4 Indexes

My parents introduced me to the wonders of the public library at a young age. It held an immense number of books on every subject you could imagine.

But wait: with so many books on so many subjects by so many authors, how can you possibly find what you want or even know what's available? The answer is an index. In those days, libraries typically had three different indexes found in the card catalog (hundreds of drawers full of index cards). These cards allowed you to find books (a) by author, (b) by title, or (c) by subject. Beyond that, the books were shelved according to their subjects, using either the Dewey decimal system or the Library of Congress system. If you were familiar with these systems, you could easily find what you were looking for: a particular book that had been mentioned in the newspaper, books written by your favorite author, or books on a particular subject you were researching for school. Nowadays, of course, the indexes are computerized, allowing you to find books more flexibly and easily than we ever imagined in the days of the card catalog.

Could you have a library without an index? Yes, but it would be much less useful. It would be harder to find what you want, and every search would take significantly longer. How to best catalog information so it's easily findable is so important that an entire branch of academia, library science, is dedicated to it.

Just as an index can help us find books in a library, it can help us find data in pandas. We've already seen that a series has one index (for its elements) and a data frame has two (one for the rows and a second for the columns). We've seen how `.loc`, along with row selectors and column selectors, can be powerful.

But indexes in pandas are far more flexible than we've seen so far: we can make an existing column into an index or turn an index back into a regular column. We can combine multiple columns into a hierarchical *multi-index* and then perform searches on specific parts of that hierarchy. Indeed, knowing how to create, query, and manipulate multi-indexed data frames is key to fluent work with pandas. We can also create *pivot tables* in which the rows and columns reflect not our original data, but rather aggregate summaries of that data.

In this chapter, we'll practice using all these techniques to better understand how to create, modify, and manipulate various types of indexes. After working through these exercises, you'll know how to use pandas indexes to retrieve data more flexibly and easily.

Table 4.1 What you need to know

Concept	What is it?	Example	To learn more
`pd.set_index`	Returns a new data frame with a new index	`df = df.set_index('name')`	http://mng.bz/MBd2
`pd.reset_index`	Returns a new data frame with a default (numeric, positional) index	`df = df.reset_index()`	http://mng.bz/a1RJ
`df.loc`	Retrieves selected rows and columns	`df.loc[:, 'passenger_count'] = df['passenger_count']`	http://mng.bz/e1QJ
`s.value_counts`	Returns a sorted (descending frequency) series counting how many times each value appears in `s`	`s.value_counts()`	http://mng.bz/Y1r7
`s.isin`	Returns a boolean series indicating whether a value in `s` is an element of the argument	`s.isin(['A', 'B', 'C'])`	http://mng.bz/9D08

`df.pivot`	Creates a pivot table based on a data frame without aggregation	`df.pivot(index='month', columns='year', values='A')`	http://mng.bz/zXjZ
`df.pivot_table`	Creates a pivot table based on a data frame, *with* aggregation, if needed	`df.pivot_table(index='month', columns= 'year', values='A')`	http://mng.bz/0K4z
`s.is_monotonic_increasing`	Contains `True` if values in the series are sorted in increasing order	`s.is_monotonic_increasing`	http://mng.bz/Ke2n
`slice`	Python builtin for creating slices	`slice(10, 20, 2)`	http://mng.bz/278g
`df.xs`	Returns a cross-section from a data frame	`df.xs(2016, level='Year')`	http://mng.bz/jPg9
`df.dropna`	Returns a new data frame without any `NaN` values	`df.dropna()`	http://mng.bz/o1PN
`IndexSlice`	Produce an object for easier	`IndexSlice[:, 2016]`	http://mng.bz/WzPX

	querying of data frames using `xs`

Exercise 21 • Parking tickets

We have already seen numerous examples of retrieving one or more rows from a data frame using `loc`. We don't necessarily *need* to use the index to select rows from a data frame, but it does make things easier to understand and yields clearer code. For this reason, we often want to use one of a data frame's existing columns as an index. Sometimes we want to do this permanently, and other times we want to do it briefly to clarify our queries.

In this exercise, I'll ask you to perform some queries on another data set from New York City: one that tracked all parking tickets during the year 2020—more than 12 million of them. You could, in theory, perform these queries without modifying the data frame's index. However, I want you to get some practice setting and resetting the index. We're going to do that a lot in this chapter, and you'll likely do it a great deal as you work with pandas with real-life data sets.

With that in mind, I want you to do the following:

1. Create a data frame from the file **nyc-parking-violations-2020.csv**. We are only interested in a handful of the columns:
   1. `Date First Observed`
   2. `Plate ID`
   3. `Registration State`
   4. `Issue Date` (a string in MM/DD/YYYY format, always followed by `12:00:00 AM`)
   5. `Vehicle Make`
   6. `Street Name`
   7. `Vehicle Color`
2. Set the data frame's index to the `Issue Date` column.

3. Determine what three makes were most frequently ticketed on January 2, 2020.
4. Determine the five streets on which cars got the most tickets on June 1, 2020.
5. Set the index to `Vehicle Color`.
6. Determine the most common make of vehicles that were either red or blue.

Working it out

We have already seen that to retrieve rows from a data frame that meet a particular condition, we can use a boolean series as a mask index. Often, especially if we are looking for specific values from a column, it makes more sense to turn that column into the data frame's index, reducing our code's complexity and length. Pandas makes it easy to do this with the `set_index` method. In this exercise, I asked you to make several queries against the data set of New York City parking tickets in 2020 and set the index to do this.

First, we read the data from a CSV file, limiting the columns from the input file:

```
filename = '../data/nyc-parking-violations-2020.csv'

df = pd.read_csv(filename,
                 usecols=[
                     'Date First Observed',
                     'Registration State', 'Plate ID',
                     'Issue Date', 'Vehicle Make',
                     'Street Name', 'Vehicle Color'])
```

Once the data frame is loaded, we perform several queries based on the parking tickets' issue date. So, it makes sense to set the index to the `Issue Date` column, as shown in figure 4.1:

```
df = df.set_index('Issue Date')
```

Notice that `set_index` returns a new data frame based on the original one, which we assign back to `df`. As of this point, if we make queries that involve the index (typically using `loc`), they will be based on the value of the issue date. Also, as far as the data frame is concerned, there is no longer an `Issue Date` column! Its identity as a named column is largely gone. Some pandas methods (e.g., `groupby`) can find and work with an index via its original name, but many others cannot.

NOTE As of this writing, the `set_index` method (along with many others in pandas) supports the `inplace` parameter. If you call `set_index` and pass `inplace=True`, the method will return `None` and will modify the data frame on which it was invoked. The core pandas developers have warned that this is a bad idea because it makes incorrect assumptions about memory and performance. They say there is no benefit to using `inplace=True`. Moreover, getting a new data frame back allows for method chaining, making long queries more readable. As a result, the `inplace` parameter will probably go away in a future version of pandas. Thus, although it may seem wasteful to call `set_index` and then assign its result back to `df`, this is the preferred, idiomatic way to do things.

With this index in place, it's relatively straightforward to find all the tickets issued on January 2:

```
df.loc['01/02/2020 12:00:00 AM']
```

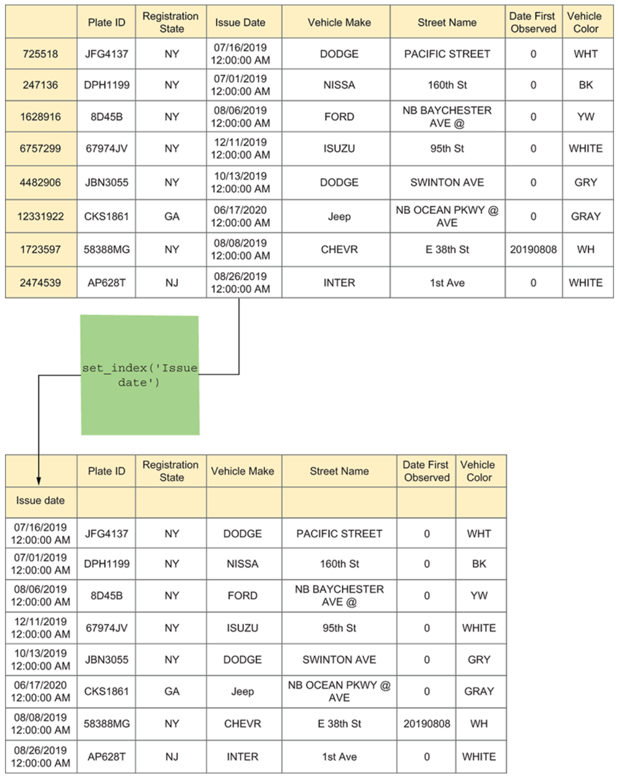


	Plate ID	Registration State	Issue Date	Vehicle Make	Street Name	Date First Observed	Vehicle Color
725518	JFG4137	NY	07/16/2019 12:00:00 AM	DODGE	PACIFIC STREET	0	WHT
247136	DPH1199	NY	07/01/2019 12:00:00 AM	NISSA	160th St	0	BK
1628916	8D45B	NY	08/06/2019 12:00:00 AM	FORD	NB BAYCHESTER AVE @	0	YW
6757299	67974JV	NY	12/11/2019 12:00:00 AM	ISUZU	95th St	0	WHITE
4482906	JBN3055	NY	10/13/2019 12:00:00 AM	DODGE	SWINTON AVE	0	GRY
12331922	CKS1861	GA	06/17/2020 12:00:00 AM	Jeep	NB OCEAN PKWY @ AVE	0	GRAY
1723597	58388MG	NY	08/08/2019 12:00:00 AM	CHEVR	E 38th St	20190808	WH
2474539	AP628T	NJ	08/26/2019 12:00:00 AM	INTER	1st Ave	0	WHITE

	Plate ID	Registration State	Vehicle Make	Street Name	Date First Observed	Vehicle Color
Issue date						
07/16/2019 12:00:00 AM	JFG4137	NY	DODGE	PACIFIC STREET	0	WHT
07/01/2019 12:00:00 AM	DPH1199	NY	NISSA	160th St	0	BK
08/06/2019 12:00:00 AM	8D45B	NY	FORD	NB BAYCHESTER AVE @	0	YW
12/11/2019 12:00:00 AM	67974JV	NY	ISUZU	95th St	0	WHITE
10/13/2019 12:00:00 AM	JBN3055	NY	DODGE	SWINTON AVE	0	GRY
06/17/2020 12:00:00 AM	CKS1861	GA	Jeep	NB OCEAN PKWY @ AVE	0	GRAY
08/08/2019 12:00:00 AM	58388MG	NY	CHEVR	E 38th St	20190808	WH
08/26/2019 12:00:00 AM	AP628T	NJ	INTER	1st Ave	0	WHITE

Figure 4.1 Graphical depiction of turning `Issue Date` from a column into the index

However, this also returns all the columns. And the first question we're trying to answer with this newly reindexed data frame is which vehicle makes received the most tickets on January 2. Let's limit the results of our query to the `Vehicle Make` column:

```
df.loc['01/02/2020 12:00:00 AM', 'Vehicle Make']
```

Once again, we see that the two-argument form of `loc` means first passing a row selector and then passing a column selector. In this case, we're only interested in a single column, `Vehicle Make`.

But we're still not done: how can we find the three most commonly ticketed vehicle makes on January 2? We use the `value_counts` method:

```
df.loc['01/02/2020 12:00:00 AM', 'Vehicle Make'].value_counts()
```

This returns a series in which the index contains the different vehicle makes and the values are the counts, sorted from highest to lowest. We can limit our results to the three most common makes by adding `head(3)` to our call:

```
df.loc['01/02/2020 12:00:00 AM', 'Vehicle Make'].value_counts().head(3)
```

Once we have this information, we can also check other columns. For example, on what five streets were the most tickets issued on June 1?

```
df.loc['06/01/2020 12:00:00 AM', 'Street Name'].value_counts().head(5)
```

Again, we select rows via the index and then select a column. We pass this to `value_counts` and get the top five results.

But now we want to make queries against the `Vehicle Color` column. We thus need to remove `Issue Date` as the index and put `Vehicle Color` in its place. We could, in theory, do this in two lines of code:

```
df = df.reset_index()
df = df.set_index('Vehicle Color')
```

Thanks to method chaining, we can do it in a single line of code:

```
df = df.reset_index().set_index('Vehicle Color')
```

We could equivalently split it up across several lines, using parentheses (see figure 4.2):

```
df = (
    df
    .reset_index()
    .set_index('Vehicle Color')
    )
```

The information in our data frame hasn't changed, but the index has—thus giving us easier access to the data from this perspective.

That will come in handy when answering the next question, which asks which vehicle make received the most parking tickets, if we only consider blue and red cars.

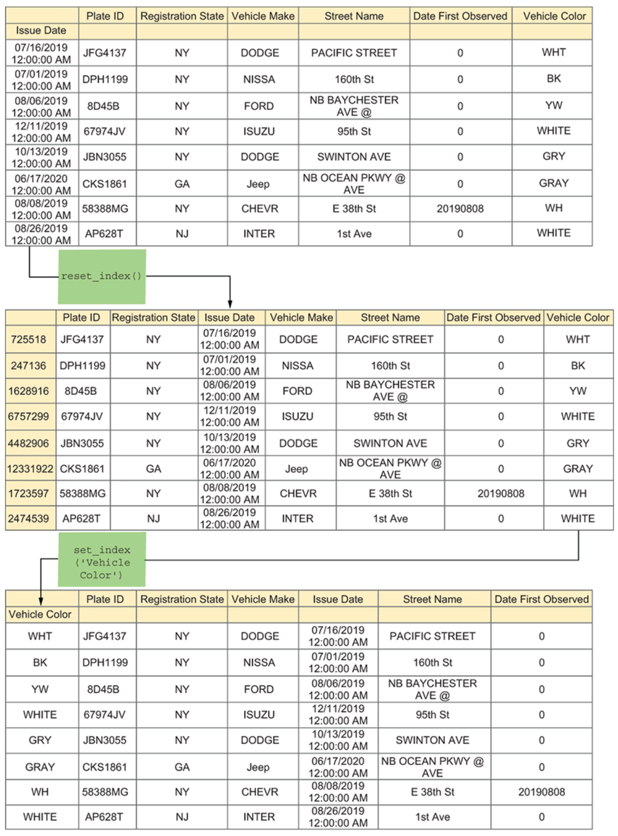


	Plate ID	Registration State	Vehicle Make	Street Name	Date First Observed	Vehicle Color
Issue Date						
07/16/2019 12:00:00 AM	JFG4137	NY	DODGE	PACIFIC STREET	0	WHT
07/01/2019 12:00:00 AM	DPH1199	NY	NISSA	160th St	0	BK
08/06/2019 12:00:00 AM	8D45B	NY	FORD	NB BAYCHESTER AVE @	0	YW
12/11/2019 12:00:00 AM	67974JV	NY	ISUZU	95th St	0	WHITE
10/13/2019 12:00:00 AM	JBN3055	NY	DODGE	SWINTON AVE	0	GRY
06/17/2020 12:00:00 AM	CKS1861	GA	Jeep	NB OCEAN PKWY @ AVE	0	GRAY
08/08/2019 12:00:00 AM	58388MG	NY	CHEVR	E 38th St	20190808	WH
08/26/2019 12:00:00 AM	AP628T	NJ	INTER	1st Ave	0	WHITE

	Plate ID	Registration State	Issue Date	Vehicle Make	Street Name	Date First Observed	Vehicle Color
725518	JFG4137	NY	07/16/2019 12:00:00 AM	DODGE	PACIFIC STREET	0	WHT
247136	DPH1199	NY	07/01/2019 12:00:00 AM	NISSA	160th St	0	BK
1628916	8D45B	NY	08/06/2019 12:00:00 AM	FORD	NB BAYCHESTER AVE @	0	YW
6757299	67974JV	NY	12/11/2019 12:00:00 AM	ISUZU	95th St	0	WHITE
4482906	JBN3055	NY	10/13/2019 12:00:00 AM	DODGE	SWINTON AVE	0	GRY
12331922	CKS1861	GA	06/17/2020 12:00:00 AM	Jeep	NB OCEAN PKWY @ AVE	0	GRAY
1723597	58388MG	NY	08/08/2019 12:00:00 AM	CHEVR	E 38th St	20190808	WH
2474539	AP628T	NJ	08/26/2019 12:00:00 AM	INTER	1st Ave	0	WHITE

	Plate ID	Registration State	Vehicle Make	Issue Date	Street Name	Date First Observed
Vehicle Color						
WHT	JFG4137	NY	DODGE	07/16/2019 12:00:00 AM	PACIFIC STREET	0
BK	DPH1199	NY	NISSA	07/01/2019 12:00:00 AM	160th St	0
YW	8D45B	NY	FORD	08/06/2019 12:00:00 AM	NB BAYCHESTER AVE @	0
WHITE	67974JV	NY	ISUZU	12/11/2019 12:00:00 AM	95th St	0
GRY	JBN3055	NY	DODGE	10/13/2019 12:00:00 AM	SWINTON AVE	0
GRAY	CKS1861	GA	Jeep	06/17/2020 12:00:00 AM	NB OCEAN PKWY @ AVE	0
WH	58388MG	NY	CHEVR	08/08/2019 12:00:00 AM	E 38th St	20190808
WHITE	AP628T	NJ	INTER	08/26/2019 12:00:00 AM	1st Ave	0

Figure 4.2 Graphical depiction of returning `Issue Date` from the index to a column and making `Vehicle Color` the new index

First, we need to find only those cars that are blue or red. We can do that by passing a list to `loc`:

```
df.loc[['BLUE', 'RED']]
```

Once we’ve done that, we can apply a column selector:

```
df.loc[['BLUE', 'RED'], 'Vehicle Make']
```

This returns all the rows in the data frame that have a blue or red car, but only the `Vehicle Make` column. With that in place, we can use `value_counts` to find the most common make and restrict it to the top-ranking brand with `head(1)`:

```
(
    df
    .loc[['BLUE', 'RED'], 'Vehicle Make']
    .value_counts()
    .head(1)
)
```

Solution

```
filename = '../data/nyc-parking-violations-2020.csv'

df = pd.read_csv(filename,
    usecols=['Date First Observed', 'Registration State', 'Plate ID',
        'Issue Date', 'Vehicle Make', 'Street Name', 'Vehicle Color'])
df = df.set_index('Issue Date')                                    ①
df.loc['01/02/2020 12:00:00 AM',
       'Vehicle Make'].value_counts().head(3)                      ②
df.loc['06/01/2020 12:00:00 AM',
       'Street Name'].value_counts().head(5)                       ③
df = df.reset_index().set_index('Vehicle Color')                   ④
df.loc[['BLUE', 'RED'],
       'Vehicle Make'].value_counts().head(1)                      ⑤
```

① Sets the data frame's index to be the Issue Date column

② Finds all rows on January 2 and just the Vehicle Make column, then gets the first three elements from the resulting series

③ Finds all rows on January 2 and just the Street Name column, then gets the first five elements from the resulting series

④ Removes Vehicle Make from being an index and then sets Vehicle Color to be the index

⑤ Finds all rows with a color of red or blue and gets the Vehicle Make column, then gets the most common make

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

Beyond the exercise

Just as changing your perspective on a problem can often help you solve it, setting (or resetting) the index on a data frame can dramatically simplify the code you need to write. Here are some additional problems based on the data frame we created in this exercise:

- What three car makes were most often ticketed from January 2 through January 10?
- How many tickets did the second-most-ticketed car get in 2020? (And why am I not interested in the most-ticketed plate?) What state was that car from, and was it always ticketed in the same location?

Working with multi-indexes

Every data frame has an index, giving labels to the rows. We have already seen that we can use the `loc` accessor to retrieve one or more rows using the index. For example, we can say

```
df.loc['a']
```

to retrieve all the rows with the index value `a`. Remember that the index doesn't necessarily contain unique values; `loc['a']` may return a series of values representing a single row, but it also may return a data frame whose rows all have the index value `a`.

This sort of index often serves us well. But in many cases it's not enough. That's because the world is full of hierarchical information, or information that is easier to process if we make it hierarchical.

For example, every business wants to know its sales figures. But getting a single number doesn't let you analyze the information in a useful way. So you may want to break it down by product to know how well each product is selling well and which contributes the most to the bottom line. (We saw a version of this in exercise 8.) However, even that isn't enough; you probably want to know how well each product is selling per month. If your store has been around for a while, you may want to break it down even further than that, finding the quantity of each product sold per month, per year. A multi-index will let you do precisely that.

For example, let's create some random sales data for three products (cleverly called A, B, and C) over the 36 months from January 2018 through December 2020:

```
# let's assume 3 products * 3 years * 12 months = 108 sales figures

g = np.random.default_rng(0)
df = DataFrame(g.integers(0, 100, [36,3]),
              columns=list('ABC'))
df['year'] = [2018] * 12 + [2019] * 12 + [2020] * 12
df['month'] = """Jan Feb Mar Apr May Jun
                 Jul Aug Sep Oct Nov Dec""".split() * 3    (1)
```

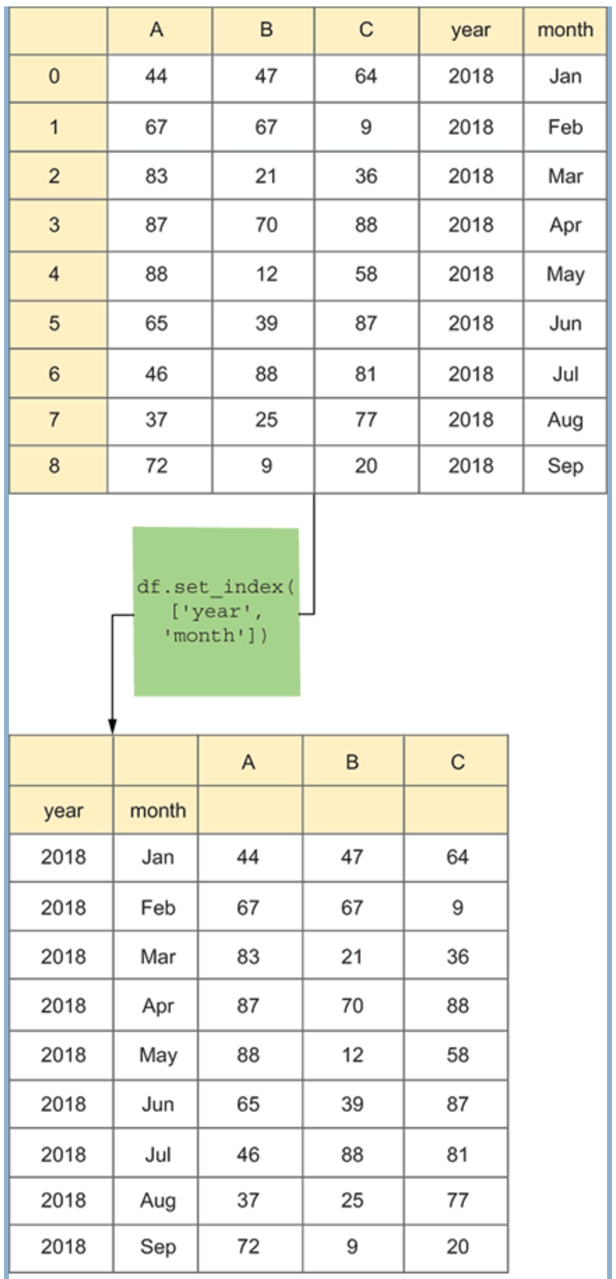
(1) Triple-quoted strings allow newlines in strings

We could set the index, based on the `year` column, as follows:

```
df = df.set_index('year')
```

But that wouldn't give us any special access to the month data, which we would like to have part of our index. We can create a multi-index by passing a list of columns to `set_index`:

```
df = df.set_index(['year', 'month'])
```



	A	B	C	year	month
0	44	47	64	2018	Jan
1	67	67	9	2018	Feb
2	83	21	36	2018	Mar
3	87	70	88	2018	Apr
4	88	12	58	2018	May
5	65	39	87	2018	Jun
6	46	88	81	2018	Jul
7	37	25	77	2018	Aug
8	72	9	20	2018	Sep

```
df.set_index(
['year',
'month'])
```

		A	B	C
year	month			
2018	Jan	44	47	64
2018	Feb	67	67	9
2018	Mar	83	21	36
2018	Apr	87	70	88
2018	May	88	12	58
2018	Jun	65	39	87
2018	Jul	46	88	81
2018	Aug	37	25	77
2018	Sep	72	9	20

Graphical depiction of creating a multi-index from the `year` and `month` columns

Remember that when creating a multi-index, we want the most general part to be on the outside and thus be mentioned first. If you create a multi-index with dates, you use year, month, and day, in that order. If you create a multi-index for your company's sales data, you might use region, country, and city to retrieve all rows for a given region, country, or city relatively easily. Usually (but not always), a multi-index reflects a hierarchy.

With this in place, we can retrieve one or more parts of the data frame in a variety of different ways. For example, we can get the sales data for all products in 2018:

```
df.loc[2018]
```

We can get all sales data for just products A and C in 2018:

```
df.loc[2018, ['A', 'C']]
```

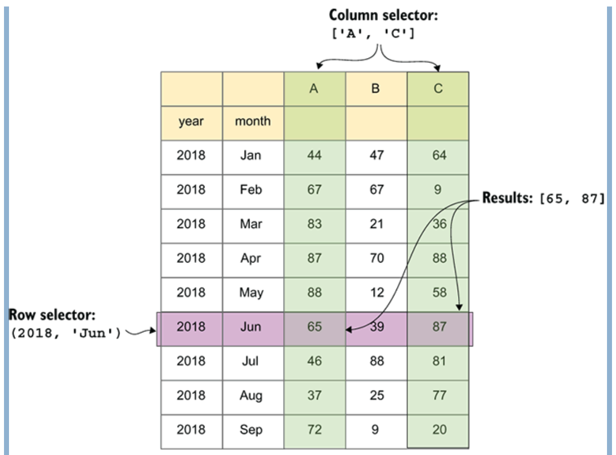
Notice that we're still applying the same rule we've always used with `loc`: the first argument describes the row(s) we want, and the second argument describes the column(s) we want. Without a second argument, we get all the columns.

We have a multi-index on this data frame, which means we can break the data down not just by year but also by month. For example, what did it look like for all three products in June 2018?

```
df.loc[(2018, 'Jun')]
```

We're still invoking `loc` with square brackets. However, the first (and only) argument is a tuple (i.e., round parentheses). Tuples are typically used in a multi-index situation when we want to specify a specific combination of index levels and values. For example, we're looking for 2018 and June—the outermost level and the inner level—so we use the tuple `(2018, 'Jun')`. We can, of course, retrieve the sales data just for products A and C:

```
df.loc[(2018, 'Jun'), ['A', 'C']]
```



		A	B	C
year	month			
2018	Jan	44	47	64
2018	Feb	67	67	9
2018	Mar	83	21	36
2018	Apr	87	70	88
2018	May	88	12	58
2018	Jun	65	39	87
2018	Jul	46	88	81
2018	Aug	37	25	77
2018	Sep	72	9	20

Graphical depiction of retrieving rows from June 2018, columns A and C, from a multi-index

What if we want to see more than one year at a time? For example, let's say we want all data for 2018 and 2020:

```
df.loc[[2018, 2020]]
```

And if we want all data for 2018 and 2020, but only products B and C?

```
df.loc[[2018, 2020], ['B', 'C']]
```

What if we want to get all the data from June in both 2018 and 2020? It's a little complicated:

- We use square brackets with `loc`.
- The first argument in the square brackets describes the rows we want (i.e., a row selector).
- We want all columns, so there isn't a second argument to `loc`.
- We want to select multiple combinations from the multi-index, so we need a list.
- Each year-month combination is a separate tuple in the list.

The result is

```
df.loc[[(2018, 'Jun'), (2020, 'Jun')]]
```

What if we want to look at all values from June, July, or August across all three years? We could, of course, do it manually:

```
df.loc[[(2018, 'Jun'), (2018, 'Jul'), (2018, 'Aug'),
        (2019, 'Jun'), (2019, 'Jul'), (2019, 'Aug'),
        (2020, 'Jun'), (2020, 'Jul'), (2020, 'Aug')]]
```

This works well, but it seems wordy. Is there another, shorter way? You might guess that we could tell pandas we want all the years (2018, 2019, and 2020) and only three months (June, July, and August) by writing the following:

```
df.loc[([2018, 2019, 2020], ['Jun', 'Jul', 'Aug'])]
```

But this won't work! It's rather surprising and confusing to find that it doesn't work, when it seems obvious and intuitive, given everything else we know about pandas. What's missing is an indicator of which columns we want:

```
df.loc[([2018, 2019, 2020], ['Jun', 'Jul', 'Aug']),
        ['A', 'B', 'C']]
```

Although the second argument (our column selector) is generally optional when using `loc`, here it isn't: we need to indicate which column, or columns, we want, along with the rows. Typically, you won't want all the columns because the analysis you'll want to do will involve a subset of the full data frame.

We can do this explicitly, as we did earlier, or we can use Python's "slice" syntax:

```
df.loc[([2018, 2019, 2020], ['Jun', 'Jul', 'Aug']),
       'A':'C']
```

For all columns, use a colon by itself:

```
df.loc[([2018, 2019, 2020], ['Jun', 'Jul', 'Aug']),
        :]
```

Assuming the index is sorted, we can even select the years using a slice:

```
df.loc[(:, ['Jun', 'Jul', 'Aug']), 'A':'B']      (1)
```

(1) This won't work!

Oh, wait—actually, we can't do that here, because Python only allows the colon within square brackets. We tried to use the colon within a tuple, which uses regular, round parentheses. Instead, we can use the builtin `slice` function with `None` as an argument for the same result:

```
df.loc[(slice(None), ['Jun', 'Jul', 'Aug']), 'A':'B']
```

And sure enough, that works. You can think of `slice(None)` as a way of indicating to pandas that you are willing to have all values as a wildcard.

As you can see, `loc` is extremely versatile, allowing us to retrieve from a multi-index various ways.

xercise 22 • State SAT scores

Setting the index can make it easier to create queries about our data. But sometimes our data is hierarchical in nature. That's where the pandas concept of a multi-index comes into play. With a multi-index, we can set the index not just to a single column but rather to multiple columns. Imagine, for example, a data frame containing sales data: we may want sales broken down by year and then further broken down by region. Once you use the phrase "further broken down by," a multi-index is almost certainly a good idea. (See the earlier sidebar “Working with multi-indexes” for a fuller description.)

In this exercise, we look at a summary of scores from the SAT, a standardized university admissions test widely used in the United States. The CSV file (sat-scores.csv) has 99 columns and 577 rows describing all 50 US states and three nonstates (Puerto Rico, the Virgin Islands, and Washington, DC) from 2005 through 2015.

In this exercise, I want you to

1. Read in the scores file, only keeping the `Year`, `State.Code`, `Total.Math`, `Total.Test-takers`, and `Total.Verbal` columns.
2. Create a multi-index based on the year and the two-letter state code.
3. Determine how many people took the SAT in 2005.
4. Determine the average SAT math score in 2010 from New York (`NY`), New Jersey (`NJ`), Massachusetts (`MA`), and Illinois (`IL`).
5. Determine the average SAT verbal score in 2012–2015 from Arizona (`AZ`), California (`CA`), and Texas (`TX`).

Working it out

In this exercise, you begin to discover the power and flexibility of a multi-index. I asked you to load the CSV file and create a multi-index based on the `Year` and `State.Code` columns. We can do this in two stages, first reading the file, including the columns we want, into a data frame and then choosing two columns to serve as our index:

```
filename = '../data/sat-scores.csv'

df = pd.read_csv(filename,
                 usecols=['Year', 'State.Code',
                 'Total.Math', 'Total.Test-takers',
                 'Total.Verbal'])
df = df.set_index(['Year', 'State.Code'])
```

As always, the result of `set_index` is a new data frame, which we assign back to `df`.

You may remember that `read_csv` also has an `index_col` parameter. If we pass an argument to that parameter, we can tell `read_csv` to do it all in one step—reading in the data frame and setting the index as the column we request. We can pass a list of columns as the argument to `index_col`, thus creating the multi-index as the data frame is collected. For example:

```
filename = '../data/sat-scores.csv'

df = pd.read_csv(filename,
                usecols=['Year', 'State.Code',
                'Total.Math', 'Total.Test-takers',
                'Total.Verbal'],
                index_col=['Year', 'State.Code'])
```

Now that we have loaded our data frame, we can explore our data and answer some questions.

First, we want to know how many people took the SAT in 2005. This means finding all rows from 2005 and the column `Total.Test-takers`, which tells us how many people took the test in each year, for each state, and summing those values:

```
df.loc[2005,                ①
       'Total.Test-takers'  ②
       ].sum()
```

① Row selector

② Column selector

Next, we want to determine the mean math score for students in four states—New York, New Jersey, Massachusetts, and Illinois—in 2010. As usual, we can use `loc` to retrieve the data that's of interest to us. But we need to combine three things to create the right query:

- From the first part (`Year`) of the multi-index, we only want 2010.
- From the second part (`State.Code`) of the multi-index, we only want `NY`, `NJ`, `MA`, and `IL`.
- From the columns, we are interested in `Total.Math`.

When retrieving from a multi-index, we need to put the parts together inside a tuple. Moreover, we can indicate that we want more than one value by using a list. The result is

```
df.loc[(2010, ['NY', 'NJ', 'MA', 'IL']),     ①
       'Total.Math'].mean()                  ②
```

① Multi-index row selector for rows in 2010 from four specific states

② Column selector, indicating just the Total.Math column

This query retrieves rows with a year of 2010 coming from any of those four states. We only get the `Total.Math` column, on which we then calculate the mean (figure 4.3).

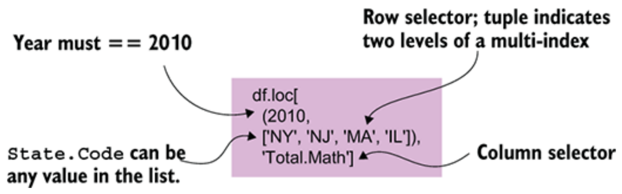

Year must == 2010
Row selector; tuple indicates two levels of a multi-index
df.loc[
(2010,
['NY', 'NJ', 'MA', 'IL']),
'Total.Math']
State.Code can be any value in the list.
Column selector

Figure 4.3 Graphical breakdown of `.loc` with a multi-index

The next question asks for a similar calculation but on several years and several states. Once again, that's not a problem if we think carefully about how to construct the query:

- From the first part (`Year`) of the multi-index, we want 2012, 2013, 2014, and 2015.
- From the second part (`State.Code`) of the multi-index, we want `AZ`, `CA`, and `TX`.
- From the columns, we are again interested in `Total.Math`.

The query then becomes

```
df.loc[([2012,2013,2014,2015], ['AZ', 'CA', 'TX']),   ①
       'Total.Math'].mean()                            ②
```

① Multi-index row selector for rows in 2012–2015 and three specific states

② Column selector, indicating just the Total.Math column

Notice how pandas figures out how to combine the parts of our multi-index so we get only the rows matching both parts.

Solution

```
filename = '../data/sat-scores.csv'

df = pd.read_csv(filename,
                 usecols=['Year',
                          'State.Code',
                          'Total.Math',
                          'Total.Test-takers',
                          'Total.Verbal'])

df = df.set_index(['Year', 'State.Code'])    ①
df.loc[2005, 'Total.Test-takers'].sum()      ②

df.loc[(2010, ['NY', 'NJ', 'MA', 'IL']),
       'Total.Math'].mean()                  ③

df.loc[([2012,2013,2014,2015],
        ['AZ', 'CA', 'TX']),
       'Total.Math'].mean()                  ④
```

① Sets the index to be a combination of Year and State.code

② Retrieves rows with 2005 and the column Total.Test-takers and then sums those values

③ Retrieves rows with 2010 and any of those four states and the column Total.Math and then gets the average

④ Retrieves rows from 2012–2015 with those three states and the column Total.Math and then gets the average

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

Beyond the exercise

- What were the average math and verbal scores for Florida, Indiana, and Idaho across all years? (Don’t break out the values by state.)
- Which state received the highest verbal score, and in which year?
- Was the average math score in 2005 higher or lower than that in 2015?

Sorting by index

When we talk about sorting in pandas, we’re usually referring to sorting the data. For example, we may want the rows in our data frame sorted by price or regional sales code. We’ll talk more about that kind of sorting in Chapters 6 and 7.

But pandas also lets us sort data frames based on the index. We can do that with the `sort_index` method, which (like so many others) returns a new data frame with the same content as the original, with rows sorted based on the index’s values. We can thus say

```
df = df.sort_index()
```

If your data frame contains a multi-index, the sorting will be done primarily along the first level, then along the second level, and so forth.

In addition to having some aesthetic benefits, sorting a data frame by index can make certain tasks easier or possible. For example, if you try to retrieve a slice, such as `df.loc['a':'c']`, pandas will insist that the index be sorted, to avoid problems if `a` and `c` are interspersed.

If your data frame is unsorted and has a multi-index, performing some operations may result in a warning:

```
PerformanceWarning: indexing past lexsort depth may impact performance
```

This is pandas trying to tell you that the combination of large size, multi-index, and an unsorted index is likely to cause you trouble. You can avoid the warning by sorting your data frame via its index.

If you want to check whether a data frame is sorted, you can check this attribute:

```
df.index.is_monotonic_increasing
```

Saying that the index is "monotonically increasing," by the way, simply means it only goes up. Similarly, if the values only go down, we say it's "monotonically decreasing," which we can check with `is_monotonic_decreasing`. Note that these are *not* methods but rather boolean attributes. They exist on all series objects, not just on indexes. Some older documentation and blogs mention the method `is_lexsorted`, which has been deprecated in recent versions of pandas.

xercise 23 • Olympic games

The modern-day Olympic games have been around for more than a century, and even people like me who rarely pay attention to sports are often excited to see a variety of international competitions take place. Fortunately, the Olympics aren't only about sports; they also generate a great deal of data, which we can enjoy and analyze using pandas.

In the previous exercise, we initially looked at building and using a multi-index. A multi-index doesn't have to stop at just two levels; pandas will, in theory, allow us to set as many as we want. Consider a large corporation that has broken down sales reports by region, country, and department; a multi-index would make it possible to retrieve that data in a variety of different ways, be it from the top of the hierarchy or by reaching "inside" the multi-index and creating a cross-regional departmental report.

In this exercise, we're going to build a deep multi-index, allowing us to retrieve data from various levels and in several ways. Specifically, I want you to do the following:

1. Read the data file (olympic_athlete_events.csv) into a data frame. We only care about some of the columns: `Age`, `Height`, `Team`, `Year`, `Season`, `City`, `Sport`, `Event`, and `Medal`. The multi-index should be based on `Year`, `Season`, `Sport`, and `Event`.
2. Answer these questions:
   1. What is the average age of winning athletes in summer games held between 1936 and 2000?
   2. What team has won the most medals in all archery events?
   3. Starting in 1980, what is the average height of the "Table Tennis Women's Team" event?
   4. Starting in 1980, what is the average height of both "Table Tennis Women's Team" and "Table Tennis Men's Team"?
   5. How tall was the tallest-ever tennis player in Olympic games from 1980 until 2016?

Working it out

In this exercise, we create a multi-index with four levels and then use those levels to ask and answer a variety of questions. The exercise shows you how powerful multi-indexes can be.

First, we have to load the data. As before, we load a subset of the columns and use four of them as a multi-index:

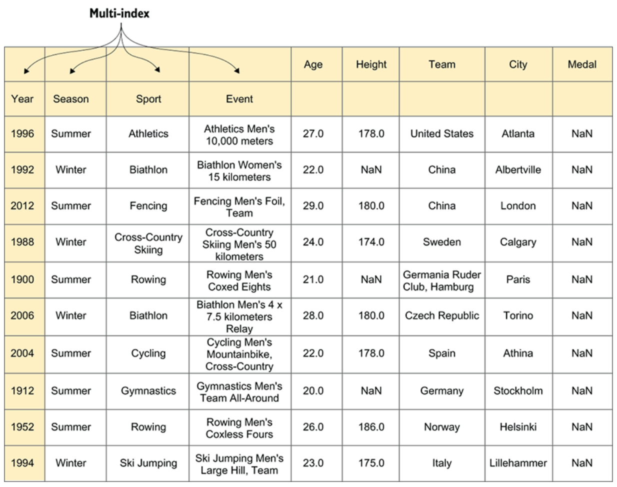
```
filename = '../data/olympic_athlete_events.csv'

df = pd.read_csv(filename,
                 index_col=['Year', 'Season',
                            'Sport', 'Event'],          ①
                 usecols=['Age', 'Height', 'Team',
                          'Year', 'Season', 'City',
                          'Sport', 'Event', 'Medal'])    ②
df = df.sort_index()                                     ③
```

① Specifies the components and order of the multi-index in index_col

② Reads the CSV file into a data frame with nine columns, four of which are used in our index

③ Sorts the rows of the data frame according to the index

By passing a list of columns to the `index_col` parameter, we create the multi-index while creating the data frame, rather than doing it in a separate second step (see figure 4.4).


Multi-index

				Age	Height	Team	City	Medal
Year	Season	Sport	Event					
1996	Summer	Athletics	Athletics Men's 10,000 meters	27.0	178.0	United States	Atlanta	NaN
1992	Winter	Biathlon	Biathlon Women's 15 kilometers	22.0	NaN	China	Albertville	NaN
2012	Summer	Fencing	Fencing Men's Foil, Team	29.0	180.0	China	London	NaN
1988	Winter	Cross-Country Skiing	Cross-Country Skiing Men's 50 kilometers	24.0	174.0	Sweden	Calgary	NaN
1900	Summer	Rowing	Rowing Men's Coxed Eights	21.0	NaN	Germania Ruder Club, Hamburg	Paris	NaN
2006	Winter	Biathlon	Biathlon Men's 4 x 7.5 kilometers Relay	28.0	180.0	Czech Republic	Torino	NaN
2004	Summer	Cycling	Cycling Men's Mountainbike, Cross-Country	22.0	178.0	Spain	Athina	NaN
1912	Summer	Gymnastics	Gymnastics Men's Team All-Around	20.0	NaN	Germany	Stockholm	NaN
1952	Summer	Rowing	Rowing Men's Coxless Fours	26.0	186.0	Norway	Helsinki	NaN
1994	Winter	Ski Jumping	Ski Jumping Men's Large Hill, Team	23.0	175.0	Italy	Lillehammer	NaN

Figure 4.4 Graphical depiction of our data frame with four columns in its multi-index

We then use `sort_index`, which returns a new data frame containing the same data we read from the CSV file but with the rows ordered according to the multi-index. When running `sort_index` on a multi-indexed data frame, we first index on the first level (i.e., `Year`), then on `Season`, then on `Sport`, and finally on `Event`.

NOTE You can invoke `set_index` with `inplace=True`. If you do, `set_index` will modify the existing data frame object and return `None`. But as with all other uses of `inplace=True` in pandas, the core developers strongly recommend against doing this. Instead, you should invoke it regularly (i.e., with a default value of `inplace=False`) and then assign the result to a variable—which could be the variable already referring to the data frame, as we do here.

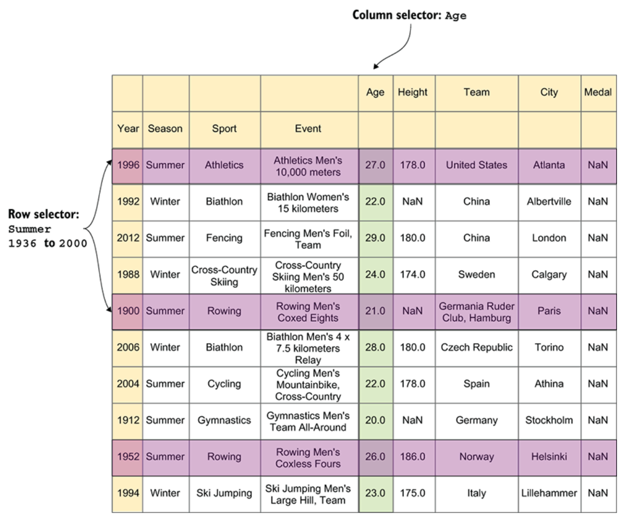
Although we don't necessarily need to sort our data frame by its index, certain pandas operations will work better if we do. Moreover, if we don't sort the data frame, we may get the `PerformanceWarning` mentioned earlier in this chapter. So, especially when we're doing operations with a multi-index, it's a good idea to sort by the index at the outset.

Now that we have our data frame, we can answer the questions I posed. For starters, I asked you to find the average age of winning athletes who participated in summer games held between 1936 and 2000. This means we want a subset of the years (i.e., the first level of our multi-index) and a subset of the seasons (i.e., just the games for which the second level of the multi-index, the `Season` column, has a value of `Summer`). We want all the values from the third and fourth levels of the multi-index, which means we can ignore them in our query; by ignoring them, we get all the values.

In other words, we want our query to retrieve the following (see figure 4.5):

- All years from 1936 to 2000, which we can express as `slice(1936,2000)`
- All games in which `Season` is set to `Summer`

- The `Age` column from the resulting data frame


Column selector: Age
Row selector: Summer 1936 to 2000

				Age	Height	Team	City	Medal
Year	Season	Sport	Event					
1996	Summer	Athletics	Athletics Men's 10,000 meters	27.0	178.0	United States	Atlanta	NaN
1992	Winter	Biathlon	Biathlon Women's 15 kilometers	22.0	NaN	China	Albertville	NaN
2012	Summer	Fencing	Fencing Men's Foil, Team	29.0	180.0	China	London	NaN
1988	Winter	Cross-Country Skiing	Cross-Country Skiing Men's 50 kilometers	24.0	174.0	Sweden	Calgary	NaN
1900	Summer	Rowing	Rowing Men's Coxed Eights	21.0	NaN	Germania Ruder Club, Hamburg	Paris	NaN
2006	Winter	Biathlon	Biathlon Men's 4 x 7.5 kilometers Relay	28.0	180.0	Czech Republic	Torino	NaN
2004	Summer	Cycling	Cycling Men's Mountainbike, Cross-Country	22.0	178.0	Spain	Athina	NaN
1912	Summer	Gymnastics	Gymnastics Men's Team All-Around	20.0	NaN	Germany	Stockholm	NaN
1952	Summer	Rowing	Rowing Men's Coxless Fours	26.0	186.0	Norway	Helsinki	NaN
1994	Winter	Ski Jumping	Ski Jumping Men's Large Hill, Team	23.0	175.0	Italy	Lillehammer	NaN

Figure 4.5 Graphical depiction of applying our multi-index row selector

Finally, we want to find the mean of those ages. We can express this as

```
df.loc[(slice(1936,2000), 'Summer'),    ①
       'Age'                            ②
       ].mean()                         ③
```

① Row selector: years 1936–2000, and summer games from the first two parts of the multi-index

② Column selector: we only want the Age column.

③ Applies mean to the resulting series

The answer is a float, 25.026883940421765.

Next, I asked you to find which team won the most medals in all archery events. How do we construct this query? We need to think through each level in our multi-index:

- We're interested in all years, so we specify `slice(None)` for the first index level.
- Archery is only a summer sport, so we can either indicate `Summer` for the second level or use `slice(None)`.

- In the third level, we explicitly specify `Archery` so we only get rows for archery events.
- We ignore the fourth level, effectively making it a wildcard.

We're interested in calculating which team won the most medals. So, we ask for the `Team` column. Then we can run `value_counts` to identify which team won the most events. The query thus looks like this:

```
df.loc[(slice(None), 'Summer', 'Archery'),     ①
       'Team'                                   ②
       ].value_counts()                         ③
```

① Row selector: all years, summer games, all competitions within archery

② Column selector: we only want the Team column.

③ Applies value_counts to the resulting series

But wait: this counts *all* participants in archery events. We are only interested in the medalists. We can thus start our query by removing all rows in which `Medal` contains a `NaN` value, calling `dropna` and passing `subset='Medal'`. It's probably easier to understand if we use method-chaining syntax and formatting:

```
(
    df
    .dropna(subset='Medal')
    .loc[(slice(None), 'Summer', 'Archery'), 'Team']
    .value_counts()
)
```

Here are the first five results:

```
Team
South Korea              69
Belgium                  52
France                   48
```

```
United States          41
China                  19
```

Because `value_counts` sorts its values in descending order, we see that South Korea has had the most archery medalists, with Belgium, France, the US, and China in the next few places.

Next, I asked you to find the average height of athletes in one specific event: “Table Tennis Women’s Team.” Again, we can consider all the parts of our multi-index:

- We want to get results from 1980 onward.
- Table tennis is only played in the summer games, so we can specify either `Summer` or `slice(None)`.
- The sport is “Table tennis,” so we can specify that if we want to—but given that all these events fall under the same sport, we can also leave it as a wildcard with `slice(None)`.
- We specify “Table Tennis Women’s Team” for the event.

We are only interested in the `Height` column, so our query looks like this:

```
df.loc[(slice(1980, None),                    ①
       'Summer',                              ②
       slice(None),                           ③
       "Table Tennis Women's Team"),          ④
       'Height'                               ⑤
       ].mean()                               ⑥
```

① Row selector: 1980 and onward

② Row selector, part 2: summer games

③ Row selector, part 3: all sports

④ Row selector, part 4: only one event, "Table Tennis Women's Team"

⑤ Column selector: Height column

⑥ Applies mean to the resulting series

The answer from our data set is the float 165.04827586206898, or just over 165 cm.

For the next query, we expand our population, looking at not just the women's team version of table tennis but also the men's version. Our first three selectors are identical to what we did before, but the final (fourth) multi-index selector is a list rather than a string:

```
df.loc[(slice(1980, None),                          ①
       'Summer',                                    ②
       slice(None),                                 ③
       ["Table Tennis Men's Team",
        "Table Tennis Women's Team"]),              ④
       'Height'                                     ⑤
       ].mean()                                     ⑥
```

① Row selector: all years from 1980

② Row selector, part 2: summer games

③ Row selector, part 3: all sports

④ Row selector, part 4: two events: "Table Tennis Women's Team" and "Table Tennis Men's Team"

⑤ Column selector: Height column

⑥ Applies mean to the resulting series

Given that men are generally taller than women, it's not a surprise that adding men's events has greatly increased the average athlete's height. The answer is 171.26643598615917.

Finally, I was curious to know the height of the tallest-ever tennis player from 1980 until 2020. Once again, let's go through our query-building process:

- We want years from 1980 through 2016. This can be handled most easily with `slice(1980,2016)`.

- Because tennis is only at summer games, it doesn't matter whether we specify the `Season` selector as `Summer` or use `slice(None)`.
- We specify "Tennis" as the sport
- We'll allow any events, so we don't need to pass a fourth element in the tuple.

Finally, we're looking for the `Height` column, so we specify that in our query. And we want the maximum value for `Height`, so we use the `max` method. The final query looks like this:

```
df.loc[(slice(1980,2016),    ①
        'Summer',            ②
        'Tennis'),           ③
        'Height'             ④
        ].max()              ⑤
```

① Row selector: years 1980–2016

② Row selector, part 2: summer games

③ Row selector, part 3: only tennis

④ Column selector: Height column

⑤ Applies max to the resulting series

The tallest-ever tennis player was 208 cm tall—known in some countries as 6 feet, 10 inches. That's pretty tall!

Solution

```
filename = '../data/olympic_athlete_events.csv'

df = pd.read_csv(filename,
                 index_col=['Year', 'Season',
                            'Sport', 'Event'],
                 usecols=['Age', 'Height', 'Team',
                          'Year', 'Season', 'City',
                          'Sport', 'Event', 'Medal'])    ①
df = df.sort_index()                                     ②
df.loc[(slice(1936,2000), 'Summer'), 'Age'].mean()       ③
df.dropna(subset='Medal').loc[
    (slice(None), 'Summer', 'Archery'),
           'Team'].value_counts()                                         ④
  df.loc[(slice(1980, None), 'Summer', slice(None),
          "Table Tennis Women's Team"),
          'Height'].mean()                                                ⑤
  df.loc[(slice(1980, None),
          'Summer', slice(None),
          ["Table Tennis Men's Team",
          "Table Tennis Women's Team"]),
          'Height'].mean()                                                ⑥
  df.loc[(slice(1980,2016),
          'Summer',
          'Tennis'), 'Height'].max()                                      ⑦
```

① Reads the CSV file into a data frame with nine total columns, four of which are used in our index

② Sorts the rows of the data frame according to the index

③ Gets the average age of summer athletes from 1936 to 2000

④ Which teams got the most medals in all archery events?

⑤ What was the average height of participants in "Table Tennis Women's Team" events from 1980?

⑥ What was the average height of participants in both "Table Tennis Women's Team" and "Table Tennis Men's Team" events from 1980?

⑦ How tall was the tallest tennis player from 1980 to 2016?

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

Going deep

As we have already seen, `loc` makes retrieving data from multi-indexed data frames pretty straightforward. However, sometimes we may want to use a multi-index differently. Pandas provides two other methods: `xs` and `IndexSlice`.

Because multi-indexed data frames are both common and important, pandas provides several ways to retrieve data from them. Let's start with `xs`, which lets us accomplish what we did in exercise 23: find matches for certain levels within a multi-index. For example, one question in the previous exercise asked you to find the mean height of participants in the "Table Tennis Women's Team" event from all Olympics years. Using `loc`, we had to tell pandas to accept all values for `Year`, all values for `Season`, and all values for `Sport`—in other words, we only checked the fourth level of the multi-index: the event. Our query looked like this:

```
df.loc[(slice(None),                              ①
        'Summer',                                 ②
        slice(None),                              ③
        "Table Tennis Women's Team"),             ④
        'Height'                                  ⑤
        ].mean()                                  ⑥
```

① Row selector: all years

② Row selector, part 2: summer games

③ Row selector, part 3: all sports

④ Row selector, part 4: one event: "Table Tennis Women's Team"

⑤ Column selector: Height column

⑥ Applies mean to the resulting series

Using `xs`, we can shorten that query to

```
df.xs("Table Tennis Women's Team",    ①
      level='Event'                   ②
      ).mean()                        ③
```

① Finds rows matching "Table Tennis Women's Team"

② The match should come in the multi-index level called Event.

③ Applies mean to the resulting series

You may have noticed that I lied when I said we didn’t search by season. As you can see in the `loc`-based query, we did include that in our search. Fortunately, we can handle that by passing a list of levels to the `level` parameter and a tuple of values as the first argument:

```
df.xs(('Summer', "Table Tennis Women's Team"),    ①
      level=['Season', 'Event']).mean()           ②
```

① Passes a two-element tuple to match two levels of the multi-index

② The argument passed to level indicates which levels need to match.

Notice that `xs` is a method and is thus invoked with round parentheses. By contrast, `loc` is an accessor attribute and is invoked with square brackets. And yes, it’s often hard to keep track of these things.

You can, by the way, use integers as the arguments to `level` rather than names. I find column names far easier to understand, though, and I encourage you to use them.

A more general way to retrieve from a multi-index is `IndexSlice`. Remember when I mentioned earlier that we cannot use `:` inside round parentheses and thus need to say `slice(None)`? Well, `IndexSlice` solves that problem: it uses square brackets and can use slice syntax for any set of values. For example, we can say

```
from pandas import IndexSlice as idx
df.loc[idx[1980:2016, :, 'Swimming':'Table tennis'], :]    (1)
```

(1) Years 1980–2016, all seasons, and all sports from "Swimming" to "Table tennis"

This code allows us to select a range of values for each level of the multi-index. We no longer need to call the `slice` function. Now we can use the standard Python `:` syntax for slicing within each level. The result of calling `IndexSlice` (or `idx`, as we aliased it here) is a tuple of Python `slice` objects:

```
(slice(1980, 2016, None),
 slice(None, None, None),
 slice('Swimming', 'Table tennis', None))
```

In other words, `IndexSlice` is syntactic sugar, allowing pandas to look and feel more like a standard Python data structure, even when the index is far more complex.

One final note: a data frame can have a multi-index on its rows, its columns, or both. By default, `xs` assumes the multi-index is on the rows. If and when you want to use it on multi-index columns, pass `axis='columns'` as a keyword argument.

Beyond the exercise

- Events occur in either summer or winter Olympic games, but not both. As a result, the `"Season"` level in our multi-index is often unnecessary. Remove the `"Season"` level, and then find (again) the height of the tallest tennis player between 1980 and 2016.
- In which city were the most gold medals awarded from 1980 onward?
- How many gold medals were received by the United States since 1980? (Use the index to select the values.)

Pivot tables

So far, we have seen how to use indexes to restructure our data, making it easier to retrieve different slices of the information it contains and thus answer particular questions more easily. But the questions we have been asking have all had a single answer. We often want to apply a particular aggregate function to many different combinations of columns and rows. One of the most common and powerful ways to accomplish this is with a *pivot table*.

A pivot table allows us to create a new table (data frame) from a subset of an existing data frame. Here's the basic idea:

- Our data frame contains two columns with categorical, repeating, nonhierarchical data. For example: years, country names, colors, and company divisions.
- Our data frame has a third column that is numeric.
- We create a new data frame from those three columns, as follows:
  - All unique values from the first categorical column become the index or row labels.
  - All unique values from the second categorical column become the column labels.
  - Wherever the two categories match, we get either the single value where those two intersect or the mean of all values where they intersect.

It takes a while to understand how a pivot table works. But once you get it, it's hard to un-see: you start finding uses everywhere.

For example, consider this simple data frame:

```
g = np.random.default_rng(0)
df = DataFrame(g.integers(0, 100, [8,3]),
               columns=list('ABC'))
```

```
df['year'] = [2018] * 4 + [2019] * 4
df['quarter'] = 'Q1 Q2 Q3 Q4'.split() * 2
```

This table shows the sales of each product per year and quarter. And you can certainly understand the data if you look at it a certain way. But what if we were interested in seeing sales figures for product A? It may make more sense, and be easier to parse, if we use the quarters (a categorical, repeating value) as the rows, the years (again, a categorical, repeating value) as the columns, and the figures for product A as the values. We can create such a pivot table as follows:

```
df.pivot_table(index='quarter',    ①
               columns='year',     ②
               values='A')         ③
```

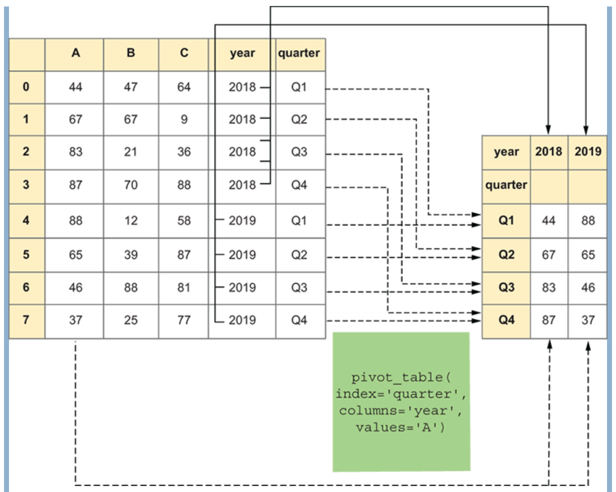
① Rows (index) are unique values from quarter.

② Columns are unique values from year.

③ Values are the mean of each year—quarter intersection.

The result, on my computer, is a data frame that looks like this:

```
year    2018    2019
quarter
Q1      44      88
Q2      67      65
Q3      83      46
Q4      87      37
```



Graphical depiction of creating a pivot table with index `quarter`, columns `year`, and values `A`

The quarters are sorted in alphabetical order, which is fine here. In some cases, such as using month names for your index, you can pass `sort=False`.

What if more than one row has the same values for year and month? By default, `pivot_table` runs the `mean` aggregation method on all values. (Pandas also offers a `pivot` method, which doesn't do aggregation and cannot handle duplicate values for index-column combinations. I never use it.) To use a different aggregation function, pass an argument to `aggfunc` in your call to `pivot_table`. For example, you can count the values in each intersection box by passing the `size` function:

```
df.pivot_table(index='quarter',          ①
               columns='year',           ②
               values='A',               ③
               sort=False,               ④
               aggfunc='size')           ⑤
```

① Index (rows): unique values from quarter

② Columns: unique values from year

③ Values: from column A

④ Don't sort the values.

⑤ Use size on the values.

Note that here, it doesn't matter whether we use `size` or `count`, because there aren't any `NaN` values. Generally speaking, `size` includes all values, including `NaN`, and `count` ignores them.

The result of this data frame isn't very interesting because there aren't any repeated intersections:

```
year     2018     2019
quarter
Q1       1        1
Q2       1        1
Q3       1        1
Q4       1        1
```

Remember that a pivot table will have one row for each unique value in your first chosen column and a column for each unique value in your second chosen column. If there are hundreds of unique values in either (or, even worse, in both), you could end up with a gargantuan pivot table. This will be hard to understand and analyze and also will consume large amounts of memory. Moreover, if your data isn't very lean (see Chapter 5), you may find junk values in your pivot table's index and columns.

Exercise 24 • Olympic pivots

In this exercise, we examine the Olympic data one more time—but using pivot tables, so we can examine and compare more information at a time than we could before. Pivot tables are a popular way to summarize information in a larger, more complex table.

I want you to do the following:

1. Read in our Olympic data again.
    1. Only use these columns: `Age`, `Height`, `Team`, `Year`, `Season`, `Sport`, `Medal`.
    2. Only include games from 1980 to the present.
    3. Only include data from these countries: Great Britain, France, United States, Switzerland, China, and India.
2. Answer these questions:
    1. What was the average age of Olympic athletes? In which country do players appear to consistently be the youngest?
    2. How tall were the tallest athletes in each sport in each year?
    3. How many medals did each country earn each year?

Working it out

The first challenge in this exercise is to create the data frame on which to base our pivot tables. We load the same CSV file as in the previous exercise, but we're interested in fewer rows and columns.

The first step is to read the CSV file into a data frame, limiting the columns we request:

```
df = pd.read_csv(filename,
                 usecols=['Age', 'Height',
                          'Team', 'Year',
                          'Season', 'Sport',
                          'Medal'])
```

Notice that we don't set the index. That's because we ignore the index in this exercise, focusing instead on pivot tables. Because the pivot tables are constructed based on actual columns and not the index, we'll stick with the default numeric index that pandas assigns to every data frame.

Now we want to remove all the rows that aren't from the countries we've named. (I chose these countries because I traveled there in the months before the pandemic. This is not meant to be any sort of representative sample, except where I've done corporate training in Python and data science.) We often keep (or remove) rows with a particular value, but how can we keep rows whose `Team` column is one of several values? We could use a query with `|` (the boolean "or" operator), but it would be long and complex.

Instead, we can use the `isin` method, which allows us to pass a list of possibilities and get a `True` value whenever the `Team` column equals one of those possible strings. In my experience, the `isin` method is one of those things that seems obvious when you start to use it but is far from obvious until you know to look for it.

We can keep only those countries this way:

```
df = df.loc[df['Team'].isin(['Great Britain', 'France',
                             'United States', 'Switzerland',
                             'China', 'India'])]
```

Now we remove any rows in which `Year` is before 1980. This is a more standard operation, one we've done many times before:

```
df = df.loc[df['Year'] >= 1980]
```

With our data frame in place, we can create pivot tables to examine our data from a new perspective. I first asked you to compare the average age of players for each team, for all sports and all years. As usual, when creating pivot tables, we need to consider what will be the rows, columns, and values:

- The rows (index) will be the unique values from the `Year` column.
- The columns will be the unique values from the `Team` column.
- The values will be from the `Age` column.

Sure enough, we can create our pivot table as follows:

```
df.pivot_table(index='Year',      ①
               columns='Team',    ②
               values='Age')      ③
```

① Index: unique values of Year in df

② Columns: unique values of Team in df

③ Values: mean of Age for each year—team combination

These numbers are across all sports, and not every country has entrants in every sport. But if we take the numbers at face value, we see that China consistently has younger athletes at Olympic games. Here is the output from the query:

```
Team    China      France     Great      India      Switzerland United
                              Britain                           States


Year
1980    21.868421  23.524590  22.882507  25.506667  24.557823   22.770992
1984    22.076336  24.369830  24.445423  24.905660  23.589744   24.437118
1988    22.358447  24.520076  25.439560  24.000000  26.218868   24.904977
1992    21.955752  25.140187  25.584055  24.184615  25.413194   25.474866
1994    20.627907  24.601307  25.282051  NaN        25.500000   24.976744
1996    22.021531  25.296629  26.746032  24.629630  27.122093   26.273277
1998    21.784091  25.462069  27.243902  16.000000  25.641509   25.146154
2000    22.515306  25.982833  26.406948  25.400000  27.376812   26.576203
2002    23.127451  25.737805  26.833333  20.000000  26.238710   25.726316
2004    23.006122  26.139073  26.303977  24.728395  27.343284   26.439093
2006    23.457143  26.303226  26.851852  25.200000  26.284848   25.637288
2008    23.903955  26.285714  25.200969  25.402985  27.312500   26.225806
2010    23.239669  25.911458  26.147059  25.666667  26.548387   25.841584
2012    23.894168  26.606635  25.922619  25.637363  27.172131   26.461883
2014    23.400000  25.708995  25.628571  25.000000  25.855814   26.189189
2016    23.873706  27.095238  26.653191  26.100000  25.891892   26.217454
```

Next, we want to find the tallest players in each sport from each year. Given that we are looking at a large number of sports and a relatively small number of years, it is wise to use the years in the columns this time:

- The rows (index) will be the unique values from the `Sport` column.
- The columns will be the unique values from the `Year` column.
- The values will come from the `Height` column. We're interested in the highest value and will thus provide a function argument to the `aggfunc` parameter: `max`.

NOTE In previous versions of pandas, it was common to specify the aggregation method by passing a NumPy method, such as `np.max` or `np.size`. However, pandas now prefers to get a string (e.g., `'max'` or `'size'`), which translates into an internal function name or reference.

In the end, we create the pivot table as follows:

```
df.pivot_table(index='Sport',      ①
               columns='Year',     ②
               values='Height',    ③
               aggfunc='max')      ④
```

① Index: unique values of Sport in df

② Columns: unique values of Year in df

③ Values: maximum value for Height

④ We use max as our aggregation function.

From the large number of `NaN` values, we can see that height information isn't as readily available for all sports and teams as many other measurements. This is not an unusual problem to face with real-world data; sometimes you have to make do with the data that is available, even if it's far from reliable and complete.

Finally, I asked you to determine how many medals each country received at each game. Once again, let's do a bit of planning before creating our pivot table:

- The rows (index) will be the unique values from the `Year` column.

- The columns will be the unique values from the `Team` column.
- We want to count the number of medals, not get their average values (as if that's even possible). This means we must provide a function argument to the `aggfunc` parameter. This can usually be a string referring to a method, such as `'max'`. We first have to remove all rows for which `Medal` has a `NaN` value indicating that no medal was won.

Our code to create the pivot table can look like this:

```
pd.pivot_table(df.dropna(subset='Medal'),      ①
               index='Year',                   ②
               columns='Team',                 ③
               values='Medal',                 ④
               aggfunc='max')                  ⑤
```

① Index: unique values of Sport in df

② Only uses the subset of df where Medal isn't NaN

③ Columns: unique values of Team in df

④ Values: sum of values in the Medal column

⑤ We use max as our aggregation function.

Solution

```
filename = '../data/olympic_athlete_events.csv'
df = pd.read_csv(filename,
                usecols=['Age', 'Height', 'Team',
                         'Year', 'Season',
                         'Sport', 'Medal'])                ①

df = df.loc[df['Team'].isin(['Great Britain', 'France',
                            'United States', 'Switzerland',
                            'China', 'India'])]            ②
df = df.loc[df['Year'] >= 1980]                            ③

df.pivot_table(index='Year', columns='Team',
               values='Age')                               ④
```

```
df.pivot_table(index='Sport',
               columns='Year', values='Height',
               aggfunc='max')                                    ⑤

pd.pivot_table(df.dropna(subset='Medal'),
               index='Year',
               columns='Team',
               values='Medal',
               aggfunc='size')                                   ⑥
```

① Loads only five columns; we ignore the index

② Removes rows in which the team isn't one of the six we're looking for

③ Removes rows in which the year is before 1980

④ Pivot table from Year (index), Team (columns), and mean Age

⑤ Pivot table from Sport (index), Year (columns), and the max value of Height

⑥ Pivot table from Year (index), Team (columns), and the number of medals

You can explore a version of this, in color, in the Pandas Tutor at https://pandastutor.com/vis.html#.

Beyond the exercise

- Create a pivot table that shows the number of medals each team won per year, with the index including the year and the season in which the games took place.
- Create a pivot table that shows both the average age and the average height per year per team.
- Create a pivot table that shows the average age and the average height per year, per team, broken up by year and season.

ummary

In this chapter, we saw that a data frame's index is not just a way to keep track of the rows but one that can be used to reshape a data frame, making it easier to extract useful information. This is particularly true when we create pivot tables, choosing values from an existing data frame for comparison.