6 Grouping, joining, and sorting

So far, we have looked at how to create data frames, read data into them, clean the data, and then analyze that clean, imported data in a number of ways. But analysis sometimes requires more than just the basics: we often need to break our input data apart, zoom in on particularly interesting subsets, combine data from different sources, transform the data into a new format or value, and then sort it according to a variety of criteria. This type of action is known in the pandas world as *split-apply-combine*, and it is our focus in this chapter. If you have experience with SQL and relational databases, you'll find many similarities, in both principle and name, to functionality in pandas.

For example, a company may want to determine its total sales in the last quarter. It may also want to learn which countries have done particularly well (or poorly). Or perhaps the head of sales would like to see how much each individual salesperson has brought in, or how much each product has contributed to the company's income.

These types of questions can be answered using a technique known as *grouping*. Much like the `GROUP BY` clause in an SQL query, we can use grouping in pandas to ask the same question for various subsets of our data.

Another common SQL technique is *joining*, which lets us keep our data in small, specific data frames and combine them only when we need to. For example, one data frame may list each sales region and that region's manager, and a second may contain this quarter's regional sales results. To show the monthly sales results for each region along with each region's manager, we'll want to join the data frames together.

A third technique, which you have likely seen in other languages and frameworks, is *sorting*. In chapter 5, we saw how to use `sort_index` to order a data frame's rows by the values in the index. In this chapter, we'll look at `sort_values`, which reorders the rows based on the values in one or more columns.

You'll want to have these techniques—grouping, joining, and sorting—at your fingertips when solving problems with pandas. In this chapter, you'll see how to use them to solve some of the most common types of problems you'll encounter. These topics are so central to data analysis that one chapter wasn't enough; the next chapter builds on the techniques you'll practice here, showing more advanced uses of the split-apply-combine paradigm.

Table 6.1 What you need to know

Concept	What is it?	Example	To learn more
`s.isnull`	Returns a boolean series indicating where there are null (typically `NaN`) values in the series `s`	`s.isnull()`	http://mng.bz/Jgyp
`df.sort_index`	Reorder the rows of a data frame based on the values in its index, in ascending order	`df = df.sort_index()`	http://mng.bz/wvB7
`df.sort_values`	Reorder the rows of a data frame based on the values in one or more specified columns	`df = df.sort_values('distance')`	http://mng.bz/qrMK
`df.transpose()` or `df.T`	Returns a new data frame with the same values as `df` but with the columns and index exchanged	`df.transpose()` or `df.T`	http://mng.bz/7DXx
`df.expanding`	Lets us run window functions on an expanding (growing) set of rows	`df.expanding().sum()`	http://mng.bz/mVBn
`df.rolling`	Lets us run window functions on a fixed-size window that moves through the data frame	`df.rolling(3).mean()`	http://mng.bz/5wp4
`df.pct_change`	For a given data frame, indicates the percentage difference between each cell and the corresponding	`df.pct_change()`	http://mng.bz/4DBB

	cell in the previous row		
`df.diff`	For a given data frame, indicates the difference between each cell and the corresponding cell in the previous row	`df.diff()`	http://mng.bz/OPDE
`df.groupby`	Allows us to invoke one or more aggregate methods for each value in a particular column.	`df.groupby('year')`	http://mng.bz/vn9x
`df.loc`	Retrieves selected rows and columns	`df.loc[:, 'passenger_count'] = df['passenger_count']`	http://mng.bz/nWzv
`s.iloc`	Accesses elements of a series by position	`s.iloc[0]`	http://mng.bz/QPxm
`df.dropna`	Removes rows with `NaN` values	`df = df.dropna()`	http://mng.bz/XN0Y
`s.unique`	Gets the unique values in a series (Pandas' `drop_duplicates` is better)	`s.unique()`	http://mng.bz/yQrJ
`df.join`	Joins two data frames based on their indexes	`df.join(other_df)`	http://mng.bz/MBo2
`df.merge`	Joins two data frames based on any columns	`df.merge(other_df)`	http://mng.bz/a1wJ
`df.corr`	Shows the correlation between the numeric columns of a data frame	`df.corr()`	http://mng.bz/gBgR
`s.to_frame`	Turns a series into a one-column	`s.to_frame()`	http://mng.bz/5wp1

	data frame		
`s.removesuffix`	Returns a new string with the same contents as `s` but without a specified suffix (if it's there)	`s.removesuffix('.csv')`	http://mng.bz/6DAD
`s.removeprefix`	Returns a new string with the same contents as `s` but without a specified prefix (if it's there)	`s.removeprefix('abcd')`	http://mng.bz/o1Rr
`s.title`	Returns a new string based on `s` in which each word starts with a capital letter	`s.title('hello out there')`	http://mng.bz/nWzg
`pd.concat`	Returns one new data frame based on a list of data frames passed to `pd.concat`	`pd.concat([df1, df2, df3])`	http://mng.bz/vn9J
`df.assign`	Adds one or more columns to a data frame	`df.assign(a=df['x']*3)`	http://mng.bz/YR2A
`DataFrame-GroupBy.agg`	Applies multiple aggregation methods to a `groupby`	`df.groupby('a')['b'].agg(['mean', 'std'])`	http://mng.bz/G90O
`DataFrame-GroupBy.filter`	Keeps those rows whose group results in `True` from an outside function	`df.groupby('a').filter(filter_func)`	http://mng.bz/z0BQ
`DataFrameGroupBy.transform`	Modifies those rows based on an outside function	`df.groupby('a').transform(transform_func)`	http://mng.bz/0l26
`df.rename`	Renames columns in a data frame	`df.rename(columns={'a':'b', 'c':'d'})`	http://mng.bz/K9W0
`df.drop_duplicates`	Returns a data frame whose	`df.drop_duplicates()`	http://mng.bz/9Qv1

	rows contain distinct values		
`df.drop`	Removes rows or columns from a data frame, returning a new one	`df.drop('a', axis='columns')`	http://mng.bz/j1eP

Exercise 29 • Longest taxi rides

When I first started to work with relational (SQL) databases, I was surprised that data isn't stored in any particular order. As I soon learned, there are several reasons for this:

- The order in which the rows are stored doesn't affect many queries.
- It's more efficient for the database itself to figure out the order in which rows should be stored.
- There are so many ways in which we may want to sort the data that the database shouldn't guess. Rather, it should allow us to choose how we want to sort and extract the information.

Pandas does keep the rows of our data frame ordered, so it's not exactly like a relational database. But for many types of analysis, the order of the rows doesn't matter. After all, if we're calculating a column's mean, it doesn't matter where we start or end.

If we want to display data—say, sales records, network statistics, or inflation projections—we'll likely want to order it. How we order it depends on the context, though. Sales records may need to be ordered by department, network statistics may need to be ordered by subnets, and inflation projections may need to be ordered chronologically.

Another reason to sort is to get the highest or lowest values from a particular column in the data frame. And in this exercise, I'm asking you to do exactly that. Specifically, I want you to make a few queries using the New York City taxi data from January 2019:

1. Load the CSV file **nyc_taxi_2019-01.csv** into a data frame using only the columns `passenger_count`, `trip_distance`, and `total_amount`.
2. Using a *descending* sort, find the average cost of the 20 longest (in distance) taxi rides in January 2019.
3. Using an *ascending* sort, find the average cost of the 20 longest (in distance) taxi rides in January 2019. Are the results any different?
4. Sort by ascending passenger count and descending trip distance. (So, start with the longest trip with 0 passengers and end with the shortest trip with 9 passengers.) What is the average price paid for the top 50 rides?

Working it out

When we want to sort a data frame in pandas, we first have to decide whether to sort it via the index or the values. We've already seen that if we invoke `sort_index` on a data frame, we get back a new data frame whose rows are identical to the existing data frame but ordered such that the index is ascending.

In this exercise, we again want to sort the rows of our data frame—but we want to do it based on the values in a particular column rather than the index. You could argue that there isn't much difference between the two; we could take a column, temporarily make it the index, sort by the index, and then return the column back to the data frame. But the difference between `sort_index` and `sort_values` isn't just technical. We're thinking about our data and how to access it in different ways.

`sort_values` is also different from `sort_index` in that we can sort by any number of columns. Again, imagine that a data frame contains sales data. We may want to sort it by price, region, or salesperson—or even a combination of these. When we sort by the index, by contrast, we're effectively sorting by a single column.

In the first part of the exercise, I asked you to create a data frame with our favorite (and familiar) columns, `passenger_count`, `trip_distance`, and `total_amount`:

```
filename = '../data/nyc_taxi_2019-01.csv'

df = pd.read_csv(filename,
                 usecols=['passenger_count',
                          'trip_distance',
                          'total_amount'])
```

With the data frame in place, we can start to analyze the data. The first task is to find the 20 longest (in distance) taxi rides in our data set and then find their average cost. We thus need to sort our data set by distance—and I asked you to do that via a descending sort.

To sort our data frame by the `trip_distance` column, we can say

```
df.sort_values('trip_distance')
```

This returns a new data frame identical to `df`, but with the rows sorted according to `trip_distance` in ascending order. Although we could (and will) work with the data in this form, I find it easier in such cases to sort in descending order. We can do that by passing `False` as an argument to the `ascending` parameter (figure 6.1):

```
df.sort_values('trip_distance',
                ascending=False)
```

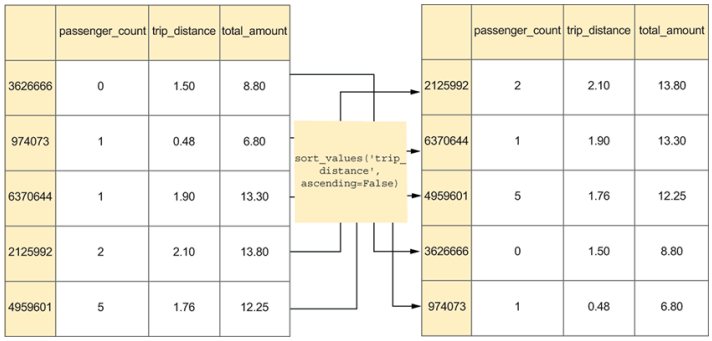


Figure 6.1 Running `sort_values` on a data frame returns a new data frame with the same rows but ordered according to the named column.

Our analysis is of the `total_amount` column. With the data already sorted by `trip_ distance`, we can now retrieve just that one column using square brackets (figure 6.2):

```
df.sort_values('trip_distance',
               ascending=False
               )['total_amount']
```

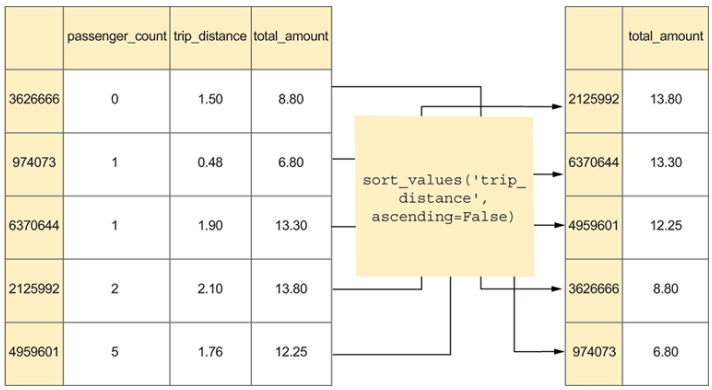


Figure 6.2 Running `sort_values` on a data frame, keeping only one column

But we're not interested in calculating the mean of all rows in `total_amount`, only those from the 20 longest trips. How can we retrieve the top 20 rows? One way would be to use `head(20)`. Another possibility, which we use here, is to retrieve the first 20 rows via `iloc` (figure 6.3):

```
df.sort_values('trip_distance',
               ascending=False
               )['total_amount'].iloc[:20]
```

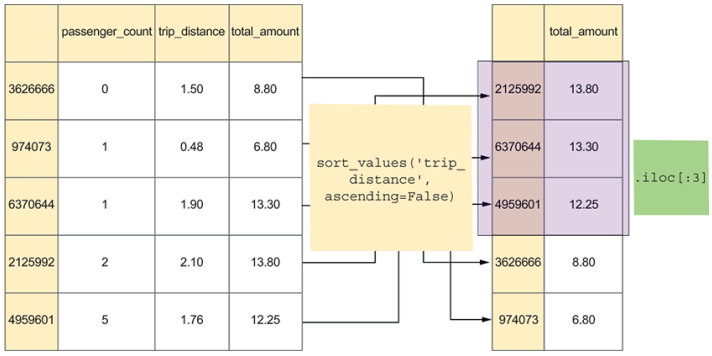


Figure 6.3 Running `sort_values` on a data frame, then keeping only one column, and then getting only the first rows with `iloc`

Notice that we have to use `iloc` here, not `loc`. That's because `loc` works with the actual index values—which, now that we've sorted the data frame by `trip_distance`, are unordered. Asking for `loc[:20]` will return many more than 20 rows.

Having retrieved `total_amount` from the 20 longest-distance taxi rides, we can finally calculate the mean value:

```
df.sort_values('trip_distance',
               ascending=False
               )['total_amount'].iloc[:20].mean()
```

We get a result of 290.00999999999993, which I think we can reasonably round to an average of $290 for those 20 longest taxi rides. We could even use the `round` method to round it to two digits. Here, we rewrite it using line-by-line method chaining for easier reading:

```
(
    df
    .sort_values('trip_distance',
                 ascending=False)
    ['total_amount']
    .iloc[:20]
    .mean()
    .round(2)
)
```

Next I asked you to make the same calculation but this time do an *ascending* sort. First we sort our data frame by values:

```
df.sort_values('trip_distance')
```

Remember that, by default, `sort_values` sorts in ascending order, so we don't need to specify anything there. We keep only the `total_amount` column:

```
df.sort_values('trip_distance')['total_amount']
```

And again, we're only interested in the 20 longest trips. This time, however, we sort in ascending order, which means the 20 longest trips are at the end of the series rather than at the top.

As before, we have two basic ways to do this. One would be to use `tail(20)` to retrieve the final 20 elements. But we're going to again use `iloc` and get the 20 final rows from our new data frame:

```
df.sort_values('trip_distance')[
    'total_amount'].iloc[-20:]
```

Remember that in Python, a negative index means we count from the end of the data structure rather than from the beginning. Thus index –1 gives us the final element, –2 the second-to-final element, and so forth. Moreover, our slice can be empty on one side, indicating that we want to go through the end of that side. Here, the use of `iloc[-20:]` means we want the final 20 elements in the series.

NOTE Wondering whether it's faster to run `tail` or `iloc` with a slice? Some performance checks I did showed that they were almost exactly the same.

Finally, we invoke `mean()` on the 20 longest-ride fares:

```
df.sort_values('trip_distance')[
    'total_amount'].iloc[-20:].mean()
```

And the result is . . . 290.01000000000005, which is, let's face it, basically the same as 290, our result from before. We can round this result as well, again rewritten to use method chaining:

```
(
    df
    .sort_values('trip_distance')
    ['total_amount']
    .iloc[-20:]
    .mean()
    .round(2)
)
```

Again, the rounded result is 290.01.

But let's ignore the rounded results and look at the original results, 290.00999999999993 and 290.0100000000001. The differences are slight, but they're real. What is going on here?

The answer, simply put, is that floating-point math is strange and can surprise you. Check out **https://0.30000000000000004.com** for a good, full explanation of floating-point problems; but is there anything we can do to avoid such problems?

The answer is: sort of. If we use longer (i.e., more bits) floats, such problems will crop up less often. For example, we can instruct pandas to read the `total_amount` column into 128-bit floats, rather than 64-bit floats, which are the default:

```
df = pd.read_csv(filename,
                 usecols=['passenger_count',
                          'trip_distance',
                          'total_amount'],
                 dtype={'total_amount':np.float128})
```

With this in place, both of our calculations—forward and backward—give the same result: 290.01000000000000076. But, of course, now our column consumes twice as much memory as before.

NOTE If 128-bit floats are the most accurate, why not always use them? First, because they're very large, at 16 bytes (!) per number. If you have 1 million floats, that translates into about 16 MB of data. Not every problem you're trying to solve needs such extreme accuracy. But 128-bit floats can also cause some problems. On my Mac, some pandas methods don't work when my columns have a `dtype` of `np.float128`. And it seems that `np.float128` doesn't even exist on computers running Windows. So if you need the precision and if you're on a platform that supports them, and if the pandas methods you need can use them, then sure—use `np.float128`. But keep in mind that doing so will make your program less portable.

Next, I asked you to sort by two columns. This is something we do naturally all the time without thinking about it. For example, telephone books are—or "were," I guess—sorted first by last name and then by first name, so the names appear in alphabetical order by last name. If more than one person has the same last name, we order the people by first name.

The sort that I asked you to do primarily looked at `passenger_count`, meaning we should sort the rows of `df` in ascending order from the smallest number of passengers to the greatest number of passengers. And in resolving ties between rows with the same passenger count, I asked you to use the `trip_distance` column. However, whereas `passenger_count` is sorted in ascending order, I asked you to sort `trip_distance` in descending order.

Pandas allows us to do this by passing a list of columns as the first argument to `sort_values`. We then pass a list of boolean values to `ascending`, with each element in the list corresponding to one of the sort columns (figure 6.4):

```
df.sort_values(['passenger_count', 'trip_distance'],
               ascending=[True, False])
```

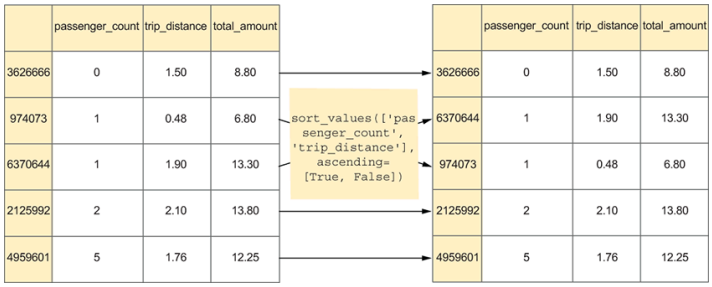


Figure 6.4 Sorting a data frame by `passenger_count` (ascending order) and then `trip_distance` (descending order)

This code returns a new data frame with three columns in which the rows are first sorted by (ascending) `passenger_count` and then by (descending) `trip_distance`. The first row of the returned data frame has the longest trip for the fewest passengers, and its final row has the shortest trip for the most passengers.

We then retrieve the `total_amount` column from the returned data frame, grab its first 50 rows using `iloc` (although we could just as easily use `head(50)` and calculate the mean using method-chaining syntax):

```
(
    df
    .sort_values(['passenger_count',
                  'trip_distance'],
                  ascending=[True, False])
   ['total_amount']
   .iloc[:50]
   .mean()
)
```

We get a result of 135.4974000000001.

Solution

```
filename = '../data/nyc_taxi_2019-01.csv'

df = pd.read_csv(filename,
                 usecols=['passenger_count',
                          'trip_distance',
                          'total_amount'],
                 dtype={'total_amount':np.float128})

df.sort_values('trip_distance',
                 ascending=False)[
                   'total_amount'].iloc[:20].mean()    ①
df.sort_values('trip_distance')[
                   'total_amount'].iloc[-20:].mean()   ②

(
    df
    .sort_values(['passenger_count',
                  'trip_distance'],
                  ascending=[True, False])
   ['total_amount']
   .iloc[:50]
```

```
    .mean()
)                                                    ③
```

① Sorts by descending values of trip_distance, gets only the total_amount column, grabs the first 20 rows, and then calculates their mean

② Sorts by ascending values of trip_distance, gets only the total_amount column, grabs the final 20 rows, and then takes their mean

③ Sorts by ascending passenger_count and then descending trip_distance, gets the total_amount column, grabs the first 50 rows, and takes their mean

You can explore a version of this in the Pandas Tutor at **http://mng.bz/W1Z1**.

Beyond the exercise

- In which five rides did people pay the most per mile? How far did people go on those trips?
- Let's assume that multipassenger rides are split evenly among the passengers. Given that assumption, in which 10 multipassenger rides did each individual pay the greatest amount?
- In the exercise solution, I showed that we needed to use `iloc` or `head`/`tail` to retrieve the first/last 20 rows because the index was scrambled after our sort operation. But you can pass `ignore_index=True` to `sort_values`: then the resulting data frame has a numeric index starting at 0. Use this option and `loc` to get the mean `total_amount` for the 20 longest trips.

Grouping

We've already seen how aggregate functions, such as `mean` and `std`, allow us to better understand our data. But sometimes we want to run an aggregate function on each piece of our data. For example, we may want to know the number of sales per region, the average cost of living per city, or the standard deviation for each age group in a population. We could, of course, run the aggregate function numerous times, each time retrieving a different group from the data frame. But that gets tedious—and why work hard, when pandas can do it for us?

This functionality, known as *grouping*, should also be familiar if you've worked with relational databases. In this exercise, we'll try to learn whether the number of people taking a taxi affects, on average, the distance the taxi has to travel. In other words, if we're a taxi driver who moonlights as a data analyst (or if you prefer, a data analyst who moonlights as a taxi driver), and we can choose between one rider and a group of riders, which is likelier to go farther—and thus pay us more?

As an example, let's go back to the data frame of products that we created back in chapter 2:

```
df = DataFrame([{'product_id':23, 'name':'computer',
                 'wholesale_price': 500,
                 'retail_price':1000, 'sales':100,
                 'department':'electronics'},
                {'product_id':96, 'name':'Python Workout',
                 'wholesale_price': 35,
                 'retail_price':75, 'sales':1000,
                 'department':'books'},
                {'product_id':97, 'name':'Pandas Workout',
                 'wholesale_price': 35,
                 'retail_price':75, 'sales':500,
                 'department':'books'},
                {'product_id':15, 'name':'banana',
                 'wholesale_price': 0.5,
                 'retail_price':1, 'sales':200,
                 'department':'food'},
                {'product_id':87, 'name':'sandwich',
                 'wholesale_price': 3,
                 'retail_price':5, 'sales':300,
                 'department': 'food'},
                ])
```

As you may have noticed, we've modified the data frame ever so slightly by adding a new column, `department`, that contains a string value. We'll use it in a moment.

To determine how many products we sell in a store (i.e., how many rows are in the data frame), we can use the `count` method:

```
df.count()
```

This is certainly interesting and useful information, but we may want to break it down further. For example, how many products are we selling in each department? To answer that question, we use the `groupby` method on our data frame:

```
df.groupby('department')
```

Notice that the argument to `groupby` needs to be the name of a column. And the result of running the `groupby` method?

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x13174f970>
```

We get a `DataFrameGroupBy` object, which is useful because of the aggregate methods we can invoke on it. For example, we can call `count` and thus determine how many items we have in each department:

```
df.groupby('department').count()
```

The result of this code is a data frame whose columns are the same as `df` and whose rows are the different values in the `department` column. Because there are three distinct departments in our store, there are three rows: `electronics`, `books`, and `food`.

Much of the time we don't want all the columns returned to us, just a subset of them. We could, in theory, use square brackets on the result of this code. For example, we could count `product_id`:

```
df.groupby('department').count()['product_id']
```

The result is a series whose index contains the different values in `department` and whose values contain the count of items per department. And the answer is accurate.

However, this code is unnecessarily wasteful. We first apply `count` to the `DataFrameGroupBy` object and only after that remove all columns by `product_id`. It's far more efficient, especially with a large data frame, to apply the square brackets to the `DataFrameGroupBy` object and then invoke our method:

```
df.groupby('department')['product_id'].count()
```

You can see this visually at **http://mng.bz/84nw**. Again, we get the same results—but this second version runs more quickly.

Although we've used `count` in the examples here, we can use any aggregation method when grouping, such as `mean`, `std`, `min`, `max`, or `sum`. So we can get the average product price per department in our store as follows:

```
df.groupby('department')['retail_price'].mean()
```

What if we want to know both the mean and the standard deviation of prices in our store, grouped by department? We can do that by altering the syntax somewhat: instead of calling an aggregation method directly, we can apply the `agg` method to our `DataFrameGroupBy` object. That method takes a list of methods, each of which is applied to the `GroupBy` object:

```
df.groupby('department')['retail_price'].agg(['mean', 'std'])
```

Notice that we pass a list of strings to `agg`. It used to be common to pass methods, such as `np.mean` and `np.std`, but that has fallen out of favor in recent years; the current standard is to pass strings and let pandas apply the appropriate methods.

Using `agg` this way returns a data frame with two columns (`mean` and `std`) and three rows (for each of the departments in our data frame). We can now determine the mean and standard

deviation for the retail prices in each department. You can see this visually in Pandas Tutor at **http://mng.bz/E9GO**.

What if we want to run multiple aggregations on separate columns? In such a case, we don't need to filter columns via square brackets. Rather, we can pass the entire `DataFrameGroupBy` object to `agg`. We then pass multiple keyword arguments to `agg`:

- The key of each keyword argument is the name of an output column.
- The value of each keyword argument is a two-element tuple:
  - The first element in the tuple is a string: the name of the column in the original data frame we want to analyze.
  - The second element in the tuple is also a string: the name (yes, as a string) of an aggregation method we wish to run on that column.

For example, we can get the mean and standard deviation of `retail_price` per department as well as find the max sales for each department:

```
df.groupby('department').agg(mean_price=('retail_price', 'mean'),
                             std_price=('retail_price', 'std'),
                            max_sales=('sales', 'max'))
```

Unsorting group keys

Normally, `groupby` sorts the group keys. If you don't want to see this, or if you are concerned that it's making your query too slow, you can pass `sort=False` to `groupby`:

```
df.groupby('department', sort=False)['retail_price'].agg(['mean', 'std'])
```

Exercise 30 • Taxi ride comparison

So far, we have taken several looks at our January 2019 taxi data. But we've always examined the overall data or effectively done manual grouping. In this exercise, we're going to use grouping to get a better understanding of the data. Specifically, I'd like you to

1. Load taxi data from January 2019 into a data frame using only the columns `passenger_count`, `trip_distance`, and `total_amount`.
2. For each number of passengers, find the mean cost of a taxi ride. Sort this result from lowest (i.e., cheapest) to highest (i.e., most expensive).
3. Sort the results again by increasing the number of passengers.

4. Create a new column, `trip_distance_group`, in which the values are `short` (< 2 miles), `medium` (≥ 2 miles and ≤ 10 miles), and `long` (> 10 miles). What is the average number of passengers per trip length category? Sort this result from highest (most passengers) to lowest (fewest passengers).

Working it out

Grouping is a simple idea, but it has profound implications. It means we can measure different parts of our data in a single query, producing a data frame that can itself then be analyzed, sorted, and displayed. In this exercise, we load the CSV file **nyc_taxi_2019-01.csv** into a data frame:

```
filename = '../data/nyc_taxi_2019-01.csv'

df = pd.read_csv(filename,
                 usecols=['passenger_count',
                          'trip_distance',
                          'total_amount'])
```

I then asked you to find the mean cost of a taxi ride for each number of passengers. When using `groupby`, we have to keep several things in mind:

- On what data frame are we operating?
- Which column will supply the groups? This column is almost always categorical in nature, either with a limited number of string values or with a limited set of integers (as is the case here). The distinct values from this column are the rows in the output from our aggregation method.
- Which column(s) do we analyze? That is, on which columns will we run our aggregation methods?
- Which aggregation method(s) will we be running?

In this case, the question provides all the answers:

- We'll work on the data frame `df`.
- We'll get our groups from `passenger_count`.
- We'll analyze `total_amount`.
- We'll run the `mean` method.

In other words, we're going to do the following:

```
df.groupby('passenger_count')['total_amount'].mean()
```

This returns a series. The index in the series contains each of the unique values in the `passenger_count` column. The values in the series are the result of running `mean` on each subset of `df['total_amount']`. You can think of this as similar to the following:

```
for i in range(df['passenger_count'].max() + 1):
    print(i,                                    ①
          df.loc[df['passenger_count'] == i,    ②
                 'total_amount'                 ③
          ].mean())                             ④
```

① Prints the current value of passenger_count

② Our row selector retrieves rows where passenger_count is i.

③ Our column selector retrieves the total_amount column.

④ Calculates the mean of total_amount for one value of passenger_count

This code uses a Python `for` loop to iterate over each value in `df['passenger_ count']` and then runs `mean` on that subset of the `total_amount` column. It calculates the same results, but it's far less efficient than using `groupby`. Moreover, it doesn't put the results in a data structure that we can use easily. For these and other reasons, it's almost never a good idea to use a `for` loop on pandas data structures—you should aim to use `groupby` and other native pandas functionality instead. That said, seeing this `for` loop can give us an idea of what's happening inside of the `groupby` and what values we get in the series it returns (figure 6.5).

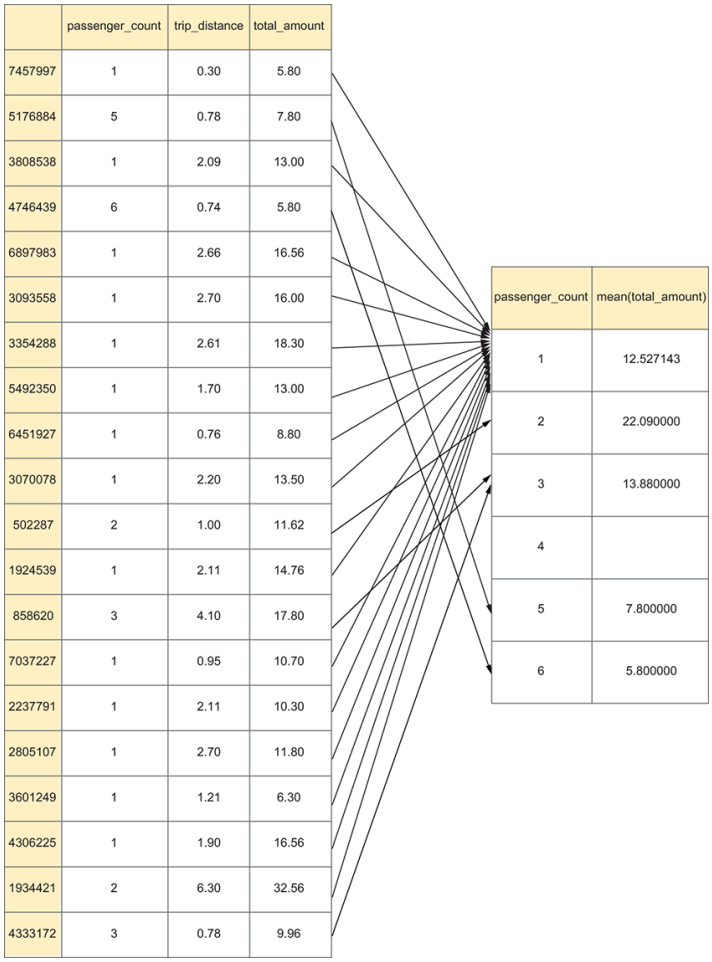


Figure 6.5 Graphical depiction of how `groupby` and then `mean` work together

Now that we have the mean price of a taxi fare for each number of passengers, we want to sort it by value in ascending order. We can do that by applying `sort_values` to the resulting series (figure 6.6):

```
df.groupby('passenger_count')[
    'total_amount'].mean().sort_values()
```

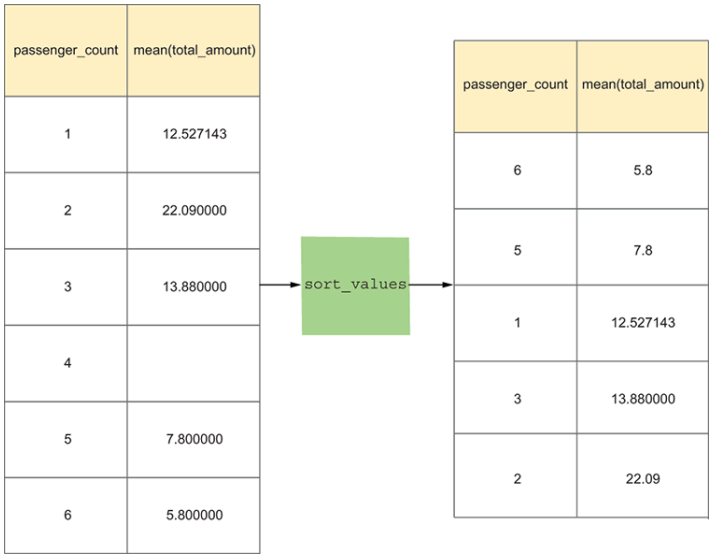


Figure 6.6 Graphical depiction of how you can run `sort_values` on the `groupby` result

The next request was for you to perform the same calculation but to sort the result by the number of passengers in ascending order. Remember that when we invoke `mean` on the grouped result, we get a series. The index of the series contains the unique values from `df['passenger_count']`. To sort by the number of passengers, we need to sort this series by its index:

```
df.groupby('passenger_count')[
    'total_amount'].mean().sort_index()
```

Next, I asked you to create a new column, `trip_distance_group`, whose values are `short`, `medium`, and `long`, corresponding to trips up to 2 miles, from 2 to 10 miles, and greater than 10 miles. We can accomplish this with `pd.cut`, which takes our column and lets us set the values we want to set as separators and the strings to assign to each category:

```
df['trip_distance_group'] = pd.cut(          ①
    df['trip_distance'],                     ②
    [df['trip_distance'].min(), 2, 10,
     df['trip_distance'].max()],             ③
    labels=['short', 'medium', 'long'],      ④
    include_lowest=True)                     ⑤
```

① Runs pd.cut to get categorical values from numeric ones

② Bases the categories on the trip_distance column

③ Our cut points are the minimum, 2, 10, and max.

④ Puts values into one of three categories: short, medium, or long

⑤ Ensures that the first category includes the left side

With this new column in place, we can use it in a `groupby` query. Specifically, I asked you to find the average number of passengers for each passenger group. We can do this as follows:

```
df.groupby('trip_distance_group')[
  'passenger_count'].mean().sort_values(ascending=False)
```

This says that we want to get the mean passenger count for each distinct value of `trip_distance_group`. We get those results back in a series, where the index is the distinct values of `trip_distance_group` and the values are the means we calculated for each trip-distance category.

Once we're done with those calculations, we sort the values of the resulting data frame in descending order. And in doing so, we find that there's very little difference between these averages. In other words, our moonlighting data scientist/taxi driver has no financial incentive to pick up a large group versus a small one because they'll likely get paid the same.

Solution

```
filename = '../data/nyc_taxi_2019-01.csv'

df = pd.read_csv(filename,
                 usecols=['passenger_count',
                          'trip_distance',
                          'total_amount'])

df.groupby('passenger_count')['total_amount'
    ].mean().sort_values()                          ①
df.groupby('passenger_count')['total_amount'
    ].mean().sort_index()                           ②

df['trip_distance_group'] = pd.cut(
    df['trip_distance'],
       [df['trip_distance'].min(), 2, 10,
        df['trip_distance'].max()],
    labels=['short', 'medium', 'long'])             ③
df.groupby('trip_distance_group')['passenger_count'
    ].mean().sort_values(ascending=False)           ④
```

① Returns the mean value of total_amount for each value of passenger_count and then sorts the resulting series by value (i.e., mean of total_amount)

② Returns the mean value of total_amount for each value of passenger_count and then sorts the resulting series by index (i.e., value of passenger_count)

③ Uses pd.cut to get a series of strings back from trip_distance, and assigns it to df['trip_distance_group']

④ For each value of trip_distance_group, gets the mean of passenger_count and sorts the values in descending order

You can explore a version of this in the Pandas Tutor at **http://mng.bz/NVN1**.

Beyond the exercise

- Create a single data frame containing rides from both January 2019 and January 2020, with a column `year` indicating which year the ride comes from. Use `groupby` to compare the average cost of a taxi in January from each of these two years.
- Create a two-level grouping, first by year and then by `passenger_count`.
- The `corr` method allows us to see how strongly two columns correlate with one another. Use `corr` and then `sort_values` to find which columns have the highest correlation.

Joining

Like grouping, joining is a concept you may have encountered previously when working with relational databases. The joining functionality in pandas is similar to that sort of a database, although the syntax is different.

Consider, for example, the data frame we looked at earlier in this chapter:

```
df = DataFrame([{'product_id':23, 'name':'computer',
                 'wholesale_price': 500,
                 'retail_price':1000, 'sales':100,
                 'department':'electronics'},
                {'product_id':96, 'name':'Python Workout',
                 'wholesale_price': 35,
                 'retail_price':75, 'sales':1000,
                 'department':'books'},
                {'product_id':97, 'name':'Pandas Workout',
                 'wholesale_price': 35,
                 'retail_price':75, 'sales':500,
                 'department':'books'},
                {'product_id':15, 'name':'banana',
                 'wholesale_price': 0.5,
                 'retail_price':1, 'sales':200,
                 'department':'food'},
                {'product_id':87, 'name':'sandwich',
                 'wholesale_price': 3,
                 'retail_price':5, 'sales':300,
                 'department': 'food'},
                ])
```

But now consider that instead of keeping track of sales numbers in this data frame, we instead break the data into two parts:

- One data frame will describe each of the products we sell.
- A second data frame will describe each sale we make.

Here is a simple example of how we can divide the data:

```
products_df = DataFrame([{'product_id':23, 'name':'computer',
                          'wholesale_price': 500,
                          'retail_price':1000,
                          'department':'electronics'},
                         {'product_id':96, 'name':'Python Workout',
                          'wholesale_price': 35,
                          'retail_price':75, 'department':'books'},
                         {'product_id':97, 'name':'Pandas Workout',
                          'wholesale_price': 35,
                          'retail_price':75, 'department':'books'},
                         {'product_id':15, 'name':'banana',
                          'wholesale_price': 0.5,
                          'retail_price':1, 'department':'food'},
                         {'product_id':87, 'name':'sandwich',
                          'wholesale_price': 3,
                          'retail_price':5, 'department': 'food'},
                         ])
sales_df = DataFrame([{'product_id': 23, 'date':'2021-August-10',
                       'quantity':1},
                      {'product_id': 96, 'date':'2021-August-10',
                       'quantity':5},
                      {'product_id': 15, 'date':'2021-August-10',
                       'quantity':3},
                      {'product_id': 87, 'date':'2021-August-10',
                       'quantity':2},
                      {'product_id': 15, 'date':'2021-August-11',
                       'quantity':1},
                      {'product_id': 96, 'date':'2021-August-11',
                       'quantity':1},
                      {'product_id': 23, 'date':'2021-August-11',
                       'quantity':2},
                      {'product_id': 87, 'date':'2021-August-12',
                       'quantity':2},
                      {'product_id': 97, 'date':'2021-August-12',
                       'quantity':6},
                      {'product_id': 97, 'date':'2021-August-12',
                       'quantity':1},
                      {'product_id': 87, 'date':'2021-August-13',
                       'quantity':2},
                      {'product_id': 23, 'date':'2021-August-13',
                       'quantity':1},
                      {'product_id': 15, 'date':'2021-August-14',
                       'quantity':2}
                      ])
```

What have we done here? We've put all our product information, which is less likely to change, in `products_df`. Every time we add a new product to the store or change the name or price of an existing product, we update that data frame. But each time we make a sale, we don't touch `products_df`. Rather, we add a new row to `sales_df`, describing which product was sold, how many we sold, and when we sold it. You can see these data frames in the following figures.

	product_id	name	wholesale_price	retail_price	department
0	23	computer	500.0	1000	electronics
1	96	Python Workout	35.0	75	books
2	97	Pandas Workout	35.0	75	books
3	15	banana	0.5	1	food
4	87	sandwich	3.0	5	food

Graphical depiction of `products_df`

This is all well and good, but how can we describe how much of each product has been sold? This is where joining comes in: we can combine `products_df` and `sales_df` into a new, single data frame that contains all the columns from both of the input data frames.

	product_id	date	quantity
0	23	2021-August-10	1
1	96	2021-August-10	5
2	15	2021-August-10	3
3	87	2021-August-10	2
4	15	2021-August-11	1
5	96	2021-August-11	1
6	23	2021-August-11	2
7	87	2021-August-12	2
8	97	2021-August-12	6
9	97	2021-August-12	1
10	87	2021-August-13	2
11	23	2021-August-13	1
12	15	2021-August-14	2

Graphical depiction of `sales_df`

But wait a second—how does pandas know which rows on the left should be joined with which rows on the right? The answer, at least by default, is that it uses the index. Wherever the index of

the left side matches the index of the right side, it joins them together, giving them a new row that contains all columns from both left and right.

This means we need to change our data frames such that both are using the same values for their indexes. The obvious choice here is `product_id`, which appears in both `products_df` and `sales_df` (see the following two figures):

```
products_df = products_df.set_index('product_id')
sales_df = sales_df.set_index('product_id')
```

product_id	name	wholesale_price	retail_price	department
23	computer	500.0	1000	electronics
96	Python Workout	35.0	75	books
97	Pandas Workout	35.0	75	books
15	banana	0.5	1	food
87	sandwich	3.0	5	food

Graphical depiction of `products_df` with `product_id` as the index

Now that our data frames have a common reference point in the index, we can create a new data frame combining the two:

```
products_df.join(sales_df)
```

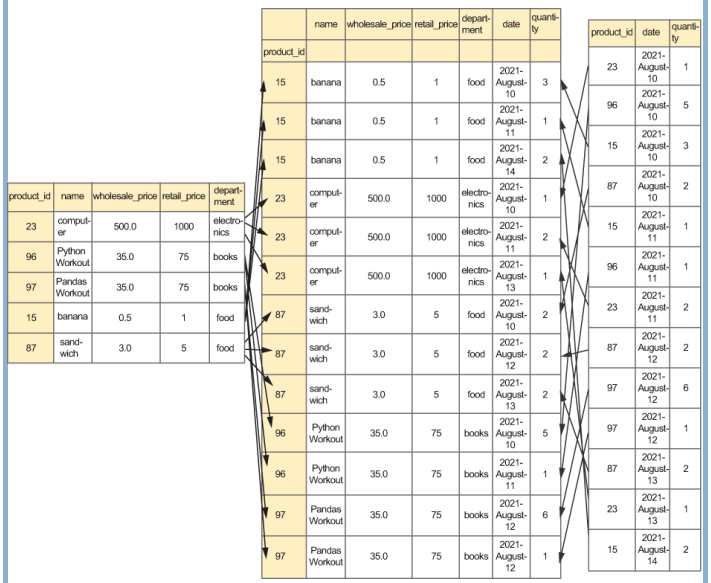
The result of this join is a new table with 13 rows and 6 columns. The columns combine all the columns from `products_df` and then all the columns from `sales_df`:

- `name`
- `wholesale_price`
- `retail_price`
- `department`
- `date`
- `quantity`

product_id	date	quantity
23	2021-August-10	1
96	2021-August-10	5
15	2021-August-10	3
87	2021-August-10	2
15	2021-August-11	1
96	2021-August-11	1
23	2021-August-11	2
87	2021-August-12	2
97	2021-August-12	6
97	2021-August-12	1
87	2021-August-13	2
23	2021-August-13	1
15	2021-August-14	2

Graphical depiction of `sales_df` with `product_id` as the index

Each row is the result of a match between the index (`product_id`) on the left (from `products_df`) and the index (`product_id`) on the right (from `sales_df`). Because several products have multiple sales, we end up with more rows than either of the original tables contained. The join is shown in the following figure.



Graphical depiction of joining `products_df` and `sales_df`

We can now perform whatever queries we like on this new, combined data frame. For example, we can determine how many

of each product have been sold:

```
products_df.join(sales_df).groupby(
    'name')['quantity'].sum()
```

Or we can determine how much income we get from each product and then sort the products from lowest to highest source of income:

```
products_df.join(sales_df).groupby(
    'name')['retail_price'].sum().sort_values()
```

We can even determine how much income we had on each individual day:

```
products_df.join(sales_df).groupby(
    'date')['retail_price'].sum().sort_index()
```

And although our data set is tiny, we can determine how much each product contributed to our income per day:

```
products_df.join(sales_df).groupby(
    ['date','name'])['retail_price'].sum().sort_index()
```

Separating data into two or more pieces so each piece of information appears only a single time is known as *normalization*. There are all sorts of formal theories and descriptions of normalization, but it boils down to keeping the information in separate places and joining data frames when necessary.

Sometimes you'll normalize your own data. But other times, you'll receive data that has been normalized and then separated into separate pieces. For example, many data sets are distributed in separate CSV files, which almost always means you'll need to join two or more data frames to analyze the information. Or you may want to normalize the data yourself to gain flexibility or performance.

One final point: the join I've shown you here is known as a *left join* because values of `product_id` on the left (i.e., in `products_df`) drive which rows are selected on the right (i.e., `sales_df`). More advanced joins called *outer joins* allow us to tell pandas that even if there isn't a corresponding row on the left or the right, we want to have a row in the result, albeit one filled with null values. We'll explore those in exercise 35 in the next chapter.

Exercise 31 • Tourist spending per country

Before the Covid-19 pandemic, I traveled internationally on a regular basis, both for work (giving classes to companies around the world) and for pleasure. The pandemic, of course, changed all that, with many countries restricting who could enter and leave and under what circumstances.

This was certainly a serious problem for corporate Python trainers. But it was an even bigger problem for the tourism industry. That's because tourists generate a great deal of money to countries around the world. In this exercise, we'll look at prepandemic data from the OECD (Organization for Economic Cooperation and Development), which the *Economist* describes as "a club of mostly-rich countries," to see how much they were earning in tourist dollars. As we'll see, the data covers countries beyond the OECD itself.

Here's what I would like you to do:

1. Load the OECD tourism data (from **oecd_tourism.oecd**) into a data frame. We're interested in the following columns:
    1. `LOCATION` —A three-letter abbreviation for the country name
    2. `SUBJECT` —Either `INT_REC` (for tourist funds received) or `INT-EXP` (for tourist expenses).
    3. `TIME` —A year (integer)
    4. `Value` —A float indicating thousands of dollars
2. Find the five countries that received the greatest amount of tourist dollars, on average, across years in the data set.
3. Find the five countries whose citizens spent the least amount of tourist dollars, on average, across years in the data set.
4. The separate CSV file **oecd_locations.csv** has two columns: one contains the three-letter abbreviated location name from the first CSV file, and the second is the full country name. Load this into a data frame, using the abbreviated data as an index.
5. Join these two data frames together into a new one. In the new data frame, there is no `LOCATION` column. Instead, there is a `name` column with the full name of the country.
6. Rerun the queries from steps 2 and 3, finding the five countries that spent and received the most, on average, from tourism. But this time, get the name of each country, rather than its abbreviation, for your reports.
7. Ignoring the names, did you get the same results as before? Why or why not?

NOTE The column names and values in this data set demonstrate the type of inconsistency that can creep into a project. The `SUBJECT` column can contain one of two strings, `INT_REC` or `INT-EXP`. Why does one use an underscore and the other a hyphen? Good question! Similarly, why are all column names in all caps, whereas `Value` has only its first letter capitalized? Another good question! This happens in many real-world data sets. Be on the lookout for these sorts of problems when you first see a data set.

And if you're creating a data set for others, try to keep things as consistent as possible.

Working it out

In this exercise, we create two separate data frames and then join them. In so doing, we create a report that uses countries' full names rather than three-letter abbreviations. Let's walk through each of the steps to achieve that.

For starters, I asked you to load the OECD tourism data into a data frame. This CSV file includes a number of columns that wouldn't help with our analysis, so I asked you to select a subset of them:

```
tourism_filename = '../data/oecd_tourism.csv'
tourism_df = pd.read_csv(tourism_filename,
                         usecols=['LOCATION',
                                  'SUBJECT',
                                  'TIME',
                                  'Value'])
```

This data frame, `tourism_df`, contains information about the total amount spent and the total amount received by a number of countries over about a decade. For example, if we want to determine how much money the French economy received in total from tourists during 2016, we can look at the row in which `SUBJECT` is `INT_REC`, `LOCATION` is `FRA`, and `TIME` is 2016. That returns a single row from the data frame; if we retrieve the `Value` column in that row, we learn the total amount of tourism income.

What if we want to determine the average amount of income that countries received in our data set? We can say, using method-chaining syntax,

```
(
    tourism_df
    .loc[tourism_df['SUBJECT'] == 'INT_REC']
    ['Value']
    .mean()
)
```

But this isn't very useful. (You could even say it isn't very "meaningful".) That's because countries differ in how much tourist income they receive. Breaking it apart by country gives many more insights than an overall mean.

How can we get the mean tourist income per country? By grouping the call to `mean` by the `LOCATION` column:

```
(
    tourism_df
    .loc[tourism_df['SUBJECT'] == 'INT_REC']
    .groupby('LOCATION')['Value']
```

```
    .mean()
)
```

Here's what we do in this code:

1. Select rows in which `SUBJECT` is `INT_REC`, for received tourism funds.
2. Group by `LOCATION`, meaning we'll get one result per value of `LOCATION`, aka country.
3. Ask for only the `Value` column.
4. Invoke the `mean` method on each location's values.

This produces a series: a single column in which the index contains the three-letter country abbreviations and with the values being the mean income per country.

I then asked you to find the five countries that received the most (on average per year) from tourism. To do this, we sort our results in descending order and then use `head` to get the five top-grossing locations:

```
(
    tourism_df
    .loc[tourism_df['SUBJECT'] == 'INT_REC']
    .groupby('LOCATION')['Value']
    .mean()
    .sort_values(ascending=False)
    .head()
)
```

Next, I asked you to perform a second, similar query, finding the countries that spent the least amount on tourism. In other words, we're now interested in the `INT-EXP` value from `SUBJECT`, and we want to look at the five lowest-spending (on average, per year) tourism countries. The solution is

```
(
    tourism_df
    .loc[tourism_df['SUBJECT'] == 'INT-EXP']
    .groupby('LOCATION')['Value']
    .mean()
    .sort_values()
    .head()
)
```

Beyond the difference in the string we're matching in `SUBJECT`, we also reverse the call to `sort_values,` using the default of an ascending sort. This way, `head` retrieves the five least-spending countries.

With these initial queries out of the way, we can now use `join` to make an easier-to-read report from what we've created. To help with that, we create a two-column CSV file that we can read. However, this CSV file needs massaging if we're going to use it. For

one thing, there isn't a header row, so we need to state that and provide our own names.

But we're also planning to use the imported data for joining with `tourism_df`. We'll want to use the three-letter country abbreviation for joining, so we may as well make that the index of `locations_df`. Here's what we do:

```
locations_filename = '../data/oecd_locations.csv'
locations_df = pd.read_csv(locations_filename,
                           header=None,
                            names=['LOCATION', 'NAME'],
                           index_col='LOCATION')
```

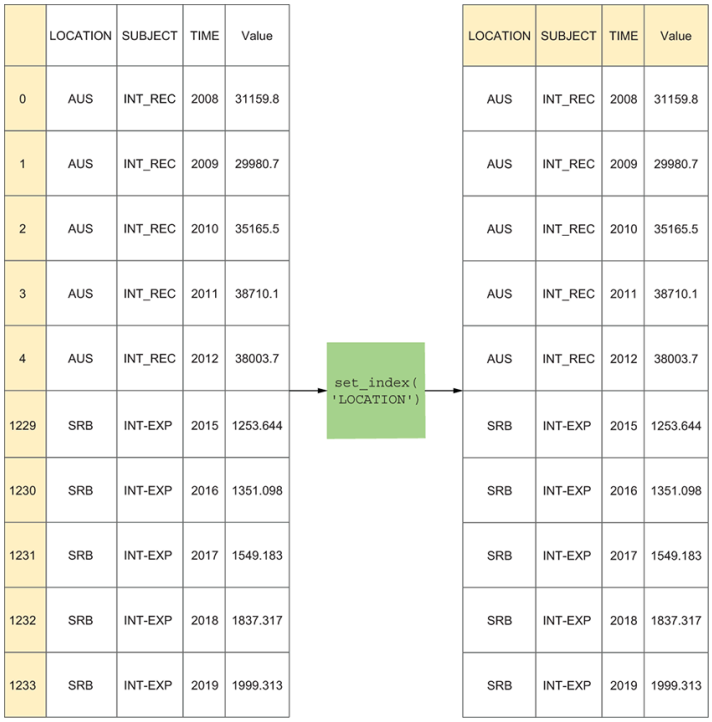
Now we'll bring this all together by creating a new data frame, the result of joining `locations_df` and `tourism_df`. The problem is that although the three-letter abbreviation (i.e., `LOCATION`) is the index of `locations_df`, it's just a plain ol' column in `tourism_df`. And yes, you can join non-index columns in pandas, but having the data frames share index values makes the code shorter and clearer.

We'll thus do the following:

1. Create a new (anonymous) data frame based on `tourism_df`, whose index is set to `LOCATION`.
2. Run `join` on `locations_df` and the new `LOCATION`-indexed version of `tourism_df`.
3. Assign this to a new data frame, which we call `fullname_df`.

You can see the setting of the indexes for our join in figures 6.7 and 6.8:

```
fullname_df = locations_df.join(tourism_df.set_index('LOCATION'))
```



	LOCATION	SUBJECT	TIME	Value
0	AUS	INT_REC	2008	31159.8
1	AUS	INT_REC	2009	29980.7
2	AUS	INT_REC	2010	35165.5
3	AUS	INT_REC	2011	38710.1
4	AUS	INT_REC	2012	38003.7
1229	SRB	INT-EXP	2015	1253.644
1230	SRB	INT-EXP	2016	1351.098
1231	SRB	INT-EXP	2017	1549.183
1232	SRB	INT-EXP	2018	1837.317
1233	SRB	INT-EXP	2019	1999.313

LOCATION	SUBJECT	TIME	Value
AUS	INT_REC	2008	31159.8
AUS	INT_REC	2009	29980.7
AUS	INT_REC	2010	35165.5
AUS	INT_REC	2011	38710.1
AUS	INT_REC	2012	38003.7
SRB	INT-EXP	2015	1253.644
SRB	INT-EXP	2016	1351.098
SRB	INT-EXP	2017	1549.183
SRB	INT-EXP	2018	1837.317
SRB	INT-EXP	2019	1999.313

Figure 6.7 Graphical depiction of making the `LOCATION` column the index of `tourism_df`

NOTE `fullname_df` is significantly smaller than `tourism_df` — 364 rows instead of 1,234. That's because the joined data frame's rows are the result of finding a match between the left and right sides of the join. `locations_df` doesn't include all the countries listed in `tourism_df`, so the result is smaller.

	LOCATION	NAME
0	AUS	Australia
1	AUT	Austria
2	BEL	Belgium
3	CAN	Canada
4	DNK	Denmark
5	FIN	Finland
6	FRA	France
7	DEU	Germany
8	HUN	Hungary
9	ITA	Italy
10	JPN	Japan
11	KOR	Korea
12	GBR	United Kingdom
13	USA	United States
14	BRA	Brazil
15	ISR	Israel

LOCATION	NAME
AUS	Australia
AUT	Austria
BEL	Belgium
CAN	Canada
DNK	Denmark
FIN	Finland
FRA	France
DEU	Germany
HUN	Hungary
ITA	Italy
JPN	Japan
KOR	Korea
GBR	United Kingdom
USA	United States
BRA	Brazil
ISR	Israel

Figure 6.8 Graphical depiction of making the `LOCATION` column the index of `locations_df`

The index of `fullname_df` is the three-character country codes. Its columns are

- `NAME` —The full name, which we get from `locations_df`
- `SUBJECT` —Tells us whether we're dealing with income or expenses
- `TIME` —Tells us the year in which the measurement was taken
- `Value` —Tells us the dollar amount that was measured

By using `NAME` for our grouping operations, we can get a report that displays each country's full name rather than the three-letter abbreviation. And indeed, I asked you to rerun our earlier queries on the result of our join (figure 6.9).

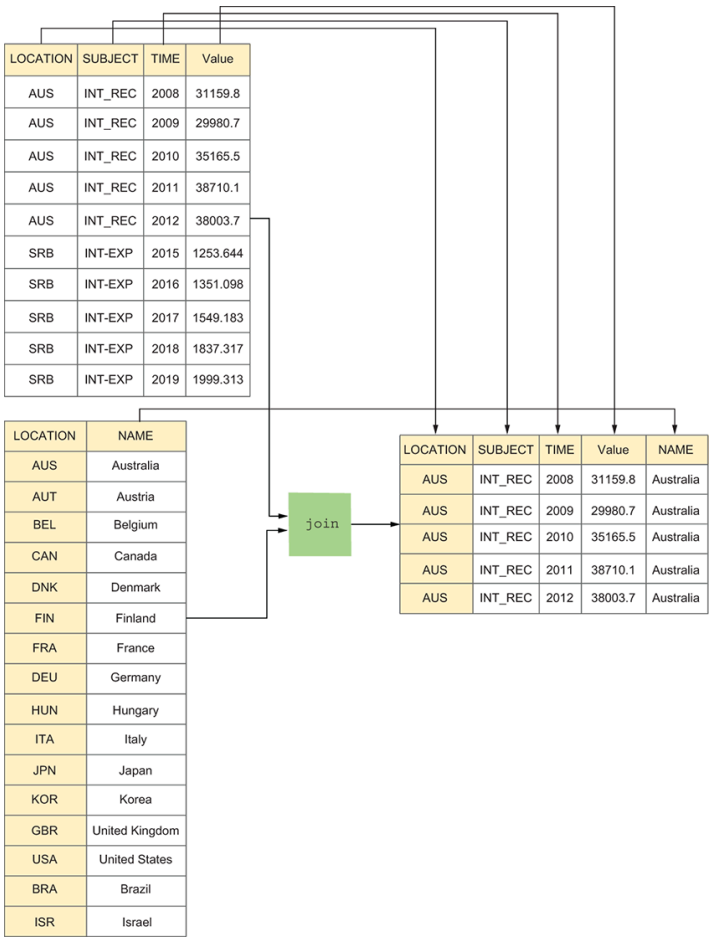


Figure 6.9 Graphical depiction of joining `locations_df` and `tourism_df`. Notice that any rows referring to a country not in `locations_df` are dropped from the result.

Here's how we can get the five countries with the greatest income from tourism, on average, during the years of the data set:

```
(
    fullname_df
    .loc[fullname_df['SUBJECT'] == 'INT_REC']
    .groupby('NAME')['Value']
    .mean()
    .sort_values(ascending=False)
    .head()
)
```

And here are the five countries that spent the least on tourism, on average, during the years of the data set:

```
(
    fullname_df
    .loc[fullname_df['SUBJECT'] == 'INT-EXP']
    .groupby('NAME')['Value']
    .mean()
    .sort_values()
    .head()
)
```

Finally, I asked whether the results are the same as before. Besides the obvious, that these results give the countries' full names rather than their abbreviations, the countries themselves are different. That's a result of `locations_df` not including all the countries in `tourism_df`. We lose some data as a result of our join.

Solution

```
tourism_filename = '../data/oecd_tourism.csv'
tourism_df = pd.read_csv(tourism_filename,                ①
                         usecols=['LOCATION',
                         'SUBJECT', 'TIME', 'Value'])

(
    tourism_df
    .loc[tourism_df['SUBJECT'] == 'INT_REC']
    .groupby('LOCATION')['Value']
    .mean()
    .sort_values(ascending=False)
    .head()
)                                                         ②

(
    tourism_df
    .loc[tourism_df['SUBJECT'] == 'INT-EXP']
    .groupby('LOCATION')['Value']
    .mean()
    .sort_values()
    .head()
)                                                         ③

locations_filename = '../data/oecd_locations.csv'
locations_df = pd.read_csv(locations_filename,
                           header=None,
                            names=['LOCATION', 'NAME'],
                           index_col='LOCATION')          ④
fullname_df = locations_df.join(
        tourism_df.set_index('LOCATION'))                 ⑤

(
    fullname_df
    .loc[fullname_df['SUBJECT'] == 'INT_REC']
    .groupby('NAME')['Value']
    .mean()
    .sort_values(ascending=False)
    .head()
)                                                         ⑥
```

```
(
    fullname_df
    .loc[fullname_df['SUBJECT'] == 'INT-EXP']
    .groupby('NAME')['Value']
    .mean()
    .sort_values()
    .head()
)
```
⑦

① Creates a data frame from four columns in the tourism data

② Chooses rows where SUBJECT is INT_REC; for each location (i.e., country), gets the mean value in the data set.; sorts those values in descending order, and takes the top five values

③ Chooses rows where SUBJECT is INT-EXP; for each location (i.e., country), gets the mean value in the data set; sorts those values in ascending order, and takes the top five values

④ Creates a data frame from the locations data, setting column names to LOCATION and NAME and making LOCATION the index.

⑤ Creates a new data frame, the result of joining tourism_df and locations_df

⑥ In the joined data, chooses rows where SUBJECT is INT_REC; for each location (i.e., country), gets the mean value in the data set; sorts those values in descending order, and takes the top five values

⑦ Chooses rows where SUBJECT is INT-EXP; for each location (i.e., country), gets the mean value in the data set; sorts those values in ascending order, and takes the top five values

You can explore a version of this in the Pandas Tutor at **http://mng.bz/D9Yw**.

Beyond the exercise

- What happens if you perform the join in the other direction? That is, what if you invoke `join` on `tourism_df`, passing it an argument of `locations_df`? Do you get the same result?
- Get the mean tourism income per year rather than by country. Do you see any evidence of less tourism income during the time of the Great Recession, which started in 2008?
- Reset the index on `locations_df` such that it has a (default) numeric index and two columns (`LOCATION` and `NAME`). Now run `join` on `locations_df`, specifying that you want to use the `LOCATION` column on the caller rather than its index. (The data frame passed as an argument to `join` will always be joined on its index.)

Summary

Once you've read data into a data frame, there are many ways in which you can split, combine, and analyze it. In this chapter, we looked at some of the most common tasks—from grouping for analysis, to grouping for including/excluding rows, to joining and merging data frames. Having these skills at your fingertips makes it easy to perform particularly complex types of analysis. The exercises in this chapter showed you how and when you can use these tools to explore your data in ways that analysts perform on a regular basis, with the "split-apply-combine" approach that's pervasive in pandas. In the next chapter, we'll build on what we've done here, using these techniques in more advanced ways.