30, 40, 50])
s.loc[[True, True, False, False, True]]        ①
```

① Notice the double square brackets! The outer pair indicates we want to retrieve from s. The inner pair defines a Python list. Returns a series containing 10, 20, and 50.



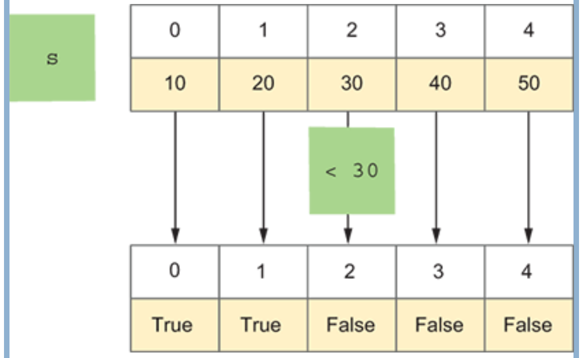
Choosing items via a mask index

Wherever we pass `True`, the value from `s` is returned, and wherever we pass `False`, the value is ignored. This is called a mask index because we're using the list of booleans as a type of sieve, or mask, to select only certain elements. Mask indexes don't transform the data but rather select specific elements from it.

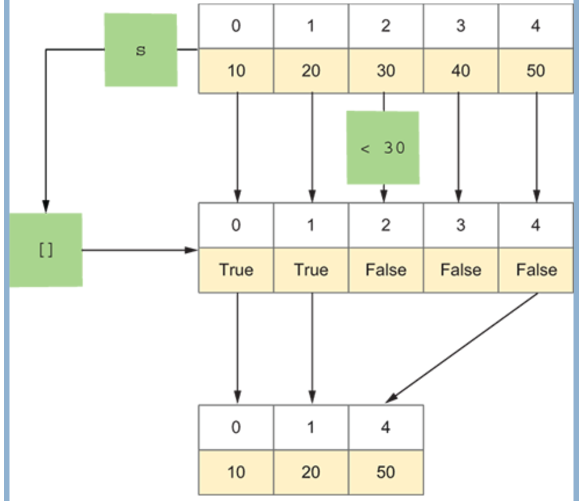
An explicitly defined list of booleans isn't very useful or common. But we can also use a series of booleans—and those are easy to create. All we need to do is use a comparison operator (e.g., `==`), which returns a boolean value. Then we can broadcast the operation and get back a series. For example:

```
s.loc[s < 30]   ①
```

① Returns the series containing 10 and 20



Generating a boolean series by broadcasting an operation



Using the boolean series as a mask index

This code looks very strange, even to experienced developers, in no small part because `s` is both outside the square brackets and inside them. Remember that we first evaluate the expression inside the square brackets. In this case, it’s `s < 30`, which returns a series of boolean values indicating whether each element of `s` is less than 30. We get back `Series([True, True, False, False, False])`.

That series of booleans is then applied to `s` as a mask index. Only those elements matching the `True` values are returned—in other words, just 10 and 20.

We can get more sophisticated, too:

```
s.loc[s <= s.mean()]     ①
```

① Returns the series containing 10, 20, and 30

Now `s` appears *three* times in the expression: once when we calculate `s.mean()`, a second when we compare the mean with each element of `s` via broadcast, and a third when we apply the resulting boolean series to `s`. We can thus see all of the elements that are less than or equal to the mean.

Finally, we can use a mask index for assignment and retrieval. For example:

```
s.loc[s <= s.mean()] = 999
```

The result?

```
0    999
1    999
2    999
3     40
4     50
dtype: int64
```

In this way, we replace elements less than or equal to the mean with 999.

This technique is worth learning and internalizing because it's both powerful and efficient. It's useful when working with individual series, as in this chapter—but it's also applicable to entire data frames, as we'll see later in the book.

One final note: given a series `s`, you can retrieve multiple items from different indexes using fancy indexing: passing a list, series, or similar iterable inside the square brackets. For example:

```
s.loc[[2,4]]
```

This code returns a series containing two values: the elements at `s.loc[2]` and `s.loc[4]`.

The outer square brackets indicate that we want to retrieve from `s` using `loc`, and the inner square brackets indicate that we want to retrieve more than one item. Pandas returns a series, keeping the original indexes and values.

Don't confuse fancy indexing with the application of a mask index; in the former case, the inner square brackets contain a list of values from the index. In the case of a mask index, the inner square brackets contain boolean (`True` and `False`) values.

## xercise 4 • Descriptive statistics

The mean, median, and standard deviation are three numbers we can use to get a better picture of our data. Adding a few other numbers can give us an even more complete picture. These *descriptive statistics* typically include the mean, standard deviation, minimum value, 25% quantile, median, 50% quantile, and maximum value. Understanding and using descriptive statistics is a key skill for anyone working with data, and in this exercise, you'll practice doing so with the following:

- Generate a series of 100,000 floats in a normal distribution with a mean of 0 and a standard deviation of 100.
- Get the descriptive statistics for this series. How close are the mean and median? (You don't need to calculate the difference; rather, consider why they aren't the same.)
- Replace the minimum value with 5 times the maximum value.
- Get the descriptive statistics again. Did the mean, median, and standard deviations change from their previous values? (Again, it's enough to see the difference without calculating it.) If so, why?

## Working it out

In this exercise, we create a slightly different distribution than before: rather than using `np.random.randint`, we instead use `g.normal`, which I described in the sidebar "Understanding mean and standard deviation," earlier in this chapter. When we invoke `g.normal`, we still get random numbers, but they are picked from the normal distribution—and we can specify both the mean and the standard deviation.

We thus create our series as follows:

```
g = np.random.default_rng(0)
s = Series(g.normal(0, 100, 100_000))    ①
```

① We can use _ in integers to separate digits.

We could call several different methods to find the descriptive statistics. But fortunately for us, pandas provides the `describe` method, which returns a series of measurements:

- `count`—The number of non-`NaN` values in the series
- `mean`—The mean, same as `s.mean()`
- `std`—The standard deviation, same as `s.std()`
- `min`—The minimum value, same as `s.min()`
- `25%`—The value in `s` you'll choose if you line up the values from smallest to largest and pick whatever is 25% of the way through, same as `s.quantile(0.25)`
- `50%`—The median value, same as `s.median()` or `s.quantile(0.5)`
- `75%`—The value in `s` you'll choose if you line up the values from smallest to largest and pick whatever is 75% of the way through, same as `s.quantile(0.75)`
- `max`—The maximum value, same as `s.max()`

You could get each of these values separately—but it's often useful to see and read them all at once.

Here's the result from calling `s.describe()`:

```
count    100000.000000
mean          0.157670
std          99.734467
min        -485.211765
25%         -66.864170
50%           0.172022
75%          67.343870
max         424.177191
dtype: float64
```

The mean is 0.157670. Not quite zero, which is what I asked for, but these are random numbers picked from a distribution, meaning there will always be wiggle room. The median, aka the 50% quantile, is 0.172022, which is very close to the mean. That makes sense, given that in a normal distribution, half of the numbers are below the mean and half are above it. The standard deviation here is roughly 100, meaning if all goes well, 68% of the values in `s` will be between –100 and +100.

What happens when we replace the minimum value with 5 times the max value? Moreover, how can we do that?

First, we need to find all the indexes at which the minimum value is located. (The `idxmin` method would return only one of the locations with this minimum value, but we want to modify as many as may exist.) The easiest way to do that is to first get a boolean series indicating which elements match the minimum value:

```
s == s.min()
```

This returns a boolean series with `True` wherever the value of `s` is the minimum. We can then apply this boolean series as a mask index:

```
s.loc[s == s.min()]
```

Now we have a series of only one element whose value is `s.min()`. We can assign a new value in its place using an assignment. But what do we want to assign? Five times the max value:

```
s.loc[s == s.min()] = 5*s.max()
```

Now that we have modified our series, we can call `s.describe()` on it again. We want to compare the mean, median, and standard deviations. What do we find?

```
count    100000.000000
mean          0.183731
std          99.947900
min        -465.995297
25%         -66.862839
50%           0.174214
75%          67.345174
max        2120.885956
dtype: float64
```

First, the mean value has gone up a bit—which makes sense, given that we took the smallest value and made it larger than the previously defined largest value. That's why the mean, although valuable, is sensitive to even a handful of very large or very small values.

Second, the standard deviation has also gone up. Again, this makes a great deal of sense, given that we have made a single value that's larger than anything we had before. True, the standard deviation didn't change much, but it reflects that values in our series are spread out more than they were previously.

Finally, the median barely shifted. That’s because it tends to be the most stable measurement, even when we have fluctuations at the extremes. This doesn’t mean you should only look at the median, but it can be useful. For example, if a country is trying to determine the threshold for government-sponsored benefits, a small number of very rich people will skew the mean upward, thus depriving more people of that help. The median will allow us to say that (for example) the bottom 20% of earners will receive help.

### Solution

```
import numpy as np
import pandas as pd
from pandas import Series, DataFrame

g = np.random.default_rng(0)
s = Series(g.normal(0, 100, 100_000))

print(s.describe())

s.loc[s == s.min()] = 5*s.max()

print(s.describe())
```

You can explore a version of this in the Pandas Tutor at http://mng.bz/JdM0.

### Beyond the exercise

- Demonstrate that 68%, 95%, and 99.7% of the values in `s` are indeed within one, two, and three standard distributions of the mean.
- Calculate the mean of numbers greater than `s.mean()`. Then calculate the mean of numbers less than `s.mean()`. Is the average of these two numbers the same as `s.mean()`?

- What is the mean of the numbers beyond three standard deviations?

## xercise 5 • Monday temperatures

Newcomers to pandas often assume that a series index must be unique. After all, the index in a Python string, list, or tuple is unique, as are the keys in a Python dictionary. But the values in a pandas index can repeat, making it easier to retrieve values with the same index. If an index contains user IDs, country codes, or email addresses, we can use it to retrieve data associated with specific index values that would otherwise require a messier and longer mask index.

In this exercise, I want you to create a series of 28 temperature readings in Celsius, representing four seven-day weeks, randomly selected from a normal distribution with a mean of 20 and a standard deviation of 5, rounded to the nearest integer. (If you're in a country that measures temperature in Fahrenheit, pretend you're looking at the weather in an exotic foreign location rather than where you live.) The index should start with `Sun`, continue through `Sat`, and repeat `Sun` through `Sat` until each temperature has a value. The question is, what was the mean temperature on Mondays during this period?

### Working it out

This exercise has two parts. First, we need to create a series that contains 28 elements but with a repeating index. Let's start by creating a random NumPy array of 28 elements drawn from a normal distribution in which the mean is 20 and the standard deviation is 5. (This means, as we’ve seen, that 95% of the values will be within 10 degrees of 20: that is, between 10 and 30. An extreme swing for one month, perhaps, but let’s assume it’s early spring or late autumn.) We can do this using `g.normal`, as we’ve seen before:

```
g = np.random.default_rng(0)
g.normal(20, 5, 28)
```

How can we create a 28-element index with the days of the week? One option is to simply create a list of 28 elements by hand. But I think that we can be a bit more clever than that, taking advantage of some core Python functionality. We can start by creating a seven-element list of strings with the days of the week:

```
days = 'Sun Mon Tue Wed Thu Fri Sat'.split()
```

If we had only seven data points in our series, we could set the index with `index=days` inside the call to `Series`. But because we have 28 data points, we want the list to repeat itself. We can create such a 28-element list by multiplying the list by 4, as in `days * 4`. This is very different behavior than the broadcast functionality of pandas!

We can thus create the series as follows:

```
s = Series(g.normal(20, 5, 28),
           index=days*4)          ①
```

① Multiplying a Python list returns a new list with the original list’s elements repeated.

But `g.normal` returns floats (specifically, `np.float64` objects). How can we turn this into a series of integers?

One way would be to use `astype(np.int8)` on our numbers. (The temperature is unlikely to get below –100 degrees or above 100 degrees, so we should be fine.) And that approach would basically work, but it would truncate the fractional part of the values rather than round them. If we want to round them to the nearest integer, we can call `round` on the series, thus getting back floats with no fractional portion. Then we can call `astype(np.int8)` on what we get back, resulting in a series of integers:

```
g = np.random.default_rng(0)
s = Series(g.normal(20, 5, 28),
           index=days*4).round().astype(np.int8)
```

We can now start to address the problem of repeated values in the index. Yes, the index can have repeated values—not just integers, but also strings (as in this example) and even other data structures, such as times and dates (as we'll see in chapter 9). Normally, when we retrieve a value from a series via `loc`, we expect to get back a single value. But if the index is repeated, we will get back multiple values. And in pandas, multiple values are returned as a series.

**NOTE** When you retrieve `s.loc[i]`, for a given index value, you can't know in advance whether you will get a single, scalar value (if the index occurs only once) or a series (if the index occurs multiple times). This is another case in which you need to know your data to know what type of value you'll get back.

In this case, we know that `Mon` exists four times in our series. And thus, when we ask for `s.loc['Mon']`, we'll get back a series of four values, all of which have `Mon` as their index:

```
s.loc['Mon']
```

We get back:

```
Mon     22
Mon     19
Mon     22
Mon     24
dtype: int8
```

Because this is a series, we can run any series methods we like on it. And because we want to know the average temperature on Mondays in this location, we can run `s.loc['Mon'].mean()`. Sure enough, we get the answer: 21.75.

### Solution

```
days = 'Sun Mon Tue Wed Thu Fri Sat'.split()

g = np.random.default_rng(0)
s = Series(g.normal(20, 5, 28),
          index=days*4).round().astype(np.int8)

s.loc['Mon'].mean()
```

You can explore a version of this in the Pandas Tutor at **http://mng.bz/wjeq**.

### Beyond the exercise

- What was the average weekend temperature (i.e., Saturdays and Sundays)?
- How many times is the change in temperature from the previous day greater than 2 degrees?
- What are the two most common temperatures in our data set, and how often does each appear?

## xercise 6 • Passenger frequency

In this exercise, we begin looking at real-world data loaded from a one-column CSV file. We'll take a deeper look at reading from and writing to files in chapter 3, but we start here by invoking `pd.read_csv`, calling `squeeze` on the one-column data frame it returns, and getting back a series.

The data is in the file **taxi-passenger-count.csv**, available along with the other data files used in this course. The data comes from 2015 data I retrieved from New York City's open data site, where you can get enormous amounts of information about taxi rides in New York City over the last few years. This file shows the number of passengers in each of 100,000 rides.

Your task in this exercise is to show what percentage of taxi rides had only one passenger versus the (theoretical) maximum of six passengers.

### Working it out

Let's start with reading the data into our series. `read_csv` is one of the most powerful and commonly used functions in pandas, reading a CSV file (or anything resembling a CSV file) into a data structure. As I mentioned, `read_csv` returns a data frame—but if we read a file containing only one column, we get a data frame with a single column. We can then invoke `squeeze` on that single-column data frame, getting back a series. Because all the values in this file are integers, pandas assumes that we want the series `dtype` to be `np.int64`.

We also set the `header` parameter to `None`, indicating that the first line in the file should not be taken as a column name but rather as data to be included in our calculations:

```
s = pd.read_csv('data/taxi-passenger-count.csv',
                header=None).squeeze()
```

The resulting series has a `name` value of 0, which we can safely ignore.

**NOTE** Although many methods operate on a series (or data frame), `read_csv` is a top-level function in the `pd` namespace. That's because we're not operating on an existing series or data frame. Rather, we're creating a new one based on the contents of a file.

Once we have read these values into a series, how can we figure out how often each value appears? One option is to use a mask index along with `count`:

```
s.loc[s==1].count()    ①
s.loc[s==6].count()    ②
```

① Results in 7207

② Results in 369

But wait: I asked you to give the proportion of elements in `s` with either 1 or 6. Thus we need to divide those results by `s.count()`:

```
s.loc[s==1].count() / s.count()   ①
s.loc[s==6].count() / s.count()   ②
```

① Results in about .720772

② Results in about .036904

There's nothing inherently wrong with doing things this way, but there's a far easier technique: `value_counts`, a series method that is one of my favorites. If we apply `value_counts` to the series `s`, we get back a new series whose keys are the distinct values in `s` and whose values are integers indicating how often each value appeared. Thus, if we invoke `s.value_counts()`, we get

```
1    7207
2    1313
5     520
3     406
6     369
4     182
0       2
Name: 0, dtype: int64
```

Notice that the values are automatically sorted from most common to least common.

Because we get back a series from `value_counts`, we can use all our series tricks on it. For example, we can invoke `head` on it to get the five most common elements. We can also use fancy indexing to retrieve the counts for specific values.

Because we're interested in the frequency of one- and six-passenger rides, we can say

```
s.value_counts()[[1,6]]
```

That returns

```
1    7207
6     369
Name: 0, dtype: int64
```

But we're interested in the percentages, not the raw values. Fortunately, `value_counts` has an optional `normalize` parameter that returns the fraction if set to `True`. We can thus say

```
s.value_counts(normalize=True)[[1,6]]
```

which returns the values

```
1    0.720772
6    0.036904
Name: 0, dtype: float64
```

### Solution

```
import pandas as pd
from pandas import Series, DataFrame

s = pd.read_csv('data/taxi-passenger-count.csv', header=None).squeeze()

s.value_counts(normalize=True)[[1,6]]
```

You can explore a version of this in the Pandas Tutor at **http://mng.bz/qjQw**.

### Beyond the exercise

Let's analyze the taxi passenger data in a few more ways:

- What are the 25%, 50% (median), and 75% quantiles for this data set? Can you guess the results before you execute the code?
- What proportion of taxi rides are for three, four, five, or six passengers?
- Consider that you're in charge of vehicle licensing for New York taxis. Given these numbers, would more people benefit from smaller taxis that can take only one or two passengers or larger taxis that can take five or six passengers?

## xercise 7 • Long, medium, and short taxi rides

In this exercise, we once again look at taxi data—but instead of the number of passengers, we examine the distance (in miles) each taxi ride went. Once again, I'll ask you to create a series based on a single-column CSV file, **taxi-distance.csv**.

First, load the data into a series. Then modify the series (or create another series) containing category names rather than numbers based on these criteria:

- Short, ≤ 2 miles
- Medium, > 2 miles but ≤ 10 miles
- Long, > 10 miles

Calculate the number of rides in each category.

### Working it out

It's not unusual to take numeric values and convert them to named categories. In this exercise, we want to turn taxi distances into short, medium, and long rides. How can we do that?

One approach is to use a combination of comparisons and assignments:

```
categories = s.astype(str)          ①
categories.loc[:] = 'medium'        ②
categories.loc[s<=2] = 'short'      ③
categories.loc[s>10] = 'long'       ④
categories.value_counts()
```

① Creates a new series the same length as s

② Assigns all the values to medium

③ Assigns small valuesto the string short

④ Assigns large values to the string long

When we call `value_counts`, we get the following:

```
short     5890
medium    3402
long       707
Name: 0, dtype: int64
```

This certainly works, but as you probably guessed, there is a more efficient approach. The `pd.cut` method allows us to set numeric boundaries and then cut a series into parts (known as *bins*) based on those boundaries. Moreover, it can assign labels to each of the bins.

Notice that `pd.cut` is not a series method but rather a function in the top-level `pd` namespace. We'll pass it a few arguments:

- Our series, `s`
- A list of four integers representing the boundaries of our three bins, assigned to the `bins` parameter

- A list of three strings, the labels for our three bins, assigned to the `labels` parameter

Note that the bin boundaries are exclusive on the left and inclusive on the right. In other words, by specifying that the medium bin is between 2 and 10, that means >2 but < = 10. This means the first boundary needs to be less than the smallest value in `s`. However, we can get around that by passing the `include_lowest=True` keyword argument, which ensures that the lowest value passed to `bins` is included in the first bin.

The result of this call to `pd.cut` is a series the same length as `s` but with the labels replacing the values:

```
pd.cut(s,
       bins=[0, 2, 10, s.max()],
       include_lowest=True,
       labels=['short', 'medium', 'long'])
```

The result, as depicted in Jupyter, is as follows:

```
0        short
1        short
2        short
3       medium
4        short
         ...
9994    medium
9995    medium
9996    medium
9997     short
9998    medium
Name: 0, Length: 9999, dtype: category                ①
Categories (3, object): ['short' < 'medium' < 'long']  ②
```

① Notice that the dtype is category. We will discuss categories later in the book.

② Shows the relative order of the categories in their description

The task I gave you for this exercise wasn't to turn the ride lengths into categories but rather to determine the number of rides in each category. For that, we need to call on our friend `value_counts`:

```
pd.cut(s,
       bins=[0, 2, 10, s.max()],
       include_lowest=True,
       labels=['short', 'medium', 'long']).value_counts()
```

And sure enough, this gives us the answer we want:

```
short     5890
medium    3402
long       707
Name: 0, dtype: int64
```

## Solution

```
import pandas as pd
from pandas import Series, DataFrame

s = pd.read_csv('data/taxi-distance.csv', header=None).squeeze()

pd.cut(s,
       bins=[0, 2, 10, s.max()],
       include_lowest=True,
       labels=['short', 'medium', 'long']).value_counts()
```

You can explore a version of this in the Pandas Tutor at http://mng.bz/7vx9.

## Beyond the exercise

- Compare the mean and median trip distances. What does that tell you about the distribution of the data?

- How many short, medium, and long trips were there for trips that had only one passenger? Note that the data for passenger count and trip length is from the same data set, meaning the indexes are the same.
- What happens if you don't pass explicit intervals and instead ask `pd.cut` to just create three bins, with `bins=3`?

## ummary

In this chapter, we saw that a pandas series provides powerful tools to analyze data. Whether it's the index, reading data from files, calculating descriptive statistics, retrieving values via fancy indexing, or even categorizing our data via numeric boundaries, we can do a lot. In the next chapter, we'll expand our reach to look at data frames, the two-dimensional data structures that most people think of when they work with pandas.