**Explicit indexes**

In chapter one, you saw that DataFrames are composed of three parts: a NumPy array for the data, and two indexes to store the row and column details.

**dog dataset, revisited**

Here's the dog dataset again.

**.columns and .index**

Recall that dot-columns contains an Index object of column names, and dot-index contains an Index object of row numbers.



**Setting a column as the index**

You can move a column from the body of the DataFrame to the index. This is called "setting an index," and it uses the set\_index method. Notice that the output has changed slightly; in particular, a quick visual clue that name is now in the index is that the index values are left-aligned rather than right-aligned.

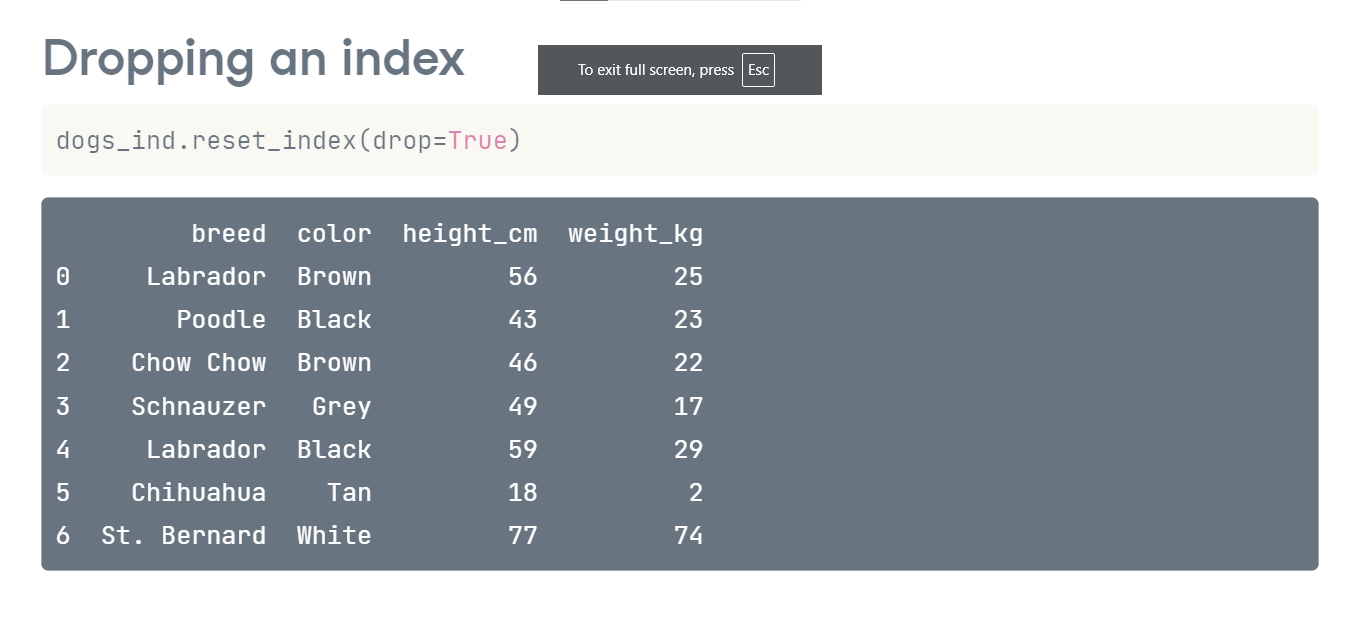


**Removing an index**

To undo what you just did, you can reset the index - that is, you remove it. This is done via reset\_index.

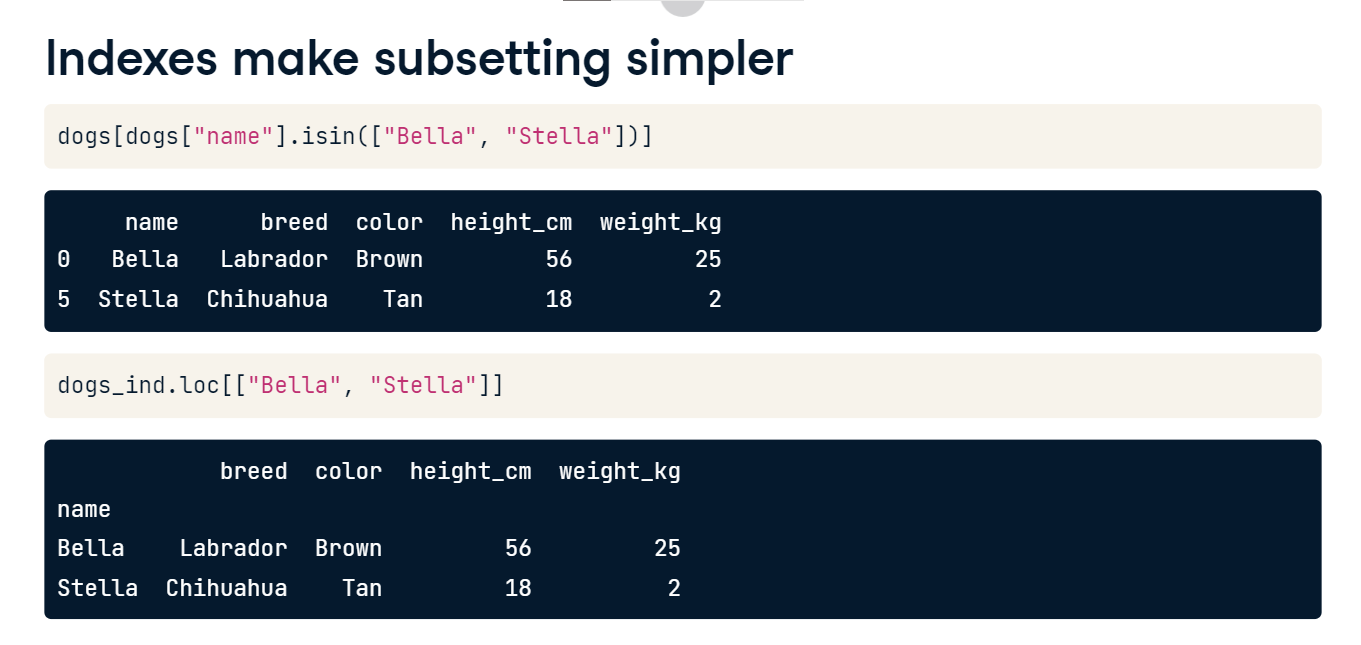
**Dropping an index**

reset\_index has a drop argument that allows you to discard an index. Here, setting drop to True entirely removes the dog names.



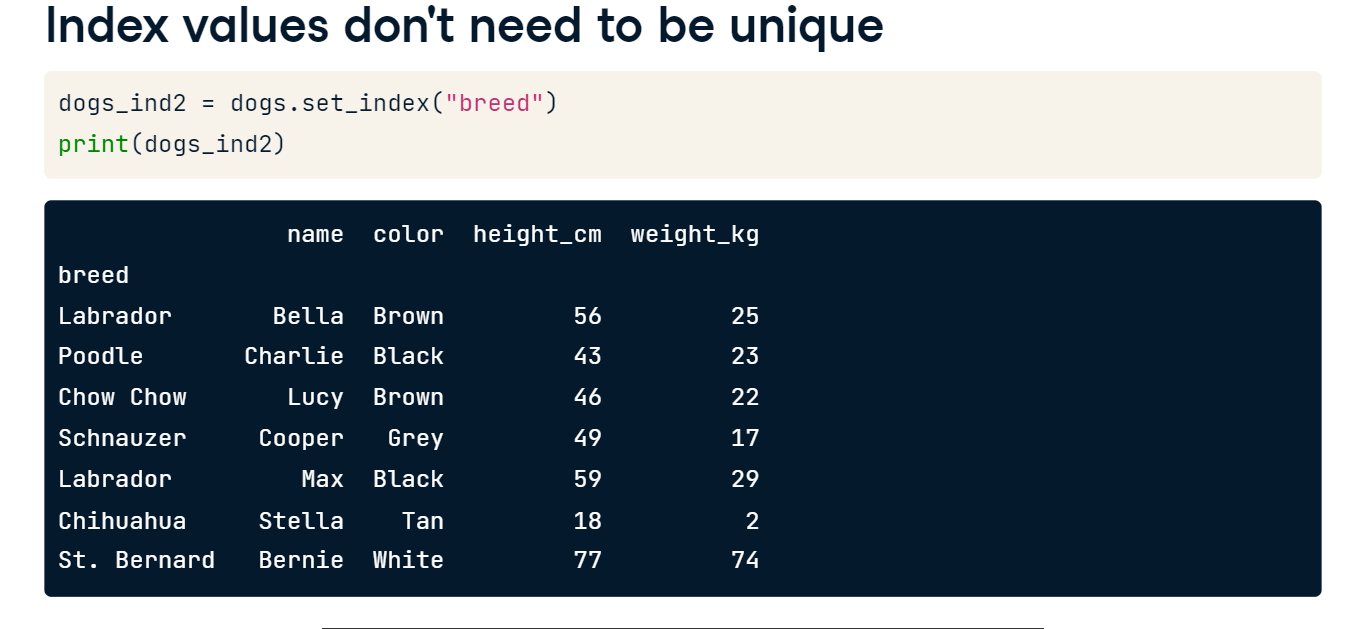
**Indexes make subsetting simpler**

You may be wondering why you should bother with indexes. The answer is that it makes subsetting code cleaner. Consider this example of subsetting for the rows where the dog is called Bella or Stella. It's a fairly tricky line of code for such a simple task. Now, look at the equivalent when the names are in the index. DataFrames have a subsetting method called "loc," which filters on index values. Here you simply pass the dog names to loc as a list. Much easier!



**Index values don't need to be unique**

The values in the index don't need to be unique. Here, there are two Labradors in the index.

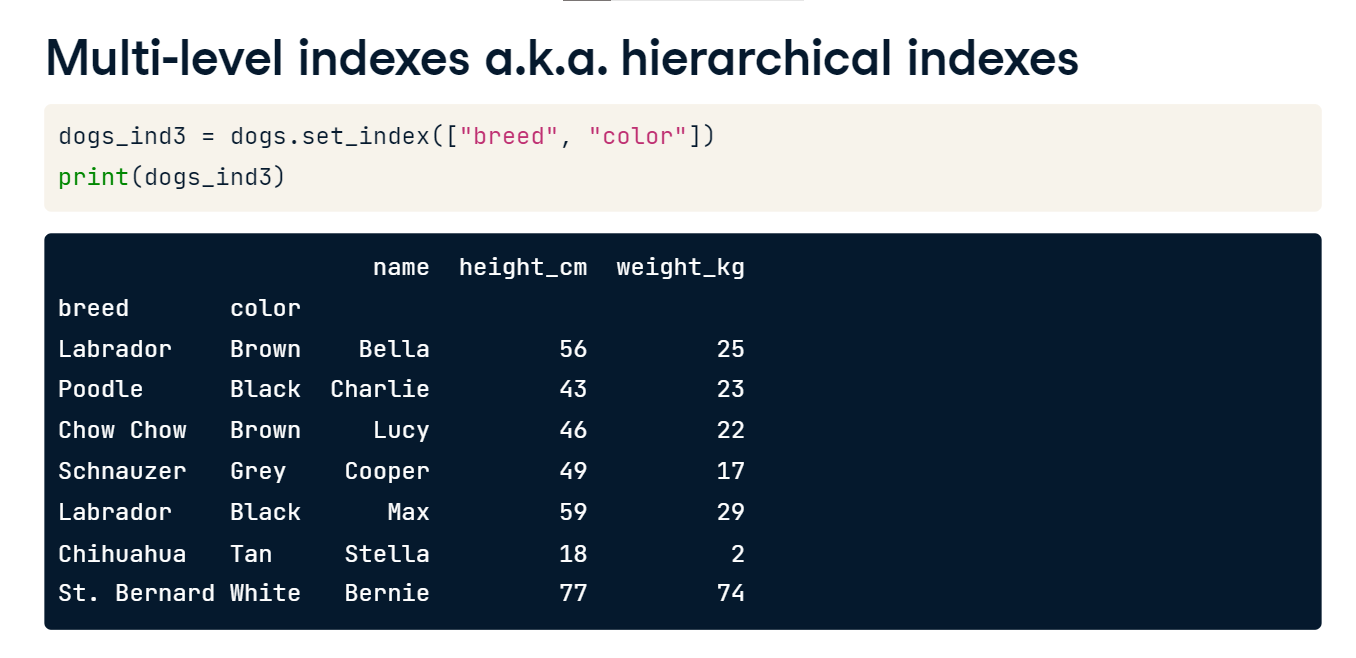


**Subsetting on duplicated index values**

Now, if you subset on "Labrador" using loc, all the Labrador data is returned.

**Multi-level indexes a.k.a. hierarchical indexes**

You can include multiple columns in the index by passing a list of column names to set\_index. Here, breed and color are included. These are called multi-level indexes, or hierarchical indexes: the terms are synonymous. There is an implication here that the inner level of index, in this case, color, is nested inside the outer level, breed.



**Subset the outer level with a list**

To take a subset of rows at the outer level index, you pass a list of index values to loc. Here, the list contains Labrador and Chihuahua, and the resulting subset contains all dogs from both breeds.



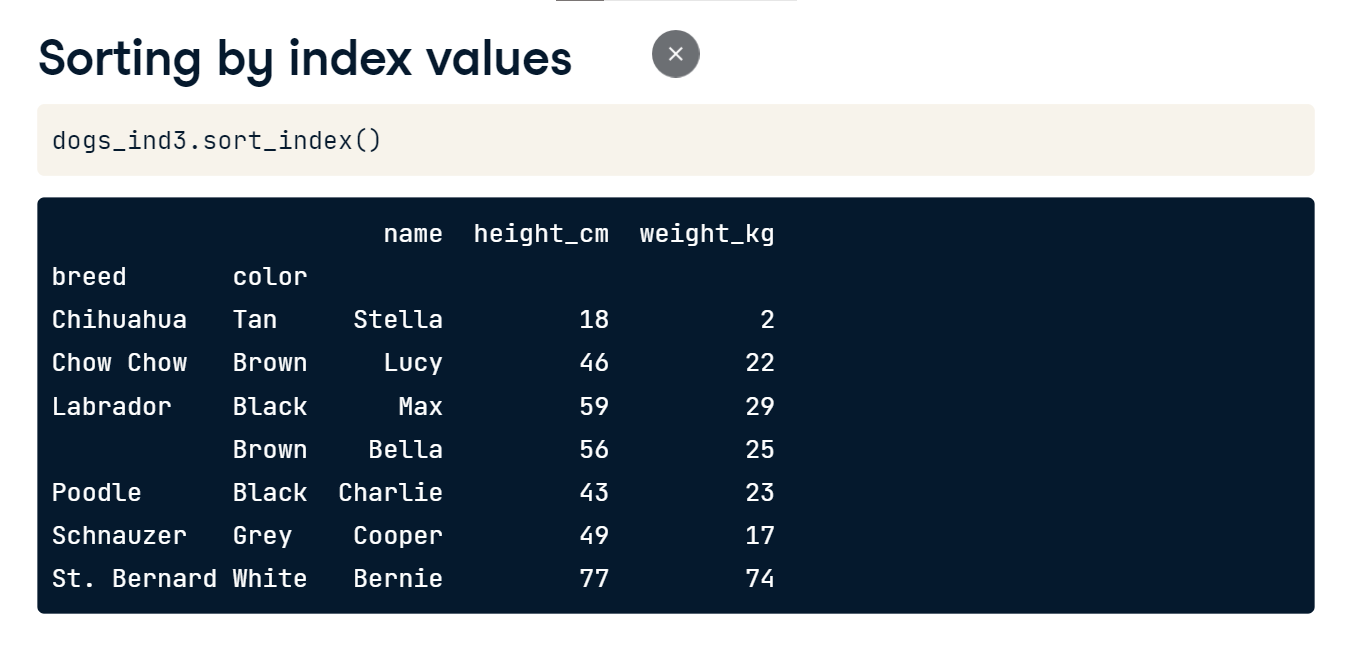
**Subset inner levels with a list of tuples**

To subset on inner levels, you need to pass a list of tuples. Here, the first tuple specifies Labrador at the outer level and Brown at the inner level. The resulting rows have to match all conditions from a tuple. For example, the black Labrador wasn't returned because the brown condition wasn't matched.



**Sorting by index values**

In chapter 1, you saw how to sort the rows of a DataFrame using sort\_values. You can also sort by index values using sort\_index. By default, it sorts all index levels from outer to inner, in ascending order.



**Controlling sort\_index**

You can control the sorting by passing lists to the level and ascending arguments.



**Now you have two problems**

Indexes are controversial. Although they simplify subsetting code, there are some downsides. Index values are just data. Storing data in multiple forms makes it harder to think about. There is a concept called "tidy data," where data is stored in tabular form - like a DataFrame. Each row contains a single observation, and each variable is stored in its own column. Indexes violate the last rule since index values don't get their own column. In pandas, the syntax for working with indexes is different from the syntax for working with columns. By using two syntaxes, your code is more complicated, which can result in more bugs. If you decide you don't want to use indexes, that's perfectly reasonable. However, it's useful to know how they work for cases when you need to read other people's code.

**Temperature dataset**

In this chapter, you'll work with a monthly time series of air temperatures in cities around the world.

**Slicing and subsetting with .loc and .iloc**

Slicing is a technique for selecting consecutive elements from objects.

**Slicing lists**

Here are the dog breeds, this time as a list. To slice the list, you pass first and last positions separated by a colon into square brackets. Remember that Python positions start from zero, so 2 refers to the third element, Chow Chow. Also remember that the last position, 5, is not included in the slice, so we finish at Labrador, not Chihuahua. If you want the slice to start from the beginning of the list, you can omit the zero. Here, using colon-3 returns the first three elements. Slicing with colon on its own returns the whole list.



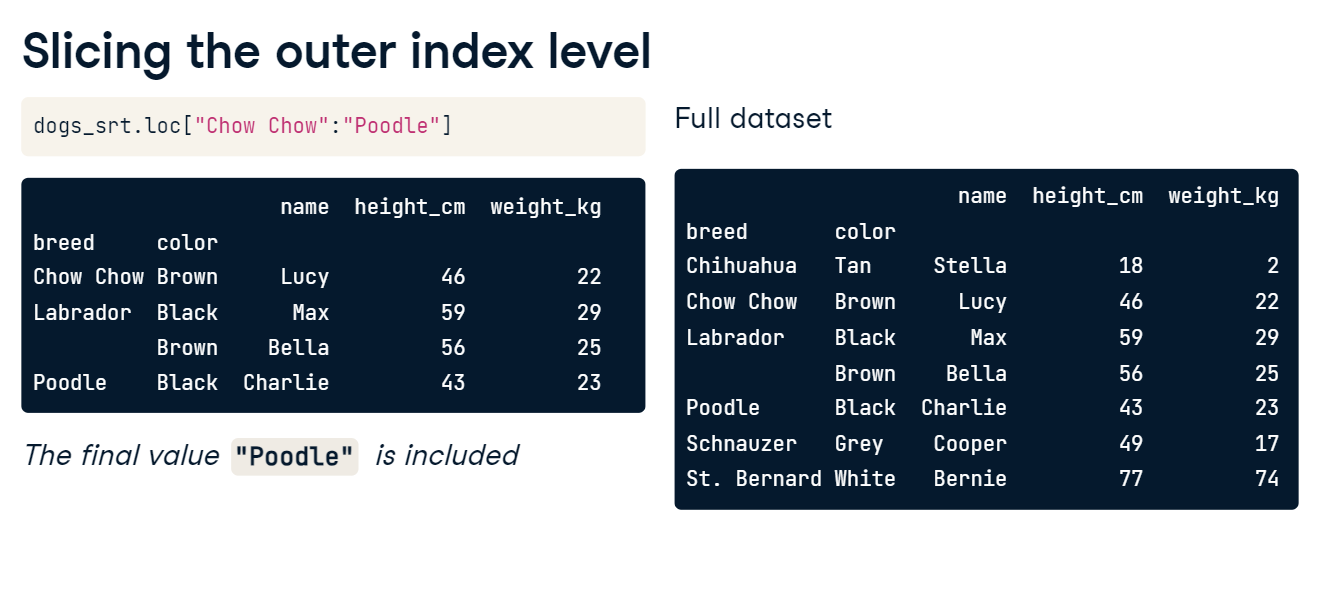
**Sort the index before you slice**

You can also slice DataFrames, but first, you need to sort the index. Here, the dogs dataset has been given a multi-level index of breed and color; then, the index is sorted with sort\_index.



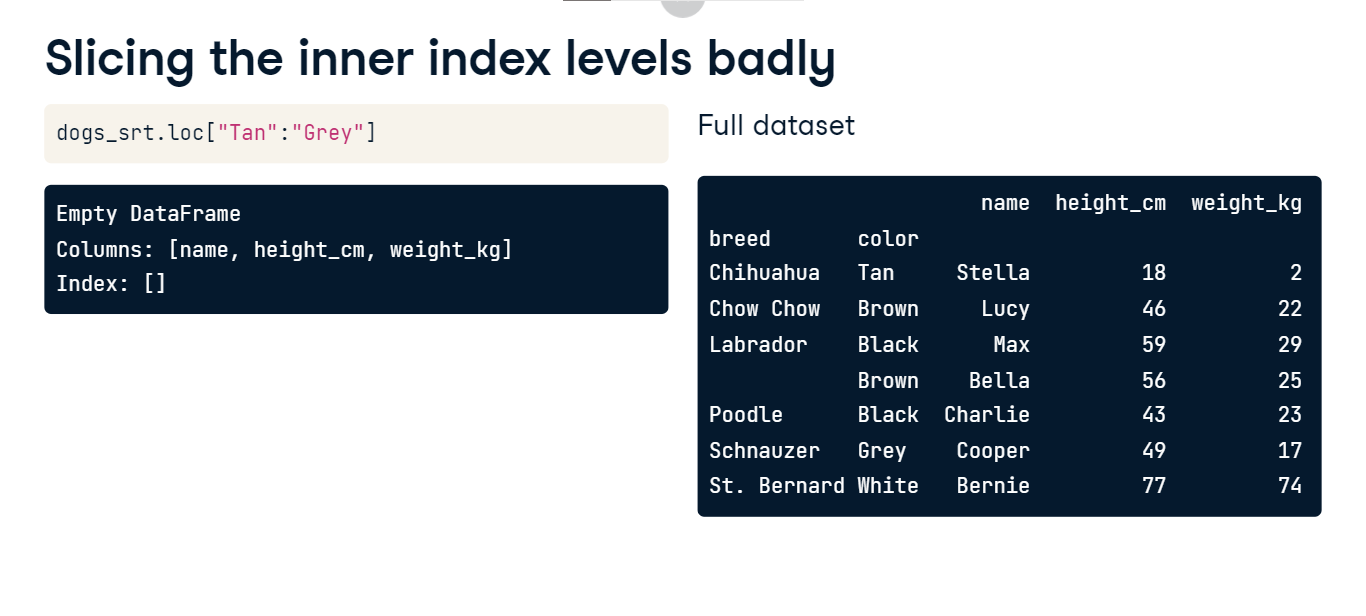
**Slicing the outer index level**

To slice rows at the outer level of an index, you call loc, passing the first and last values separated by a colon. The full dataset is shown on the right for comparison. There are two differences compared to slicing lists. Rather than specifying row numbers, you specify index values. Secondly, notice that the final value is included. Here, Poodle is included in the results.



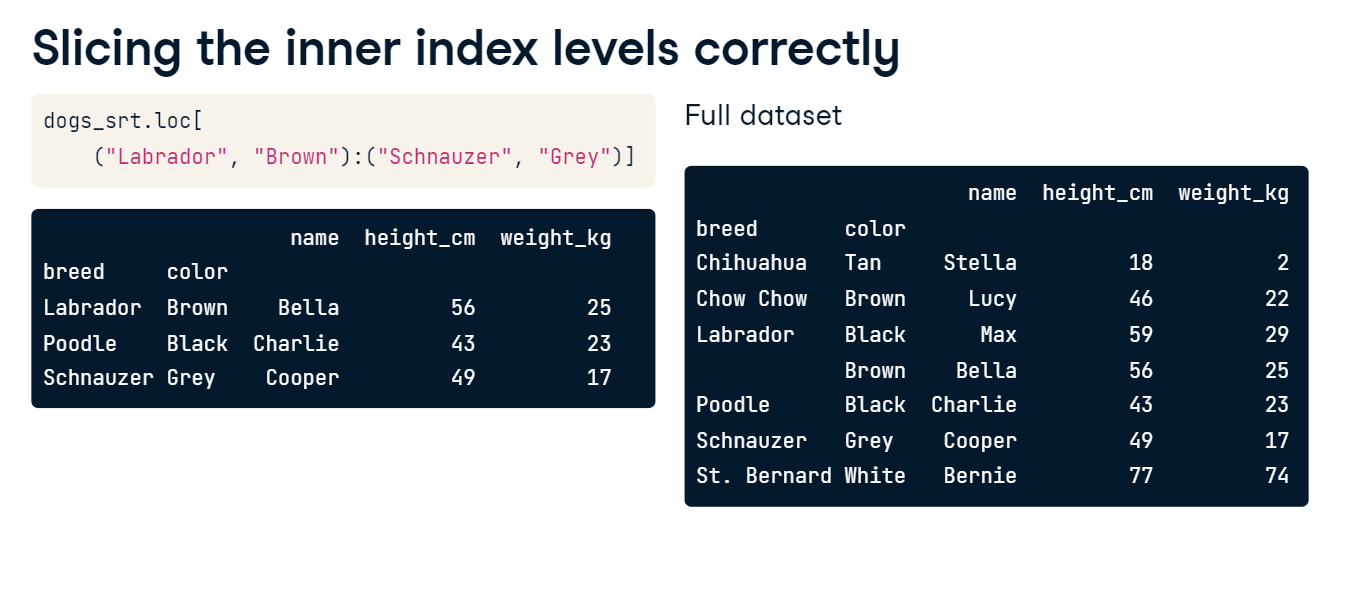
**Slicing the inner index levels badly**

The same technique doesn't work on inner index levels. Here, trying to slice from Tan to Grey returns an empty DataFrame instead of the six dogs we wanted. It's important to understand the danger here. pandas doesn't throw an error to let you know that there is a problem, so be careful when coding.



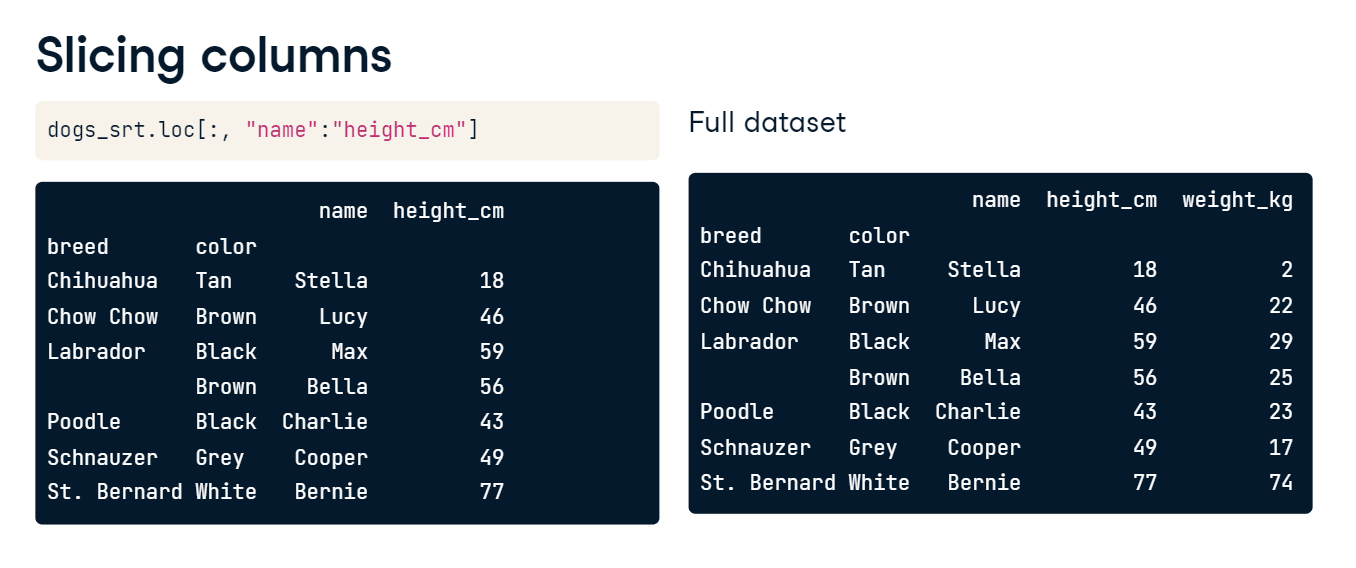
**Slicing the inner index levels correctly**

The correct approach to slicing at inner index levels is to pass the first and last positions as tuples. Here, the first element to include is a tuple of Labrador and Brown.



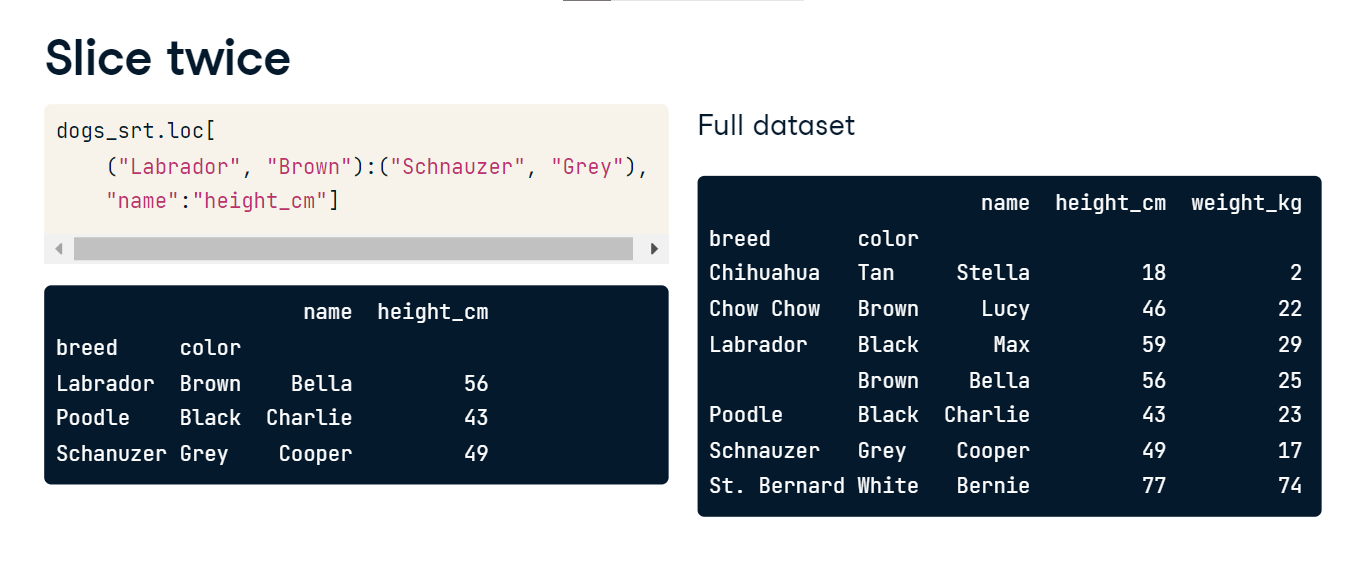
**Slicing columns**

Since DataFrames are two-dimensional objects, you can also slice columns. You do this by passing two arguments to loc. The simplest case involves subsetting columns but keeping all rows. To do this, pass a colon as the first argument to loc. As with slicing lists, a colon by itself means "keep everything." The second argument takes column names as the first and last positions to slice on.



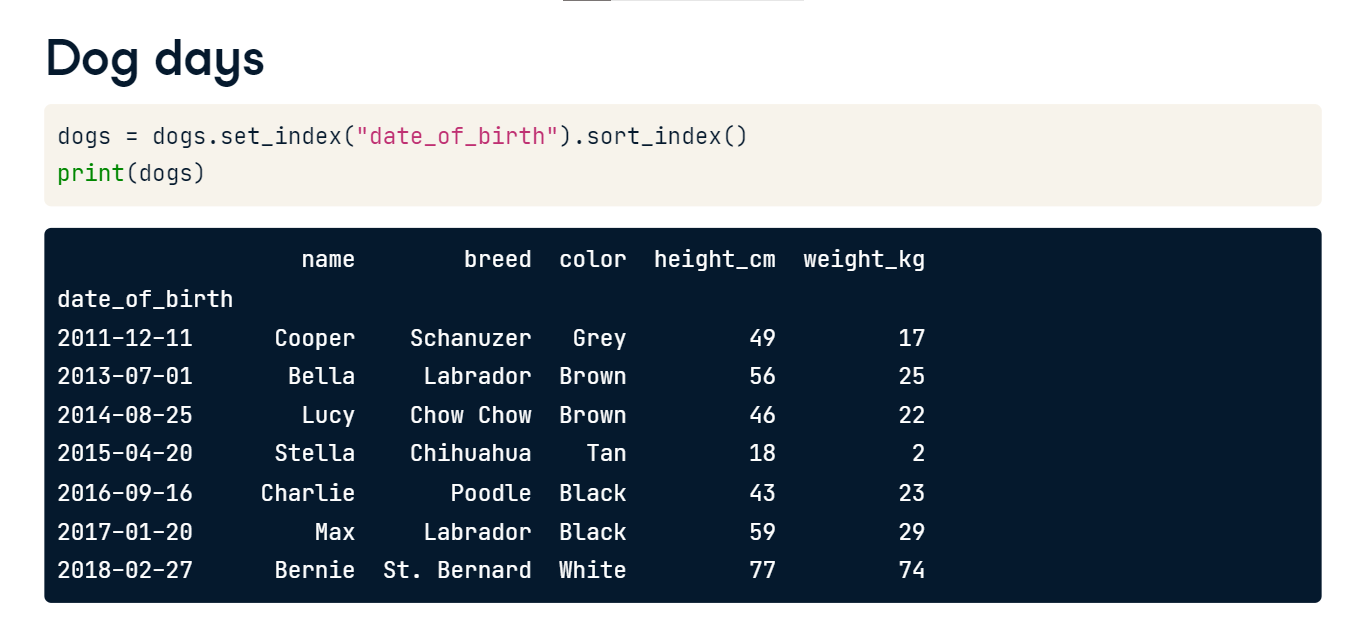
**Slice twice**

You can slice on rows and columns at the same time: simply pass the appropriate slice to each argument. Here, you see the previous two slices being performed in the same line of code.



**Dog days**

An important use case of slicing is to subset DataFrames by a range of dates. To demonstrate this, let's set the date\_of\_birth column as the index and sort by this index.



**Slicing by dates**

You slice dates with the same syntax as other types. The first and last dates are passed as strings.



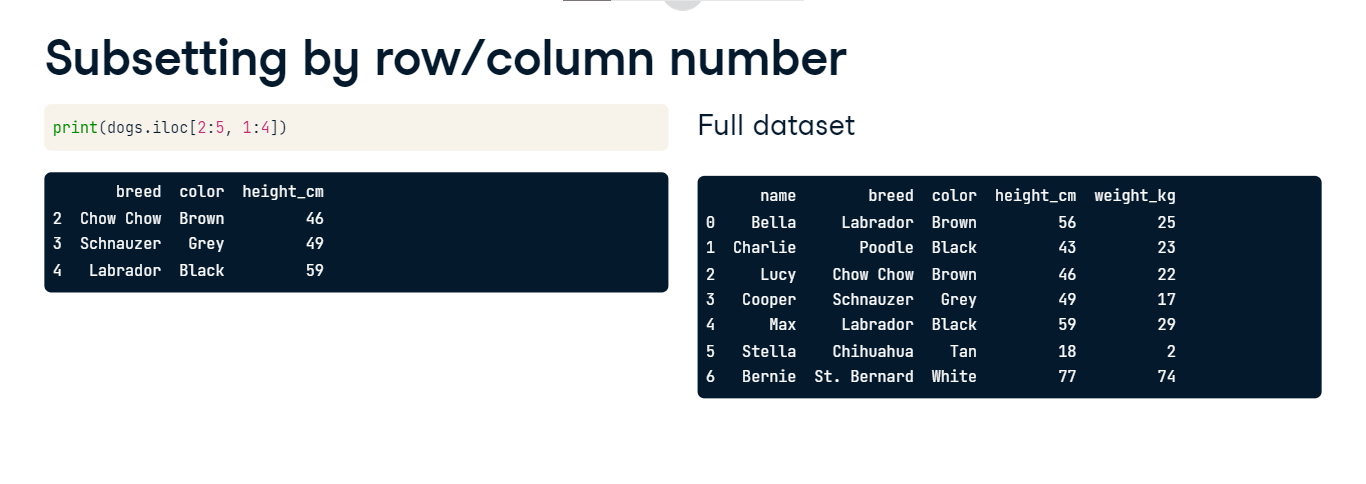
**Slicing by partial dates**

One helpful feature is that you can slice by partial dates. Here, the first and last positions are only specified as 2014 and 2016, with no month or day parts. pandas interprets this as slicing from the start of 2014 to the end of 2016; that is, all dates in 2014, 2015, and 2016.



**Subsetting by row/column number**

You can also slice DataFrames by row or column number using the iloc method. This uses a similar syntax to slicing lists, except that there are two arguments: one for rows and one for columns. Notice that, like list slicing but unlike loc, the final values aren't included in the slice. In this case, the fifth row and fourth column aren't included.



**Working with pivot tables**

You saw how to create pivot tables with pandas in chapter two. In this lesson, you'll perform subsetting and calculations on pivot tables.

**A bigger dog dataset**

Here's a larger version of the dog dataset. The extra dogs mean we have something to compute on.

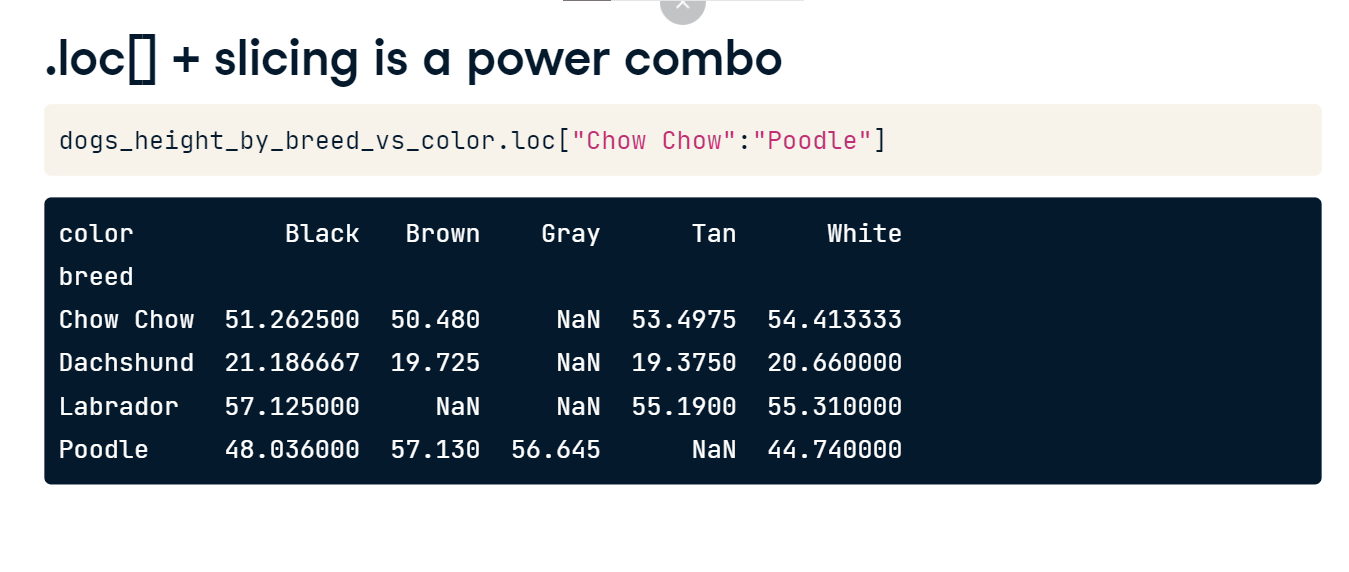
**Pivoting the dog pack**

Recall that you create a pivot table by calling dot-pivot\_table. The first argument is the column name containing values to aggregate. The index argument lists the columns to group by and display in rows, and the columns argument lists the columns to group by and display in columns. We'll use the default aggregation function, which is mean.



**.loc[] + slicing is a power combo**

Pivot tables are just DataFrames with sorted indexes. That means that all the fun stuff you've learned so far this chapter can be used on them. In particular, the loc and slicing combination is ideal for subsetting pivot tables, like so.



**The axis argument**

The methods for calculating summary statistics on a DataFrame, such as mean, have an axis argument. The default value is "index," which means "calculate the statistic across rows." Here, the mean is calculated for each color. That is, "across the breeds." The behavior is the same as if you hadn't specified the axis argument.



**Calculating summary stats across columns**

To calculate a summary statistic for each row, that is, "across the columns," you set axis to "columns." Here, the mean height is calculated for each breed. That is, "across the colors." For most DataFrames, setting the axis argument doesn't make any sense, since you'll have different data types in each column. Pivot tables are a special case since every column contains the same data type.

