# Data Manipulation with pandas

**Course Description**

pandas is the world's most popular Python library, used for everything from data manipulation to data analysis. In this course, you'll learn how to manipulate DataFrames, as you extract, filter, and transform real-world datasets for analysis. Using pandas you’ll explore all the core data science concepts. Using real-world data, including Walmart sales figures and global temperature time series, you’ll learn how to import, clean, calculate statistics, and create visualizations—using pandas to add to the power of Python!

**Daily XP1200**

# Introducing DataFrames

**50 XP**

## 1. Introducing DataFrames

Hi, I'm Richie. I'll be your tour guide through the world of pandas.

## 2. What's the point of pandas?

pandas is a Python package for data manipulation. It can also be used for data visualization; we'll get to that in Chapter 4.

## 3. Course outline

We'll start by talking about DataFrames, which form the core of pandas. In chapter 2, we'll discuss aggregating data to gather insights. In chapter 3, you'll learn all about slicing and indexing to subset DataFrames. Finally, you'll visualize your data, deal with missing data, and read data into a DataFrame. Let's dive in.

## 4. pandas is built on NumPy and Matplotlib

pandas is built on top of two essential Python packages, NumPy and Matplotlib. Numpy provides multidimensional array objects for easy data manipulation that pandas uses to store data, and Matplotlib has powerful data visualization capabilities that pandas takes advantage of.

## 5. pandas is popular

pandas has millions of users, with PyPi recording about 14 million downloads in December 2019. This represents almost the entire Python data science community!

1. 1 https://pypistats.org/packages/pandas

## 6. Rectangular data

There are several ways to store data for analysis, but rectangular data, sometimes called "tabular data" is the most common form. In this example, with dogs, each observation, or each dog, is a row, and each variable, or each dog property, is a column. pandas is designed to work with rectangular data like this.

## 7. pandas DataFrames

In pandas, rectangular data is represented as a DataFrame object. Every programming language used for data analysis has something similar to this. R also has DataFrames, while SQL has database tables. Every value within a column has the same data type, either text or numeric, but different columns can contain different data types.

## 8. Exploring a DataFrame: .head()

When you first receive a new dataset, you want to quickly explore it and get a sense of its contents. pandas has several methods for this. The first is head, which returns the first few rows of the DataFrame. We only had seven rows to begin with, so it's not super exciting, but this becomes very useful if you have many rows.

## 9. Exploring a DataFrame: .info()

The info method displays the names of columns, the data types they contain, and whether they have any missing values.

## 10. Exploring a DataFrame: .shape

A DataFrame's shape attribute contains a tuple that holds the number of rows followed by the number of columns. Since this is an attribute instead of a method, you write it without parentheses.

## 11. Exploring a DataFrame: .describe()

The describe method computes some summary statistics for numerical columns, like mean and median. "count" is the number of non-missing values in each column. describe is good for a quick overview of numeric variables, but if you want more control, you'll see how to perform more specific calculations later in the course.

## 12. Components of a DataFrame: .values

DataFrames consist of three different components, accessible using attributes. The values attribute, as you might expect, contains the data values in a 2-dimensional NumPy array.

## 13. Components of a DataFrame: .columns and .index

The other two components of a DataFrame are labels for columns and rows. The columns attribute contains column names, and the index attribute contains row numbers or row names. Be careful, since row labels are stored in dot-index, not in dot-rows. Notice that these are Index objects, which we'll cover in Chapter 3. This allows for flexibility in labels. For example, the dogs data uses row numbers, but row names are also possible.

## 14. pandas Philosophy

Python has a semi-official philosophy on how to write good code called The Zen of Python. One suggestion is that given a programming problem, there should only be one obvious solution. As you go through this course, bear in mind that pandas deliberately doesn't follow this philosophy. Instead, there are often multiple ways to solve a problem, leaving you to choose the best. In this respect, pandas is like a Swiss Army Knife, giving you a variety of tools, making it incredibly powerful, but more difficult to learn. In this course, we aim for a more streamlined approach to pandas, only covering the most important ways of doing things.

1. 1 https://www.python.org/dev/peps/pep-0020/

## 15. Let's practice!

Enough meditating, time to write some code!

**Daily XP50**

##### Exercise

##### Exercise

# Inspecting a DataFrame

When you get a new DataFrame to work with, the first thing you need to do is explore it and see what it contains. There are several useful methods and attributes for this.

* .head() returns the first few rows (the “head” of the DataFrame).
* .info() shows information on each of the columns, such as the data type and number of missing values.
* .shape returns the number of rows and columns of the DataFrame.
* .describe() calculates a few summary statistics for each column.

homelessness is a DataFrame containing estimates of homelessness in each U.S. state in 2018. The individual column is the number of homeless individuals not part of a family with children. The family\_members column is the number of homeless individuals part of a family with children. The state\_pop column is the state's total population.

pandas is imported for you.

##### Instructions 1/4

**25 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* [3](javascript:void(0))
* [4](javascript:void(0))
* Print the head of the homelessness DataFrame.

# Print the head of the homelessness data

homelessness.head

* IPython Shell
* Slides
* Notes

# Print the head of the homelessness data

homelessness.head

<bound method NDFrame.head of region state individuals family\_members state\_pop

0 East South Central Alabama 2570.0 864.0 4887681

1 Pacific Alaska 1434.0 582.0 735139

2 Mountain Arizona 7259.0 2606.0 7158024

3 West South Central Arkansas 2280.0 432.0 3009733

4 Pacific California 109008.0 20964.0 39461588

5 Mountain Colorado 7607.0 3250.0 5691287

6 New England Connecticut 2280.0 1696.0 3571520

7 South Atlantic Delaware 708.0 374.0 965479

8 South Atlantic District of Columbia 3770.0 3134.0 701547

9 South Atlantic Florida 21443.0 9587.0 21244317

10 South Atlantic Georgia 6943.0 2556.0 10511131

11 Pacific Hawaii 4131.0 2399.0 1420593

12 Mountain Idaho 1297.0 715.0 1750536

13 East North Central Illinois 6752.0 3891.0 12723071

14 East North Central Indiana 3776.0 1482.0 6695497

15 West North Central Iowa 1711.0 1038.0 3148618

16 West North Central Kansas 1443.0 773.0 2911359

17 East South Central Kentucky 2735.0 953.0 4461153

18 West South Central Louisiana 2540.0 519.0 4659690

19 New England Maine 1450.0 1066.0 1339057

20 South Atlantic Maryland 4914.0 2230.0 6035802

21 New England Massachusetts 6811.0 13257.0 6882635

22 East North Central Michigan 5209.0 3142.0 9984072

23 West North Central Minnesota 3993.0 3250.0 5606249

24 East South Central Mississippi 1024.0 328.0 2981020

25 West North Central Missouri 3776.0 2107.0 6121623

26 Mountain Montana 983.0 422.0 1060665

27 West North Central Nebraska 1745.0 676.0 1925614

28 Mountain Nevada 7058.0 486.0 3027341

29 New England New Hampshire 835.0 615.0 1353465

30 Mid-Atlantic New Jersey 6048.0 3350.0 8886025

31 Mountain New Mexico 1949.0 602.0 2092741

32 Mid-Atlantic New York 39827.0 52070.0 19530351

33 South Atlantic North Carolina 6451.0 2817.0 10381615

34 West North Central North Dakota 467.0 75.0 758080

35 East North Central Ohio 6929.0 3320.0 11676341

36 West South Central Oklahoma 2823.0 1048.0 3940235

37 Pacific Oregon 11139.0 3337.0 4181886

38 Mid-Atlantic Pennsylvania 8163.0 5349.0 12800922

39 New England Rhode Island 747.0 354.0 1058287

40 South Atlantic South Carolina 3082.0 851.0 5084156

41 West North Central South Dakota 836.0 323.0 878698

42 East South Central Tennessee 6139.0 1744.0 6771631

43 West South Central Texas 19199.0 6111.0 28628666

44 Mountain Utah 1904.0 972.0 3153550

45 New England Vermont 780.0 511.0 624358

46 South Atlantic Virginia 3928.0 2047.0 8501286

47 Pacific Washington 16424.0 5880.0 7523869

48 South Atlantic West Virginia 1021.0 222.0 1804291

49 East North Central Wisconsin 2740.0 2167.0 5807406

50 Mountain Wyoming 434.0 205.0 577601>

# Print the head of the homelessness data

print(homelessness.head())

# Print the head of the homelessness data

print(homelessness.head())

region state individuals family\_members state\_pop

0 East South Central Alabama 2570.0 864.0 4887681

1 Pacific Alaska 1434.0 582.0 735139

2 Mountain Arizona 7259.0 2606.0 7158024

3 West South Central Arkansas 2280.0 432.0 3009733

4 Pacific California 109008.0 20964.0 39461588

# Inspecting a DataFrame

When you get a new DataFrame to work with, the first thing you need to do is explore it and see what it contains. There are several useful methods and attributes for this.

* .head() returns the first few rows (the “head” of the DataFrame).
* .info() shows information on each of the columns, such as the data type and number of missing values.
* .shape returns the number of rows and columns of the DataFrame.
* .describe() calculates a few summary statistics for each column.

homelessness is a DataFrame containing estimates of homelessness in each U.S. state in 2018. The individual column is the number of homeless individuals not part of a family with children. The family\_members column is the number of homeless individuals part of a family with children. The state\_pop column is the state's total population.

pandas is imported for you.

##### Instructions 2/4

**25 XP**

* [2](javascript:void(0))
* [3](javascript:void(0))
* [4](javascript:void(0))
* Print information about the column types and missing values in homelessness.
* # Print the head of the homelessness data
* print(homelessness.head())
* # Print information about homelessness
* \_\_\_\_

# Print the head of the homelessness data

print(homelessness.head())

# Print information about homelessness

print(homelessness.info())

# Print the shape of homelessness

print(homelessness.shape)

# Print a description of homelessness

print(homelessness.describe())

# Print the head of the homelessness data

homelessness.head()

region state individuals family\_members state\_pop

0 East South Central Alabama 2570.0 864.0 4887681

1 Pacific Alaska 1434.0 582.0 735139

2 Mountain Arizona 7259.0 2606.0 7158024

3 West South Central Arkansas 2280.0 432.0 3009733

4 Pacific California 109008.0 20964.0 39461588

# Print the head of the homelessness data

print(homelessness.head())

region state individuals family\_members state\_pop

0 East South Central Alabama 2570.0 864.0 4887681

1 Pacific Alaska 1434.0 582.0 735139

2 Mountain Arizona 7259.0 2606.0 7158024

3 West South Central Arkansas 2280.0 432.0 3009733

4 Pacific California 109008.0 20964.0 39461588

<script.py> output:

region state individuals family\_members state\_pop

0 East South Central Alabama 2570.0 864.0 4887681

1 Pacific Alaska 1434.0 582.0 735139

2 Mountain Arizona 7259.0 2606.0 7158024

3 West South Central Arkansas 2280.0 432.0 3009733

4 Pacific California 109008.0 20964.0 39461588

# Print the head of the homelessness data

print(homelessness.head())

# Print information about homelessness

print(homelessness.info())

region state individuals family\_members state\_pop

0 East South Central Alabama 2570.0 864.0 4887681

1 Pacific Alaska 1434.0 582.0 735139

2 Mountain Arizona 7259.0 2606.0 7158024

3 West South Central Arkansas 2280.0 432.0 3009733

4 Pacific California 109008.0 20964.0 39461588

<class 'pandas.core.frame.DataFrame'>

Int64Index: 51 entries, 0 to 50

Data columns (total 5 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 region 51 non-null object

1 state 51 non-null object

2 individuals 51 non-null float64

3 family\_members 51 non-null float64

4 state\_pop 51 non-null int64

dtypes: float64(2), int64(1), object(2)

memory usage: 2.4+ KB

None

<script.py> output:

region state individuals family\_members state\_pop

0 East South Central Alabama 2570.0 864.0 4887681

1 Pacific Alaska 1434.0 582.0 735139

2 Mountain Arizona 7259.0 2606.0 7158024

3 West South Central Arkansas 2280.0 432.0 3009733

4 Pacific California 109008.0 20964.0 39461588

<class 'pandas.core.frame.DataFrame'>

Int64Index: 51 entries, 0 to 50

Data columns (total 5 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 region 51 non-null object

1 state 51 non-null object

2 individuals 51 non-null float64

3 family\_members 51 non-null float64

4 state\_pop 51 non-null int64

dtypes: float64(2), int64(1), object(2)

memory usage: 2.4+ KB

None

# Print the head of the homelessness data

print(homelessness.head())

# Print information about homelessness

print(homelessness.info())

# Print the shape of homelessness

print(homelessness.shape)

region state individuals family\_members state\_pop

0 East South Central Alabama 2570.0 864.0 4887681

1 Pacific Alaska 1434.0 582.0 735139

2 Mountain Arizona 7259.0 2606.0 7158024

3 West South Central Arkansas 2280.0 432.0 3009733

4 Pacific California 109008.0 20964.0 39461588

<class 'pandas.core.frame.DataFrame'>

Int64Index: 51 entries, 0 to 50

Data columns (total 5 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 region 51 non-null object

1 state 51 non-null object

2 individuals 51 non-null float64

3 family\_members 51 non-null float64

4 state\_pop 51 non-null int64

dtypes: float64(2), int64(1), object(2)

memory usage: 2.4+ KB

None

(51, 5)

<script.py> output:

region state individuals family\_members state\_pop

0 East South Central Alabama 2570.0 864.0 4887681

1 Pacific Alaska 1434.0 582.0 735139

2 Mountain Arizona 7259.0 2606.0 7158024

3 West South Central Arkansas 2280.0 432.0 3009733

4 Pacific California 109008.0 20964.0 39461588

<class 'pandas.core.frame.DataFrame'>

Int64Index: 51 entries, 0 to 50

Data columns (total 5 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 region 51 non-null object

1 state 51 non-null object

2 individuals 51 non-null float64

3 family\_members 51 non-null float64

4 state\_pop 51 non-null int64

dtypes: float64(2), int64(1), object(2)

memory usage: 2.4+ KB

None

(51, 5)

# Print the head of the homelessness data

print(homelessness.head())

# Print information about homelessness

print(homelessness.info())

# Print the shape of homelessness

print(homelessness.shape)

# Print a description of homelessness

print(homelessness.describe())

region state individuals family\_members state\_pop

0 East South Central Alabama 2570.0 864.0 4887681

1 Pacific Alaska 1434.0 582.0 735139

2 Mountain Arizona 7259.0 2606.0 7158024

3 West South Central Arkansas 2280.0 432.0 3009733

4 Pacific California 109008.0 20964.0 39461588

<class 'pandas.core.frame.DataFrame'>

Int64Index: 51 entries, 0 to 50

Data columns (total 5 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 region 51 non-null object

1 state 51 non-null object

2 individuals 51 non-null float64

3 family\_members 51 non-null float64

4 state\_pop 51 non-null int64

dtypes: float64(2), int64(1), object(2)

memory usage: 2.4+ KB

None

(51, 5)

individuals family\_members state\_pop

count 51.000 51.000 5.100e+01

mean 7225.784 3504.882 6.406e+06

std 15991.025 7805.412 7.327e+06

min 434.000 75.000 5.776e+05

25% 1446.500 592.000 1.777e+06

50% 3082.000 1482.000 4.461e+06

75% 6781.500 3196.000 7.341e+06

max 109008.000 52070.000 3.946e+07

<script.py> output:

region state individuals family\_members state\_pop

0 East South Central Alabama 2570.0 864.0 4887681

1 Pacific Alaska 1434.0 582.0 735139

2 Mountain Arizona 7259.0 2606.0 7158024

3 West South Central Arkansas 2280.0 432.0 3009733

4 Pacific California 109008.0 20964.0 39461588

<class 'pandas.core.frame.DataFrame'>

Int64Index: 51 entries, 0 to 50

Data columns (total 5 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 region 51 non-null object

1 state 51 non-null object

2 individuals 51 non-null float64

3 family\_members 51 non-null float64

4 state\_pop 51 non-null int64

dtypes: float64(2), int64(1), object(2)

memory usage: 2.4+ KB

None

(51, 5)

individuals family\_members state\_pop

count 51.000 51.000 5.100e+01

mean 7225.784 3504.882 6.406e+06

std 15991.025 7805.412 7.327e+06

min 434.000 75.000 5.776e+05

25% 1446.500 592.000 1.777e+06

50% 3082.000 1482.000 4.461e+06

75% 6781.500 3196.000 7.341e+06

max 109008.000 52070.000 3.946e+07

##### Exercise

# Parts of a DataFrame

To better understand DataFrame objects, it's useful to know that they consist of three components, stored as attributes:

* .values: A two-dimensional NumPy array of values.
* .columns: An index of columns: the column names.
* .index: An index for the rows: either row numbers or row names.

You can usually think of indexes as a list of strings or numbers, though the pandas Index data type allows for more sophisticated options. (These will be covered later in the course.)

homelessness is available.

##### Instructions

**100 XP**

* Import pandas using the alias pd.
* Print a 2D NumPy array of the values in homelessness.
* Print the column names of homelessness.
* Print the index of homelessness.

# Import pandas using the alias pd

\_\_\_\_

# Print the values of homelessness

\_\_\_\_

# Print the column index of homelessness

\_\_\_\_

# Print the row index of homelessness

\_\_\_\_

# Print the head of the homelessness data

print(homelessness.head())

# Print information about homelessness

print(homelessness.info())

# Print the shape of homelessness

print(homelessness.shape)

# Print a description of homelessness

print(homelessness.describe())

# Import pandas using the alias pd

import pandas as pd

# Print the values of homelessness

print(homelessness.values)

# Print the column index of homelessness

print(homelessness.columns)

# Print the row index of homelessness

print(homelessness.index)

[['East South Central' 'Alabama' 2570.0 864.0 4887681]

['Pacific' 'Alaska' 1434.0 582.0 735139]

['Mountain' 'Arizona' 7259.0 2606.0 7158024]

['West South Central' 'Arkansas' 2280.0 432.0 3009733]

['Pacific' 'California' 109008.0 20964.0 39461588]

['Mountain' 'Colorado' 7607.0 3250.0 5691287]

['New England' 'Connecticut' 2280.0 1696.0 3571520]

['South Atlantic' 'Delaware' 708.0 374.0 965479]

['South Atlantic' 'District of Columbia' 3770.0 3134.0 701547]

['South Atlantic' 'Florida' 21443.0 9587.0 21244317]

['South Atlantic' 'Georgia' 6943.0 2556.0 10511131]

['Pacific' 'Hawaii' 4131.0 2399.0 1420593]

['Mountain' 'Idaho' 1297.0 715.0 1750536]

['East North Central' 'Illinois' 6752.0 3891.0 12723071]

['East North Central' 'Indiana' 3776.0 1482.0 6695497]

['West North Central' 'Iowa' 1711.0 1038.0 3148618]

['West North Central' 'Kansas' 1443.0 773.0 2911359]

['East South Central' 'Kentucky' 2735.0 953.0 4461153]

['West South Central' 'Louisiana' 2540.0 519.0 4659690]

['New England' 'Maine' 1450.0 1066.0 1339057]

['South Atlantic' 'Maryland' 4914.0 2230.0 6035802]

['New England' 'Massachusetts' 6811.0 13257.0 6882635]

['East North Central' 'Michigan' 5209.0 3142.0 9984072]

['West North Central' 'Minnesota' 3993.0 3250.0 5606249]

['East South Central' 'Mississippi' 1024.0 328.0 2981020]

['West North Central' 'Missouri' 3776.0 2107.0 6121623]

['Mountain' 'Montana' 983.0 422.0 1060665]

['West North Central' 'Nebraska' 1745.0 676.0 1925614]

['Mountain' 'Nevada' 7058.0 486.0 3027341]

['New England' 'New Hampshire' 835.0 615.0 1353465]

['Mid-Atlantic' 'New Jersey' 6048.0 3350.0 8886025]

['Mountain' 'New Mexico' 1949.0 602.0 2092741]

['Mid-Atlantic' 'New York' 39827.0 52070.0 19530351]

['South Atlantic' 'North Carolina' 6451.0 2817.0 10381615]

['West North Central' 'North Dakota' 467.0 75.0 758080]

['East North Central' 'Ohio' 6929.0 3320.0 11676341]

['West South Central' 'Oklahoma' 2823.0 1048.0 3940235]

['Pacific' 'Oregon' 11139.0 3337.0 4181886]

['Mid-Atlantic' 'Pennsylvania' 8163.0 5349.0 12800922]

['New England' 'Rhode Island' 747.0 354.0 1058287]

['South Atlantic' 'South Carolina' 3082.0 851.0 5084156]

['West North Central' 'South Dakota' 836.0 323.0 878698]

['East South Central' 'Tennessee' 6139.0 1744.0 6771631]

['West South Central' 'Texas' 19199.0 6111.0 28628666]

['Mountain' 'Utah' 1904.0 972.0 3153550]

['New England' 'Vermont' 780.0 511.0 624358]

['South Atlantic' 'Virginia' 3928.0 2047.0 8501286]

['Pacific' 'Washington' 16424.0 5880.0 7523869]

['South Atlantic' 'West Virginia' 1021.0 222.0 1804291]

['East North Central' 'Wisconsin' 2740.0 2167.0 5807406]

['Mountain' 'Wyoming' 434.0 205.0 577601]]

Index(['region', 'state', 'individuals', 'family\_members', 'state\_pop'], dtype='object')

Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48,

49, 50],

dtype='int64')

# Import pandas using the alias pd

import pandas as pd

# Print the values of homelessness

print(homelessness.values)

# Print the column index of homelessness

print(homelessness.columns)

# Print the row index of homelessness

print(homelessness.index)

In this video, we'll cover the two simplest and possibly most important ways to find interesting parts of your DataFrame.

## 2. Sorting

The first thing you can do is change the order of the rows by sorting them so that the most interesting data is at the top of the DataFrame. You can sort rows using the sort\_values method, passing in a column name that you want to sort by. For example, when we apply sort\_values on the weight\_kg column of the dogs DataFrame, we get the lightest dog at the top, Stella the Chihuahua, and the heaviest dog at the bottom, Bernie the Saint Bernard.

## 3. Sorting in descending order

Setting the ascending argument to False will sort the data the other way around, from heaviest dog to lightest dog.

## 4. Sorting by multiple variables

We can sort by multiple variables by passing a list of column names to sort\_values. Here, we sort first by weight, then by height. Now, Charlie, Lucy, and Bella are ordered from shortest to tallest, even though they all weigh the same.

## 5. Sorting by multiple variables

To change the direction values are sorted in, pass a list to the ascending argument to specify which direction sorting should be done for each variable. Now, Charlie, Lucy, and Bella are ordered from tallest to shortest.

## 6. Subsetting columns

We may want to zoom in on just one column. We can do this using the name of the DataFrame, followed by square brackets with a column name inside. Here, we can look at just the name column.

## 7. Subsetting multiple columns

To select multiple columns, you need two pairs of square brackets. In this code, the inner and outer square brackets are performing different tasks. The outer square brackets are responsible for subsetting the DataFrame, and the inner square brackets are creating a list of column names to subset. This means you could provide a separate list of column names as a variable and then use that list to perform the same subsetting. Usually, it's easier to do in one line.

## 8. Subsetting rows

There are lots of different ways to subset rows. The most common way to do this is by creating a logical condition to filter against. For example, let's find all the dogs whose height is greater than 50 centimeters. Now we have a True or False value for every row.

## 9. Subsetting rows

We can use the logical condition inside of square brackets to subset the rows we're interested in to get all of the dogs taller than 50 centimeters.

## 10. Subsetting based on text data

We can also subset rows based on text data. Here, we use the double equal sign in the logical condition to filter the dogs that are Labradors.

## 11. Subsetting based on dates

We can also subset based on dates. Here, we filter all the dogs born before 2015. Notice that the dates are in quotes and are written as year then month, then day. This is the international standard date format.

## 12. Subsetting based on multiple conditions

To subset the rows that meet multiple conditions, you can combine conditions using logical operators, such as the "and" operator seen here. This means that only rows that meet both of these conditions will be subsetted. You could also do this in one line of code, but you'll also need to add parentheses around each condition.

## 13. Subsetting using .isin()

If you want to filter on multiple values of a categorical variable, the easiest way is to use the isin method. This takes in a list of values to filter for. Here, we check if the color of a dog is black or brown, and use this condition to subset the data.

## 14. Let's practice!

Now it's time to practice your sorting and subsetting!

##### Exercise

# Sorting rows

Finding interesting bits of data in a DataFrame is often easier if you change the order of the rows. You can sort the rows by passing a column name to .sort\_values().

In cases where rows have the same value (this is common if you sort on a categorical variable), you may wish to break the ties by sorting on another column. You can sort on multiple columns in this way by passing a list of column names.

| **Sort on …** | **Syntax** |
| --- | --- |
| one column | df.sort\_values("breed") |
| multiple columns | df.sort\_values(["breed", "weight\_kg"]) |

By combining .sort\_values() with .head(), you can answer questions in the form, "What are the top cases where…?".

homelessness is available and pandas is loaded as pd.

##### Instructions 1/3

**35 XP**

* [1](javascript:void(0))
  + Sort homelessness by the number of homeless individuals, from smallest to largest, and save this as homelessness\_ind.
  + Print the head of the sorted DataFrame.

 [2](javascript:void(0))

* Sort homelessness by the number of homeless family\_members in descending order, and save this as homelessness\_fam.
* Print the head of the sorted DataFrame.
* # Sort homelessness by individuals
* homelessness\_ind = \_\_\_\_
* # Print the top few rows
* print(\_\_\_\_)

# Import pandas using the alias pd import pandas as pd # Print the values of homelessness print(homelessness.values) # Print the column index of homelessness print(homelessness.columns) # Print the row index of homelessness print(homelessness.index

# Sort homelessness by individuals

homelessness\_ind = homelessness.sort\_values("individuals")

# Print the top few rows

print(homelessness\_ind.head())

region state individuals family\_members state\_pop

50 Mountain Wyoming 434.0 205.0 577601

34 West North Central North Dakota 467.0 75.0 758080

7 South Atlantic Delaware 708.0 374.0 965479

39 New England Rhode Island 747.0 354.0 1058287

45 New England Vermont 780.0 511.0 624358

<script.py> output:

region state individuals family\_members state\_pop

50 Mountain Wyoming 434.0 205.0 577601

34 West North Central North Dakota 467.0 75.0 758080

7 South Atlantic Delaware 708.0 374.0 965479

39 New England Rhode Island 747.0 354.0 1058287

45 New England Vermont 780.0 511.0 624358

# Sort homelessness by descending family members

homelessness\_fam = homelessness.sort\_values("family\_members", ascending= False)

# Print the top few rows

print(homelessness\_fam.head())

# Sort homelessness by descending family members

homelessness\_fam = homelessness.sort\_values("family\_members", ascending= False)

# Print the top few rows

print(homelessness\_fam.head())

region state individuals family\_members state\_pop

32 Mid-Atlantic New York 39827.0 52070.0 19530351

4 Pacific California 109008.0 20964.0 39461588

21 New England Massachusetts 6811.0 13257.0 6882635

9 South Atlantic Florida 21443.0 9587.0 21244317

43 West South Central Texas 19199.0 6111.0 28628666

# Sort homelessness by region, then descending family members

homelessness\_reg\_fam = homelessness.sort\_values(["region", "family\_members"], ascending = [True, False])

# Print the top few rows

print(homelessness\_reg\_fam.head())

# Sort homelessness by region, then descending family members

homelessness\_reg\_fam = homelessness.sort\_values(["region", "family\_members"], ascending = [True, False])

# Print the top few rows

print(homelessness\_reg\_fam.head())

region state individuals family\_members state\_pop

13 East North Central Illinois 6752.0 3891.0 12723071

35 East North Central Ohio 6929.0 3320.0 11676341

22 East North Central Michigan 5209.0 3142.0 9984072

49 East North Central Wisconsin 2740.0 2167.0 5807406

14 East North Central Indiana 3776.0 1482.0 6695497

**Daily XP400**

##### Exercise

##### Exercise

# Subsetting columns

When working with data, you may not need all of the variables in your dataset. Square brackets ([]) can be used to select only the columns that matter to you in an order that makes sense to you. To select only "col\_a" of the DataFrame df, use

df["col\_a"]

To select "col\_a" and "col\_b" of df, use

df[["col\_a", "col\_b"]]

homelessness is available and pandas is loaded as pd.

##### Instructions 1/3

**35 XP**

* [1](javascript:void(0))
  + Create a DataFrame called individuals that contains only the individuals column of homelessness.
  + Print the head of the result.

 [2](javascript:void(0))

* Create a DataFrame called state\_fam that contains only the state and family\_members columns of homelessness, in that order.
* Print the head of the result.

 [3](javascript:void(0))

* Create a DataFrame called ind\_state that contains the individuals and state columns of homelessness, in that order.
* Print the head of the result.

# Select the individuals column

individuals = \_\_\_\_

# Print the head of the result

\_\_\_\_

# Sort homelessness by region, then descending family members homelessness\_reg\_fam = homelessness.sort\_values(["region", "family\_members"], ascending = [True, False]) # Print the top few rows print(homelessness\_reg\_fam.head())

# Select the individuals column

individuals = homelessness["individuals"]

# Print the head of the result

print(individuals)

# Select the individuals column

individuals = homelessness["individuals"]

# Print the head of the result

print(individuals)

0 2570.0

1 1434.0

2 7259.0

3 2280.0

4 109008.0

5 7607.0

6 2280.0

7 708.0

8 3770.0

9 21443.0

10 6943.0

11 4131.0

12 1297.0

13 6752.0

14 3776.0

15 1711.0

16 1443.0

17 2735.0

18 2540.0

19 1450.0

20 4914.0

21 6811.0

22 5209.0

23 3993.0

24 1024.0

25 3776.0

26 983.0

27 1745.0

28 7058.0

29 835.0

30 6048.0

31 1949.0

32 39827.0

33 6451.0

34 467.0

35 6929.0

36 2823.0

37 11139.0

38 8163.0

39 747.0

40 3082.0

41 836.0

42 6139.0

43 19199.0

44 1904.0

45 780.0

46 3928.0

47 16424.0

48 1021.0

49 2740.0

50 434.0

Name: individuals, dtype: float64

# Select the individuals column

individuals = homelessness["individuals"]

# Print the head of the result

print(individuals.head())

# Select the individuals column

individuals = homelessness["individuals"]

# Print the head of the result

print(individuals.head())

0 2570.0

1 1434.0

2 7259.0

3 2280.0

4 109008.0

Name: individuals, dtype: float64

# Select the state and family\_members columns

state\_fam = homelessness[["state" , "family\_members"]]

# Print the head of the result

print(state\_fam.head())

# Select the state and family\_members columns

state\_fam = homelessness[["state" , "family\_members"]]

# Print the head of the result

print(state\_fam.head())

state family\_members

0 Alabama 864.0

1 Alaska 582.0

2 Arizona 2606.0

3 Arkansas 432.0

4 California 20964.0

**Daily XP500**

##### Exercise

##### Exercise

# Subsetting rows

A large part of data science is about finding which bits of your dataset are interesting. One of the simplest techniques for this is to find a subset of rows that match some criteria. This is sometimes known as filtering rows or selecting rows.

There are many ways to subset a DataFrame, perhaps the most common is to use relational operators to return True or False for each row, then pass that inside square brackets.

dogs[dogs["height\_cm"] > 60]

dogs[dogs["color"] == "tan"]

You can filter for multiple conditions at once by using the "bitwise and" operator, &.

dogs[(dogs["height\_cm"] > 60) & (dogs["color"] == "tan")]

homelessness is available and pandas is loaded as pd.

##### Instructions 1/3

**35 XP**

* [1](javascript:void(0))

Filter homelessness for cases where the number of individuals is greater than ten thousand, assigning to ind\_gt\_10k. View the printed result.

 [2](javascript:void(0))

Filter homelessness for cases where the USA Census region is "Mountain", assigning to mountain\_reg. View the printed result.

 [3](javascript:void(0))

Filter homelessness for cases where the number of family\_members is less than one thousand and the region is "Pacific", assigning to fam\_lt\_1k\_pac. View the printed result.

# Filter for rows where individuals is greater than 10000

ind\_gt\_10k = \_\_\_\_

# See the result

print(ind\_gt\_10k)

# Select only the individuals and state columns, in that order ind\_state = homelessness[["individuals", "state"]] # Print the head of the result print(ind\_state.head())

# Filter for rows where individuals is greater than 10000

ind\_gt\_10k = homelessness[homelessness["individuals"] >10000]

# See the result

print(ind\_gt\_10k)

# Filter for rows where individuals is greater than 10000

ind\_gt\_10k = homelessness[homelessness["individuals"] >10000]

# See the result

print(ind\_gt\_10k)

region state individuals family\_members state\_pop

4 Pacific California 109008.0 20964.0 39461588

9 South Atlantic Florida 21443.0 9587.0 21244317

32 Mid-Atlantic New York 39827.0 52070.0 19530351

37 Pacific Oregon 11139.0 3337.0 4181886

43 West South Central Texas 19199.0 6111.0 28628666

47 Pacific Washington 16424.0 5880.0 7523869

# Filter for rows where region is Mountain

mountain\_reg = \_\_\_\_

# See the result

\_\_\_\_

# Filter for rows where region is Mountain

mountain\_reg = homelessness[homelessness["region"] == "Mountain"]

# See the result

print(mountain\_reg)

# Filter for rows where region is Mountain

mountain\_reg = homelessness[homelessness["region"] == "Mountain"]

# See the result

print(mountain\_reg)

region state individuals family\_members state\_pop

2 Mountain Arizona 7259.0 2606.0 7158024

5 Mountain Colorado 7607.0 3250.0 5691287

12 Mountain Idaho 1297.0 715.0 1750536

26 Mountain Montana 983.0 422.0 1060665

28 Mountain Nevada 7058.0 486.0 3027341

31 Mountain New Mexico 1949.0 602.0 2092741

44 Mountain Utah 1904.0 972.0 3153550

50 Mountain Wyoming 434.0 205.0 577601

# Filter for rows where family\_members is less than 1000

# and region is Pacific

fam\_lt\_1k\_pac = \_\_\_\_

# See the result

print(fam\_lt\_1k\_pac)

# Filter for rows where family\_members is less than 1000

# and region is Pacific

fam\_lt\_1k\_pac = homelessness[(homelessness["family\_members"]<1000) & (homelessness["region"] == "Pacific")]

# See the result

print(fam\_lt\_1k\_pac)

# Filter for rows where family\_members is less than 1000

# and region is Pacific

fam\_lt\_1k\_pac = homelessness[(homelessness["family\_members"]<1000) & (homelessness["region"] == "Pacific")]

# See the result

print(fam\_lt\_1k\_pac)

region state individuals family\_members state\_pop

1 Pacific Alaska 1434.0 582.0 735139

# Select only the individuals and state columns, in that order

ind\_state = homelessness[["individuals", "state"]]

# Print the head of the result

print(ind\_state.head())

# Filter for rows where individuals is greater than 10000

ind\_gt\_10k = homelessness[homelessness["individuals" >10000]]

# See the result

print(ind\_gt\_10k)

Traceback (most recent call last):

File "<stdin>", line 72, in exceptionCatcher

raise exception

File "<stdin>", line 3361, in run\_ast\_nodes

if (await self.run\_code(code, result, async\_=asy)):

File "<stdin>", line 3458, in run\_code

self.showtraceback(running\_compiled\_code=True)

File "<stdin>", line 2066, in showtraceback

self.\_showtraceback(etype, value, stb)

File "<stdin>", line 72, in exceptionCatcher

raise exception

File "<stdin>", line 3441, in run\_code

exec(code\_obj, self.user\_global\_ns, self.user\_ns)

File "<stdin>", line 2, in <module>

ind\_gt\_10k = homelessness[homelessness["individuals" >10000]]

TypeError: '>' not supported between instances of 'str' and 'int'

# Filter for rows where individuals is greater than 10000

ind\_gt\_10k = homelessness[homelessness["individuals"] >10000]

# See the result

print(ind\_gt\_10k)

region state individuals family\_members state\_pop

4 Pacific California 109008.0 20964.0 39461588

9 South Atlantic Florida 21443.0 9587.0 21244317

32 Mid-Atlantic New York 39827.0 52070.0 19530351

37 Pacific Oregon 11139.0 3337.0 4181886

43 West South Central Texas 19199.0 6111.0 28628666

47 Pacific Washington 16424.0 5880.0 7523869

<script.py> output:

region state individuals family\_members state\_pop

4 Pacific California 109008.0 20964.0 39461588

9 South Atlantic Florida 21443.0 9587.0 21244317

32 Mid-Atlantic New York 39827.0 52070.0 19530351

37 Pacific Oregon 11139.0 3337.0 4181886

43 West South Central Texas 19199.0 6111.0 28628666

47 Pacific Washington 16424.0 5880.0 7523869

# Filter for rows where region is Mountain

mountain\_reg = homelessness[homelessness["region"] == "Mountain"]

# See the result

print(mountain\_reg)

region state individuals family\_members state\_pop

2 Mountain Arizona 7259.0 2606.0 7158024

5 Mountain Colorado 7607.0 3250.0 5691287

12 Mountain Idaho 1297.0 715.0 1750536

26 Mountain Montana 983.0 422.0 1060665

28 Mountain Nevada 7058.0 486.0 3027341

31 Mountain New Mexico 1949.0 602.0 2092741

44 Mountain Utah 1904.0 972.0 3153550

50 Mountain Wyoming 434.0 205.0 577601

<script.py> output:

region state individuals family\_members state\_pop

2 Mountain Arizona 7259.0 2606.0 7158024

5 Mountain Colorado 7607.0 3250.0 5691287

12 Mountain Idaho 1297.0 715.0 1750536

26 Mountain Montana 983.0 422.0 1060665

28 Mountain Nevada 7058.0 486.0 3027341

31 Mountain New Mexico 1949.0 602.0 2092741

44 Mountain Utah 1904.0 972.0 3153550

50 Mountain Wyoming 434.0 205.0 577601

# Filter for rows where family\_members is less than 1000

# and region is Pacific

fam\_lt\_1k\_pac = homelessness[(homelessness["family\_members"]<1000) & (homelessness["region"] == "Pacific")]

# See the result

print(fam\_lt\_1k\_pac)

region state individuals family\_members state\_pop

1 Pacific Alaska 1434.0 582.0 735139

# Filter for rows where individuals is greater than 10000

ind\_gt\_10k = homelessness[homelessness["individuals"] >10000]

# See the result

print(ind\_gt\_10k)

region state individuals family\_members state\_pop

4 Pacific California 109008.0 20964.0 39461588

9 South Atlantic Florida 21443.0 9587.0 21244317

32 Mid-Atlantic New York 39827.0 52070.0 19530351

37 Pacific Oregon 11139.0 3337.0 4181886

43 West South Central Texas 19199.0 6111.0 28628666

47 Pacific Washington 16424.0 5880.0 7523869

<script.py> output:

region state individuals family\_members state\_pop

4 Pacific California 109008.0 20964.0 39461588

9 South Atlantic Florida 21443.0 9587.0 21244317

32 Mid-Atlantic New York 39827.0 52070.0 19530351

37 Pacific Oregon 11139.0 3337.0 4181886

43 West South Central Texas 19199.0 6111.0 28628666

47 Pacific Washington 16424.0 5880.0 7523869

# Filter for rows where region is Mountain

mountain\_reg = homelessness[homelessness["region"] == "Mountain"]

# See the result

print(mountain\_reg)

region state individuals family\_members state\_pop

2 Mountain Arizona 7259.0 2606.0 7158024

5 Mountain Colorado 7607.0 3250.0 5691287

12 Mountain Idaho 1297.0 715.0 1750536

26 Mountain Montana 983.0 422.0 1060665

28 Mountain Nevada 7058.0 486.0 3027341

31 Mountain New Mexico 1949.0 602.0 2092741

44 Mountain Utah 1904.0 972.0 3153550

50 Mountain Wyoming 434.0 205.0 577601

<script.py> output:

region state individuals family\_members state\_pop

2 Mountain Arizona 7259.0 2606.0 7158024

5 Mountain Colorado 7607.0 3250.0 5691287

12 Mountain Idaho 1297.0 715.0 1750536

26 Mountain Montana 983.0 422.0 1060665

28 Mountain Nevada 7058.0 486.0 3027341

31 Mountain New Mexico 1949.0 602.0 2092741

44 Mountain Utah 1904.0 972.0 3153550

50 Mountain Wyoming 434.0 205.0 577601

# Filter for rows where family\_members is less than 1000

# and region is Pacific

fam\_lt\_1k\_pac = homelessness[(homelessness["family\_members"]<1000) & (homelessness["region"] == "Pacific")]

# See the result

print(fam\_lt\_1k\_pac)

region state individuals family\_members state\_pop

1 Pacific Alaska 1434.0 582.0 735139

# gorical variables

Subsetting data based on a categorical variable often involves using the "or" operator (|) to select rows from multiple categories. This can get tedious when you want all states in one of three different regions, for example. Instead, use the .isin() method, which will allow you to tackle this problem by writing one condition instead of three separate ones.

colors = ["brown", "black", "tan"]

condition = dogs["color"].isin(colors)

dogs[condition]

homelessness is available and pandas is loaded as pd.

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))

Filter homelessness for cases where the USA census region is "South Atlantic" or it is "Mid-Atlantic", assigning to south\_mid\_atlantic. View the printed result.

 [2](javascript:void(0))

Filter homelessness for cases where the USA census state is in the list of Mojave states, canu, assigning to mojave\_homelessness. View the printed result.

# Subset for rows in South Atlantic or Mid-Atlantic regions

south\_mid\_atlantic = \_\_\_\_

# See the result

print(south\_mid\_atlantic)

# Filter for rows where family\_members is less than 1000 # and region is Pacific fam\_lt\_1k\_pac = homelessness[(homelessness["family\_members"]<1000) & (homelessness["region"] == "Pacific")] # See the result print(fam\_lt\_1k\_pac)

# Subset for rows in South Atlantic or Mid-Atlantic regions

south\_mid\_atlantic = homelessness["region"] .isin(["South Atlantic", "Mid\_Atlantic"])

# See the result

print(south\_mid\_atlantic)

# Subset for rows in South Atlantic or Mid-Atlantic regions

south\_mid\_atlantic = homelessness["region"] .isin(["South Atlantic", "Mid\_Atlantic"])

# See the result

print(south\_mid\_atlantic)

0 False

1 False

2 False

3 False

4 False

5 False

6 False

7 True

8 True

9 True

10 True

11 False

12 False

13 False

14 False

15 False

16 False

17 False

18 False

19 False

20 True

21 False

22 False

23 False

24 False

25 False

26 False

27 False

28 False

29 False

30 False

31 False

32 False

33 True

34 False

35 False

36 False

37 False

38 False

39 False

40 True

41 False

42 False

43 False

44 False

45 False

46 True

47 False

48 True

49 False

50 False

Name: region, dtype: bool

# Subset for rows in South Atlantic or Mid-Atlantic regions

south\_mid\_atlantic = homelessness[(homelessness["region"] .isin(["South Atlantic", "Mid-Atlantic"]))]

# See the result

print(south\_mid\_atlantic)

region state individuals family\_members state\_pop

7 South Atlantic Delaware 708.0 374.0 965479

8 South Atlantic District of Columbia 3770.0 3134.0 701547

9 South Atlantic Florida 21443.0 9587.0 21244317

10 South Atlantic Georgia 6943.0 2556.0 10511131

20 South Atlantic Maryland 4914.0 2230.0 6035802

30 Mid-Atlantic New Jersey 6048.0 3350.0 8886025

32 Mid-Atlantic New York 39827.0 52070.0 19530351

33 South Atlantic North Carolina 6451.0 2817.0 10381615

38 Mid-Atlantic Pennsylvania 8163.0 5349.0 12800922

40 South Atlantic South Carolina 3082.0 851.0 5084156

46 South Atlantic Virginia 3928.0 2047.0 8501286

48 South Atlantic West Virginia 1021.0 222.0 1804291

<script.py> output:

region state individuals family\_members state\_pop

7 South Atlantic Delaware 708.0 374.0 965479

8 South Atlantic District of Columbia 3770.0 3134.0 701547

9 South Atlantic Florida 21443.0 9587.0 21244317

10 South Atlantic Georgia 6943.0 2556.0 10511131

20 South Atlantic Maryland 4914.0 2230.0 6035802

30 Mid-Atlantic New Jersey 6048.0 3350.0 8886025

32 Mid-Atlantic New York 39827.0 52070.0 19530351

33 South Atlantic North Carolina 6451.0 2817.0 10381615

38 Mid-Atlantic Pennsylvania 8163.0 5349.0 12800922

40 South Atlantic South Carolina 3082.0 851.0 5084156

46 South Atlantic Virginia 3928.0 2047.0 8501286

48 South Atlantic West Virginia 1021.0 222.0 1804291

# The Mojave Desert states

canu = ["California", "Arizona", "Nevada", "Utah"]

# Filter for rows in the Mojave Desert states

mojave\_homelessness = \_\_\_\_

# See the result

print(mojave\_homelessness)

# The Mojave Desert states

canu = ["California", "Arizona", "Nevada", "Utah"]

# Filter for rows in the Mojave Desert states

mojave\_homelessness = homelessness[(homelessness["state"] .isin(canu))]

# See the result

print(mojave\_homelessness)

# Filter for rows in the Mojave Desert states

mojave\_homelessness = homelessness[(homelessness["state"] .isin(canu))]

# See the result

print(mojave\_homelessness)

region state individuals family\_members state\_pop

2 Mountain Arizona 7259.0 2606.0 7158024

4 Pacific California 109008.0 20964.0 39461588

28 Mountain Nevada 7058.0 486.0 3027341

44 Mountain Utah 1904.0 972.0 3153550

**Daily XP100**

# New columns

**50 XP**

## 1. New columns

In the last lesson, you saw how to subset and sort a DataFrame to extract interesting bits. However, often when you first receive a DataFrame, the contents aren't exactly what you want. You may have to add new columns derived from existing columns.

## 2. Adding a new column

Creating and adding new columns can go by many names, including mutating a DataFrame, transforming a DataFrame, and feature engineering. Let's say we want to add a new column to our DataFrame that has each dog's height in meters instead of centimeters. On the left-hand side of the equals, we use square brackets with the name of the new column we want to create. On the right-hand side, we have the calculation. Notice that both the existing column and the new column we just created are in the DataFrame.

## 3. Doggy mass index

Let's see what the results are if we calculate the body mass index, or BMI, of these dogs. BMI is usually calculated by taking a person's weight in kilograms and dividing it by their height in meters, squared. Instead of doing this with people, we'll try it out with dogs. Again, the new column is on the left-hand side of the equals, but this time, our calculation involves two columns.

## 4. Multiple manipulations

The real power of pandas comes in when you combine all the skills you've learned so far. Let's figure out the names of skinny, tall dogs. First, to define the skinny dogs, we take the subset of the dogs who have a BMI of under 100. Next, we sort the result in descending order of height to get the tallest skinny dogs at the top. Finally, we keep only the columns we're interested in. Here, you can see that Max is the tallest dog with a BMI of under 100.

## 5. Let's practice!

Time to practice your pandas powers!

**Daily XP350**

# Introducing DataFrames

**50 XP**

## 1. Introducing DataFrames

Hi, I'm Richie. I'll be your tour guide through the world of pandas.

## 2. What's the point of pandas?

pandas is a Python package for data manipulation. It can also be used for data visualization; we'll get to that in Chapter 4.

## 3. Course outline

We'll start by talking about DataFrames, which form the core of pandas. In chapter 2, we'll discuss aggregating data to gather insights. In chapter 3, you'll learn all about slicing and indexing to subset DataFrames. Finally, you'll visualize your data, deal with missing data, and read data into a DataFrame. Let's dive in.

## 4. pandas is built on NumPy and Matplotlib

pandas is built on top of two essential Python packages, NumPy and Matplotlib. Numpy provides multidimensional array objects for easy data manipulation that pandas uses to store data, and Matplotlib has powerful data visualization capabilities that pandas takes advantage of.

## 5. pandas is popular

pandas has millions of users, with PyPi recording about 14 million downloads in December 2019. This represents almost the entire Python data science community!

1. 1 https://pypistats.org/packages/pandas

## 6. Rectangular data

There are several ways to store data for analysis, but rectangular data, sometimes called "tabular data" is the most common form. In this example, with dogs, each observation, or each dog, is a row, and each variable, or each dog property, is a column. pandas is designed to work with rectangular data like this.

## 7. pandas DataFrames

In pandas, rectangular data is represented as a DataFrame object. Every programming language used for data analysis has something similar to this. R also has DataFrames, while SQL has database tables. Every value within a column has the same data type, either text or numeric, but different columns can contain different data types.

## 8. Exploring a DataFrame: .head()

When you first receive a new dataset, you want to quickly explore it and get a sense of its contents. pandas has several methods for this. The first is head, which returns the first few rows of the DataFrame. We only had seven rows to begin with, so it's not super exciting, but this becomes very useful if you have many rows.

## 9. Exploring a DataFrame: .info()

The info method displays the names of columns, the data types they contain, and whether they have any missing values.

## 10. Exploring a DataFrame: .shape

A DataFrame's shape attribute contains a tuple that holds the number of rows followed by the number of columns. Since this is an attribute instead of a method, you write it without parentheses.

## 11. Exploring a DataFrame: .describe()

The describe method computes some summary statistics for numerical columns, like mean and median. "count" is the number of non-missing values in each column. describe is good for a quick overview of numeric variables, but if you want more control, you'll see how to perform more specific calculations later in the course.

## 12. Components of a DataFrame: .values

DataFrames consist of three different components, accessible using attributes. The values attribute, as you might expect, contains the data values in a 2-dimensional NumPy array.

## 13. Components of a DataFrame: .columns and .index

The other two components of a DataFrame are labels for columns and rows. The columns attribute contains column names, and the index attribute contains row numbers or row names. Be careful, since row labels are stored in dot-index, not in dot-rows. Notice that these are Index objects, which we'll cover in Chapter 3. This allows for flexibility in labels. For example, the dogs data uses row numbers, but row names are also possible.

## 14. pandas Philosophy

Python has a semi-official philosophy on how to write good code called The Zen of Python. One suggestion is that given a programming problem, there should only be one obvious solution. As you go through this course, bear in mind that pandas deliberately doesn't follow this philosophy. Instead, there are often multiple ways to solve a problem, leaving you to choose the best. In this respect, pandas is like a Swiss Army Knife, giving you a variety of tools, making it incredibly powerful, but more difficult to learn. In this course, we aim for a more streamlined approach to pandas, only covering the most important ways of doing things.

1. 1 https://www.python.org/dev/peps/pep-0020/

## 15. Let's practice!

Enough meditating, time to write some code!

**Daily XP350**

##### Exercise

##### Exercise

# Inspecting a DataFrame

When you get a new DataFrame to work with, the first thing you need to do is explore it and see what it contains. There are several useful methods and attributes for this.

* .head() returns the first few rows (the “head” of the DataFrame).
* .info() shows information on each of the columns, such as the data type and number of missing values.
* .shape returns the number of rows and columns of the DataFrame.
* .describe() calculates a few summary statistics for each column.

homelessness is a DataFrame containing estimates of homelessness in each U.S. state in 2018. The individual column is the number of homeless individuals not part of a family with children. The family\_members column is the number of homeless individuals part of a family with children. The state\_pop column is the state's total population.

pandas is imported for you.

##### Instructions 1/4

**25 XP**

* Print the head of the homelessness DataFrame.
* # Print the head of the homelessness data
* print(homelessness.head())

# Print the head of the homelessness data

print(homelessness.head())

region state individuals family\_members state\_pop

0 East South Central Alabama 2570.0 864.0 4887681

1 Pacific Alaska 1434.0 582.0 735139

2 Mountain Arizona 7259.0 2606.0 7158024

3 West South Central Arkansas 2280.0 432.0 3009733

4 Pacific California 109008.0 20964.0 39461588

**Daily XP350**

##### Exercise

##### Exercise

# Parts of a DataFrame

To better understand DataFrame objects, it's useful to know that they consist of three components, stored as attributes:

* .values: A two-dimensional NumPy array of values.
* .columns: An index of columns: the column names.
* .index: An index for the rows: either row numbers or row names.

You can usually think of indexes as a list of strings or numbers, though the pandas Index data type allows for more sophisticated options. (These will be covered later in the course.)

homelessness is available.

##### Instructions

**100 XP**

* Import pandas using the alias pd.
* Print a 2D NumPy array of the values in homelessness.
* Print the column names of homelessness.
* Print the index of homelessness.
* # Import pandas using the alias pd
* import pandas as pd
* # Print the values of homelessness
* print(homelessness.values)
* # Print the column index of homelessness
* print(homelessness.columns)
* # Print the row index of homelessness
* print(homelessness.index)
* IPython Shell

# Print the head of the homelessness data

print(homelessness.head())

# Import pandas using the alias pd

import pandas as pd

# Print the values of homelessness

print(homelessness.values)

# Print the column index of homelessness

print(homelessness.columns)

# Print the row index of homelessness

print(homelessness.index)

[['East South Central' 'Alabama' 2570.0 864.0 4887681]

['Pacific' 'Alaska' 1434.0 582.0 735139]

['Mountain' 'Arizona' 7259.0 2606.0 7158024]

['West South Central' 'Arkansas' 2280.0 432.0 3009733]

['Pacific' 'California' 109008.0 20964.0 39461588]

['Mountain' 'Colorado' 7607.0 3250.0 5691287]

['New England' 'Connecticut' 2280.0 1696.0 3571520]

['South Atlantic' 'Delaware' 708.0 374.0 965479]

['South Atlantic' 'District of Columbia' 3770.0 3134.0 701547]

['South Atlantic' 'Florida' 21443.0 9587.0 21244317]

['South Atlantic' 'Georgia' 6943.0 2556.0 10511131]

['Pacific' 'Hawaii' 4131.0 2399.0 1420593]

['Mountain' 'Idaho' 1297.0 715.0 1750536]

['East North Central' 'Illinois' 6752.0 3891.0 12723071]

['East North Central' 'Indiana' 3776.0 1482.0 6695497]

['West North Central' 'Iowa' 1711.0 1038.0 3148618]

['West North Central' 'Kansas' 1443.0 773.0 2911359]

['East South Central' 'Kentucky' 2735.0 953.0 4461153]

['West South Central' 'Louisiana' 2540.0 519.0 4659690]

['New England' 'Maine' 1450.0 1066.0 1339057]

['South Atlantic' 'Maryland' 4914.0 2230.0 6035802]

['New England' 'Massachusetts' 6811.0 13257.0 6882635]

['East North Central' 'Michigan' 5209.0 3142.0 9984072]

['West North Central' 'Minnesota' 3993.0 3250.0 5606249]

['East South Central' 'Mississippi' 1024.0 328.0 2981020]

['West North Central' 'Missouri' 3776.0 2107.0 6121623]

['Mountain' 'Montana' 983.0 422.0 1060665]

['West North Central' 'Nebraska' 1745.0 676.0 1925614]

['Mountain' 'Nevada' 7058.0 486.0 3027341]

['New England' 'New Hampshire' 835.0 615.0 1353465]

['Mid-Atlantic' 'New Jersey' 6048.0 3350.0 8886025]

['Mountain' 'New Mexico' 1949.0 602.0 2092741]

['Mid-Atlantic' 'New York' 39827.0 52070.0 19530351]

['South Atlantic' 'North Carolina' 6451.0 2817.0 10381615]

['West North Central' 'North Dakota' 467.0 75.0 758080]

['East North Central' 'Ohio' 6929.0 3320.0 11676341]

['West South Central' 'Oklahoma' 2823.0 1048.0 3940235]

['Pacific' 'Oregon' 11139.0 3337.0 4181886]

['Mid-Atlantic' 'Pennsylvania' 8163.0 5349.0 12800922]

['New England' 'Rhode Island' 747.0 354.0 1058287]

['South Atlantic' 'South Carolina' 3082.0 851.0 5084156]

['West North Central' 'South Dakota' 836.0 323.0 878698]

['East South Central' 'Tennessee' 6139.0 1744.0 6771631]

['West South Central' 'Texas' 19199.0 6111.0 28628666]

['Mountain' 'Utah' 1904.0 972.0 3153550]

['New England' 'Vermont' 780.0 511.0 624358]

['South Atlantic' 'Virginia' 3928.0 2047.0 8501286]

['Pacific' 'Washington' 16424.0 5880.0 7523869]

['South Atlantic' 'West Virginia' 1021.0 222.0 1804291]

['East North Central' 'Wisconsin' 2740.0 2167.0 5807406]

['Mountain' 'Wyoming' 434.0 205.0 577601]]

Index(['region', 'state', 'individuals', 'family\_members', 'state\_pop'], dtype='object')

Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48,

49, 50],

dtype='int64')

**Daily XP350**

# Sorting and subsetting

**50 XP**

## 1. Sorting and subsetting

In this video, we'll cover the two simplest and possibly most important ways to find interesting parts of your DataFrame.

## 2. Sorting

The first thing you can do is change the order of the rows by sorting them so that the most interesting data is at the top of the DataFrame. You can sort rows using the sort\_values method, passing in a column name that you want to sort by. For example, when we apply sort\_values on the weight\_kg column of the dogs DataFrame, we get the lightest dog at the top, Stella the Chihuahua, and the heaviest dog at the bottom, Bernie the Saint Bernard.

## 3. Sorting in descending order

Setting the ascending argument to False will sort the data the other way around, from heaviest dog to lightest dog.

## 4. Sorting by multiple variables

We can sort by multiple variables by passing a list of column names to sort\_values. Here, we sort first by weight, then by height. Now, Charlie, Lucy, and Bella are ordered from shortest to tallest, even though they all weigh the same.

## 5. Sorting by multiple variables

To change the direction values are sorted in, pass a list to the ascending argument to specify which direction sorting should be done for each variable. Now, Charlie, Lucy, and Bella are ordered from tallest to shortest.

## 6. Subsetting columns

We may want to zoom in on just one column. We can do this using the name of the DataFrame, followed by square brackets with a column name inside. Here, we can look at just the name column.

## 7. Subsetting multiple columns

To select multiple columns, you need two pairs of square brackets. In this code, the inner and outer square brackets are performing different tasks. The outer square brackets are responsible for subsetting the DataFrame, and the inner square brackets are creating a list of column names to subset. This means you could provide a separate list of column names as a variable and then use that list to perform the same subsetting. Usually, it's easier to do in one line.

## 8. Subsetting rows

There are lots of different ways to subset rows. The most common way to do this is by creating a logical condition to filter against. For example, let's find all the dogs whose height is greater than 50 centimeters. Now we have a True or False value for every row.

## 9. Subsetting rows

We can use the logical condition inside of square brackets to subset the rows we're interested in to get all of the dogs taller than 50 centimeters.

## 10. Subsetting based on text data

We can also subset rows based on text data. Here, we use the double equal sign in the logical condition to filter the dogs that are Labradors.

## 11. Subsetting based on dates

We can also subset based on dates. Here, we filter all the dogs born before 2015. Notice that the dates are in quotes and are written as year then month, then day. This is the international standard date format.

## 12. Subsetting based on multiple conditions

To subset the rows that meet multiple conditions, you can combine conditions using logical operators, such as the "and" operator seen here. This means that only rows that meet both of these conditions will be subsetted. You could also do this in one line of code, but you'll also need to add parentheses around each condition.

## 13. Subsetting using .isin()

If you want to filter on multiple values of a categorical variable, the easiest way is to use the isin method. This takes in a list of values to filter for. Here, we check if the color of a dog is black or brown, and use this condition to subset the data.

## 14. Let's practice!

Now it's time to practice your sorting and subsetting!

**Daily XP350**

##### Exercise

##### Exercise

# Sorting rows

Finding interesting bits of data in a DataFrame is often easier if you change the order of the rows. You can sort the rows by passing a column name to .sort\_values().

In cases where rows have the same value (this is common if you sort on a categorical variable), you may wish to break the ties by sorting on another column. You can sort on multiple columns in this way by passing a list of column names.

| **Sort on …** | **Syntax** |
| --- | --- |
| one column | df.sort\_values("breed") |
| multiple columns | df.sort\_values(["breed", "weight\_kg"]) |

By combining .sort\_values() with .head(), you can answer questions in the form, "What are the top cases where…?".

homelessness is available and pandas is loaded as pd.

##### Instructions 1/3

**35 XP**

* + Sort homelessness by the number of homeless individuals, from smallest to largest, and save this as homelessness\_ind.
  + Print the head of the sorted DataFrame.
  + Sort homelessness by the number of homeless family\_members in descending order, and save this as homelessness\_fam.
  + Print the head of the sorted DataFrame.
  + Sort homelessness first by region (ascending), and then by number of family members (descending). Save this as homelessness\_reg\_fam.
  + Print the head of the sorted DataFrame.
* # Sort homelessness by individuals
* homelessness\_ind = homelessness.sort\_values("individuals")
* # Print the top few rows
* print(homelessness\_ind.head())

# Import pandas using the alias pd

import pandas as pd

# Print the values of homelessness

print(homelessness.values)

# Print the column index of homelessness

print(homelessness.columns)

# Print the row index of homelessness

print(homelessness.index)

# Sort homelessness by individuals

homelessness\_ind = homelessness.sort\_values("individuals")

# Print the top few rows

print(homelessness\_ind.head())

region state individuals family\_members state\_pop

50 Mountain Wyoming 434.0 205.0 577601

34 West North Central North Dakota 467.0 75.0 758080

7 South Atlantic Delaware 708.0 374.0 965479

39 New England Rhode Island 747.0 354.0 1058287

45 New England Vermont 780.0 511.0 624358

**Daily XP350**

##### Exercise

##### Exercise

# Subsetting columns

When working with data, you may not need all of the variables in your dataset. Square brackets ([]) can be used to select only the columns that matter to you in an order that makes sense to you. To select only "col\_a" of the DataFrame df, use

df["col\_a"]

To select "col\_a" and "col\_b" of df, use

df[["col\_a", "col\_b"]]

homelessness is available and pandas is loaded as pd.

##### Instructions 1/3

**35 XP**

* + Create a DataFrame called individuals that contains only the individuals column of homelessness.
  + Print the head of the result.
  + Create a DataFrame called state\_fam that contains only the state and family\_members columns of homelessness, in that order.
  + Print the head of the result.
  + Create a DataFrame called ind\_state that contains the individuals and state columns of homelessness, in that order.
  + Print the head of the result.
* # Select the individuals column
* individuals = homelessness["individuals"]
* # Print the head of the result
* print(individuals.head())

# Sort homelessness by individuals

homelessness\_ind = homelessness.sort\_values("individuals")

# Print the top few rows

print(homelessness\_ind.head())

# Select the individuals column

individuals = homelessness["individuals"]

# Print the head of the result

print(individuals.head())

0 2570.0

1 1434.0

2 7259.0

3 2280.0

4 109008.0

Name: individuals, dtype: float64

**Daily XP350**

##### Exercise

##### Exercise

# Subsetting rows

A large part of data science is about finding which bits of your dataset are interesting. One of the simplest techniques for this is to find a subset of rows that match some criteria. This is sometimes known as filtering rows or selecting rows.

There are many ways to subset a DataFrame, perhaps the most common is to use relational operators to return True or False for each row, then pass that inside square brackets.

dogs[dogs["height\_cm"] > 60]

dogs[dogs["color"] == "tan"]

You can filter for multiple conditions at once by using the "bitwise and" operator, &.

dogs[(dogs["height\_cm"] > 60) & (dogs["color"] == "tan")]

homelessness is available and pandas is loaded as pd.

##### Instructions 1/3

**35 XP**

* Filter homelessness for cases where the number of individuals is greater than ten thousand, assigning to ind\_gt\_10k. View the printed result.
* Filter homelessness for cases where the USA Census region is "Mountain", assigning to mountain\_reg. View the printed result.
* Filter homelessness for cases where the number of family\_members is less than one thousand and the region is "Pacific", assigning to fam\_lt\_1k\_pac. View the printed result.
* # Filter for rows where individuals is greater than 10000
* ind\_gt\_10k = homelessness[homelessness["individuals"] >10000]
* # See the result
* print(ind\_gt\_10k)

# Select the individuals column

individuals = homelessness["individuals"]

# Print the head of the result

print(individuals.head())

# Filter for rows where individuals is greater than 10000

ind\_gt\_10k = homelessness[homelessness["individuals"] >10000]

# See the result

print(ind\_gt\_10k)

region state individuals family\_members state\_pop

4 Pacific California 109008.0 20964.0 39461588

9 South Atlantic Florida 21443.0 9587.0 21244317

32 Mid-Atlantic New York 39827.0 52070.0 19530351

37 Pacific Oregon 11139.0 3337.0 4181886

43 West South Central Texas 19199.0 6111.0 28628666

47 Pacific Washington 16424.0 5880.0 7523869

##### Exercise

# Subsetting rows by categorical variables

Subsetting data based on a categorical variable often involves using the "or" operator (|) to select rows from multiple categories. This can get tedious when you want all states in one of three different regions, for example. Instead, use the .isin() method, which will allow you to tackle this problem by writing one condition instead of three separate ones.

colors = ["brown", "black", "tan"]

condition = dogs["color"].isin(colors)

dogs[condition]

homelessness is available and pandas is loaded as pd.

##### Instructions 1/2

**50 XP**

* Filter homelessness for cases where the USA census region is "South Atlantic" or it is "Mid-Atlantic", assigning to south\_mid\_atlantic. View the printed result.
* Filter homelessness for cases where the USA census state is in the list of Mojave states, canu, assigning to mojave\_homelessness. View the printed result.
* # Subset for rows in South Atlantic or Mid-Atlantic regions
* south\_mid\_atlantic = homelessness[(homelessness["region"] .isin(["South Atlantic", "Mid-Atlantic"]))]
* # See the result
* print(south\_mid\_atlantic)

# Filter for rows where individuals is greater than 10000

ind\_gt\_10k = homelessness[homelessness["individuals"] >10000]

# See the result

print(ind\_gt\_10k)

# Subset for rows in South Atlantic or Mid-Atlantic regions

south\_mid\_atlantic = homelessness[(homelessness["region"] .isin(["South Atlantic", "Mid-Atlantic"]))]

# See the result

print(south\_mid\_atlantic)

region state individuals family\_members state\_pop

7 South Atlantic Delaware 708.0 374.0 965479

8 South Atlantic District of Columbia 3770.0 3134.0 701547

9 South Atlantic Florida 21443.0 9587.0 21244317

10 South Atlantic Georgia 6943.0 2556.0 10511131

20 South Atlantic Maryland 4914.0 2230.0 6035802

30 Mid-Atlantic New Jersey 6048.0 3350.0 8886025

32 Mid-Atlantic New York 39827.0 52070.0 19530351

33 South Atlantic North Carolina 6451.0 2817.0 10381615

38 Mid-Atlantic Pennsylvania 8163.0 5349.0 12800922

40 South Atlantic South Carolina 3082.0 851.0 5084156

46 South Atlantic Virginia 3928.0 2047.0 8501286

48 South Atlantic West Virginia 1021.0 222.0 1804291

## 1. New columns

In the last lesson, you saw how to subset and sort a DataFrame to extract interesting bits. However, often when you first receive a DataFrame, the contents aren't exactly what you want. You may have to add new columns derived from existing columns.

## 2. Adding a new column

Creating and adding new columns can go by many names, including mutating a DataFrame, transforming a DataFrame, and feature engineering. Let's say we want to add a new column to our DataFrame that has each dog's height in meters instead of centimeters. On the left-hand side of the equals, we use square brackets with the name of the new column we want to create. On the right-hand side, we have the calculation. Notice that both the existing column and the new column we just created are in the DataFrame.

## 3. Doggy mass index

Let's see what the results are if we calculate the body mass index, or BMI, of these dogs. BMI is usually calculated by taking a person's weight in kilograms and dividing it by their height in meters, squared. Instead of doing this with people, we'll try it out with dogs. Again, the new column is on the left-hand side of the equals, but this time, our calculation involves two columns.

## 4. Multiple manipulations

The real power of pandas comes in when you combine all the skills you've learned so far. Let's figure out the names of skinny, tall dogs. First, to define the skinny dogs, we take the subset of the dogs who have a BMI of under 100. Next, we sort the result in descending order of height to get the tallest skinny dogs at the top. Finally, we keep only the columns we're interested in. Here, you can see that Max is the tallest dog with a BMI of under 100.

## 5. Let's practice!

Time to practice your pandas powers!

**Daily XP350**

##### Exercise

##### Exercise

# Adding new columns

You aren't stuck with just the data you are given. Instead, you can add new columns to a DataFrame. This has many names, such as transforming, mutating, and feature engineering.

You can create new columns from scratch, but it is also common to derive them from other columns, for example, by adding columns together or by changing their units.

homelessness is available and pandas is loaded as pd.

##### Instructions

**100 XP**

* Add a new column to homelessness, named total, containing the sum of the individuals and family\_members columns.
* Add another column to homelessness, named p\_individuals, containing the proportion of homeless people in each state who are individuals.
* # Add total col as sum of individuals and family\_members
* homelessness['total'] = homelessness['individuals'] + homelessness['family\_members']
* # Add p\_individuals col as proportion of total that are individual
* homelessness['p\_individuals'] = homelessness['individuals'] / homelessness['total']
* # See the result
* print(homelessness)

# Subset for rows in South Atlantic or Mid-Atlantic regions

south\_mid\_atlantic = homelessness[(homelessness["region"] .isin(["South Atlantic", "Mid-Atlantic"]))]

# See the result

print(south\_mid\_atlantic)

# Add total col as sum of individuals and family\_members

homelessness['total'] = homelessness['individuals'] + homelessness['family\_members']

# Add p\_individuals col as proportion of total that are individual

homelessness['p\_individuals'] = homelessness['individuals'] / homelessness['total']

# See the result

print(homelessness)

region state individuals family\_members state\_pop total p\_individuals

0 East South Central Alabama 2570.0 864.0 4887681 3434.0 0.748

1 Pacific Alaska 1434.0 582.0 735139 2016.0 0.711

2 Mountain Arizona 7259.0 2606.0 7158024 9865.0 0.736

3 West South Central Arkansas 2280.0 432.0 3009733 2712.0 0.841

4 Pacific California 109008.0 20964.0 39461588 129972.0 0.839

5 Mountain Colorado 7607.0 3250.0 5691287 10857.0 0.701

6 New England Connecticut 2280.0 1696.0 3571520 3976.0 0.573

7 South Atlantic Delaware 708.0 374.0 965479 1082.0 0.654

8 South Atlantic District of Columbia 3770.0 3134.0 701547 6904.0 0.546

9 South Atlantic Florida 21443.0 9587.0 21244317 31030.0 0.691

10 South Atlantic Georgia 6943.0 2556.0 10511131 9499.0 0.731

11 Pacific Hawaii 4131.0 2399.0 1420593 6530.0 0.633

12 Mountain Idaho 1297.0 715.0 1750536 2012.0 0.645

13 East North Central Illinois 6752.0 3891.0 12723071 10643.0 0.634

14 East North Central Indiana 3776.0 1482.0 6695497 5258.0 0.718

15 West North Central Iowa 1711.0 1038.0 3148618 2749.0 0.622

16 West North Central Kansas 1443.0 773.0 2911359 2216.0 0.651

17 East South Central Kentucky 2735.0 953.0 4461153 3688.0 0.742

18 West South Central Louisiana 2540.0 519.0 4659690 3059.0 0.830

19 New England Maine 1450.0 1066.0 1339057 2516.0 0.576

20 South Atlantic Maryland 4914.0 2230.0 6035802 7144.0 0.688

21 New England Massachusetts 6811.0 13257.0 6882635 20068.0 0.339

22 East North Central Michigan 5209.0 3142.0 9984072 8351.0 0.624

23 West North Central Minnesota 3993.0 3250.0 5606249 7243.0 0.551

24 East South Central Mississippi 1024.0 328.0 2981020 1352.0 0.757

25 West North Central Missouri 3776.0 2107.0 6121623 5883.0 0.642

26 Mountain Montana 983.0 422.0 1060665 1405.0 0.700

27 West North Central Nebraska 1745.0 676.0 1925614 2421.0 0.721

28 Mountain Nevada 7058.0 486.0 3027341 7544.0 0.936

29 New England New Hampshire 835.0 615.0 1353465 1450.0 0.576

30 Mid-Atlantic New Jersey 6048.0 3350.0 8886025 9398.0 0.644

31 Mountain New Mexico 1949.0 602.0 2092741 2551.0 0.764

32 Mid-Atlantic New York 39827.0 52070.0 19530351 91897.0 0.433

33 South Atlantic North Carolina 6451.0 2817.0 10381615 9268.0 0.696

34 West North Central North Dakota 467.0 75.0 758080 542.0 0.862

35 East North Central Ohio 6929.0 3320.0 11676341 10249.0 0.676

36 West South Central Oklahoma 2823.0 1048.0 3940235 3871.0 0.729

37 Pacific Oregon 11139.0 3337.0 4181886 14476.0 0.769

38 Mid-Atlantic Pennsylvania 8163.0 5349.0 12800922 13512.0 0.604

39 New England Rhode Island 747.0 354.0 1058287 1101.0 0.678

40 South Atlantic South Carolina 3082.0 851.0 5084156 3933.0 0.784

41 West North Central South Dakota 836.0 323.0 878698 1159.0 0.721

42 East South Central Tennessee 6139.0 1744.0 6771631 7883.0 0.779

43 West South Central Texas 19199.0 6111.0 28628666 25310.0 0.759

44 Mountain Utah 1904.0 972.0 3153550 2876.0 0.662

45 New England Vermont 780.0 511.0 624358 1291.0 0.604

46 South Atlantic Virginia 3928.0 2047.0 8501286 5975.0 0.657

47 Pacific Washington 16424.0 5880.0 7523869 22304.0 0.736

48 South Atlantic West Virginia 1021.0 222.0 1804291 1243.0 0.821

49 East North Central Wisconsin 2740.0 2167.0 5807406 4907.0 0.558

50 Mountain Wyoming 434.0 205.0 577601 639.0 0.679

##### Exercise

# Combo-attack!

You've seen the four most common types of data manipulation: sorting rows, subsetting columns, subsetting rows, and adding new columns. In a real-life data analysis, you can mix and match these four manipulations to answer a multitude of questions.

In this exercise, you'll answer the question, "Which state has the highest number of homeless individuals per 10,000 people in the state?" Combine your new pandas skills to find out.

##### Instructions

**100 XP**

* Add a column to homelessness, indiv\_per\_10k, containing the number of homeless individuals per ten thousand people in each state.
* Subset rows where indiv\_per\_10k is higher than 20, assigning to high\_homelessness.
* Sort high\_homelessness by descending indiv\_per\_10k, assigning to high\_homelessness\_srt.
* Select only the state and indiv\_per\_10k columns of high\_homelessness\_srt and save as result. Look at the *result*.
* # Create indiv\_per\_10k col as homeless individuals per 10k state pop
* homelessness["indiv\_per\_10k"] = 10000 \* homelessness["individuals"] / homelessness["state\_pop"]
* #print(homelessness)
* # Subset rows for indiv\_per\_10k greater than 20
* high\_homelessness = homelessness[homelessness["indiv\_per\_10k"] >20]
* #print(high\_homelessness)
* # Sort high\_homelessness by descending indiv\_per\_10k
* high\_homelessness\_srt = high\_homelessness.sort\_values("indiv\_per\_10k" ,ascending=False)
* #print(high\_homelessness\_srt)
* # From high\_homelessness\_srt, select the state and indiv\_per\_10k cols
* result = high\_homelessness\_srt[["state", "indiv\_per\_10k"]]
* # See the result
* print(result)

# Add total col as sum of individuals and family\_members

homelessness['total'] = homelessness['individuals'] + homelessness['family\_members']

# Add p\_individuals col as proportion of total that are individual

homelessness['p\_individuals'] = homelessness['individuals'] / homelessness['total']

# See the result

print(homelessness)

# Create indiv\_per\_10k col as homeless individuals per 10k state pop

homelessness["indiv\_per\_10k"] = 10000 \* homelessness["individuals"] / homelessness["state\_pop"]

#print(homelessness)

# Subset rows for indiv\_per\_10k greater than 20

high\_homelessness = homelessness[homelessness["indiv\_per\_10k"] >20]

#print(high\_homelessness)

# Sort high\_homelessness by descending indiv\_per\_10k

high\_homelessness\_srt = high\_homelessness.sort\_values("indiv\_per\_10k" ,ascending=False)

#print(high\_homelessness\_srt)

# From high\_homelessness\_srt, select the state and indiv\_per\_10k cols

result = high\_homelessness\_srt[["state", "indiv\_per\_10k"]]

# See the result

print(result)

state indiv\_per\_10k

8 District of Columbia 53.738

11 Hawaii 29.079

4 California 27.624

37 Oregon 26.636

28 Nevada 23.314

47 Washington 21.829

32 New York 20.392

**Daily XP350**

# Summary statistics

**50 XP**

## 1. Summary statistics

Hi, I'm Maggie, and I'll be the other instructor for this course. In the first chapter, you learned about DataFrames, how to sort and subset them, and how to add new columns to them. In this chapter, we'll talk about aggregating data, starting with summary statistics. Summary statistics, as follows from their name, are numbers that summarize and tell you about your dataset.

## 2. Summarizing numerical data

One of the most common summary statistics for numeric data is the mean, which is one way of telling you where the "center" of your data is. You can calculate the mean of a column by selecting the column with square brackets and calling dot-mean. There are lots of other summary statistics that you can compute on columns, like median and mode, minimum and maximum, and variance and standard deviation. You can also take sums and calculate quantiles.

## 3. Summarizing dates

You can also get summary statistics for date columns. For example, we can find the oldest dog's date of birth by taking the minimum of the date of birth column. Similarly, we can take the maximum to see that the youngest dog was born in 2018.

## 4. The .agg() method

The aggregate, or agg, method allows you to compute custom summary statistics. Here, we create a function called pct30 that computes the thirtieth percentile of a DataFrame column. Don't worry if this code doesn't make sense to you -- just know that the function takes in a column and spits out the column's thirtieth percentile. Now we can subset the weight column and call dot-agg, passing in the name of our function, pct30. It gives us the thirtieth percentile of the dogs' weights.

## 5. Summaries on multiple columns

agg can also be used on more than one column. By selecting the weight and height columns before calling agg, we get the thirtieth percentile for both columns.

## 6. Multiple summaries

We can also use agg to get multiple summary statistics at once. Here's another function that computes the fortieth percentile called pct40. We can pass a list of functions into agg, in this case, pct30 and pct40, which will return the thirtieth and fortieth percentiles of the dogs' weights.

## 7. Cumulative sum

pandas also has methods for computing cumulative statistics, for example, the cumulative sum. Calling cumsum on a column returns not just one number, but a number for each row of the DataFrame. The first number returned, or the number in the zeroth index, is the first dog's weight. The next number is the sum of the first and second dogs' weights. The third number is the sum of the first, second, and third dogs' weights, and so on. The last number is the sum of all the dogs' weights.

## 8. Cumulative statistics

pandas also has methods for other cumulative statistics, such as the cumulative maximum, cumulative minimum, and the cumulative product. These all return an entire column of a DataFrame, rather than a single number.

## 9. Walmart

In this chapter, you'll be working with data on Walmart stores, which is a chain of department stores in the US. The dataset contains weekly sales in US dollars in various stores. Each store has an ID number and a specific store type. The sales are also separated by department ID. Along with weekly sales, there is information about whether it was a holiday week or not, the average temperature during the week in that location, the average fuel price in dollars per liter that week, and the national unemployment rate that week.

## 10. Let's practice!

Time to practice your summary statistics skills!

**Daily XP350**

##### Exercise

##### Exercise

# Mean and median

Summary statistics are exactly what they sound like - they summarize many numbers in one statistic. For example, mean, median, minimum, maximum, and standard deviation are summary statistics. Calculating summary statistics allows you to get a better sense of your data, even if there's a lot of it.

sales is available and pandas is loaded as pd.

##### Instructions

**100 XP**

* Explore your new DataFrame first by printing the first few rows of the sales DataFrame.
* Print information about the columns in sales.
* Print the mean of the weekly\_sales column.
* Print the median of the weekly\_sales column.

# Print the head of the sales DataFrame

print(sales.head())

# Print the info about the sales DataFrame

print(sales.info())

# Print the mean of weekly\_sales

print(sales['weekly\_sales'].mean())

# Print the median of weekly\_sales

print(sales['weekly\_sales'].median())

# Create indiv\_per\_10k col as homeless individuals per 10k state pop

homelessness["indiv\_per\_10k"] = 10000 \* homelessness["individuals"] / homelessness["state\_pop"]

#print(homelessness)

# Subset rows for indiv\_per\_10k greater than 20

high\_homelessness = homelessness[homelessness["indiv\_per\_10k"] >20]

#print(high\_homelessness)

# Sort high\_homelessness by descending indiv\_per\_10k

high\_homelessness\_srt = high\_homelessness.sort\_values("indiv\_per\_10k" ,ascending=False)

#print(high\_homelessness\_srt)

# From high\_homelessness\_srt, select the state and indiv\_per\_10k cols

result = high\_homelessness\_srt[["state", "indiv\_per\_10k"]]

# See the result

print(result)

# Print the head of the sales DataFrame

print(sales.head())

# Print the info about the sales DataFrame

print(sales.info())

# Print the mean of weekly\_sales

print(sales['weekly\_sales'].mean())

# Print the median of weekly\_sales

print(sales['weekly\_sales'].median())

store type department date weekly\_sales is\_holiday temperature\_c fuel\_price\_usd\_per\_l unemployment

0 1 A 1 2010-02-05 24924.50 False 5.728 0.679 8.106

1 1 A 1 2010-03-05 21827.90 False 8.056 0.693 8.106

2 1 A 1 2010-04-02 57258.43 False 16.817 0.718 7.808

3 1 A 1 2010-05-07 17413.94 False 22.528 0.749 7.808

4 1 A 1 2010-06-04 17558.09 False 27.050 0.715 7.808

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 10774 entries, 0 to 10773

Data columns (total 9 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 store 10774 non-null int64

1 type 10774 non-null object

2 department 10774 non-null int32

3 date 10774 non-null datetime64[ns]

4 weekly\_sales 10774 non-null float64

5 is\_holiday 10774 non-null bool

6 temperature\_c 10774 non-null float64

7 fuel\_price\_usd\_per\_l 10774 non-null float64

8 unemployment 10774 non-null float64

dtypes: bool(1), datetime64[ns](1), float64(4), int32(1), int64(1), object(1)

memory usage: 641.9+ KB

None

23843.95014850566

12049.064999999999

**Daily XP350**

##### Exercise

##### Exercise

# Summarizing dates

Summary statistics can also be calculated on date columns that have values with the data type datetime64. Some summary statistics — like mean — don't make a ton of sense on dates, but others are super helpful, for example, minimum and maximum, which allow you to see what time range your data covers.

sales is available and pandas is loaded as pd.

##### Instructions

**100 XP**

* Print the maximum of the date column.
* Print the minimum of the date column.
* # Print the maximum of the date column
* print(sales['date'].max())
* # Print the minimum of the date column
* print(sales['date'].min())

# Print the head of the sales DataFrame

print(sales.head())

# Print the info about the sales DataFrame

print(sales.info())

# Print the mean of weekly\_sales

print(sales['weekly\_sales'].mean())

# Print the median of weekly\_sales

print(sales['weekly\_sales'].median())

# Print the maximum of the date column

print(sales['date'].max())

# Print the minimum of the date column

print(sales['date'].min())

2012-10-26 00:00:00

2010-02-05 00:00:00

**Daily XP350**

##### Exercise

##### Exercise

# Efficient summaries

While pandas and NumPy have tons of functions, sometimes, you may need a different function to summarize your data.

The .agg() method allows you to apply your own custom functions to a DataFrame, as well as apply functions to more than one column of a DataFrame at once, making your aggregations super-efficient. For example,

df['column'].agg(function)

In the custom function for this exercise, "IQR" is short for inter-quartile range, which is the 75th percentile minus the 25th percentile. It's an alternative to standard deviation that is helpful if your data contains outliers.

sales is available and pandas is loaded as pd.

##### Instructions 1/3

**35 XP**

* + Use the custom iqr function defined for you along with .agg() to print the IQR of the temperature\_c column of sales.
  + Update the column selection to use the custom iqr function with .agg() to print the IQR of temperature\_c, fuel\_price\_usd\_per\_l, and unemployment, in that order.
  + Update the aggregation functions called by .agg(): include iqr and np.median in that order.
* # A custom IQR function
* def iqr(column):
* return column.quantile(0.75) - column.quantile(0.25)
* # Print IQR of the temperature\_c column
* print(sales['temperature\_c'].agg(iqr))

# Print the maximum of the date column

print(sales['date'].max())

# Print the minimum of the date column

print(sales['date'].min())

# A custom IQR function

def iqr(column):

return column.quantile(0.75) - column.quantile(0.25)

# Print IQR of the temperature\_c column

print(sales['temperature\_c'].agg(iqr))

16.583333333333336

**Daily XP350**

##### Exercise

##### Exercise

# Cumulative statistics

Cumulative statistics can also be helpful in tracking summary statistics over time. In this exercise, you'll calculate the cumulative sum and cumulative max of a department's weekly sales, which will allow you to identify what the total sales were so far as well as what the highest weekly sales were so far.

A DataFrame called sales\_1\_1 has been created for you, which contains the sales data for department 1 of store 1. pandas is loaded as pd.

##### Instructions

**100 XP**

* Sort the rows of sales\_1\_1 by the date column in ascending order.
* Get the cumulative sum of weekly\_sales and add it as a new column of sales\_1\_1 called cum\_weekly\_sales.
* Get the cumulative maximum of weekly\_sales, and add it as a column called cum\_max\_sales.
* Print the date, weekly\_sales, cum\_weekly\_sales, and cum\_max\_sales columns.
* # Sort sales\_1\_1 by date
* sales\_1\_1 = sales\_1\_1.sort\_values("date", ascending=True)
* #print(sales\_1\_1)
* # Get the cumulative sum of weekly\_sales, add as cum\_weekly\_sales col
* sales\_1\_1["cum\_weekly\_sales"] = sales\_1\_1["weekly\_sales"].cumsum()
* #print(sales\_1\_1)
* # Get the cumulative max of weekly\_sales, add as cum\_max\_sales col
* sales\_1\_1["cum\_max\_sales"] = sales\_1\_1["weekly\_sales"].cummax()
* #print(sales\_1\_1)
* # See the columns you calculated
* print(sales\_1\_1[["date", "weekly\_sales", "cum\_weekly\_sales", "cum\_max\_sales"]])

# A custom IQR function

def iqr(column):

return column.quantile(0.75) - column.quantile(0.25)

# Print IQR of the temperature\_c column

print(sales['temperature\_c'].agg(iqr))

# Sort sales\_1\_1 by date

sales\_1\_1 = sales\_1\_1.sort\_values("date", ascending=True)

#print(sales\_1\_1)

# Get the cumulative sum of weekly\_sales, add as cum\_weekly\_sales col

sales\_1\_1["cum\_weekly\_sales"] = sales\_1\_1["weekly\_sales"].cumsum()

#print(sales\_1\_1)

# Get the cumulative max of weekly\_sales, add as cum\_max\_sales col

sales\_1\_1["cum\_max\_sales"] = sales\_1\_1["weekly\_sales"].cummax()

#print(sales\_1\_1)

# See the columns you calculated

print(sales\_1\_1[["date", "weekly\_sales", "cum\_weekly\_sales", "cum\_max\_sales"]])

date weekly\_sales cum\_weekly\_sales cum\_max\_sales

0 2010-02-05 24924.50 24924.50 24924.50

1 2010-03-05 21827.90 46752.40 24924.50

2 2010-04-02 57258.43 104010.83 57258.43

3 2010-05-07 17413.94 121424.77 57258.43

4 2010-06-04 17558.09 138982.86 57258.43

5 2010-07-02 16333.14 155316.00 57258.43

6 2010-08-06 17508.41 172824.41 57258.43

7 2010-09-03 16241.78 189066.19 57258.43

8 2010-10-01 20094.19 209160.38 57258.43

9 2010-11-05 34238.88 243399.26 57258.43

10 2010-12-03 22517.56 265916.82 57258.43

11 2011-01-07 15984.24 281901.06 57258.43

**Daily XP350**

# Counting

**50 XP**

## 1. Counting

So far, in this chapter, you've learned how to summarize numeric variables. In this video, you'll learn how to summarize categorical data using counting.

## 2. Avoiding double counting

Counting dogs is no easy task when they're running around the park. It's hard to keep track of who you have and haven't counted!

## 3. Vet visits

Here's a DataFrame that contains vet visits. The vet's office wants to know how many dogs of each breed have visited their office. However, some dogs have been to the vet more than once, like Max and Stella, so we can't just count the number of each breed in the breed column.

## 4. Dropping duplicate names

Let's try to fix this by removing rows that contain a dog name already listed earlier in the dataset, or in other words; we'll extract a dog with each name from the dataset once. We can do this using the drop\_duplicates method. It takes an argument, subset, which is the column we want to find our duplicates based on - in this case, we want all the unique names. Now we have a list of dogs where each one appears once. We have Max the Chow Chow, but where did Max the Labrador go? Because we have two different dogs with the same name, we'll need to consider more than just name when dropping duplicates.

## 5. Dropping duplicate pairs

Since Max and Max are different breeds, we can drop the rows with pairs of name and breed listed earlier in the dataset. To base our duplicate dropping on multiple columns, we can pass a list of column names to the subset argument, in this case, name and breed. Now both Maxes have been included, and we can start counting.

## 6. Easy as 1, 2, 3

To count the dogs of each breed, we'll subset the breed column and use the value\_counts method. We can also use the sort argument to get the breeds with the biggest counts on top.

## 7. Proportions

The normalize argument can be used to turn the counts into proportions of the total. 25% of the dogs that go to this vet are Labradors.

## 8. Let's practice!

Time to commence counting!

**Daily XP350**

##### Exercise

##### Exercise

# Dropping duplicates

Removing duplicates is an essential skill to get accurate counts because often, you don't want to count the same thing multiple times. In this exercise, you'll create some new DataFrames using unique values from sales.

sales is available and pandas is imported as pd.

##### Instructions

**100 XP**

* Remove rows of sales with duplicate pairs of store and type and save as store\_types and print the head.
* Remove rows of sales with duplicate pairs of store and department and save as store\_depts and print the head.
* Subset the rows that are holiday weeks using the is\_holiday column, and drop the duplicate dates, saving as holiday\_dates.
* Select the date column of holiday\_dates, and print.
* # Drop duplicate store/type combinations
* store\_types = sales.drop\_duplicates(subset=['store','type'])
* print(store\_types.head())
* # Drop duplicate store/department combinations
* store\_depts = sales.drop\_duplicates(subset=['store', 'department'])
* print(store\_depts.head())
* # Subset the rows where is\_holiday is True and drop duplicate dates
* holiday\_dates = sales[sales['is\_holiday']].drop\_duplicates(subset='date')
* # Print date col of holiday\_dates
* print(holiday\_dates['date'])

# Sort sales\_1\_1 by date

sales\_1\_1 = sales\_1\_1.sort\_values("date", ascending=True)

#print(sales\_1\_1)

# Get the cumulative sum of weekly\_sales, add as cum\_weekly\_sales col

sales\_1\_1["cum\_weekly\_sales"] = sales\_1\_1["weekly\_sales"].cumsum()

#print(sales\_1\_1)

# Get the cumulative max of weekly\_sales, add as cum\_max\_sales col

sales\_1\_1["cum\_max\_sales"] = sales\_1\_1["weekly\_sales"].cummax()

#print(sales\_1\_1)

# See the columns you calculated

print(sales\_1\_1[["date", "weekly\_sales", "cum\_weekly\_sales", "cum\_max\_sales"]])

# Drop duplicate store/type combinations

store\_types = sales.drop\_duplicates(subset=['store','type'])

print(store\_types.head())

# Drop duplicate store/department combinations

store\_depts = sales.drop\_duplicates(subset=['store', 'department'])

print(store\_depts.head())

# Subset the rows where is\_holiday is True and drop duplicate dates

holiday\_dates = sales[sales['is\_holiday']].drop\_duplicates(subset='date')

# Print date col of holiday\_dates

print(holiday\_dates['date'])

store type department date weekly\_sales is\_holiday temperature\_c fuel\_price\_usd\_per\_l unemployment

0 1 A 1 2010-02-05 24924.50 False 5.728 0.679 8.106

901 2 A 1 2010-02-05 35034.06 False 4.550 0.679 8.324

1798 4 A 1 2010-02-05 38724.42 False 6.533 0.686 8.623

2699 6 A 1 2010-02-05 25619.00 False 4.683 0.679 7.259

3593 10 B 1 2010-02-05 40212.84 False 12.411 0.782 9.765

store type department date weekly\_sales is\_holiday temperature\_c fuel\_price\_usd\_per\_l unemployment

0 1 A 1 2010-02-05 24924.50 False 5.728 0.679 8.106

12 1 A 2 2010-02-05 50605.27 False 5.728 0.679 8.106

24 1 A 3 2010-02-05 13740.12 False 5.728 0.679 8.106

36 1 A 4 2010-02-05 39954.04 False 5.728 0.679 8.106

48 1 A 5 2010-02-05 32229.38 False 5.728 0.679 8.106

498 2010-09-10

691 2011-11-25

2315 2010-02-12

6735 2012-09-07

6810 2010-12-31

6815 2012-02-10

6820 2011-09-09

Name: date, dtype: datetime64[ns]

**Daily XP350**

##### Exercise

##### Exercise

# Counting categorical variables

Counting is a great way to get an overview of your data and to spot curiosities that you might not notice otherwise. In this exercise, you'll count the number of each type of store and the number of each department number using the DataFrames you created in the previous exercise:

# Drop duplicate store/type combinations

store\_types = sales.drop\_duplicates(subset=["store", "type"])

# Drop duplicate store/department combinations

store\_depts = sales.drop\_duplicates(subset=["store", "department"])

The store\_types and store\_depts DataFrames you created in the last exercise are available, and pandas is imported as pd.

##### Instructions

**100 XP**

* Count the number of stores of each store type in store\_types.
* Count the proportion of stores of each store type in store\_types.
* Count the number of different departments in store\_depts, sorting the counts in descending order.
* Count the proportion of different departments in store\_depts, sorting the proportions in descending order.
* # Count the number of stores of each type
* store\_counts = store\_types['type'].value\_counts()
* print(store\_counts)
* # Get the proportion of stores of each type
* store\_props = store\_types['type'].value\_counts(normalize=True)
* print(store\_props)
* # Count the number of each department number and sort
* dept\_counts\_sorted = store\_depts['department'].value\_counts(sort=True)
* print(dept\_counts\_sorted)
* # Get the proportion of departments of each number and sort
* dept\_props\_sorted = store\_depts['department'].value\_counts(sort=True, normalize=True)
* print(dept\_props\_sorted)

# Drop duplicate store/type combinations

store\_types = sales.drop\_duplicates(subset=['store','type'])

print(store\_types.head())

# Drop duplicate store/department combinations

store\_depts = sales.drop\_duplicates(subset=['store', 'department'])

print(store\_depts.head())

# Subset the rows where is\_holiday is True and drop duplicate dates

holiday\_dates = sales[sales['is\_holiday']].drop\_duplicates(subset='date')

# Print date col of holiday\_dates

print(holiday\_dates['date'])

# Count the number of stores of each type

store\_counts = store\_types['type'].value\_counts()

print(store\_counts)

# Get the proportion of stores of each type

store\_props = store\_types['type'].value\_counts(normalize=True)

print(store\_props)

# Count the number of each department number and sort

dept\_counts\_sorted = store\_depts['department'].value\_counts(sort=True)

print(dept\_counts\_sorted)

# Get the proportion of departments of each number and sort

dept\_props\_sorted = store\_depts['department'].value\_counts(sort=True, normalize=True)

print(dept\_props\_sorted)

A 11

B 1

Name: type, dtype: int64

A 0.917

B 0.083

Name: type, dtype: float64

1 12

55 12

72 12

71 12

67 12

..

37 10

48 8

50 6

39 4

43 2

Name: department, Length: 80, dtype: int64

1 0.013

55 0.013

72 0.013

71 0.013

67 0.013

...

37 0.011

48 0.009

50 0.006

39 0.004

43 0.002

Name: department, Length: 80, dtype: float64

**Daily XP350**

# Grouped summary statistics

**50 XP**

## 1. Grouped summary statistics

So far, you've been calculating summary statistics for all rows of a dataset, but summary statistics can be useful to compare different groups.

## 2. Summaries by group

While computing summary statistics of entire columns may be useful, you can gain many insights from summaries of individual groups. For example, does one color of dog weigh more than another on average? Are female dogs taller than males? You can already answer these questions with what you've learned so far! We can subset the dogs into groups based on their color, and take the mean of each. But that's a lot of work, and the duplicated code means you can easily introduce copy and paste bugs.

## 3. Grouped summaries

That's where the groupby method comes in. We can group by the color variable, select the weight column, and take the mean. This will give us the mean weight for each dog color. This was just one line of code compared to the five we had to write before to get the same results.

## 4. Multiple grouped summaries

Just like with ungrouped summary statistics, we can use the agg method to get multiple statistics. Here, we pass a list of functions into agg after grouping by color. This gives us the minimum, maximum, and sum of the different colored dogs' weights.

## 5. Grouping by multiple variables

You can also group by multiple columns and calculate summary statistics. Here, we group by color and breed, select the weight column and take the mean. This gives us the mean weight of each breed of each color.

## 6. Many groups, many summaries

You can also group by multiple columns and aggregate by multiple columns.

## 7. Let's practice!

Now that we've talked about grouping, it's time to practice grouped summary statistics.

**Daily XP350**

##### Exercise

##### Exercise

# What percent of sales occurred at each store type?

While .groupby() is useful, you can calculate grouped summary statistics without it.

Walmart distinguishes three types of stores: "supercenters," "discount stores," and "neighborhood markets," encoded in this dataset as type "A," "B," and "C." In this exercise, you'll calculate the total sales made at each store type, without using .groupby(). You can then use these numbers to see what proportion of Walmart's total sales were made at each type.

sales is available and pandas is imported as pd.

##### Instructions

**100 XP**

* Calculate the total weekly\_sales over the whole dataset.
* Subset for type "A" stores, and calculate their total weekly sales.
* Do the same for type "B" and type "C" stores.
* Combine the A/B/C results into a list, and divide by sales\_all to get the proportion of sales by type.
* # Calc total weekly sales
* sales\_all = sales["weekly\_sales"].sum()
* # Subset for type A stores, calc total weekly sales
* sales\_A = sales[sales["type"] == "A"]["weekly\_sales"].sum()
* # Subset for type B stores, calc total weekly sales
* sales\_B = sales[sales["type"] == "B"]["weekly\_sales"].sum()
* # Subset for type C stores, calc total weekly sales
* sales\_C = sales[sales["type"] == "C"]["weekly\_sales"].sum()
* # Get proportion for each type
* sales\_propn\_by\_type = [sales\_A, sales\_B, sales\_C] / sales\_all
* print(sales\_propn\_by\_type)

# Count the number of stores of each type

store\_counts = store\_types['type'].value\_counts()

print(store\_counts)

# Get the proportion of stores of each type

store\_props = store\_types['type'].value\_counts(normalize=True)

print(store\_props)

# Count the number of each department number and sort

dept\_counts\_sorted = store\_depts['department'].value\_counts(sort=True)

print(dept\_counts\_sorted)

# Get the proportion of departments of each number and sort

dept\_props\_sorted = store\_depts['department'].value\_counts(sort=True, normalize=True)

print(dept\_props\_sorted)

# Calc total weekly sales

sales\_all = sales["weekly\_sales"].sum()

# Subset for type A stores, calc total weekly sales

sales\_A = sales[sales["type"] == "A"]["weekly\_sales"].sum()

# Subset for type B stores, calc total weekly sales

sales\_B = sales[sales["type"] == "B"]["weekly\_sales"].sum()

# Subset for type C stores, calc total weekly sales

sales\_C = sales[sales["type"] == "C"]["weekly\_sales"].sum()

# Get proportion for each type

sales\_propn\_by\_type = [sales\_A, sales\_B, sales\_C] / sales\_all

print(sales\_propn\_by\_type)

[0.9097747 0.0902253 0. ]

##### Exercise

# Calculations with .groupby()

The .groupby() method makes life much easier. In this exercise, you'll perform the same calculations as last time, except you'll use the .groupby() method. You'll also perform calculations on data grouped by two variables to see if sales differ by store type depending on if it's a holiday week or not.

sales is available and pandas is loaded as pd.

##### Instructions 2/2

**50 XP**

* Group sales by "type" and "is\_holiday", take the sum of weekly\_sales, and store as sales\_by\_type\_is\_holiday.
* # From previous step
* sales\_by\_type = sales.groupby("type")["weekly\_sales"].sum()
* #print(sales\_by\_type)
* # Group by type and is\_holiday; calc total weekly sales
* sales\_by\_type\_is\_holiday = sales.groupby(["type", "is\_holiday"])["weekly\_sales"].sum()
* print(sales\_by\_type\_is\_holiday)

# Calc total weekly sales

sales\_all = sales["weekly\_sales"].sum()

# Subset for type A stores, calc total weekly sales

sales\_A = sales[sales["type"] == "A"]["weekly\_sales"].sum()

# Subset for type B stores, calc total weekly sales

sales\_B = sales[sales["type"] == "B"]["weekly\_sales"].sum()

# Subset for type C stores, calc total weekly sales

sales\_C = sales[sales["type"] == "C"]["weekly\_sales"].sum()

# Get proportion for each type

sales\_propn\_by\_type = [sales\_A, sales\_B, sales\_C] / sales\_all

print(sales\_propn\_by\_type)

# From previous step

sales\_by\_type = sales.groupby("type")["weekly\_sales"].sum()

#print(sales\_by\_type)

# Group by type and is\_holiday; calc total weekly sales

sales\_by\_type\_is\_holiday = sales.groupby(["type", "is\_holiday"])["weekly\_sales"].sum()

print(sales\_by\_type\_is\_holiday)

type is\_holiday

A False 2.337e+08

True 2.360e+04

B False 2.318e+07

True 1.621e+03

Name: weekly\_sales, dtype: float64

**Daily XP350**

##### Exercise

##### Exercise

# Multiple grouped summaries

Earlier in this chapter, you saw that the .agg() method is useful to compute multiple statistics on multiple variables. It also works with grouped data. NumPy, which is imported as np, has many different summary statistics functions, including: np.min, np.max, np.mean, and np.median.

sales is available and pandas is imported as pd.

##### Instructions

**100 XP**

* Import numpy with the alias np.
* Get the min, max, mean, and median of weekly\_sales for each store type using .groupby() and .agg(). Store this as sales\_stats. Make sure to use numpy functions!
* Get the min, max, mean, and median of unemployment and fuel\_price\_usd\_per\_l for each store type. Store this as unemp\_fuel\_stats.
* # Import numpy with the alias np
* import numpy as np
* # For each store type, aggregate weekly\_sales: get min, max, mean, and median
* sales\_stats = sales.groupby("type")["weekly\_sales"].agg([np.min, np.max, np.mean, np.median ])
* # Print sales\_stats
* print(sales\_stats)
* # For each store type, aggregate unemployment and fuel\_price\_usd\_per\_l: get min, max, mean, and median
* unemp\_fuel\_stats = sales.groupby("type")["unemployment", "fuel\_price\_usd\_per\_l"].agg([np.min, np.max, np.mean, np.median ])
* # Print unemp\_fuel\_stats
* print(unemp\_fuel\_stats)

# From previous step

sales\_by\_type = sales.groupby("type")["weekly\_sales"].sum()

#print(sales\_by\_type)

# Group by type and is\_holiday; calc total weekly sales

sales\_by\_type\_is\_holiday = sales.groupby(["type", "is\_holiday"])["weekly\_sales"].sum()

print(sales\_by\_type\_is\_holiday)

# Import numpy with the alias np

import numpy as np

# For each store type, aggregate weekly\_sales: get min, max, mean, and median

sales\_stats = sales.groupby("type")["weekly\_sales"].agg([np.min, np.max, np.mean, np.median ])

# Print sales\_stats

print(sales\_stats)

# For each store type, aggregate unemployment and fuel\_price\_usd\_per\_l: get min, max, mean, and median

unemp\_fuel\_stats = sales.groupby("type")["unemployment", "fuel\_price\_usd\_per\_l"].agg([np.min, np.max, np.mean, np.median ])

# Print unemp\_fuel\_stats

print(unemp\_fuel\_stats)

amin amax mean median

type

A -1098.0 293966.05 23674.667 11943.92

B -798.0 232558.51 25696.678 13336.08

unemployment fuel\_price\_usd\_per\_l

amin amax mean median amin amax mean median

type

A 3.879 8.992 7.973 8.067 0.664 1.107 0.745 0.735

B 7.170 9.765 9.279 9.199 0.760 1.108 0.806 0.803

**Daily XP350**

# Pivot tables

**50 XP**

## 1. Pivot tables

Pivot tables are another way of calculating grouped summary statistics. If you've ever used a spreadsheet, chances are you've used a pivot table. Let's see how to create pivot tables in pandas.

## 2. Group by to pivot table

In the last lesson, we grouped the dogs by color and calculated their mean weights. We can do the same thing using the pivot\_table method. The "values" argument is the column that you want to summarize, and the index column is the column that you want to group by. By default, pivot\_table takes the mean value for each group.

## 3. Different statistics

If we want a different summary statistic, we can use the aggfunc argument and pass it a function. Here, we take the median for each dog color using NumPy's median function.

## 4. Multiple statistics

To get multiple summary statistics at a time, we can pass a list of functions to the aggfunc argument. Here, we get the mean and median for each dog color.

## 5. Pivot on two variables

You also previously computed the mean weight grouped by two variables: color and breed. We can also do this using the pivot\_table method. To group by two variables, we can pass a second variable name into the columns argument. While the result looks a little different than what we had before, it contains the same numbers. There are NaNs, or missing values, because there are no black Chihuahuas or gray Labradors in our dataset, for example.

## 6. Filling missing values in pivot tables

Instead of having lots of missing values in our pivot table, we can have them filled in using the fill\_value argument. Here, all of the NaNs get filled in with zeros.

## 7. Summing with pivot tables

If we set the margins argument to True, the last row and last column of the pivot table contain the mean of all the values in the column or row, not including the missing values that were filled in with Os. For example, in the last row of the Labrador column, we can see that the mean weight of the Labradors is 26 kilograms. In the last column of the Brown row, the mean weight of the Brown dogs is 24 kilograms. The value in the bottom right, in the last row and last column, is the mean weight of all the dogs in the dataset. Using margins equals True allows us to see a summary statistic for multiple levels of the dataset: the entire dataset, grouped by one variable, by another variable, and by two variables.

## 8. Let's practice!

Time to practice aggregating data using pivot tables!

**Daily XP350**

##### Exercise

##### Exercise

# Pivoting on one variable

Pivot tables are the standard way of aggregating data in spreadsheets. In pandas, pivot tables are essentially just another way of performing grouped calculations. That is, the .pivot\_table() method is just an alternative to .groupby().

In this exercise, you'll perform calculations using .pivot\_table() to replicate the calculations you performed in the last lesson using .groupby().

sales is available and pandas is imported as pd.

##### Instructions 1/3

**35 XP**

* + Get the mean weekly\_sales by type using .pivot\_table() and store as mean\_sales\_by\_type.
  + Get the mean and median (using NumPy functions) of weekly\_sales by type using .pivot\_table() and store as mean\_med\_sales\_by\_type.
  + Get the mean of weekly\_sales by type and is\_holiday using .pivot\_table() and store as mean\_sales\_by\_type\_holiday.
* # Pivot for mean weekly\_sales for each store type
* mean\_sales\_by\_type = sales.pivot\_table(values= "weekly\_sales", index= "type")
* # Print mean\_sales\_by\_type
* print(mean\_sales\_by\_type)

# Import numpy with the alias np

import numpy as np

# For each store type, aggregate weekly\_sales: get min, max, mean, and median

sales\_stats = sales.groupby("type")["weekly\_sales"].agg([np.min, np.max, np.mean, np.median ])

# Print sales\_stats

print(sales\_stats)

# For each store type, aggregate unemployment and fuel\_price\_usd\_per\_l: get min, max, mean, and median

unemp\_fuel\_stats = sales.groupby("type")["unemployment", "fuel\_price\_usd\_per\_l"].agg([np.min, np.max, np.mean, np.median ])

# Print unemp\_fuel\_stats

print(unemp\_fuel\_stats)

# Pivot for mean weekly\_sales for each store type

mean\_sales\_by\_type = sales.pivot\_table(values= "weekly\_sales", index= "type")

# Print mean\_sales\_by\_type

print(mean\_sales\_by\_type)

weekly\_sales

type

A 23674.667

B 25696.678

**Daily XP350**

##### Exercise

##### Exercise

# Fill in missing values and sum values with pivot tables

The .pivot\_table() method has several useful arguments, including fill\_value and margins.

* fill\_value replaces missing values with a real value (known as imputation). What to replace missing values with is a topic big enough to have its own course ([Dealing with Missing Data in Python](https://www.datacamp.com/courses/dealing-with-missing-data-in-python)), but the simplest thing to do is to substitute a dummy value.
* margins is a shortcut for when you pivoted by two variables, but also wanted to pivot by each of those variables separately: it gives the row and column totals of the pivot table contents.

In this exercise, you'll practice using these arguments to up your pivot table skills, which will help you crunch numbers more efficiently!

sales is available and pandas is imported as pd.

##### Instructions 1/2

**50 XP**

* + Print the mean weekly\_sales by department and type, filling in any missing values with 0.
  + Print the mean weekly\_sales by department and type, filling in any missing values with 0 and summing all rows and columns.
* # Print mean weekly\_sales by department and type; fill missing values with 0
* print(sales.pivot\_table(values= "weekly\_sales", index= "department", columns= "type", fill\_value= 0))
* IPython Shell

# Pivot for mean weekly\_sales for each store type

mean\_sales\_by\_type = sales.pivot\_table(values= "weekly\_sales", index= "type")

# Print mean\_sales\_by\_type

print(mean\_sales\_by\_type)

# Print mean weekly\_sales by department and type; fill missing values with 0

print(sales.pivot\_table(values= "weekly\_sales", index= "department", columns= "type", fill\_value= 0))

type A B

department

1 30961.725 44050.627

2 67600.159 112958.527

3 17160.003 30580.655

4 44285.399 51219.654

5 34821.011 63236.875

... ... ...

95 123933.787 77082.102

96 21367.043 9528.538

97 28471.267 5828.873

98 12875.423 217.428

99 379.124 0.000

[80 rows x 2 columns]

**Daily XP350**

# Explicit indexes

**50 XP**

## 1. 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.

## 2. The dog dataset, revisited

Here's the dog dataset again.

## 3. .columns and .index

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

## 4. 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.

## 5. 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.

## 6. 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.

## 7. 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!

## 8. 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.

## 9. Subsetting on duplicated index values

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

## 10. 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.

## 11. 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.

## 12. 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.

## 13. 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.

## 14. Controlling sort\_index

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

## 15. 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.

## 16. Temperature dataset

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

## 17. Let's practice!

Let's get indexing!

**Daily XP400**

##### Exercise

##### Exercise

# Setting and removing indexes

pandas allows you to designate columns as an index. This enables cleaner code when taking subsets (as well as providing more efficient lookup under some circumstances).

In this chapter, you'll be exploring temperatures, a DataFrame of average temperatures in cities around the world. pandas is loaded as pd.

##### Instructions

**100 XP**

* Look at *temperatures*.
* Set the index of temperatures to "city", assigning to temperatures\_ind.
* Look at *temperatures\_ind*. How is it different from *temperatures*?
* Reset the index of temperatures\_ind, keeping its contents.
* Reset the index of temperatures\_ind, dropping its contents.
* # Look at temperatures
* print(\_\_\_\_)
* # Set the index of temperatures to city
* temperatures\_ind = \_\_\_\_
* # Look at temperatures\_ind
* print(\_\_\_\_)
* # Reset the temperatures\_ind index, keeping its contents
* print(\_\_\_\_)
* # Reset the temperatures\_ind index, dropping its contents
* print(\_\_\_\_)

# Print mean weekly\_sales by department and type; fill missing values with 0

print(sales.pivot\_table(values= "weekly\_sales", index= "department", columns= "type", fill\_value= 0))

# Look at temperatures

print(temperatures)

# Set the index of temperatures to city

temperatures\_ind = temperatures.set\_index("city")

# Look at temperatures\_ind

print(temperatures\_ind)

# Reset the temperatures\_ind index, keeping its contents

print(temperatures\_ind.reset\_index())

# Reset the temperatures\_ind index, dropping its contents

print(temperatures\_ind.reset\_index(drop=True))

# Look at temperatures

print(temperatures)

# Set the index of temperatures to city

temperatures\_ind = temperatures.set\_index("city")

# Look at temperatures\_ind

print(temperatures\_ind)

# Reset the temperatures\_ind index, keeping its contents

print(temperatures\_ind.reset\_index())

# Reset the temperatures\_ind index, dropping its contents

print(temperatures\_ind.reset\_index(drop=True))

date city country avg\_temp\_c

0 2000-01-01 Abidjan Côte D'Ivoire 27.293

1 2000-02-01 Abidjan Côte D'Ivoire 27.685

2 2000-03-01 Abidjan Côte D'Ivoire 29.061

3 2000-04-01 Abidjan Côte D'Ivoire 28.162

4 2000-05-01 Abidjan Côte D'Ivoire 27.547

... ... ... ... ...

16495 2013-05-01 Xian China 18.979

16496 2013-06-01 Xian China 23.522

16497 2013-07-01 Xian China 25.251

16498 2013-08-01 Xian China 24.528

16499 2013-09-01 Xian China NaN

[16500 rows x 4 columns]

date country avg\_temp\_c

city

Abidjan 2000-01-01 Côte D'Ivoire 27.293

Abidjan 2000-02-01 Côte D'Ivoire 27.685

Abidjan 2000-03-01 Côte D'Ivoire 29.061

Abidjan 2000-04-01 Côte D'Ivoire 28.162

Abidjan 2000-05-01 Côte D'Ivoire 27.547

... ... ... ...

Xian 2013-05-01 China 18.979

Xian 2013-06-01 China 23.522

Xian 2013-07-01 China 25.251

Xian 2013-08-01 China 24.528

Xian 2013-09-01 China NaN

[16500 rows x 3 columns]

city date country avg\_temp\_c

0 Abidjan 2000-01-01 Côte D'Ivoire 27.293

1 Abidjan 2000-02-01 Côte D'Ivoire 27.685

2 Abidjan 2000-03-01 Côte D'Ivoire 29.061

3 Abidjan 2000-04-01 Côte D'Ivoire 28.162

4 Abidjan 2000-05-01 Côte D'Ivoire 27.547

... ... ... ... ...

16495 Xian 2013-05-01 China 18.979

16496 Xian 2013-06-01 China 23.522

16497 Xian 2013-07-01 China 25.251

16498 Xian 2013-08-01 China 24.528

16499 Xian 2013-09-01 China NaN

[16500 rows x 4 columns]

date country avg\_temp\_c

0 2000-01-01 Côte D'Ivoire 27.293

1 2000-02-01 Côte D'Ivoire 27.685

2 2000-03-01 Côte D'Ivoire 29.061

3 2000-04-01 Côte D'Ivoire 28.162

4 2000-05-01 Côte D'Ivoire 27.547

... ... ... ...

16495 2013-05-01 China 18.979

16496 2013-06-01 China 23.522

16497 2013-07-01 China 25.251

16498 2013-08-01 China 24.528

16499 2013-09-01 China NaN

[16500 rows x 3 columns]

<script.py> output:

date city country avg\_temp\_c

0 2000-01-01 Abidjan Côte D'Ivoire 27.293

1 2000-02-01 Abidjan Côte D'Ivoire 27.685

2 2000-03-01 Abidjan Côte D'Ivoire 29.061

3 2000-04-01 Abidjan Côte D'Ivoire 28.162

4 2000-05-01 Abidjan Côte D'Ivoire 27.547

... ... ... ... ...

16495 2013-05-01 Xian China 18.979

16496 2013-06-01 Xian China 23.522

16497 2013-07-01 Xian China 25.251

16498 2013-08-01 Xian China 24.528

16499 2013-09-01 Xian China NaN

[16500 rows x 4 columns]

date country avg\_temp\_c

city

Abidjan 2000-01-01 Côte D'Ivoire 27.293

Abidjan 2000-02-01 Côte D'Ivoire 27.685

Abidjan 2000-03-01 Côte D'Ivoire 29.061

Abidjan 2000-04-01 Côte D'Ivoire 28.162

Abidjan 2000-05-01 Côte D'Ivoire 27.547

... ... ... ...

Xian 2013-05-01 China 18.979

Xian 2013-06-01 China 23.522

Xian 2013-07-01 China 25.251

Xian 2013-08-01 China 24.528

Xian 2013-09-01 China NaN

[16500 rows x 3 columns]

city date country avg\_temp\_c

0 Abidjan 2000-01-01 Côte D'Ivoire 27.293

1 Abidjan 2000-02-01 Côte D'Ivoire 27.685

2 Abidjan 2000-03-01 Côte D'Ivoire 29.061

3 Abidjan 2000-04-01 Côte D'Ivoire 28.162

4 Abidjan 2000-05-01 Côte D'Ivoire 27.547

... ... ... ... ...

16495 Xian 2013-05-01 China 18.979

16496 Xian 2013-06-01 China 23.522

16497 Xian 2013-07-01 China 25.251

16498 Xian 2013-08-01 China 24.528

16499 Xian 2013-09-01 China NaN

[16500 rows x 4 columns]

date country avg\_temp\_c

0 2000-01-01 Côte D'Ivoire 27.293

1 2000-02-01 Côte D'Ivoire 27.685

2 2000-03-01 Côte D'Ivoire 29.061

3 2000-04-01 Côte D'Ivoire 28.162

4 2000-05-01 Côte D'Ivoire 27.547

... ... ... ...

16495 2013-05-01 China 18.979

16496 2013-06-01 China 23.522

16497 2013-07-01 China 25.251

16498 2013-08-01 China 24.528

16499 2013-09-01 China NaN

[16500 rows x 3 columns]

**Daily XP500**

##### Exercise

##### Exercise

# Subsetting with .loc[]

The killer feature for indexes is .loc[]: a subsetting method that accepts index values. When you pass it a single argument, it will take a subset of rows.

The code for subsetting using .loc[] can be easier to read than standard square bracket subsetting, which can make your code less burdensome to maintain.

pandas is loaded as pd. temperatures and temperatures\_ind are available; the latter is indexed by city.

##### Instructions

**100 XP**

* Create a list called cities that contains "Moscow" and "Saint Petersburg".
* Use [] subsetting to filter temperatures for rows where the city column takes a value in the cities list.
* Use .loc[] subsetting to filter temperatures\_ind for rows where the city is in the cities list.
* # Make a list of cities to subset on
* cities = ["\_\_\_\_", "\_\_\_\_"]
* # Subset temperatures using square brackets
* print(temperatures[\_\_\_\_])
* # Subset temperatures\_ind using .loc[]
* print(temperatures\_ind.loc[\_\_\_\_])

# Look at temperatures

print(temperatures)

# Set the index of temperatures to city

temperatures\_ind = temperatures.set\_index("city")

# Look at temperatures\_ind

print(temperatures\_ind)

# Reset the temperatures\_ind index, keeping its contents

print(temperatures\_ind.reset\_index())

# Reset the temperatures\_ind index, dropping its contents

print(temperatures\_ind.reset\_index(drop=True))

# Make a list of cities to subset on

cities = ["Moscow", "Saint Petersburg"]

# Subset temperatures using square brackets

print(temperatures[temperatures["city"].isin(["Moscow", "Saint Petersburg"])])

# Subset temperatures\_ind using .loc[]

print(temperatures\_ind.loc[["Moscow", "Saint Petersburg"]])

# Make a list of cities to subset on

cities = ["Moscow", "Saint Petersburg"]

# Subset temperatures using square brackets

print(temperatures[temperatures["city"].isin(["Moscow", "Saint Petersburg"])])

# Subset temperatures\_ind using .loc[]

print(temperatures\_ind.loc[["Moscow", "Saint Petersburg"]])

date city country avg\_temp\_c

10725 2000-01-01 Moscow Russia -7.313

10726 2000-02-01 Moscow Russia -3.551

10727 2000-03-01 Moscow Russia -1.661

10728 2000-04-01 Moscow Russia 10.096

10729 2000-05-01 Moscow Russia 10.357

... ... ... ... ...

13360 2013-05-01 Saint Petersburg Russia 12.355

13361 2013-06-01 Saint Petersburg Russia 17.185

13362 2013-07-01 Saint Petersburg Russia 17.234

13363 2013-08-01 Saint Petersburg Russia 17.153

13364 2013-09-01 Saint Petersburg Russia NaN

[330 rows x 4 columns]

date country avg\_temp\_c

city

Moscow 2000-01-01 Russia -7.313

Moscow 2000-02-01 Russia -3.551

Moscow 2000-03-01 Russia -1.661

Moscow 2000-04-01 Russia 10.096

Moscow 2000-05-01 Russia 10.357

... ... ... ...

Saint Petersburg 2013-05-01 Russia 12.355

Saint Petersburg 2013-06-01 Russia 17.185

Saint Petersburg 2013-07-01 Russia 17.234

Saint Petersburg 2013-08-01 Russia 17.153

Saint Petersburg 2013-09-01 Russia NaN

[330 rows x 3 columns]

<script.py> output:

date city country avg\_temp\_c

10725 2000-01-01 Moscow Russia -7.313

10726 2000-02-01 Moscow Russia -3.551

10727 2000-03-01 Moscow Russia -1.661

10728 2000-04-01 Moscow Russia 10.096

10729 2000-05-01 Moscow Russia 10.357

... ... ... ... ...

13360 2013-05-01 Saint Petersburg Russia 12.355

13361 2013-06-01 Saint Petersburg Russia 17.185

13362 2013-07-01 Saint Petersburg Russia 17.234

13363 2013-08-01 Saint Petersburg Russia 17.153

13364 2013-09-01 Saint Petersburg Russia NaN

[330 rows x 4 columns]

date country avg\_temp\_c

city

Moscow 2000-01-01 Russia -7.313

Moscow 2000-02-01 Russia -3.551

Moscow 2000-03-01 Russia -1.661

Moscow 2000-04-01 Russia 10.096

Moscow 2000-05-01 Russia 10.357

... ... ... ...

Saint Petersburg 2013-05-01 Russia 12.355

Saint Petersburg 2013-06-01 Russia 17.185

Saint Petersburg 2013-07-01 Russia 17.234

Saint Petersburg 2013-08-01 Russia 17.153

Saint Petersburg 2013-09-01 Russia NaN

[330 rows x 3 columns]

**Daily XP570**

##### Exercise

##### Exercise

# Setting multi-level indexes

Indexes can also be made out of multiple columns, forming a multi-level index (sometimes called a hierarchical index). There is a trade-off to using these.

The benefit is that multi-level indexes make it more natural to reason about nested categorical variables. For example, in a clinical trial, you might have control and treatment groups. Then each test subject belongs to one or another group, and we can say that a test subject is nested inside the treatment group. Similarly, in the temperature dataset, the city is located in the country, so we can say a city is nested inside the country.

The main downside is that the code for manipulating indexes is different from the code for manipulating columns, so you have to learn two syntaxes and keep track of how your data is represented.

pandas is loaded as pd. temperatures is available.

##### Instructions

**100 XP**

* Set the index of temperatures to the "country" and "city" columns, and assign this to temperatures\_ind.
* Specify two country/city pairs to keep: "Brazil"/"Rio De Janeiro" and "Pakistan"/"Lahore", assigning to rows\_to\_keep.
* Print and subset temperatures\_ind for rows\_to\_keep using .loc[].
* # Index temperatures by country & city
* temperatures\_ind = \_\_\_\_
* # List of tuples: Brazil, Rio De Janeiro & Pakistan, Lahore
* rows\_to\_keep = [\_\_\_\_]
* # Subset for rows to keep
* print(temperatures\_ind.\_\_\_\_)

# Make a list of cities to subset on cities = ["Moscow", "Saint Petersburg"] # Subset temperatures using square brackets print(temperatures[temperatures["city"].isin(["Moscow", "Saint Petersburg"])]) # Subset temperatures\_ind using .loc[] print(temperatures\_ind.loc[["Moscow", "Saint Petersburg"]])

# Index temperatures by country & city

temperatures\_ind = temperatures.set\_index(["country", "city"])

# List of tuples: Brazil, Rio De Janeiro & Pakistan, Lahore

rows\_to\_keep = [("Brazil", "Rio De Janeiro") , ("Pakistan", "Lahore")]

# Subset for rows to keep

print(temperatures\_ind.loc[rows\_to\_keep])

# Index temperatures by country & city

temperatures\_ind = temperatures.set\_index(["country", "city"])

# List of tuples: Brazil, Rio De Janeiro & Pakistan, Lahore

rows\_to\_keep = [("Brazil", "Rio De Janeiro") , ("Pakistan", "Lahore")]

# Subset for rows to keep

print(temperatures\_ind.loc[rows\_to\_keep])

date avg\_temp\_c

country city

Brazil Rio De Janeiro 2000-01-01 25.974

Rio De Janeiro 2000-02-01 26.699

Rio De Janeiro 2000-03-01 26.270

Rio De Janeiro 2000-04-01 25.750

Rio De Janeiro 2000-05-01 24.356

... ... ...

Pakistan Lahore 2013-05-01 33.457

Lahore 2013-06-01 34.456

Lahore 2013-07-01 33.279

Lahore 2013-08-01 31.511

Lahore 2013-09-01 NaN

[330 rows x 2 columns]

**Daily XP670**

##### Exercise

##### Exercise

# Sorting by index values

Previously, you changed the order of the rows in a DataFrame by calling .sort\_values(). It's also useful to be able to sort by elements in the index. For this, you need to use .sort\_index().

pandas is loaded as pd. temperatures\_ind has a multi-level index of country and city, and is available.

##### Instructions

**100 XP**

* Sort temperatures\_ind by the index values.
* Sort temperatures\_ind by the index values at the "city" level.
* Sort temperatures\_ind by ascending country then descending city.
* # Sort temperatures\_ind by index values
* print(\_\_\_\_)
* # Sort temperatures\_ind by index values at the city level
* print(\_\_\_\_)
* # Sort temperatures\_ind by country then descending city
* print(\_\_\_\_)

# Index temperatures by country & city temperatures\_ind = temperatures.set\_index(["country", "city"]) # List of tuples: Brazil, Rio De Janeiro & Pakistan, Lahore rows\_to\_keep = [("Brazil", "Rio De Janeiro") , ("Pakistan", "Lahore")] # Subset for rows to keep print(temperatures\_ind.loc[rows\_to\_keep])

# Sort temperatures\_ind by index values

print(temperatures\_ind.sort\_index())

# Sort temperatures\_ind by index values at the city level

print(temperatures\_ind.sort\_index(level="city"))

# Sort temperatures\_ind by country then descending city

print(temperatures\_ind.sort\_index(level= ["country","city"], ascending=[True, False]))

# Sort temperatures\_ind by index values

print(temperatures\_ind.sort\_index())

# Sort temperatures\_ind by index values at the city level

print(temperatures\_ind.sort\_index(level="city"))

# Sort temperatures\_ind by country then descending city

print(temperatures\_ind.sort\_index(level= ["country","city"], ascending=[True, False]))

date avg\_temp\_c

country city

Afghanistan Kabul 2000-01-01 3.326

Kabul 2000-02-01 3.454

Kabul 2000-03-01 9.612

Kabul 2000-04-01 17.925

Kabul 2000-05-01 24.658

... ... ...

Zimbabwe Harare 2013-05-01 18.298

Harare 2013-06-01 17.020

Harare 2013-07-01 16.299

Harare 2013-08-01 19.232

Harare 2013-09-01 NaN

[16500 rows x 2 columns]

date avg\_temp\_c

country city

Côte D'Ivoire Abidjan 2000-01-01 27.293

Abidjan 2000-02-01 27.685

Abidjan 2000-03-01 29.061

Abidjan 2000-04-01 28.162

Abidjan 2000-05-01 27.547

... ... ...

China Xian 2013-05-01 18.979

Xian 2013-06-01 23.522

Xian 2013-07-01 25.251

Xian 2013-08-01 24.528

Xian 2013-09-01 NaN

[16500 rows x 2 columns]

date avg\_temp\_c

country city

Afghanistan Kabul 2000-01-01 3.326

Kabul 2000-02-01 3.454

Kabul 2000-03-01 9.612

Kabul 2000-04-01 17.925

Kabul 2000-05-01 24.658

... ... ...

Zimbabwe Harare 2013-05-01 18.298

Harare 2013-06-01 17.020

Harare 2013-07-01 16.299

Harare 2013-08-01 19.232

Harare 2013-09-01 NaN

[16500 rows x 2 columns]

**Daily XP100**

# Slicing and subsetting with .loc and .iloc

**50 XP**

## 1. Slicing and subsetting with .loc and .iloc

Slicing is a technique for selecting consecutive elements from objects.

## 2. 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.

## 3. 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.

## 4. 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.

## 5. 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.

## 6. 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.

## 7. 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.

## 8. 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.

## 9. 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.

## 10. Slicing by dates

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

## 11. 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.

## 12. 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.

## 13. Let's practice!

Time for a nice slice!

**Daily XP150**

##### Exercise

##### Exercise

# Slicing index values

Slicing lets you select consecutive elements of an object using first:last syntax. DataFrames can be sliced by index values or by row/column number; we'll start with the first case. This involves slicing inside the .loc[] method.

Compared to slicing lists, there are a few things to remember.

* You can only slice an index if the index is sorted (using .sort\_index()).
* To slice at the outer level, first and last can be strings.
* To slice at inner levels, first and last should be tuples.
* If you pass a single slice to .loc[], it will slice the rows.

pandas is loaded as pd. temperatures\_ind has country and city in the index, and is available.

##### Instructions

**100 XP**

* Sort the index of temperatures\_ind.
* Use slicing with .loc[] to get these subsets:
  + from Pakistan to Russia.
  + from Lahore to Moscow. (This will return nonsense.)
  + from Pakistan, Lahore to Russia, Moscow.
* # Sort the index of temperatures\_ind
* temperatures\_srt = \_\_\_\_
* # Subset rows from Pakistan to Russia
* print(\_\_\_\_)
* # Try to subset rows from Lahore to Moscow
* print(\_\_\_\_)
* # Subset rows from Pakistan, Lahore to Russia, Moscow
* print(\_\_\_\_)

# Sort the index of temperatures\_ind

temperatures\_srt = temperatures\_ind.sort\_index()

print(temperatures\_srt)

# Subset rows from Pakistan to Russia

print(temperatures\_srt.loc["Pakistan" : "Russia"])

# Try to subset rows from Lahore to Moscow

print(temperatures\_srt.loc["Lahore" : "Moscow"])

# Subset rows from Pakistan, Lahore to Russia, Moscow

print(temperatures\_srt.loc[("Pakistan","Lahore"): ("Russia", "Moscow")])

# Sort the index of temperatures\_ind

temperatures\_srt = temperatures\_ind.sort\_index(sort\_remaining=True)

#print(temperatures\_srt)

# Subset rows from Pakistan to Russia

print(temperatures\_srt.loc["Pakistan" : "Russia"])

# Try to subset rows from Lahore to Moscow

print(temperatures\_srt.loc["Lahore" : "Moscow"])

# Subset rows from Pakistan, Lahore to Russia, Moscow

print(temperatures\_srt.loc[("Pakistan","Lahore"): ("Russia", "Moscow")])

# Sort the index of temperatures\_ind

temperatures\_srt = temperatures\_ind.sort\_index(sort\_remaining=True)

#print(temperatures\_srt)

# Subset rows from Pakistan to Russia

print(temperatures\_srt.loc["Pakistan" : "Russia"])

# Try to subset rows from Lahore to Moscow

print(temperatures\_srt.loc["Lahore" : "Moscow"])

# Subset rows from Pakistan, Lahore to Russia, Moscow

print(temperatures\_srt.loc[("Pakistan","Lahore"): ("Russia", "Moscow")])

date avg\_temp\_c

country city

Pakistan Faisalabad 2000-01-01 12.792

Faisalabad 2000-02-01 14.339

Faisalabad 2000-03-01 20.309

Faisalabad 2000-04-01 29.072

Faisalabad 2000-05-01 34.845

... ... ...

Russia Saint Petersburg 2013-05-01 12.355

Saint Petersburg 2013-06-01 17.185

Saint Petersburg 2013-07-01 17.234

Saint Petersburg 2013-08-01 17.153

Saint Petersburg 2013-09-01 NaN

[1155 rows x 2 columns]

date avg\_temp\_c

country city

Mexico Mexico 2000-01-01 12.694

Mexico 2000-02-01 14.677

Mexico 2000-03-01 17.376

Mexico 2000-04-01 18.294

Mexico 2000-05-01 18.562

... ... ...

Morocco Casablanca 2013-05-01 19.217

Casablanca 2013-06-01 23.649

Casablanca 2013-07-01 27.488

Casablanca 2013-08-01 27.952

Casablanca 2013-09-01 NaN

[330 rows x 2 columns]

date avg\_temp\_c

country city

Pakistan Lahore 2000-01-01 12.792

Lahore 2000-02-01 14.339

Lahore 2000-03-01 20.309

Lahore 2000-04-01 29.072

Lahore 2000-05-01 34.845

... ... ...

Russia Moscow 2013-05-01 16.152

Moscow 2013-06-01 18.718

Moscow 2013-07-01 18.136

Moscow 2013-08-01 17.485

Moscow 2013-09-01 NaN

[660 rows x 2 columns]

<script.py> output:

date avg\_temp\_c

country city

Pakistan Faisalabad 2000-01-01 12.792

Faisalabad 2000-02-01 14.339

Faisalabad 2000-03-01 20.309

Faisalabad 2000-04-01 29.072

Faisalabad 2000-05-01 34.845

... ... ...

Russia Saint Petersburg 2013-05-01 12.355

Saint Petersburg 2013-06-01 17.185

Saint Petersburg 2013-07-01 17.234

Saint Petersburg 2013-08-01 17.153

Saint Petersburg 2013-09-01 NaN

[1155 rows x 2 columns]

date avg\_temp\_c

country city

Mexico Mexico 2000-01-01 12.694

Mexico 2000-02-01 14.677

Mexico 2000-03-01 17.376

Mexico 2000-04-01 18.294

Mexico 2000-05-01 18.562

... ... ...

Morocco Casablanca 2013-05-01 19.217

Casablanca 2013-06-01 23.649

Casablanca 2013-07-01 27.488

Casablanca 2013-08-01 27.952

Casablanca 2013-09-01 NaN

[330 rows x 2 columns]

date avg\_temp\_c

country city

Pakistan Lahore 2000-01-01 12.792

Lahore 2000-02-01 14.339

Lahore 2000-03-01 20.309

Lahore 2000-04-01 29.072

Lahore 2000-05-01 34.845

... ... ...

Russia Moscow 2013-05-01 16.152

Moscow 2013-06-01 18.718

Moscow 2013-07-01 18.136

Moscow 2013-08-01 17.485

Moscow 2013-09-01 NaN

[660 rows x 2 columns]

**Daily XP220**

##### Exercise

##### Exercise

# Slicing in both directions

You've seen slicing DataFrames by rows and by columns, but since DataFrames are two-dimensional objects, it is often natural to slice both dimensions at once. That is, by passing two arguments to .loc[], you can subset by rows and columns in one go.

pandas is loaded as pd. temperatures\_srt is indexed by country and city, has a sorted index, and is available.

##### Instructions

**100 XP**

* Use .loc[] slicing to subset rows from India, Hyderabad to Iraq, Baghdad.
* Use .loc[] slicing to subset columns from date to avg\_temp\_c.
* Slice in both directions at once from Hyderabad to Baghdad, and date to avg\_temp\_c.
* # Subset rows from India, Hyderabad to Iraq, Baghdad
* print(\_\_\_\_)
* # Subset columns from date to avg\_temp\_c
* print(\_\_\_\_)
* # Subset in both directions at once
* print(\_\_\_\_)

# Sort the index of temperatures\_ind

temperatures\_srt = temperatures\_ind.sort\_index(sort\_remaining=True)

#print(temperatures\_srt)

# Subset rows from Pakistan to Russia

print(temperatures\_srt.loc["Pakistan" : "Russia"])

# Try to subset rows from Lahore to Moscow

print(temperatures\_srt.loc["Lahore" : "Moscow"])

# Subset rows from Pakistan, Lahore to Russia, Moscow

print(temperatures\_srt.loc[("Pakistan","Lahore"): ("Russia", "Moscow")])

# Subset rows from India, Hyderabad to Iraq, Baghdad

print(temperatures\_srt.loc[("India","Hyderabad"):("Iraq","Baghdad")])

# Subset columns from date to avg\_temp\_c

print(temperatures\_srt.loc[ :, "date":"avg\_temp\_c"])

# Subset in both directions at once

print(temperatures\_srt.loc[("India","Hyderabad"):("Iraq","Baghdad") , "date":"avg\_temp\_c"])

# Subset rows from India, Hyderabad to Iraq, Baghdad

print(temperatures\_srt.loc[("India","Hyderabad"):("Iraq","Baghdad")])

# Subset columns from date to avg\_temp\_c

print(temperatures\_srt.loc[ :, "date":"avg\_temp\_c"])

# Subset in both directions at once

print(temperatures\_srt.loc[("India","Hyderabad"):("Iraq","Baghdad") , "date":"avg\_temp\_c"])

date avg\_temp\_c

country city

India Hyderabad 2000-01-01 23.779

Hyderabad 2000-02-01 25.826

Hyderabad 2000-03-01 28.821

Hyderabad 2000-04-01 32.698

Hyderabad 2000-05-01 32.438

... ... ...

Iraq Baghdad 2013-05-01 28.673

Baghdad 2013-06-01 33.803

Baghdad 2013-07-01 36.392

Baghdad 2013-08-01 35.463

Baghdad 2013-09-01 NaN

[2145 rows x 2 columns]

date avg\_temp\_c

country city

Afghanistan Kabul 2000-01-01 3.326

Kabul 2000-02-01 3.454

Kabul 2000-03-01 9.612

Kabul 2000-04-01 17.925

Kabul 2000-05-01 24.658

... ... ...

Zimbabwe Harare 2013-05-01 18.298

Harare 2013-06-01 17.020

Harare 2013-07-01 16.299

Harare 2013-08-01 19.232

Harare 2013-09-01 NaN

[16500 rows x 2 columns]

date avg\_temp\_c

country city

India Hyderabad 2000-01-01 23.779

Hyderabad 2000-02-01 25.826

Hyderabad 2000-03-01 28.821

Hyderabad 2000-04-01 32.698

Hyderabad 2000-05-01 32.438

... ... ...

Iraq Baghdad 2013-05-01 28.673

Baghdad 2013-06-01 33.803

Baghdad 2013-07-01 36.392

Baghdad 2013-08-01 35.463

Baghdad 2013-09-01 NaN

[2145 rows x 2 columns]

<script.py> output:

date avg\_temp\_c

country city

India Hyderabad 2000-01-01 23.779

Hyderabad 2000-02-01 25.826

Hyderabad 2000-03-01 28.821

Hyderabad 2000-04-01 32.698

Hyderabad 2000-05-01 32.438

... ... ...

Iraq Baghdad 2013-05-01 28.673

Baghdad 2013-06-01 33.803

Baghdad 2013-07-01 36.392

Baghdad 2013-08-01 35.463

Baghdad 2013-09-01 NaN

[2145 rows x 2 columns]

date avg\_temp\_c

country city

Afghanistan Kabul 2000-01-01 3.326

Kabul 2000-02-01 3.454

Kabul 2000-03-01 9.612

Kabul 2000-04-01 17.925

Kabul 2000-05-01 24.658

... ... ...

Zimbabwe Harare 2013-05-01 18.298

Harare 2013-06-01 17.020

Harare 2013-07-01 16.299

Harare 2013-08-01 19.232

Harare 2013-09-01 NaN

[16500 rows x 2 columns]

date avg\_temp\_c

country city

India Hyderabad 2000-01-01 23.779

Hyderabad 2000-02-01 25.826

Hyderabad 2000-03-01 28.821

Hyderabad 2000-04-01 32.698

Hyderabad 2000-05-01 32.438

... ... ...

Iraq Baghdad 2013-05-01 28.673

Baghdad 2013-06-01 33.803

Baghdad 2013-07-01 36.392

Baghdad 2013-08-01 35.463

Baghdad 2013-09-01 NaN

[2145 rows x 2 columns]

**Daily XP380**

##### Exercise

##### Exercise

# Slicing time series

Slicing is particularly useful for time series since it's a common thing to want to filter for data within a date range. Add the date column to the index, then use .loc[] to perform the subsetting. The important thing to remember is to keep your dates in ISO 8601 format, that is, "yyyy-mm-dd" for year-month-day, "yyyy-mm" for year-month, and "yyyy" for year.

Recall from Chapter 1 that you can combine multiple Boolean conditions using logical operators, such as &. To do so in one line of code, you'll need to add parentheses () around each condition.

pandas is loaded as pd and temperatures, with no index, is available.

##### Instructions

**100 XP**

* Use Boolean conditions, not .isin() or .loc[], and the full date "yyyy-mm-dd", to subset temperatures for rows in 2010 and 2011 and print the results.
* Set the index of temperatures to the date column and sort it.
* Use .loc[] to subset temperatures\_ind for rows in 2010 and 2011.
* Use .loc[] to subset temperatures\_ind for rows from Aug 2010 to Feb 2011.
* # Use Boolean conditions to subset temperatures for rows in 2010 and 2011
* temperatures\_bool = \_\_\_\_[(\_\_\_\_ >= \_\_\_\_) & (\_\_\_\_ <= \_\_\_\_)]
* print(temperatures\_bool)
* # Set date as the index and sort the index
* temperatures\_ind = temperatures.\_\_\_\_.\_\_\_\_
* # Use .loc[] to subset temperatures\_ind for rows in 2010 and 2011
* print(\_\_\_\_)
* # Use .loc[] to subset temperatures\_ind for rows from Aug 2010 to Feb 2011
* print(\_\_\_\_)

# Subset rows from India, Hyderabad to Iraq, Baghdad

print(temperatures\_srt.loc[("India","Hyderabad"):("Iraq","Baghdad")])

# Subset columns from date to avg\_temp\_c

print(temperatures\_srt.loc[ :, "date":"avg\_temp\_c"])

# Subset in both directions at once

print(temperatures\_srt.loc[("India","Hyderabad"):("Iraq","Baghdad") , "date":"avg\_temp\_c"])

# Use Boolean conditions to subset temperatures for rows in 2010 and 2011

temperatures\_bool = temperatures[(temperatures["date"] >= "2010-01-01") & (temperatures["date"] <= "2011-12-31")]

print(temperatures\_bool)

#print(temperatures)

# Set date as the index and sort the index

temperatures\_ind = temperatures.set\_index("date").sort\_index()

# Use .loc[] to subset temperatures\_ind for rows in 2010 and 2011

print(temperatures\_ind.loc["2010":"2011"])

# Use .loc[] to subset temperatures\_ind for rows from Aug 2010 to Feb 2011

print(temperatures\_ind.loc["2010-08":"2011-02"])

# Use Boolean conditions to subset temperatures for rows in 2010 and 2011

temperatures\_bool = temperatures[(temperatures["date"] >= "2010-01-01") & (temperatures["date"] <= "2011-12-31")]

print(temperatures\_bool)

#print(temperatures)

# Set date as the index and sort the index

temperatures\_ind = temperatures.set\_index("date").sort\_index()

# Use .loc[] to subset temperatures\_ind for rows in 2010 and 2011

print(temperatures\_ind.loc["2010":"2011"])

# Use .loc[] to subset temperatures\_ind for rows from Aug 2010 to Feb 2011

print(temperatures\_ind.loc["2010-08":"2011-02"])

date city country avg\_temp\_c

120 2010-01-01 Abidjan Côte D'Ivoire 28.270

121 2010-02-01 Abidjan Côte D'Ivoire 29.262

122 2010-03-01 Abidjan Côte D'Ivoire 29.596

123 2010-04-01 Abidjan Côte D'Ivoire 29.068

124 2010-05-01 Abidjan Côte D'Ivoire 28.258

... ... ... ... ...

16474 2011-08-01 Xian China 23.069

16475 2011-09-01 Xian China 16.775

16476 2011-10-01 Xian China 12.587

16477 2011-11-01 Xian China 7.543

16478 2011-12-01 Xian China -0.490

[2400 rows x 4 columns]

ERROR! Session/line number was not unique in database. History logging moved to new session 29

city country avg\_temp\_c

date

2010-01-01 Faisalabad Pakistan 11.810

2010-01-01 Melbourne Australia 20.016

2010-01-01 Chongqing China 7.921

2010-01-01 São Paulo Brazil 23.738

2010-01-01 Guangzhou China 14.136

... ... ... ...

2011-12-01 Nagoya Japan 6.476

2011-12-01 Hyderabad India 23.613

2011-12-01 Cali Colombia 21.559

2011-12-01 Lima Peru 18.293

2011-12-01 Bangkok Thailand 25.021

[2400 rows x 3 columns]

city country avg\_temp\_c

date

2010-08-01 Calcutta India 30.226

2010-08-01 Pune India 24.941

2010-08-01 Izmir Turkey 28.352

2010-08-01 Tianjin China 25.543

2010-08-01 Manila Philippines 27.101

... ... ... ...

2011-02-01 Kabul Afghanistan 3.914

2011-02-01 Chicago United States 0.276

2011-02-01 Aleppo Syria 8.246

2011-02-01 Delhi India 18.136

2011-02-01 Rangoon Burma 26.631

[700 rows x 3 columns]

**Daily XP480**

##### Exercise

##### Exercise

# Subsetting by row/column number

The most common ways to subset rows are the ways we've previously discussed: using a Boolean condition or by index labels. However, it is also occasionally useful to pass row numbers.

This is done using .iloc[], and like .loc[], it can take two arguments to let you subset by rows and columns.

pandas is loaded as pd. temperatures (without an index) is available.

##### Instructions

**100 XP**

Use .iloc[] on temperatures to take subsets.

* Get the 23rd row, 2nd column (index positions 22 and 1).
* Get the first 5 rows (index positions 0 to 5).
* Get all rows, columns 3 and 4 (index positions 2 to 4).
* Get the first 5 rows, columns 3 and 4.
* # Get 23rd row, 2nd column (index 22, 1)
* print(temperatures.iloc[22,1])
* # Use slicing to get the first 5 rows
* print(temperatures.iloc[:5])
* # Use slicing to get columns 3 to 4
* print(temperatures.iloc[,3:4])
* # Use slicing in both directions at once
* print(\_\_\_\_)

# Use Boolean conditions to subset temperatures for rows in 2010 and 2011 temperatures\_bool = temperatures[(temperatures["date"] >= "2010-01-01") & (temperatures["date"] <= "2011-12-31")] print(temperatures\_bool) #print(temperatures) # Set date as the index and sort the index temperatures\_ind = temperatures.set\_index("date").sort\_index() # Use .loc[] to subset temperatures\_ind for rows in 2010 and 2011 print(temperatures\_ind.loc["2010":"2011"]) # Use .loc[] to subset temperatures\_ind for rows from Aug 2010 to Feb 2011 print(temperatures\_ind.loc["2010-08":"2011-02"])

# Get 23rd row, 2nd column (index 22, 1)

print(temperatures.iloc[22,1])

# Use slicing to get the first 5 rows

print(temperatures.iloc[:5])

# Use slicing to get columns 3 to 4

print(temperatures.iloc[:,2:4])

# Use slicing in both directions at once

print(temperatures.iloc[:5,2:4])

# Get 23rd row, 2nd column (index 22, 1)

print(temperatures.iloc[22,1])

# Use slicing to get the first 5 rows

print(temperatures.iloc[:5])

# Use slicing to get columns 3 to 4

print(temperatures.iloc[:,2:4])

# Use slicing in both directions at once

print(temperatures.iloc[:5,2:4])

Abidjan

date city country avg\_temp\_c

0 2000-01-01 Abidjan Côte D'Ivoire 27.293

1 2000-02-01 Abidjan Côte D'Ivoire 27.685

2 2000-03-01 Abidjan Côte D'Ivoire 29.061

3 2000-04-01 Abidjan Côte D'Ivoire 28.162

4 2000-05-01 Abidjan Côte D'Ivoire 27.547

country avg\_temp\_c

0 Côte D'Ivoire 27.293

1 Côte D'Ivoire 27.685

2 Côte D'Ivoire 29.061

3 Côte D'Ivoire 28.162

4 Côte D'Ivoire 27.547

... ... ...

16495 China 18.979

16496 China 23.522

16497 China 25.251

16498 China 24.528

16499 China NaN

[16500 rows x 2 columns]

country avg\_temp\_c

0 Côte D'Ivoire 27.293

1 Côte D'Ivoire 27.685

2 Côte D'Ivoire 29.061

3 Côte D'Ivoire 28.162

4 Côte D'Ivoire 27.547

<script.py> output:

Abidjan

date city country avg\_temp\_c

0 2000-01-01 Abidjan Côte D'Ivoire 27.293

1 2000-02-01 Abidjan Côte D'Ivoire 27.685

2 2000-03-01 Abidjan Côte D'Ivoire 29.061

3 2000-04-01 Abidjan Côte D'Ivoire 28.162

4 2000-05-01 Abidjan Côte D'Ivoire 27.547

country avg\_temp\_c

0 Côte D'Ivoire 27.293

1 Côte D'Ivoire 27.685

2 Côte D'Ivoire 29.061

3 Côte D'Ivoire 28.162

4 Côte D'Ivoire 27.547

... ... ...

16495 China 18.979

16496 China 23.522

16497 China 25.251

16498 China 24.528

16499 China NaN

[16500 rows x 2 columns]

country avg\_temp\_c

0 Côte D'Ivoire 27.293

1 Côte D'Ivoire 27.685

2 Côte D'Ivoire 29.061

3 Côte D'Ivoire 28.162

4 Côte D'Ivoire 27.547

Daily XP

580

Working with pivot tables

50 XP

1. 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.

2. A bigger dog dataset

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

3. 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.

4. .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.

5. 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.

6. 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.

7. Let's practice!

Time to play with pivot tables!

**Daily XP630**

##### Exercise

##### Exercise

# Pivot temperature by city and year

It's interesting to see how temperatures for each city change over time—looking at every month results in a big table, which can be tricky to reason about. Instead, let's look at how temperatures change by year.

You can access the components of a date (year, month and day) using code of the form dataframe["column"].dt.component. For example, the month component is dataframe["column"].dt.month, and the year component is dataframe["column"].dt.year.

Once you have the year column, you can create a pivot table with the data aggregated by city and year, which you'll explore in the coming exercises.

pandas is loaded as pd. temperatures is available.

##### Instructions

**100 XP**

* Add a year column to temperatures, from the year component of the date column.
* Make a pivot table of the avg\_temp\_c column, with country and city as rows, and year as columns. Assign to temp\_by\_country\_city\_vs\_year, and look at the result.
* # Add a year column to temperatures
* \_\_\_\_
* # Pivot avg\_temp\_c by country and city vs year
* temp\_by\_country\_city\_vs\_year = \_\_\_\_
* # See the result
* print(temp\_by\_country\_city\_vs\_year)

# Get 23rd row, 2nd column (index 22, 1) print(temperatures.iloc[22,1]) # Use slicing to get the first 5 rows print(temperatures.iloc[:5]) # Use slicing to get columns 3 to 4 print(temperatures.iloc[:,2:4]) # Use slicing in both directions at once print(temperatures.iloc[:5,2:4])

# Add a year column to temperatures

temperatures["year"] = temperatures["date"].dt.year

#print(temperatures)

# Pivot avg\_temp\_c by country and city vs year

temp\_by\_country\_city\_vs\_year = temperatures.pivot\_table("avg\_temp\_c", index=["country", "city"], columns= "year")

# See the result

print(temp\_by\_country\_city\_vs\_year)

* script.py

1

2

4

5

6

7

3

8



* IPython Shell
* Slides
* Notes

# Add a year column to temperatures

print(temperatures)

# Pivot avg\_temp\_c by country and city vs year

temp\_by\_country\_city\_vs\_year = \_\_\_\_

# See the result

print(temp\_by\_country\_city\_vs\_year)

date city country avg\_temp\_c

0 2000-01-01 Abidjan Côte D'Ivoire 27.293

1 2000-02-01 Abidjan Côte D'Ivoire 27.685

2 2000-03-01 Abidjan Côte D'Ivoire 29.061

3 2000-04-01 Abidjan Côte D'Ivoire 28.162

4 2000-05-01 Abidjan Côte D'Ivoire 27.547

... ... ... ... ...

16495 2013-05-01 Xian China 18.979

16496 2013-06-01 Xian China 23.522

16497 2013-07-01 Xian China 25.251

16498 2013-08-01 Xian China 24.528

16499 2013-09-01 Xian China NaN

[16500 rows x 4 columns]

Traceback (most recent call last):

File "<stdin>", line 72, in exceptionCatcher

raise exception

File "<stdin>", line 3361, in run\_ast\_nodes

if (await self.run\_code(code, result, async\_=asy)):

File "<stdin>", line 3458, in run\_code

self.showtraceback(running\_compiled\_code=True)

File "<stdin>", line 2066, in showtraceback

self.\_showtraceback(etype, value, stb)

File "<stdin>", line 72, in exceptionCatcher

raise exception

File "<stdin>", line 3441, in run\_code

exec(code\_obj, self.user\_global\_ns, self.user\_ns)

File "<stdin>", line 5, in <module>

temp\_by\_country\_city\_vs\_year = \_\_\_\_

NameError: name '\_\_\_\_' is not defined

# Add a year column to temperatures

print(temperatures)

temperatures["year"] = temperatures["date"].dt.year

print(temperatures)

# Pivot avg\_temp\_c by country and city vs year

temp\_by\_country\_city\_vs\_year = \_\_\_\_

# See the result

print(temp\_by\_country\_city\_vs\_year)

date city country avg\_temp\_c

0 2000-01-01 Abidjan Côte D'Ivoire 27.293

1 2000-02-01 Abidjan Côte D'Ivoire 27.685

2 2000-03-01 Abidjan Côte D'Ivoire 29.061

3 2000-04-01 Abidjan Côte D'Ivoire 28.162

4 2000-05-01 Abidjan Côte D'Ivoire 27.547

... ... ... ... ...

16495 2013-05-01 Xian China 18.979

16496 2013-06-01 Xian China 23.522

16497 2013-07-01 Xian China 25.251

16498 2013-08-01 Xian China 24.528

16499 2013-09-01 Xian China NaN

[16500 rows x 4 columns]

date city country avg\_temp\_c year

0 2000-01-01 Abidjan Côte D'Ivoire 27.293 2000

1 2000-02-01 Abidjan Côte D'Ivoire 27.685 2000

2 2000-03-01 Abidjan Côte D'Ivoire 29.061 2000

3 2000-04-01 Abidjan Côte D'Ivoire 28.162 2000

4 2000-05-01 Abidjan Côte D'Ivoire 27.547 2000

... ... ... ... ... ...

16495 2013-05-01 Xian China 18.979 2013

16496 2013-06-01 Xian China 23.522 2013

16497 2013-07-01 Xian China 25.251 2013

16498 2013-08-01 Xian China 24.528 2013

16499 2013-09-01 Xian China NaN 2013

[16500 rows x 5 columns]

Traceback (most recent call last):

File "<stdin>", line 72, in exceptionCatcher

raise exception

File "<stdin>", line 3361, in run\_ast\_nodes

if (await self.run\_code(code, result, async\_=asy)):

File "<stdin>", line 3458, in run\_code

self.showtraceback(running\_compiled\_code=True)

File "<stdin>", line 2066, in showtraceback

self.\_showtraceback(etype, value, stb)

File "<stdin>", line 72, in exceptionCatcher

raise exception

File "<stdin>", line 3441, in run\_code

exec(code\_obj, self.user\_global\_ns, self.user\_ns)

File "<stdin>", line 6, in <module>

temp\_by\_country\_city\_vs\_year = \_\_\_\_

NameError: name '\_\_\_\_' is not defined

# Add a year column to temperatures

temperatures["year"] = temperatures["date"].dt.year

#print(temperatures)

# Pivot avg\_temp\_c by country and city vs year

temp\_by\_country\_city\_vs\_year = temperatures.pivot\_table("avg\_temp\_c", index=["country", "city"], columns= "year")

# See the result

print(temp\_by\_country\_city\_vs\_year)

year 2000 2001 2002 2003 2004 ... 2009 2010 2011 2012 2013

country city ...

Afghanistan Kabul 15.823 15.848 15.715 15.133 16.128 ... 15.093 15.676 15.812 14.510 16.206

Angola Luanda 24.410 24.427 24.791 24.867 24.216 ... 24.325 24.440 24.151 24.240 24.554

Australia Melbourne 14.320 14.180 14.076 13.986 13.742 ... 14.647 14.232 14.191 14.269 14.742

Sydney 17.567 17.854 17.734 17.592 17.870 ... 18.176 17.999 17.713 17.474 18.090

Bangladesh Dhaka 25.905 25.931 26.095 25.927 26.136 ... 26.536 26.648 25.803 26.284 26.587

... ... ... ... ... ... ... ... ... ... ... ...

United States Chicago 11.090 11.703 11.532 10.482 10.943 ... 10.298 11.816 11.214 12.821 11.587

Los Angeles 16.643 16.466 16.430 16.945 16.553 ... 16.677 15.887 15.875 17.090 18.121

New York 9.969 10.931 11.252 9.836 10.389 ... 10.142 11.358 11.272 11.971 12.164

Vietnam Ho Chi Minh City 27.589 27.832 28.065 27.828 27.687 ... 27.853 28.282 27.675 28.249 28.455

Zimbabwe Harare 20.284 20.861 21.079 20.889 20.308 ... 20.524 21.166 20.782 20.523 19.756

[100 rows x 14 columns]

**Daily XP730**

##### Exercise

##### Exercise

# Subsetting pivot tables

A pivot table is just a DataFrame with sorted indexes, so the techniques you have learned already can be used to subset them. In particular, the .loc[] + slicing combination is often helpful.

pandas is loaded as pd. temp\_by\_country\_city\_vs\_year is available.

##### Instructions

**100 XP**

Use .loc[] on temp\_by\_country\_city\_vs\_year to take subsets.

* From Egypt to India.
* From Egypt, Cairo to India, Delhi.
* From Egypt, Cairo to India, Delhi, and 2005 to 2010.
* # Subset for Egypt to India
* \_\_\_\_
* # Subset for Egypt, Cairo to India, Delhi
* \_\_\_\_
* # Subset for Egypt, Cairo to India, Delhi, and 2005 to 2010
* \_\_\_\_

# Add a year column to temperatures temperatures["year"] = temperatures["date"].dt.year #print(temperatures) # Pivot avg\_temp\_c by country and city vs year temp\_by\_country\_city\_vs\_year = temperatures.pivot\_table("avg\_temp\_c", index=["country", "city"], columns= "year") # See the result print(temp\_by\_country\_city\_vs\_year)

# Subset for Egypt to India

#print(temp\_by\_country\_city\_vs\_year)

temp\_by\_country\_city\_vs\_year.loc["Egypt":"India"]

# Subset for Egypt, Cairo to India, Delhi

temp\_by\_country\_city\_vs\_year.loc[("Egypt", "Cairo") :("India", "Delhi")]

# Subset for Egypt, Cairo to India, Delhi, and 2005 to 2010

temp\_by\_country\_city\_vs\_year.loc[("Egypt", "Cairo") : ("India", "Delhi") , "2005": "2010"]

# Subset for Egypt to India

#print(temp\_by\_country\_city\_vs\_year)

temp\_by\_country\_city\_vs\_year.loc["Egypt":"India"]

# Subset for Egypt, Cairo to India, Delhi

temp\_by\_country\_city\_vs\_year.loc[("Egypt", "Cairo") :("India", "Delhi")]

# Subset for Egypt, Cairo to India, Delhi, and 2005 to 2010

temp\_by\_country\_city\_vs\_year.loc[("Egypt", "Cairo") : ("India", "Delhi") , "2005": "2010"]

year 2005 2006 2007 2008 2009 2010

country city

Egypt Cairo 22.006 22.050 22.361 22.644 22.625 23.718

Gizeh 22.006 22.050 22.361 22.644 22.625 23.718

Ethiopia Addis Abeba 18.313 18.427 18.143 18.165 18.765 18.298

France Paris 11.553 11.788 11.751 11.278 11.464 10.410

Germany Berlin 9.919 10.545 10.883 10.658 10.062 8.607

India Ahmadabad 26.828 27.283 27.511 27.049 28.096 28.018

Bangalore 25.477 25.418 25.464 25.353 25.726 25.705

Bombay 27.036 27.382 27.635 27.178 27.845 27.765

Calcutta 26.729 26.986 26.585 26.522 27.153 27.289

Delhi 25.716 26.366 26.146 25.675 26.554 26.520

# Subset for Egypt to India

#print(temp\_by\_country\_city\_vs\_year)

temp\_by\_country\_city\_vs\_year.loc["Egypt":"India"]

# Subset for Egypt, Cairo to India, Delhi

temp\_by\_country\_city\_vs\_year.loc[("Egypt", "Cairo") :("India", "Delhi")]

# Subset for Egypt, Cairo to India, Delhi, and 2005 to 2010

temp\_by\_country\_city\_vs\_year.loc[("Egypt", "Cairo") : ("India", "Delhi") , "2005": "2010"]

year 2005 2006 2007 2008 2009 2010

country city

Egypt Cairo 22.006 22.050 22.361 22.644 22.625 23.718

Gizeh 22.006 22.050 22.361 22.644 22.625 23.718

Ethiopia Addis Abeba 18.313 18.427 18.143 18.165 18.765 18.298

France Paris 11.553 11.788 11.751 11.278 11.464 10.410

Germany Berlin 9.919 10.545 10.883 10.658 10.062 8.607

India Ahmadabad 26.828 27.283 27.511 27.049 28.096 28.018

Bangalore 25.477 25.418 25.464 25.353 25.726 25.705

Bombay 27.036 27.382 27.635 27.178 27.845 27.765

Calcutta 26.729 26.986 26.585 26.522 27.153 27.289

Delhi 25.716 26.366 26.146 25.675 26.554 26.520

**Daily XP830**

##### Exercise

##### Exercise

# Calculating on a pivot table

Pivot tables are filled with summary statistics, but they are only a first step to finding something insightful. Often you'll need to perform further calculations on them. A common thing to do is to find the rows or columns where the highest or lowest value occurs.

Recall from Chapter 1 that you can easily subset a Series or DataFrame to find rows of interest using a logical condition inside of square brackets. For example: series[series > value].

pandas is loaded as pd and the DataFrame temp\_by\_country\_city\_vs\_year is available.

##### Instructions

**100 XP**

* Calculate the mean temperature for each year, assigning to mean\_temp\_by\_year.
* Filter mean\_temp\_by\_year for the year that had the highest mean temperature.
* Calculate the mean temperature for each city (across columns), assigning to mean\_temp\_by\_city.
* Filter mean\_temp\_by\_city for the city that had the lowest mean temperature.
* # Get the worldwide mean temp by year
* mean\_temp\_by\_year = temp\_by\_country\_city\_vs\_year.\_\_\_\_
* # Filter for the year that had the highest mean temp
* print(mean\_temp\_by\_year[\_\_\_\_])
* # Get the mean temp by city
* mean\_temp\_by\_city = temp\_by\_country\_city\_vs\_year.\_\_\_\_
* # Filter for the city that had the lowest mean temp
* print(mean\_temp\_by\_city[\_\_\_\_])

# Subset for Egypt to India #print(temp\_by\_country\_city\_vs\_year) temp\_by\_country\_city\_vs\_year.loc["Egypt":"India"] # Subset for Egypt, Cairo to India, Delhi temp\_by\_country\_city\_vs\_year.loc[("Egypt", "Cairo") :("India", "Delhi")] # Subset for Egypt, Cairo to India, Delhi, and 2005 to 2010 temp\_by\_country\_city\_vs\_year.loc[("Egypt", "Cairo") : ("India", "Delhi") , "2005": "2010"]

# Get the worldwide mean temp by year

mean\_temp\_by\_year = temp\_by\_country\_city\_vs\_year.mean(axis="index")

print(mean\_temp\_by\_year)

# Filter for the year that had the highest mean temp

print(mean\_temp\_by\_year[mean\_temp\_by\_year  == mean\_temp\_by\_year.max()])

# Get the mean temp by city

mean\_temp\_by\_city = temp\_by\_country\_city\_vs\_year.mean(axis="columns")

print(mean\_temp\_by\_city)

# Filter for the city that had the lowest mean temp

print(mean\_temp\_by\_city[mean\_temp\_by\_city == mean\_temp\_by\_city.min()])

# Get the worldwide mean temp by year

mean\_temp\_by\_year = temp\_by\_country\_city\_vs\_year.mean(axis="index")

print(mean\_temp\_by\_year)

# Filter for the year that had the highest mean temp

print(mean\_temp\_by\_year[mean\_temp\_by\_year == mean\_temp\_by\_year.max()])

# Get the mean temp by city

mean\_temp\_by\_city = temp\_by\_country\_city\_vs\_year.mean(axis="columns")

print(mean\_temp\_by\_city)

# Filter for the city that had the lowest mean temp

print(mean\_temp\_by\_city[mean\_temp\_by\_city == mean\_temp\_by\_city.min()])

year

2000 19.506

2001 19.679

2002 19.856

2003 19.630

2004 19.672

2005 19.607

2006 19.794

2007 19.854

2008 19.609

2009 19.834

2010 19.912

2011 19.549

2012 19.668

2013 20.312

dtype: float64

year

2013 20.312

dtype: float64

country city

Afghanistan Kabul 15.542

Angola Luanda 24.392

Australia Melbourne 14.275

Sydney 17.799

Bangladesh Dhaka 26.174

...

United States Chicago 11.331

Los Angeles 16.675

New York 10.911

Vietnam Ho Chi Minh City 27.923

Zimbabwe Harare 20.699

Length: 100, dtype: float64

country city

China Harbin 4.877

dtype: float64

**Daily XP100**

# Visualizing your data

**50 XP**

## 1. Visualizing your data

Plots are a powerful way to share the insights you've gained from your data. In this lesson, we'll use a bigger dataset of dogs, called dog\_pack, to make visualization easier.

## 2. Histograms

Remember when we talked about matplotlib at the beginning of the course? We'll need to import matplotlib-dot-pyplot as plt in order to display our visualizations. Just like pd is the standard alias for pandas, plt is the standard alias for matplotlib-dot-pyplot. Let's create a histogram, which shows the distribution of a numeric variable. We can create a histogram of the height variable by selecting the column and calling dot-hist. In order to show the plot, we need to call plt-dot-show. The x-axis represents the heights of the dogs, and the y-axis represents the number of dogs in each height range. By grouping observations into ranges, the histogram allows us to see that there are a lot of dogs around 50 to 60 centimeters tall.

## 3. Histograms

We can adjust the number of bars, or bins, using the "bins" argument. Increasing or decreasing this can give us a better idea of what the distribution looks like.

## 4. Bar plots

Bar plots can reveal relationships between a categorical variable and a numeric variable, like breed and weight. To compute the average weight of each breed, we group by breed, select the weight column, and take the mean, giving us the average weight of each breed.

## 5. Bar plots

Now we can create a bar plot from the mean weights using the plot method, setting "kind" equal to "bar." Finally, we call plt-dot-show. To add a title to our plot, we can use the title argument of the plot method. It looks like Saint Bernards are the heaviest breed on average! Woof!

## 6. Line plots

Line plots are great for visualizing changes in numeric variables over time. Lucky for us, a Labrador named Sully has been weighed by his owner every month - let's see how his weight has changed over the year. We can use the plot method again, but this time, we pass in three arguments: date as x, weight as y, and "kind" equals "line." Sully's weight has fluctuated quite a bit over the year!

## 7. Rotating axis labels

We may want to rotate the x-axis labels to make the text easier to read. This can be done by passing an angle in degrees with the "rot" argument. Here, we rotate the labels by 45 degrees.

## 8. Scatter plots

Scatter plots are great for visualizing relationships between two numeric variables. To plot each dog's height versus their weight, we call the plot method with x equal to height\_cm, y equal to weight\_kg, and "kind" equal to "scatter." From our plot, it looks like taller dogs tend to weigh more.

## 9. Layering plots

Plots can also be layered on top of one another. For example, we can create a histogram of female dogs' heights, and put a histogram of male dogs' heights on top, then call show. However, we can't tell which color represents which sex.

## 10. Add a legend

We can use plt-dot-legend, passing in a list of labels, and then call show. Now we know which color is which, but we can't see what's going on behind the orange histogram.

## 11. Transparency

Let's fix this problem by making the histograms translucent. We can use hist's alpha argument, which takes a number. 0 means completely transparent that is, invisible, and 1 means completely opaque.

## 12. Avocados

In this chapter, you'll be working with a dataset that contains weekly US avocado sales data, broken down by avocado size, and whether or not the avocados were organic.

## 13. Let's practice!

Prepare to practice your pandas plotting!

**Daily XP150**

##### Exercise

##### Exercise

# Which avocado size is most popular?

Avocados are increasingly popular and delicious in guacamole and on toast. The Hass Avocado Board keeps track of avocado supply and demand across the USA, including the sales of three different sizes of avocado. In this exercise, you'll use a bar plot to figure out which size is the most popular.

Bar plots are great for revealing relationships between categorical (size) and numeric (number sold) variables, but you'll often have to manipulate your data first in order to get the numbers you need for plotting.

pandas has been imported as pd, and avocados is available.

##### Instructions

**100 XP**

* Print the head of the avocados dataset. What columns are available?
* For each avocado size group, calculate the total number sold, storing as nb\_sold\_by\_size.
* Create a bar plot of the number of avocados sold by size.
* Show the plot.
* # Import matplotlib.pyplot with alias plt
* import matplotlib.pyplot as plt
* # Look at the first few rows of data
* print(\_\_\_\_)
* # Get the total number of avocados sold of each size
* nb\_sold\_by\_size = \_\_\_\_
* # Create a bar plot of the number of avocados sold by size
* \_\_\_\_
* # Show the plot
* \_\_\_\_

# Get the worldwide mean temp by year mean\_temp\_by\_year = temp\_by\_country\_city\_vs\_year.mean(axis="index") print(mean\_temp\_by\_year) # Filter for the year that had the highest mean temp print(mean\_temp\_by\_year[mean\_temp\_by\_year == mean\_temp\_by\_year.max()]) # Get the mean temp by city mean\_temp\_by\_city = temp\_by\_country\_city\_vs\_year.mean(axis="columns") print(mean\_temp\_by\_city) # Filter for the city that had the lowest mean temp print(mean\_temp\_by\_city[mean\_temp\_by\_city == mean\_temp\_by\_city.min()])

# Import matplotlib.pyplot with alias plt

import matplotlib.pyplot as plt

# Look at the first few rows of data

print(avocados.head())

# Get the total number of avocados sold of each size

nb\_sold\_by\_size = avocados.groupby("size")["nb\_sold"].sum()

#print(nb\_sold\_by\_size)

# Create a bar plot of the number of avocados sold by size

nb\_sold\_by\_size.plot(kind='bar')

# Show the plot

plt.show()

# Import matplotlib.pyplot with alias plt

import matplotlib.pyplot as plt

# Look at the first few rows of data

print(avocados.head())

# Get the total number of avocados sold of each size

nb\_sold\_by\_size = avocados.groupby("size")["nb\_sold"].sum()

print(nb\_sold\_by\_size)

# Create a bar plot of the number of avocados sold by size

# Show the plot

#plt.show()

date type year avg\_price size nb\_sold

0 2015-12-27 conventional 2015 0.95 small 9.627e+06

1 2015-12-20 conventional 2015 0.98 small 8.710e+06

2 2015-12-13 conventional 2015 0.93 small 9.855e+06

3 2015-12-06 conventional 2015 0.89 small 9.405e+06

4 2015-11-29 conventional 2015 0.99 small 8.095e+06

size

extra\_large 1.562e+08

large 2.015e+09

small 2.055e+09

Name: nb\_sold, dtype: float64

# Import matplotlib.pyplot with alias plt

import matplotlib.pyplot as plt

# Look at the first few rows of data

print(avocados.head())

# Get the total number of avocados sold of each size

nb\_sold\_by\_size = avocados.groupby("size")["nb\_sold"].sum()

#print(nb\_sold\_by\_size)

# Create a bar plot of the number of avocados sold by size

nb\_sold\_by\_size.plot(kind='bar')

# Show the plot

plt.show()

date type year avg\_price size nb\_sold

0 2015-12-27 conventional 2015 0.95 small 9.627e+06

1 2015-12-20 conventional 2015 0.98 small 8.710e+06

2 2015-12-13 conventional 2015 0.93 small 9.855e+06

3 2015-12-06 conventional 2015 0.89 small 9.405e+06

4 2015-11-29 conventional 2015 0.99 small 8.095e+06

**Daily XP250**

##### Exercise

##### Exercise

# Changes in sales over time

Line plots are designed to visualize the relationship between two numeric variables, where each data values is connected to the next one. They are especially useful for visualizing the change in a number over time since each time point is naturally connected to the next time point. In this exercise, you'll visualize the change in avocado sales over three years.

pandas has been imported as pd, and avocados is available.

##### Instructions

**100 XP**

* Get the total number of avocados sold on each date. The DataFrame has two rows for each date—one for organic, and one for conventional. Save this as nb\_sold\_by\_date.
* Create a line plot of the number of avocados sold.
* Show the plot.
* # Import matplotlib.pyplot with alias plt
* import matplotlib.pyplot as plt
* # Get the total number of avocados sold on each date
* nb\_sold\_by\_date = \_\_\_\_
* # Create a line plot of the number of avocados sold by date
* \_\_\_\_
* # Show the plot
* \_\_\_\_

# Import matplotlib.pyplot with alias plt import matplotlib.pyplot as plt # Look at the first few rows of data print(avocados.head()) # Get the total number of avocados sold of each size nb\_sold\_by\_size = avocados.groupby("size")["nb\_sold"].sum() #print(nb\_sold\_by\_size) # Create a bar plot of the number of avocados sold by size nb\_sold\_by\_size.plot(kind='bar') # Show the plot plt.show()

# Import matplotlib.pyplot with alias plt

import matplotlib.pyplot as plt

#print(avocados)

# Get the total number of avocados sold on each date

nb\_sold\_by\_date = avocados.groupby("date")["nb\_sold"].sum()

print(nb\_sold\_by\_date)

# Create a line plot of the number of avocados sold by date

nb\_sold\_by\_date.plot(x= "date", y="nb\_sold", kind= "line", rot=45)

# Show the plot

plt.show()

# Import matplotlib.pyplot with alias plt

import matplotlib.pyplot as plt

#print(avocados)

# Get the total number of avocados sold on each date

nb\_sold\_by\_date = avocados.groupby("date")["nb\_sold"].sum()

print(nb\_sold\_by\_date)

# Create a line plot of the number of avocados sold by date

nb\_sold\_by\_date.plot(x= "date", y="nb\_sold", kind= "line", rot=45)

# Show the plot

plt.show()

date

2015-01-04 2.728e+07

2015-01-11 2.508e+07

2015-01-18 2.496e+07

2015-01-25 2.409e+07

2015-02-01 3.984e+07

...

2018-02-25 2.543e+07

2018-03-04 2.683e+07

2018-03-11 2.609e+07

2018-03-18 2.603e+07

2018-03-25 2.748e+07

Name: nb\_sold, Length: 169, dtype: float64

<script.py> output:

date

2015-01-04 2.728e+07

2015-01-11 2.508e+07

2015-01-18 2.496e+07

2015-01-25 2.409e+07

2015-02-01 3.984e+07

...

2018-02-25 2.543e+07

2018-03-04 2.683e+07

2018-03-11 2.609e+07

2018-03-18 2.603e+07

2018-03-25 2.748e+07

Name: nb\_sold, Length: 169, dtype: float64

# Avocado supply and demand

Scatter plots are ideal for visualizing relationships between numerical variables. In this exercise, you'll compare the number of avocados sold to average price and see if they're at all related. If they're related, you may be able to use one number to predict the other.

matplotlib.pyplot has been imported as plt, pandas has been imported as pd, and avocados is available.

##### Instructions

**100 XP**

* Create a scatter plot with nb\_sold on the x-axis and avg\_price on the y-axis. Title it "Number of avocados sold vs. average price".
* Show the plot.
* # Scatter plot of avg\_price vs. nb\_sold with title
* \_\_\_\_.\_\_\_\_
* # Show the plot
* \_\_\_\_

# Import matplotlib.pyplot with alias plt import matplotlib.pyplot as plt #print(avocados) # Get the total number of avocados sold on each date nb\_sold\_by\_date = avocados.groupby("date")["nb\_sold"].sum() print(nb\_sold\_by\_date) # Create a line plot of the number of avocados sold by date nb\_sold\_by\_date.plot(x= "date", y="nb\_sold", kind= "line", rot=45) # Show the plot plt.show()

# Scatter plot of avg\_price vs. nb\_sold with title

avocados.plot(x= "nb\_sold", y= "avg\_price", kind="scatter", title="Number of avocados sold vs. average price")

# Show the plot

plt.show()

# Show the plot

plt.show()

# Scatter plot of avg\_price vs. nb\_sold with title

avocados.plot(x= "nb\_sold", y= "avg\_price", kind="scatter", title="Number of avocados sold vs. average\_price")

# Show the plot

plt.show()

# Scatter plot of avg\_price vs. nb\_sold with title

avocados.plot(x= "nb\_sold", y= "avg\_price", kind="scatter", title="Number of avocados sold vs. average price")

# Show the plot

plt.show()

**Daily XP100**

##### Exercise

##### Exercise

# Price of conventional vs. organic avocados

Creating multiple plots for different subsets of data allows you to compare groups. In this exercise, you'll create multiple histograms to compare the prices of conventional and organic avocados.

matplotlib.pyplot has been imported as plt and pandas has been imported as pd.

##### Instructions 1/3

**35 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* [3](javascript:void(0))
* Subset avocados for the conventional type, and the average price column. Create a histogram.
* Create a histogram of avg\_price for organic type avocados.
* Add a legend to your plot, with the names "conventional" and "organic".
* Show your plot.
* # Histogram of conventional avg\_price
* avocados[\_\_\_\_][\_\_\_\_].\_\_\_\_
* # Histogram of organic avg\_price
* avocados[\_\_\_\_][\_\_\_\_].\_\_\_\_
* # Add a legend
* plt.legend(\_\_\_\_)
* # Show the plot
* \_\_\_\_

# Scatter plot of avg\_price vs. nb\_sold with title avocados.plot(x= "nb\_sold", y= "avg\_price", kind="scatter", title="Number of avocados sold vs. average price") # Show the plot plt.show()

# Modify histogram transparency to 0.5

avocados[avocados["type"] == "conventional"]["avg\_price"].hist()

# Modify histogram transparency to 0.5

avocados[avocados["type"] == "organic"]["avg\_price"].hist()

# Add a legend

plt.legend(["conventional", "organic"])

# Show the plot

plt.show()

# Histogram of conventional avg\_price

avocados[avocados["type"]=="conventional"]["avg\_price"].hist()

# Histogram of organic avg\_price

avocados[avocados["type"]=="organic"].hist()

# Add a legend

plt.legend(["conventional", "organic"])

# Show the plot

plt.show()

# Histogram of conventional avg\_price

avocados[avocados["type"]=="conventional"]["avg\_price"].hist()

# Histogram of organic avg\_price

avocados[avocados["type"]=="organic"]["avg\_price"].hist()

# Add a legend

plt.legend(["conventional", "organic"])

# Show the plot

plt.show()

**Daily XP135**

##### Exercise

##### Exercise

# Price of conventional vs. organic avocados

Creating multiple plots for different subsets of data allows you to compare groups. In this exercise, you'll create multiple histograms to compare the prices of conventional and organic avocados.

matplotlib.pyplot has been imported as plt and pandas has been imported as pd.

##### Instructions 2/3

**35 XP**

* [2](javascript:void(0))
* [3](javascript:void(0))
* Modify your code to adjust the transparency of both histograms to 0.5 to see how much overlap there is between the two distributions.

# Price of conventional vs. organic avocados

Creating multiple plots for different subsets of data allows you to compare groups. In this exercise, you'll create multiple histograms to compare the prices of conventional and organic avocados.

matplotlib.pyplot has been imported as plt and pandas has been imported as pd.

##### Instructions 3/3

**30 XP**

* [3](javascript:void(0))
* Modify your code to use 20 bins in both histograms.
* Modify bins to 20
* avocados[avocados["type"] == "conventional"]["avg\_price"].hist(alpha=0.5)
* # Modify bins to 20
* avocados[avocados["type"] == "organic"]["avg\_price"].hist(alpha=0.5)
* # Add a legend
* plt.legend(["conventional", "organic"])
* # Show the plot
* plt.show()

# Histogram of conventional avg\_price

avocados[avocados["type"]=="conventional"]["avg\_price"].hist()

# Histogram of organic avg\_price

avocados[avocados["type"]=="organic"].hist()

# Add a legend

plt.legend(["conventional", "organic"])

# Show the plot

plt.show()

# Histogram of conventional avg\_price

avocados[avocados["type"]=="conventional"]["avg\_price"].hist()

# Histogram of organic avg\_price

avocados[avocados["type"]=="organic"]["avg\_price"].hist()

# Add a legend

plt.legend(["conventional", "organic"])

# Show the plot

plt.show()

# Modify histogram transparency to 0.5

avocados[avocados["type"] == "conventional"]["avg\_price"].hist(alpha=0.5)

# Modify histogram transparency to 0.5

avocados[avocados["type"] == "organic"]["avg\_price"].hist(alpha=0.5)

# Add a legend

plt.legend(["conventional", "organic"])

# Show the plot

plt.show()

# Modify bins to 20

avocados[avocados["type"] == "conventional"]["avg\_price"].hist(alpha=0.5, bins=20)

# Modify bins to 20

avocados[avocados["type"] == "organic"]["avg\_price"].hist(alpha=0.5, bins=20)

# Add a legend

plt.legend(["conventional", "organic"])

# Show the plot

plt.show()

# Histogram of conventional avg\_price

avocados[avocados["type"]=="conventional"]["avg\_price"].hist()

# Histogram of organic avg\_price

avocados[avocados["type"]=="organic"].hist()

# Add a legend

#plt.legend(\_\_\_\_)

# Show the plot

#plt.show()

array([[<AxesSubplot:title={'center':'year'}>,

<AxesSubplot:title={'center':'avg\_price'}>],

[<AxesSubplot:title={'center':'nb\_sold'}>, <AxesSubplot:>]],

dtype=object)

# Histogram of conventional avg\_price

avocados[avocados["type"]=="conventional"]["avg\_price"].hist()

# Histogram of organic avg\_price

avocados[avocados["type"]=="organic"].hist()

# Add a legend

plt.legend(["conventional", "organic"])

# Show the plot

plt.show()

# Histogram of conventional avg\_price

avocados[avocados["type"]=="conventional"]["avg\_price"].hist()

# Histogram of organic avg\_price

avocados[avocados["type"]=="organic"]["avg\_price"].hist()

# Add a legend

plt.legend(["conventional", "organic"])

# Show the plot

plt.show()

# Modify histogram transparency to 0.5

avocados[avocados["type"] == "conventional"]["avg\_price"].hist(alpha=0.5)

# Modify histogram transparency to 0.5

avocados[avocados["type"] == "organic"]["avg\_price"].hist(alpha=0.5)

# Add a legend

plt.legend(["conventional", "organic"])

# Show the plot

plt.show()

# Modify bins to 20

avocados[avocados["type"] == "conventional"]["avg\_price"].hist(alpha=0.5,bins=20)

# Modify bins to 20

avocados[avocados["type"] == "organic"]["avg\_price"].hist(alpha=0.5, bins=20)

# Add a legend

plt.legend(["conventional", "organic"])

# Show the plot

plt.show()

# Modify bins to 20

avocados[avocados["type"] == "conventional"]["avg\_price"].hist(bins=20)

# Modify bins to 20

avocados[avocados["type"] == "organic"]["avg\_price"].hist(bins=20)

# Add a legend

plt.legend(["conventional", "organic"])

# Show the plot

plt.show()

# Modify bins to 20

avocados[avocados["type"] == "conventional"]["avg\_price"].hist(alpha=0.5, bins=20)

# Modify bins to 20

avocados[avocados["type"] == "organic"]["avg\_price"].hist(alpha=0.5, bins=20)

# Add a legend

plt.legend(["conventional", "organic"])

# Show the plot

plt.show()

Ctrl+O

**Daily XP200**

# Missing values

**50 XP**

## 1. Missing values

You could be given a DataFrame that has missing values, so it's important to know how to handle them.

## 2. What's a missing value?

Most data is not perfect - there's always a possibility that there are some pieces missing from your dataset. For example, maybe on the day that Bella and Cooper's owner weighed them,

## 3. What's a missing value?

the scale was broken. Now we have two missing values in our dataset.

## 4. Missing values in pandas DataFrames

In a pandas DataFrame, missing values are indicated with N-a-N, which stands for "not a number."

## 5. Detecting missing values

When you first get a DataFrame, it's a good idea to get a sense of whether it contains any missing values, and if so, how many. That's where the isna method comes in. When we call isna on a DataFrame, we get a Boolean for every single value indicating whether the value is missing or not, but this isn't very helpful when you're working with a lot of data.

## 6. Detecting any missing values

If we chain dot-isna with dot-any, we get one value for each variable that tells us if there are any missing values in that column. Here, we see that there's at least one missing value in the weight column, but not in any of the others.

## 7. Counting missing values

Since taking the sum of Booleans is the same thing as counting the number of Trues, we can combine sum with isna to count the number of NaNs in each column.

## 8. Plotting missing values

We can use those counts to visualize the missing values in the dataset using a bar plot. Plots like this are more interesting when you have missing data across different variables, while here, only weights are missing. Now that we know there are missing values in the dataset, what can we do about them?

## 9. Removing missing values

One option is to remove the rows in the DataFrame that contain missing values. This can be done using the dropna method. However, this may not be ideal if you have a lot of missing data, since that means losing a lot of observations.

## 10. Replacing missing values

Another option is to replace missing values with another value. The fillna method takes in a value, and all NaNs will be replaced with this value. There are also many sophisticated techniques for replacing missing values, which you can learn more about in our course about missing data.

## 11. Let's practice!

Alright, time to wrangle with some missing values on your own!

**Daily XP250**

##### Exercise

##### Exercise

# Finding missing values

Missing values are everywhere, and you don't want them interfering with your work. Some functions ignore missing data by default, but that's not always the behavior you might want. Some functions can't handle missing values at all, so these values need to be taken care of before you can use them. If you don't know where your missing values are, or if they exist, you could make mistakes in your analysis. In this exercise, you'll determine if there are missing values in the dataset, and if so, how many.

pandas has been imported as pd and avocados\_2016, a subset of avocados that contains only sales from 2016, is available.

##### Instructions

**100 XP**

* Print a DataFrame that shows whether each value in avocados\_2016 is missing or not.
* Print a summary that shows whether any value in each column is missing or not.
* Create a bar plot of the total number of missing values in each column.
* # Import matplotlib.pyplot with alias plt
* import matplotlib.pyplot as plt
* # Check individual values for missing values
* print(\_\_\_\_)
* # Check each column for missing values
* print(\_\_\_\_)
* # Bar plot of missing values by variable
* \_\_\_\_
* # Show plot
* plt.show()

# Modify bins to 20 avocados[avocados["type"] == "conventional"]["avg\_price"].hist(alpha=0.5, bins=20) # Modify bins to 20 avocados[avocados["type"] == "organic"]["avg\_price"].hist(alpha=0.5, bins=20) # Add a legend plt.legend(["conventional", "organic"]) # Show the plot plt.show()

# Import matplotlib.pyplot with alias plt

import matplotlib.pyplot as plt

# Check individual values for missing values

print(avocados\_2016.isna())

# Check each column for missing values

print(avocados\_2016.isna().any())

# Bar plot of missing values by variable

avocados\_2016.isna().sum().plot(kind="bar")

# Show plot

plt.show()

<script.py> output: date avg\_price total\_sold small\_sold large\_sold xl\_sold total\_bags\_sold small\_bags\_sold large\_bags\_sold xl\_bags\_sold 0 False False False False False False False False False False 1 False False False False False False False False False False 2 False False False False True False False False False False 3 False False False False False False False False False False 4 False False False False False True False False False False 5 False False False True False False False False False False 6 False False False False False False False False False False 7 False False False False True False False False False False 8 False False False False False False False False False False 9 False False False False False False False False False False 10 False False False False True False False False False False 11 False False False False False False False False False False 12 False False False False False False False False False False 13 False False False False False False False False False False 14 False False False False False False False False False False 15 False False False False True False False False False False 16 False False False False False True False False False False 17 False False False False False False False False False False 18 False False False False False False False False False False 19 False False False False True False False False False False 20 False False False False False False False False False False 21 False False False False False False False False False False 22 False False False False False False False False False False 23 False False False False False False False False False False 24 False False False False False False False False False False 25 False False False False False False False False False False 26 False False False False False False False False False False 27 False False False False False False False False False False 28 False False False False False False False False False False 29 False False False False False False False False False False 30 False False False False False True False False False False 31 False False False False False False False False False False 32 False False False False False True False False False False 33 False False False False False False False False False False 34 False False False False False False False False False False 35 False False False False False False False False False False 36 False False False True False False False False False False 37 False False False False True False False False False False 38 False False False False False False False False False False 39 False False False False False False False False False False 40 False False False True False False False False False False 41 False False False False False False False False False False 42 False False False False False False False False False False 43 False False False False False False False False False False 44 False False False True False False False False False False 45 False False False False False False False False False False 46 False False False False False False False False False False 47 False False False False False False False False False False 48 False False False False False False False False False False 49 False False False False False False False False False False 50 False False False True False False False False False False 51 False False False True False False False False False False date False avg\_price False total\_sold False small\_sold True large\_sold True xl\_sold True total\_bags\_sold False small\_bags\_sold False large\_bags\_sold False xl\_bags\_sold False dtype: bool

**Daily XP350**

##### Exercise

##### Exercise

# Removing missing values

Now that you know there are some missing values in your DataFrame, you have a few options to deal with them. One way is to remove them from the dataset completely. In this exercise, you'll remove missing values by removing all rows that contain missing values.

pandas has been imported as pd and avocados\_2016 is available.

##### Instructions

**100 XP**

* Remove the rows of avocados\_2016 that contain missing values and store the remaining rows in avocados\_complete.
* Verify that all missing values have been removed from avocados\_complete. Calculate each column that has NAs and print.
* # Remove rows with missing values
* avocados\_complete = \_\_\_\_
* # Check if any columns contain missing values
* print(\_\_\_\_)

# Import matplotlib.pyplot with alias plt import matplotlib.pyplot as plt # Check individual values for missing values print(avocados\_2016.isna()) # Check each column for missing values print(avocados\_2016.isna().any()) # Bar plot of missing values by variable avocados\_2016.isna().sum().plot(kind="bar") # Show plot plt.show()

# Remove rows with missing values

avocados\_complete = avocados\_2016.dropna()

# Check if any columns contain missing values

print(avocados\_complete.isna().any().sum())

# Remove rows with missing values

avocados\_complete = avocados\_2016.dropna()

# Check if any columns contain missing values

print(avocados\_complete.isna().any())

date False

avg\_price False

total\_sold False

small\_sold False

large\_sold False

xl\_sold False

total\_bags\_sold False

small\_bags\_sold False

large\_bags\_sold False

xl\_bags\_sold False

dtype: bool

# Remove rows with missing values

avocados\_complete = avocados\_2016.dropna()

# Check if any columns contain missing values

print(avocados\_complete.isna().any().sum())

0

<script.py> output:

0

**Daily XP450**

##### Exercise

##### Exercise

# Replacing missing values

Another way of handling missing values is to replace them all with the same value. For numerical variables, one option is to replace values with 0— you'll do this here. However, when you replace missing values, you make assumptions about what a missing value means. In this case, you will assume that a missing number sold means that no sales for that avocado type were made that week.

In this exercise, you'll see how replacing missing values can affect the distribution of a variable using histograms. You can plot histograms for multiple variables at a time as follows:

dogs[["height\_cm", "weight\_kg"]].hist()

pandas has been imported as pd and matplotlib.pyplot has been imported as plt. The avocados\_2016 dataset is available.

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* A list has been created, cols\_with\_missing, containing the names of columns with missing values: "small\_sold", "large\_sold", and "xl\_sold".
* Create a histogram of those columns.
* Show the plot.
* # List the columns with missing values
* cols\_with\_missing = ["small\_sold", "large\_sold", "xl\_sold"]
* # Create histograms showing the distributions cols\_with\_missing
* avocados\_2016[\_\_\_\_].\_\_\_\_
* # Show the plot
* \_\_\_\_

# Remove rows with missing values avocados\_complete = avocados\_2016.dropna() # Check if any columns contain missing values print(avocados\_complete.isna().any().sum())

# List the columns with missing values

cols\_with\_missing = ["small\_sold", "large\_sold", "xl\_sold"]

# Create histograms showing the distributions cols\_with\_missing

avocados\_2016[cols\_with\_missing].hist()

# Show the plot

plt.show()

# List the columns with missing values

cols\_with\_missing = ["small\_sold", "large\_sold", "xl\_sold"]

# Create histograms showing the distributions cols\_with\_missing

avocados\_2016[["small\_sold", "large\_sold", "xl\_sold"]].hist()

# Show the plot

plt.show()

# List the columns with missing values

cols\_with\_missing = ["small\_sold", "large\_sold", "xl\_sold"]

# Create histograms showing the distributions cols\_with\_missing

avocados\_2016[cols\_with\_missing].hist()

# Show the plot

plt.show()

# From previous step

cols\_with\_missing = ["small\_sold", "large\_sold", "xl\_sold"]

avocados\_2016[cols\_with\_missing].hist()

plt.show()

# Fill in missing values with 0

avocados\_filled = avocados\_2016[cols\_with\_missing].fillna(0)

#print(avocados\_filled)

# Create histograms of the filled columns

avocados\_filled[cols\_with\_missing].hist()

# Show the plot

plt.show()

# Fill in missing values with 0

avocados\_filled = avocados\_2016[cols\_with\_missing].fillna(0)

print(avocados\_filled)

# Create histograms of the filled columns

# Show the plot

#plt.show()

small\_sold large\_sold xl\_sold

0 9.255e+06 1.028e+07 5.420e+05

1 9.394e+06 1.034e+07 4.279e+05

2 9.010e+06 0.000e+00 4.030e+05

3 1.104e+07 9.909e+06 4.280e+05

4 7.891e+06 7.337e+06 0.000e+00

5 0.000e+00 8.034e+06 4.076e+05

6 8.235e+06 7.760e+06 4.775e+05

7 7.804e+06 0.000e+00 5.053e+05

8 7.100e+06 6.852e+06 4.531e+05

9 7.580e+06 8.105e+06 4.524e+05

10 8.811e+06 0.000e+00 4.473e+05

11 9.553e+06 9.579e+06 4.910e+05

12 9.535e+06 9.124e+06 4.901e+05

13 1.055e+07 9.256e+06 4.926e+05

14 1.134e+07 9.481e+06 4.862e+05

15 1.296e+07 0.000e+00 5.122e+05

16 1.464e+07 1.108e+07 0.000e+00

17 1.256e+07 1.042e+07 5.637e+05

18 1.273e+07 9.906e+06 6.097e+05

19 1.235e+07 0.000e+00 7.492e+05

20 1.220e+07 1.073e+07 7.545e+05

21 1.017e+07 1.042e+07 7.839e+05

22 1.033e+07 1.080e+07 9.480e+05

23 1.032e+07 1.063e+07 9.536e+05

24 1.146e+07 1.168e+07 1.266e+06

25 1.251e+07 1.332e+07 1.761e+06

26 1.219e+07 1.203e+07 1.625e+06

27 1.349e+07 1.284e+07 1.643e+06

28 1.229e+07 1.153e+07 1.607e+06

29 1.381e+07 1.340e+07 1.771e+06

30 1.368e+07 1.383e+07 0.000e+00

31 1.171e+07 1.298e+07 1.641e+06

32 1.245e+07 1.444e+07 0.000e+00

33 1.422e+07 1.790e+07 1.994e+06

34 1.375e+07 1.590e+07 1.414e+06

35 1.291e+07 1.438e+07 1.600e+06

36 0.000e+00 1.290e+07 1.896e+06

37 1.288e+07 0.000e+00 1.666e+06

38 1.132e+07 1.151e+07 1.732e+06

39 1.125e+07 1.275e+07 1.880e+06

40 0.000e+00 1.373e+07 1.773e+06

41 1.166e+07 1.310e+07 1.800e+06

42 1.182e+07 1.220e+07 1.686e+06

43 1.191e+07 1.321e+07 1.560e+06

44 0.000e+00 1.200e+07 1.375e+06

45 1.191e+07 1.319e+07 1.811e+06

46 1.657e+07 2.047e+07 2.546e+06

47 1.110e+07 1.328e+07 1.644e+06

48 7.020e+06 1.405e+07 1.440e+06

49 1.004e+07 1.383e+07 1.419e+06

50 0.000e+00 1.323e+07 1.283e+06

51 0.000e+00 1.605e+07 1.560e+06

# From previous step

cols\_with\_missing = ["small\_sold", "large\_sold", "xl\_sold"]

avocados\_2016[cols\_with\_missing].hist()

plt.show()

# Fill in missing values with 0

avocados\_filled = avocados\_2016[cols\_with\_missing].fillna(0)

#print(avocados\_filled)

# Create histograms of the filled columns

avocados\_filled[cols\_with\_missing].hist()

# Show the plot

#plt.show()

array([[<AxesSubplot:title={'center':'small\_sold'}>,

<AxesSubplot:title={'center':'large\_sold'}>],

[<AxesSubplot:title={'center':'xl\_sold'}>, <AxesSubplot:>]],

dtype=object)

**Daily XP550**

# Creating DataFrames

**50 XP**

## 1. Creating DataFrames

Now that you've learned a lot about how to work with pandas DataFrames, how do you get data into a DataFrame in the first place?

## 2. Dictionaries

Before creating your own DataFrames, let's talk about dictionaries. A dictionary is a way of storing data in Python. It holds a set of key-value pairs. You can create a dictionary like this, using curly braces. Inside, each key-value pair is written as "key colon value." Let's create a dictionary that holds information about a book. "Title" is a key in the dictionary, and "Charlotte's Web" is its corresponding value, and so on. You can access values of a dictionary via their keys in square brackets. For example, we can access the value of "title" like this.

## 3. Creating DataFrames

There are many ways to create DataFrames from scratch, but we'll discuss two ways: from a list of dictionaries and from a dictionary of lists. In the first method, the DataFrame is built up row by row, while in the second method, the DataFrame is built up column by column.

## 4. List of dictionaries - by row

We have some new dog data to put into a DataFrame. Let's start with the first method to do this, creating a list of dictionaries. First, we'll create a new list using square brackets to hold our dictionaries. Then, we'll go through the first row of our data and put it in a dictionary. Each key, on the left of each colon, will become a column name. Each value is one dog's data for that column. Here, the first key is "name," which is the first column name, and its corresponding value is "Ginger," the name of the first dog. The second key is the second column name, "breed," and its value is "Dachshund," which is the first dog's breed. Then we have the dog's height and weight. For the next row, we create another dictionary that follows the same format.

## 5. List of dictionaries - by row

Now that we have our list of dictionaries, we can pass it into pd-dot-DataFrame to convert it into DataFrame form.

## 6. Dictionary of lists - by column

Now let's talk about the dictionary of lists method. When using this method, we need to go through the data column by column. Remember that keys are to the left of a colon, and values are to the right. Each key will be a column name, and each value will be a list of the values in the column. First, we'll create a dictionary using curly braces. Let's start with the first column, which is called "name," so the first key is "name." The value is a list containing each name, from top to bottom. In this case, it's "Ginger" and "Scout." Next, we have the "breed" column, so we add "breed" as a key, and its corresponding value is a list containing "Dachshund" and "Dalmatian." Then we have height\_cm, which is 22 and 59, and weight\_kg, which is 10 and 25. Now that we have our dictionary of lists set up, we can pass it into pd-dot-DataFrame to convert it into a pandas DataFrame.

## 7. Dictionary of lists - by column

If we print the new DataFrame, we can see that it's exactly what we wanted.

## 8. Let's practice!

Time to practice creating your own DataFrames!

**Daily XP50**

##### Exercise

##### Exercise

# List of dictionaries

You recently got some new avocado data from 2019 that you'd like to put in a DataFrame using the list of dictionaries method. Remember that with this method, you go through the data row by row.

| **date** | **small\_sold** | **large\_sold** |
| --- | --- | --- |
| "2019-11-03" | 10376832 | 7835071 |
| "2019-11-10" | 10717154 | 8561348 |

pandas as pd is imported.

##### Instructions

**100 XP**

* Create a list of dictionaries with the new data called avocados\_list.
* Convert the list into a DataFrame called avocados\_2019.
* Print your new DataFrame.
* # Create a list of dictionaries with new data
* avocados\_list = [
* {\_\_\_\_: \_\_\_\_, \_\_\_\_: \_\_\_\_, \_\_\_\_: \_\_\_\_},
* {\_\_\_\_: \_\_\_\_, \_\_\_\_: \_\_\_\_, \_\_\_\_: \_\_\_\_},
* ]
* # Convert list into DataFrame
* avocados\_2019 = \_\_\_\_
* # Print the new DataFrame
* \_\_\_\_

# From previous step cols\_with\_missing = ["small\_sold", "large\_sold", "xl\_sold"] avocados\_2016[cols\_with\_missing].hist() plt.show() # Fill in missing values with 0 avocados\_filled = avocados\_2016[cols\_with\_missing].fillna(0) #print(avocados\_filled) # Create histograms of the filled columns avocados\_filled[cols\_with\_missing].hist() # Show the plot #plt.show()

# Create a list of dictionaries with new data

avocados\_list = [

    {"date":"2019-11-03" , "small\_sold": 10376832, "large\_sold": 7835071},

    {"date":"2019-11-10", "small\_sold": 10717154, "large\_sold": 8561348},

]

# Convert list into DataFrame

avocados\_2019 = pd.DataFrame(avocados\_list)

# Print the new DataFrame

print(avocados\_2019)

# Create a list of dictionaries with new data

avocados\_list = [

{"date":"2019-11-03" , "small\_sold": 10376832, "large\_sold": 7835071},

{"date":"2019-11-10", "small\_sold": 10717154, "large\_sold": 8561348},

]

# Convert list into DataFrame

avocados\_2019 = pd.DataFrame(avocados\_list)

# Print the new DataFrame

print(avocados\_2019)

date small\_sold large\_sold

0 2019-11-03 10376832 7835071

1 2019-11-10 10717154 8561348

<script.py> output:

date small\_sold large\_sold

0 2019-11-03 10376832 7835071

1 2019-11-10 10717154 8561348

**Daily XP150**

##### Exercise

##### Exercise

# Dictionary of lists

Some more data just came in! This time, you'll use the dictionary of lists method, parsing the data column by column.

| **date** | **small\_sold** | **large\_sold** |
| --- | --- | --- |
| "2019-11-17" | 10859987 | 7674135 |
| "2019-12-01" | 9291631 | 6238096 |

pandas as pd is imported.

##### Instructions

**100 XP**

* Create a dictionary of lists with the new data called avocados\_dict.
* Convert the dictionary to a DataFrame called avocados\_2019.
* Print your new DataFrame.
* # Create a dictionary of lists with new data
* avocados\_dict = {
* "\_\_\_\_": [\_\_\_\_],
* "\_\_\_\_": [\_\_\_\_],
* "\_\_\_\_": [\_\_\_\_]
* }
* # Convert dictionary into DataFrame
* avocados\_2019 = \_\_\_\_
* # Print the new DataFrame
* \_\_\_\_

# Create a list of dictionaries with new data avocados\_list = [ {"date":"2019-11-03" , "small\_sold": 10376832, "large\_sold": 7835071}, {"date":"2019-11-10", "small\_sold": 10717154, "large\_sold": 8561348}, ] # Convert list into DataFrame avocados\_2019 = pd.DataFrame(avocados\_list) # Print the new DataFrame print(avocados\_2019)

# Create a dictionary of lists with new data

avocados\_dict = {

  "date": ["2019-11-17", "2019-12-01"],

  "small\_sold": [10859987, 9291631],

  "large\_sold": [7674135, 6238096]

}

# Convert dictionary into DataFrame

avocados\_2019 = pd.DataFrame(avocados\_dict)

# Print the new DataFrame

print(avocados\_2019)

# Create a dictionary of lists with new data

avocados\_dict = {

"date": ["2019-11-17", "2019-12-01"],

"small\_sold": [10859987, 9291631],

"large\_sold": [7674135, 6238096]

}

# Convert dictionary into DataFrame

avocados\_2019 = pd.DataFrame(avocados\_dict)

# Print the new DataFrame

print(avocados\_2019)

date small\_sold large\_sold

0 2019-11-17 10859987 7674135

1 2019-12-01 9291631 6238096

<script.py> output:

date small\_sold large\_sold

0 2019-11-17 10859987 7674135

1 2019-12-01 9291631 6238096

# Create a list of dictionaries with new data

avocados\_list = [

{"date":"2019-11-03" , "small\_sold": 10376832, "large\_sold": 7835071},

{"date":"2019-11-10", "small\_sold": 10717154, "large\_sold": 8561348},

]

# Convert list into DataFrame

avocados\_2019 = pd.DataFrame(avocados\_list)

# Print the new DataFrame

print(avocados\_2019)

# Create a dictionary of lists with new data

avocados\_dict = {

"date": ["2019-11-17", "2019-12-01"],

"small\_sold": [10859987, 9291631],

"large\_sold": [7674135, 6238096]

}

# Convert dictionary into DataFrame

avocados\_2019 = pd.DataFrame(avocados\_dict)

# Print the new DataFrame

print(avocados\_2019)

date small\_sold large\_sold

0 2019-11-17 10859987 7674135

1 2019-12-01 9291631 6238096

<script.py> output:

date small\_sold large\_sold

0 2019-11-17 10859987 7674135

1 2019-12-01 9291631 6238096

## 1. Reading and writing CSVs

You now know how to create your own DataFrames, but typing out your data entry-by-entry isn't usually the most efficient way to get your data into a DataFrame. In this video, you'll learn how to pull data from CSV files.

## 2. What's a CSV file?

CSV, or comma-separated values, is a common data storage file type. It's designed to store tabular data, just like a pandas DataFrame. It's a text file, where each row of data has its own line, and each value is separated by a comma. Almost every database, programming language, and piece of data analysis software can read and write CSV files. That makes it a good storage format if you need to share your data with other people who may be using different tools than you.

## 3. Example CSV file

Remember the dogs from the last video? Their data is stored in a CSV file called new\_dogs-dot-csv, which looks like this.

## 4. CSV to DataFrame

We can put this data in a DataFrame using the handy pandas function, read-underscore-csv, and pass it the file path of the CSV.

## 5. DataFrame manipulation

Now that the data is in DataFrame form, we can manipulate it using some of the functions from earlier in the course. Here, we'll add a body mass index column.

## 6. DataFrame to CSV

Now that we've changed the data let's create an updated CSV file to share with the dogs' owners. To convert a DataFrame to a CSV, we can use new\_dogs dot to-underscore-csv, and pass in a new file path. If we take a look at the new file, it contains the BMI column.

## 7. Let's practice!

Now it's time to practice getting data in and out of pandas!

**Daily XP300**

##### Exercise

##### Exercise

# CSV to DataFrame

You work for an airline, and your manager has asked you to do a competitive analysis and see how often passengers flying on other airlines are involuntarily bumped from their flights. You got a CSV file (airline\_bumping.csv) from the Department of Transportation containing data on passengers that were involuntarily denied boarding in 2016 and 2017, but it doesn't have the exact numbers you want. In order to figure this out, you'll need to get the CSV into a pandas DataFrame and do some manipulation!

pandas is imported for you as pd. "airline\_bumping.csv" is in your working directory.

##### Instructions 1/4

**25 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* [3](javascript:void(0))
* [4](javascript:void(0))
* Read the CSV file "airline\_bumping.csv" and store it as a DataFrame called airline\_bumping.
* Print the first few rows of airline\_bumping.
* # Read CSV as DataFrame called airline\_bumping
* airline\_bumping = \_\_\_\_
* # Take a look at the DataFrame
* print(\_\_\_\_)

# Create a dictionary of lists with new data avocados\_dict = { "date": ["2019-11-17", "2019-12-01"], "small\_sold": [10859987, 9291631], "large\_sold": [7674135, 6238096] } # Convert dictionary into DataFrame avocados\_2019 = pd.DataFrame(avocados\_dict) # Print the new DataFrame print(avocados\_2019)

# Read CSV as DataFrame called airline\_bumping

airline\_bumping = pd.read\_csv("airline\_bumping.csv")

# Take a look at the DataFrame

print(airline\_bumping.head())

# Read CSV as DataFrame called airline\_bumping

airline\_bumping = pd.read\_csv("airline\_bumping.csv")

# Take a look at the DataFrame

print(airline\_bumping.head())

airline year nb\_bumped total\_passengers

0 DELTA AIR LINES 2017 679 99796155

1 VIRGIN AMERICA 2017 165 6090029

2 JETBLUE AIRWAYS 2017 1475 27255038

3 UNITED AIRLINES 2017 2067 70030765

4 HAWAIIAN AIRLINES 2017 92 8422734

# CSV to DataFrame

You work for an airline, and your manager has asked you to do a competitive analysis and see how often passengers flying on other airlines are involuntarily bumped from their flights. You got a CSV file (airline\_bumping.csv) from the Department of Transportation containing data on passengers that were involuntarily denied boarding in 2016 and 2017, but it doesn't have the exact numbers you want. In order to figure this out, you'll need to get the CSV into a pandas DataFrame and do some manipulation!

pandas is imported for you as pd. "airline\_bumping.csv" is in your working directory.

##### Instructions 2/4

**25 XP**

* [2](javascript:void(0))
* [3](javascript:void(0))
* [4](javascript:void(0))
* For each airline group, select the nb\_bumped, and total\_passengers columns, and calculate the sum (for both years). Store this as airline\_totals.
* # From previous step
* airline\_bumping = pd.read\_csv("airline\_bumping.csv")
* print(airline\_bumping.head())
* # For each airline, select nb\_bumped and total\_passengers and sum
* airline\_totals = airline\_bumping.groupby()[[\_\_\_\_]].\_\_\_\_

# Read CSV as DataFrame called airline\_bumping

airline\_bumping = pd.read\_csv("airline\_bumping.csv")

# Take a look at the DataFrame

print(airline\_bumping.head())

airline year nb\_bumped total\_passengers

0 DELTA AIR LINES 2017 679 99796155

1 VIRGIN AMERICA 2017 165 6090029

2 JETBLUE AIRWAYS 2017 1475 27255038

3 UNITED AIRLINES 2017 2067 70030765

4 HAWAIIAN AIRLINES 2017 92 8422734

<script.py> output:

airline year nb\_bumped total\_passengers

0 DELTA AIR LINES 2017 679 99796155

1 VIRGIN AMERICA 2017 165 6090029

2 JETBLUE AIRWAYS 2017 1475 27255038

3 UNITED AIRLINES 2017 2067 70030765

4 HAWAIIAN AIRLINES 2017 92 8422734

# From previous step

airline\_bumping = pd.read\_csv("airline\_bumping.csv")

print(airline\_bumping.head())

# For each airline, select nb\_bumped and total\_passengers and sum

airline\_totals = airline\_bumping.groupby("airline")[["nb\_bumped", "total\_passengers"]].sum()

airline year nb\_bumped total\_passengers

0 DELTA AIR LINES 2017 679 99796155

1 VIRGIN AMERICA 2017 165 6090029

2 JETBLUE AIRWAYS 2017 1475 27255038

3 UNITED AIRLINES 2017 2067 70030765

4 HAWAIIAN AIRLINES 2017 92 8422734

<script.py> output:

airline year nb\_bumped total\_passengers

0 DELTA AIR LINES 2017 679 99796155

1 VIRGIN AMERICA 2017 165 6090029

2 JETBLUE AIRWAYS 2017 1475 27255038

3 UNITED AIRLINES 2017 2067 70030765

4 HAWAIIAN AIRLINES 2017 92 8422734

**Daily XP350**

##### Exercise

##### Exercise

# CSV to DataFrame

You work for an airline, and your manager has asked you to do a competitive analysis and see how often passengers flying on other airlines are involuntarily bumped from their flights. You got a CSV file (airline\_bumping.csv) from the Department of Transportation containing data on passengers that were involuntarily denied boarding in 2016 and 2017, but it doesn't have the exact numbers you want. In order to figure this out, you'll need to get the CSV into a pandas DataFrame and do some manipulation!

pandas is imported for you as pd. "airline\_bumping.csv" is in your working directory.

##### Instructions 3/4

**25 XP**

* [3](javascript:void(0))
* [4](javascript:void(0))
* Create a new column of airline\_totals called bumps\_per\_10k, which is the number of passengers bumped per 10,000 passengers in 2016 and 2017.
* # From previous steps
* airline\_bumping = pd.read\_csv("airline\_bumping.csv")
* print(airline\_bumping.head())
* airline\_totals = airline\_bumping.groupby("airline")[["nb\_bumped", "total\_passengers"]].sum()
* # Create new col, bumps\_per\_10k: no. of bumps per 10k passengers for each airline
* airline\_totals["bumps\_per\_10k"] = \_\_\_\_ / \_\_\_\_ \* 10000

# From previous steps

airline\_bumping = pd.read\_csv("airline\_bumping.csv")

print(airline\_bumping.head())

airline\_totals = airline\_bumping.groupby("airline")[["nb\_bumped", "total\_passengers"]].sum()

print(airline\_totals)

# Create new col, bumps\_per\_10k: no. of bumps per 10k passengers for each airline

airline\_totals["bumps\_per\_10k"] = (airline\_totals["nb\_bumped"]/ airline\_totals["total\_passengers"]\* 10000)

# From previous steps

airline\_bumping = pd.read\_csv("airline\_bumping.csv")

print(airline\_bumping.head())

airline\_totals = airline\_bumping.groupby("airline")[["nb\_bumped", "total\_passengers"]].sum()

airline\_totals["bumps\_per\_10k"] = airline\_totals["nb\_bumped"] / airline\_totals["total\_passengers"] \* 10000

# Print airline\_totals

print(airline\_totals)

# From previous steps

airline\_bumping = pd.read\_csv("airline\_bumping.csv")

print(airline\_bumping.head())

airline\_totals = airline\_bumping.groupby("airline")[["nb\_bumped", "total\_passengers"]].sum()

print(airline\_totals)

# Create new col, bumps\_per\_10k: no. of bumps per 10k passengers for each airline

airline\_totals["bumps\_per\_10k"] = (airline\_totals["nb\_bumped"]/ airline\_totals["total\_passengers"]\* 10000)

print(airline\_totals)

print(airline\_bumping[airline\_bumping["airline"]=='ALASKA AIRLINES'])

airline year nb\_bumped total\_passengers

0 DELTA AIR LINES 2017 679 99796155

1 VIRGIN AMERICA 2017 165 6090029

2 JETBLUE AIRWAYS 2017 1475 27255038

3 UNITED AIRLINES 2017 2067 70030765

4 HAWAIIAN AIRLINES 2017 92 8422734

nb\_bumped total\_passengers

airline

ALASKA AIRLINES 1392 36543121

AMERICAN AIRLINES 11115 197365225

DELTA AIR LINES 1591 197033215

EXPRESSJET AIRLINES 3326 27858678

FRONTIER AIRLINES 1228 22954995

HAWAIIAN AIRLINES 122 16577572

JETBLUE AIRWAYS 3615 53245866

SKYWEST AIRLINES 3094 47091737

SOUTHWEST AIRLINES 18585 228142036

SPIRIT AIRLINES 2920 32304571

UNITED AIRLINES 4941 134468897

VIRGIN AMERICA 242 12017967

nb\_bumped total\_passengers bumps\_per\_10k

airline

ALASKA AIRLINES 1392 36543121 0.381

AMERICAN AIRLINES 11115 197365225 0.563

DELTA AIR LINES 1591 197033215 0.081

EXPRESSJET AIRLINES 3326 27858678 1.194

FRONTIER AIRLINES 1228 22954995 0.535

HAWAIIAN AIRLINES 122 16577572 0.074

JETBLUE AIRWAYS 3615 53245866 0.679

SKYWEST AIRLINES 3094 47091737 0.657

SOUTHWEST AIRLINES 18585 228142036 0.815

SPIRIT AIRLINES 2920 32304571 0.904

UNITED AIRLINES 4941 134468897 0.367

VIRGIN AMERICA 242 12017967 0.201

airline year nb\_bumped total\_passengers

8 ALASKA AIRLINES 2017 658 18817924

20 ALASKA AIRLINES 2016 734 17725197

<script.py> output:

airline year nb\_bumped total\_passengers

0 DELTA AIR LINES 2017 679 99796155

1 VIRGIN AMERICA 2017 165 6090029

2 JETBLUE AIRWAYS 2017 1475 27255038

3 UNITED AIRLINES 2017 2067 70030765

4 HAWAIIAN AIRLINES 2017 92 8422734

# From previous steps

airline\_bumping = pd.read\_csv("airline\_bumping.csv")

print(airline\_bumping.head())

airline\_totals = airline\_bumping.groupby("airline")[["nb\_bumped", "total\_passengers"]].sum()

airline\_totals["bumps\_per\_10k"] = airline\_totals["nb\_bumped"] / airline\_totals["total\_passengers"] \* 10000

# Print airline\_totals

print(airline\_totals)

airline year nb\_bumped total\_passengers

0 DELTA AIR LINES 2017 679 99796155

1 VIRGIN AMERICA 2017 165 6090029

2 JETBLUE AIRWAYS 2017 1475 27255038

3 UNITED AIRLINES 2017 2067 70030765

4 HAWAIIAN AIRLINES 2017 92 8422734

nb\_bumped total\_passengers bumps\_per\_10k

airline

ALASKA AIRLINES 1392 36543121 0.381

AMERICAN AIRLINES 11115 197365225 0.563

DELTA AIR LINES 1591 197033215 0.081

EXPRESSJET AIRLINES 3326 27858678 1.194

FRONTIER AIRLINES 1228 22954995 0.535

HAWAIIAN AIRLINES 122 16577572 0.074

JETBLUE AIRWAYS 3615 53245866 0.679

SKYWEST AIRLINES 3094 47091737 0.657

SOUTHWEST AIRLINES 18585 228142036 0.815

SPIRIT AIRLINES 2920 32304571 0.904

UNITED AIRLINES 4941 134468897 0.367

VIRGIN AMERICA 242 12017967 0.201

**Daily XP400**

##### Exercise

##### Exercise

# DataFrame to CSV

You're almost there! To make things easier to read, you'll need to sort the data and export it to CSV so that your colleagues can read it.

pandas as pd has been imported for you.

##### Instructions

**100 XP**

* Sort airline\_totals by the values of bumps\_per\_10k from highest to lowest, storing as airline\_totals\_sorted.
* Print your sorted DataFrame.
* Save the sorted DataFrame as a CSV called "airline\_totals\_sorted.csv".
* # Create airline\_totals\_sorted
* airline\_totals\_sorted = \_\_\_\_
* # Print airline\_totals\_sorted
* \_\_\_\_
* # Save as airline\_totals\_sorted.csv
* \_\_\_\_

# From previous steps airline\_bumping = pd.read\_csv("airline\_bumping.csv") print(airline\_bumping.head()) airline\_totals = airline\_bumping.groupby("airline")[["nb\_bumped", "total\_passengers"]].sum() airline\_totals["bumps\_per\_10k"] = airline\_totals["nb\_bumped"] / airline\_totals["total\_passengers"] \* 10000 # Print airline\_totals print(airline\_totals)

# Create airline\_totals\_sorted

airline\_totals\_sorted = airline\_totals.sort\_values("bumps\_per\_10k",ascending=False)

# Print airline\_totals\_sorted

print(airline\_totals\_sorted)

# Save as airline\_totals\_sorted.csv

airline\_totals\_sorted.to\_csv("airline\_totals\_sorted.csv")

# Create airline\_totals\_sorted

airline\_totals\_sorted = airline\_totals.sort\_values("bumps\_per\_10k",ascending=False)

# Print airline\_totals\_sorted

print(airline\_totals\_sorted)

# Save as airline\_totals\_sorted.csv

airline\_totals\_sorted.to\_csv("airline\_totals\_sorted.csv")

nb\_bumped total\_passengers bumps\_per\_10k

airline

EXPRESSJET AIRLINES 3326 27858678 1.194

SPIRIT AIRLINES 2920 32304571 0.904

SOUTHWEST AIRLINES 18585 228142036 0.815

JETBLUE AIRWAYS 3615 53245866 0.679

SKYWEST AIRLINES 3094 47091737 0.657

AMERICAN AIRLINES 11115 197365225 0.563

FRONTIER AIRLINES 1228 22954995 0.535

ALASKA AIRLINES 1392 36543121 0.381

UNITED AIRLINES 4941 134468897 0.367

VIRGIN AMERICA 242 12017967 0.201

DELTA AIR LINES 1591 197033215 0.081

HAWAIIAN AIRLINES 122 16577572 0.074

# Create airline\_totals\_sorted

airline\_totals\_sorted = airline\_totals.sort\_values("bumps\_per\_10k",ascending=False)

# Print airline\_totals\_sorted

print(airline\_totals\_sorted)

# Save as airline\_totals\_sorted.csv

airline\_totals\_sorted.to\_csv("airline\_totals\_sorted.csv")

airline\_totals\_sorted.csv

nb\_bumped total\_passengers bumps\_per\_10k

airline

EXPRESSJET AIRLINES 3326 27858678 1.194

SPIRIT AIRLINES 2920 32304571 0.904

SOUTHWEST AIRLINES 18585 228142036 0.815

JETBLUE AIRWAYS 3615 53245866 0.679

SKYWEST AIRLINES 3094 47091737 0.657

AMERICAN AIRLINES 11115 197365225 0.563

FRONTIER AIRLINES 1228 22954995 0.535

ALASKA AIRLINES 1392 36543121 0.381

UNITED AIRLINES 4941 134468897 0.367

VIRGIN AMERICA 242 12017967 0.201

DELTA AIR LINES 1591 197033215 0.081

HAWAIIAN AIRLINES 122 16577572 0.074

Traceback (most recent call last):

File "<stdin>", line 72, in exceptionCatcher

raise exception

File "<stdin>", line 3361, in run\_ast\_nodes

if (await self.run\_code(code, result, async\_=asy)):

File "<stdin>", line 3458, in run\_code

self.showtraceback(running\_compiled\_code=True)

File "<stdin>", line 2066, in showtraceback

self.\_showtraceback(etype, value, stb)

File "<stdin>", line 72, in exceptionCatcher

raise exception

File "<stdin>", line 3441, in run\_code

exec(code\_obj, self.user\_global\_ns, self.user\_ns)

File "<stdin>", line 9, in <module>

airline\_totals\_sorted.csv

File "<stdin>", line 5487, in \_\_getattr\_\_

return object.\_\_getattribute\_\_(self, name)

AttributeError: 'DataFrame' object has no attribute 'csv'

# Create airline\_totals\_sorted

airline\_totals\_sorted = airline\_totals.sort\_values("bumps\_per\_10k",ascending=False)

# Print airline\_totals\_sorted

print(airline\_totals\_sorted)

# Save as airline\_totals\_sorted.csv

airline\_totals\_sorted.to\_csv("airline\_totals\_sorted.csv")

print(airline\_totals\_sorted.csv)

nb\_bumped total\_passengers bumps\_per\_10k

airline

EXPRESSJET AIRLINES 3326 27858678 1.194

SPIRIT AIRLINES 2920 32304571 0.904

SOUTHWEST AIRLINES 18585 228142036 0.815

JETBLUE AIRWAYS 3615 53245866 0.679

SKYWEST AIRLINES 3094 47091737 0.657

AMERICAN AIRLINES 11115 197365225 0.563

FRONTIER AIRLINES 1228 22954995 0.535

ALASKA AIRLINES 1392 36543121 0.381

UNITED AIRLINES 4941 134468897 0.367

VIRGIN AMERICA 242 12017967 0.201

DELTA AIR LINES 1591 197033215 0.081

HAWAIIAN AIRLINES 122 16577572 0.074

## 1. Wrap-up

Congratulations! You've now covered the basics of using pandas.

## 2. Recap

In chapter 1, you saw how to subset and sort DataFrames and how to add new columns. In chapter 2, you saw several methods for aggregating and grouping data to calculate summary statistics. In chapter 3, you saw how using indexing and slicing allows for simpler subsetting. In chapter 4, you saw how to visualize a DataFrame, and how to read data from and write data to CSV files.

## 3. More to learn

I hope you are convinced that pandas is a powerful tool to analyze tabular data. In fact, pandas is so powerful that there are many features that we didn't get around to discussing in this course. To begin with, everything in this course involved a single DataFrame, but sometimes you need to join or "merge" several DataFrames together. Reading from CSV files barely scratches the surface of the options for importing data into pandas. You can also perform very sophisticated exploratory data analysis using pandas.

## 4. Congratulations!

Congratulations, and have fun learning!