15.1 Categorical data: counts of occurrences

Let's say you had access to a poll on people's favorite pop stars. You import this into a big ol' Pandas `Series` called `faves`:

```
print(faves)
```

```
0          Katy Perry
1             Rihanna
2       Justin Bieber
3               Drake
4             Rihanna
5        Taylor Swift
6               Adele
7               Adele
8        Taylor Swift
9       Justin Bieber
...
1395       Katy Perry
dtype: object
```

That's great, but it's also kinda TMI. You probably don't care who the *first* person's idol is, nor the fifteenth, nor the last. Much more interesting is simply *how many times* each value appears in the `Series`. This information is available from the Pandas `.value_counts()` method:

```
counts = faves.value_counts()
print(counts)
```

```
Taylor Swift     388
Katy Perry       265
Drake            261
Adele            212
Rihanna          136
Justin Bieber    134
dtype: int64
```

The `.value_counts()` method returns another `Series`, but the *values* of the original `Series` become the *keys* of the new one. This tells us at a glance how popular each answer is relative to the others.

To get percentages instead of totals, just divide by the total and multiply by 100, of course:

```
print(counts / len(counts) * 100)
```

```
Taylor Swift     27.7937
Katy Perry       18.9828
Drake            18.6963
Adele            15.1862
Rihanna          09.7421
Justin Bieber    09.5989
dtype: float64
```

Recall (p. 45) that the **mode** is the only measure of central tendency that makes sense for categorical data. And all you have to do is call `.value_counts()` and look at the top result. (In this case, `Taylor Swift`.)

Note that `.value_counts()` is a Pandas `Series` method, not a NumPy method. If you find yourself with a NumPy array instead, you can just **wrap** it in a `Series` as we did in Section 11.1 (p. 106):

```
my_array = np.array(['red','blue','red','green','green',
    'green','blue'])
print(pd.Series(my_array).value_counts())
```

```
green    3
red      2
blue     2
dtype: int64
```

15.2 Numerical data: quantiles

A **quantile** is a real number between 0 and 1 that represents a "**cut point**" of a numerical data set: roughly speaking, it's the number for which a certain fraction of the values are *less than* that number. So the ".2-quantile" (pronounced "point two quantile") of a variable containing the heights of third-graders might be 50 inches. If that's the case, it would indicate that *20% of the third-graders are less than 50 inches tall.*

Quantiles are very revealing, but underappreciated. Most people don't seem to know how to interpret them. But once you figure it out, you'll realize that quantiles tell you almost everything possible to know about a numeric variable: by dialing the quantile between 0 and 1, you can tell exactly how common values in certain ranges are.

In Python, you simply call the `.quantile()` method on a `Series`, passing a number between 0 and 1 as an argument, and it tells you exactly where that cut point is.

Now there's a little bit of weirdness around the edges, depending on the exact definition used to calculate the quantiles. Let's say I collected some salary data, and got these responses ("k" means "thousand dollars per year," and "M" means "million dollars per year"):

35k 22k 67k 45k 35k 8M 94k 51k 53k 64k 54k

How would I calculate, say, the .7-quantile? First, sort the numbers:

22k 35k 35k 45k 51k 53k 54k 64k 67k 94k 8M

(yes, we *do* include the 35k value twice; don't eliminate duplicates) and then spread out the quantiles "evenly" from 0 to 1:

value:	22k	35k	35k	45k	51k	53k	54k	64k	67k	94k	8M
	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
quantile:	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	1.0

Don't get picky on me. If you were picky, you could quibble at saying "the .3-quantile is 45k" since it's technically *not* true that 30% of the values are less than 45k: in truth, 3 out of 11 (27.3%) of them are. Whatever, whatever. The point is that 45k is at the "cut point" that's $\frac{3}{10}$ths of the way through the values from min to max. Quantiles aren't about laser precision anyway: they're about understanding the general pattern of the data.

"Special" quantiles

You'll realize as an immediate consequence of the above that the **median** is just another name for the **.5-quantile**. It's the value for which half the data points are below it, and half above. Also, the **0-quantile** is just the minimum of the data set, and the **1-quantile** the maximum.

Other kinds of "-tiles"

This whole idea may ring bells with other names for you: qua**r**tiles, qu**i**ntiles, deciles, or (most probably) percentiles. All of those are basically special cases of quantiles. They split the data into evenly-sized groups:

- "quartiles": split the data into four groups, with the split points being the .25-, .5-, and .75-quantiles.
- "quintiles": split the data into five groups, with the split points being the .2-, .4-, .6, and .8-quantiles.
- "deciles": split the data into ten groups, with the split points being the .1-, .2-, .3-, .4-, .5-, .6-, .7-, .8-, and .9-quantiles.
- "percentiles": split the data into 100 groups, with the split points being the .01-, .02-, .03-, ..., .98-, and .99-quantiles.

Be careful to understand that "evenly-sized groups" does not mean "groups with the same-sized range," but rather "groups with *the same number of data points in them.*" Normally, in fact, the ranges will *not* be the same size. The lowest quintile for a data set of IQs might range from 47 (the lowest IQ in the data set) all the way up to 83, whereas the IQs in the middle quintile might all be in the narrow range 96 to 104.

The IQR (interquartile range)

Speaking of quartiles, you'll commonly hear data scientists cite the **IQR**, or **interquartile range**, as a measure of how widely varying a univariate data set is. It's simply the distance between the .25-quantile and the .75-quantile; or in quartile terms, the difference between the "upper" and "lower" quartiles.

Because of how quantiles work, exactly 50% of the data points are between the .25- and .75-quantiles. This means that the more spread out the data points are, the larger the IQR, and vice versa. In this sense, it's akin to the standard deviation (see p. 153) which you may be familiar with.

A quantile example

Let's nail this down with an example. I have a (fictitious) data set containing the number of YouTube plays for each of a selection of videos. It's called `num_plays`. Here are the first few values:

```
0     791
1    3133
2       0
3    1789
4     297
5     219
6    1688
7     209
8     422
9   91454
dtype: int64
```

That's great, but it's both too much information and too little: we can pore through the plays for every single video, but it's hard to get our head around what the overall contents are. So let's run some quantiles. We'll start with the .1-quantile:

```
print(num_plays.quantile(.1))
```

```
0.0
```

Whoa. The .1-quantile is *zero*. Think about what that means. Pictorially, sorting the data would give this:

value:	0	0	0	0	0	0	0	0	···	0	0	···
	↑										↑	
quantile:	.0										.1	

Put another way, that means that (at least) *10% of our videos have no plays at all.*

Let's try the .2-quantile:

```
print(num_plays.quantile(.2))
```

```
15.0
```

Okay, now at least we have a pulse. But in case we thought this was data set was packed with big hits, think again: a full 20% of these videos have fewer than 15 plays.

The median is:

```
print(num_plays.quantile(.5))
```

```
263.0
```

That's quite a bit higher. How about the 90% mark?

```
print(num_plays.quantile(.9))
```

```
1378.0
```

All right, so the upper end of these videos are in the thousands. Finally, let's look at the max:

```
print(num_plays.quantile(1))
```

```
982221.0
```

!!

Believe it or not, this sort of thing isn't unusual, especially with data from social phenomena. The tiny fraction of the data at the upper end of the range is *vastly* higher than everything else is. Get your head around that: the median number of plays was a couple hundred, but the maximum number of plays was nearly a *million*.

Computing the IQR of this data set is as simple as finding the difference between the .25 and .75 quantiles:

```
print(num_plays.quantile(.75) - num_plays.quantile(.25))
```

```
399.75
```

15.3 Numerical data: other summary statistics

That YouTube data set is a good segue to talking about that most overused of all statistics: the **mean**. Nearly everyone, if you ask them "what's the typical number of plays for these videos?" will use the mean, or average, to get at the answer. After all, isn't that what we mean by "the average number of plays?"

The answer is: not really, and not usually. Look what happens if we compute the mean (using the `.mean()` `Series` method) in this case:

```
print(num_plays.mean())
```

```
14018.888235294118
```

Consider just how misleading that really is. The "average" number of plays is over 14,000. Yet the *.9*-quantile was less than $\frac{1}{10}$th of that! In fact, even the .97-quantile is only:

```
print(num_plays.quantile(.97))
```

```
3836.0
```

So *over 97% of the videos have less than the mean of 14,000 plays.* I think you'll agree that it is nonsensical to claim that "the typical number of plays is 14,018," no matter how you slice it.

We'll see in the next section why the mean is hopelessly skewed here. Basically, unless the data is symmetrical and "bell-curvy," it gives a meaningless number. It is almost *always* safer and more illuminating to look at the median (or other quantiles).

For completeness, one other commonly cited summary statistic is the **standard deviation**, which can be computed with the `.std()` method:

```
print(num_plays.std())
```

```
93031.835
```

The standard deviation, like the IQR, is a measure of the "spread" of a data set – a high number means (in this example) higher variability in the number of plays from video to video. As with the mean, it's essentially meaningless (no pun intended) unless the data is nice and bell-curve shaped.

Speaking of which, we'll never be able to judge the "shape" of anything unless we get some graphical plots involved. So let's turn our focus to that.

15.4 Plotting univariate data

There are basically two useful ways of plotting a `Series` with univariate data. In one, you care about the specific labels (*i.e.* keys, or "index") of the values in the `Series`, and you want them to be prominent in the plot. In the other, you don't; you just want to show the values themselves, so you can visualize how they are distributed irrespective of what label they might have.

Let's do the first one first.

Bar charts of labeled data

Let's read a data set on the world countries with the highest GDP (Gross Domestic Product). Here's a CSV file called `gdp.csv`[1]:

```
Nation,Trillions
Italy,2.26
Germany,4.42
Brazil,2.26
United States,21.41
France,3.06
Canada,1.91
Japan,5.36
China,15.54
India,3.16
United Kingdom,3.02
```

We'll read that into a `Series` using our technique from p. 107:

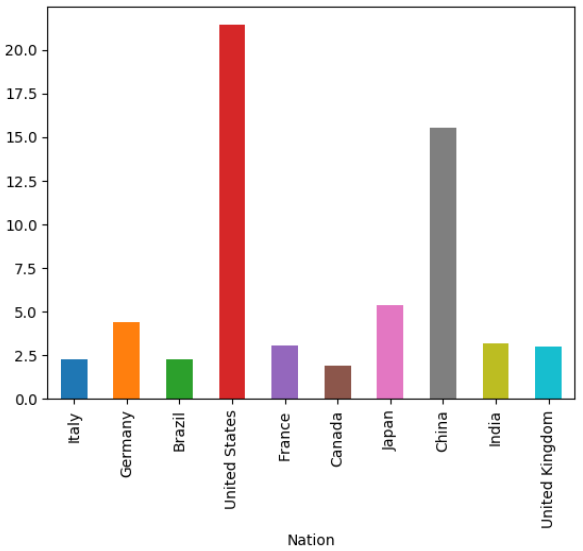
```
gdp = pd.read_csv('gdp.csv', squeeze=True, index_col=0,
    header=None)
print(gdp)
```

[1] Recall the caveat about filename extensions in the p. 69 footnote.

```
0
Nation            Trillions
Italy                  2.26
Germany                4.42
Brazil                 2.26
United States         21.41
France                 3.06
Canada                 1.91
Japan                  5.36
China                 15.54
India                  3.16
United Kingdom         3.02
Name: 1, dtype: object
```

and now, we can visualize the relative sizes of these economies with the `.plot()` method. The `.plot()` method takes, among other things, a "`kind`" argument which specifies what kind of plot you want. In this case, a **bar chart** is the correct thing:

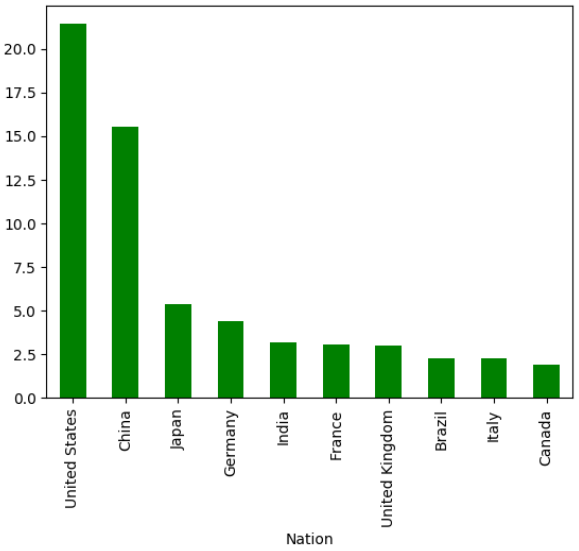
```
gdp.plot(kind='bar')
```



There are a zillion ways to customize these plots, and I'll only mention a very, very few. A more complete list of options is available by Googling, or going to `https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.plot.html`

For instance, to make all the bars the same color, we can pass "`color="blue"`". Sorting the values is something we already know how to do, with `.sort_values()`:

```
gdp.sort_values(ascending=False).plot(kind='bar')
```



You see what I mean about "caring about the labels/keys/index" for this sort of plot: if we hadn't labeled the bars, the plot would tell us nothing useful.

I'm sure you've seen lots of bar charts in your life, so this is nothing new. But consider how much information is embedded in this infographic. Not only can we tell that the U.S. and China are the two biggest economies, we can tell that they are *far and away* the two biggest, with Japan and Germany (the next two highest) only a fraction.

Bar charts of occurrence counts

A very common special case of a bar chart is one where we combine it with the `.value_counts()` method. Let's go back to Taylor vs. Katy:

```
print(faves)
```

```
0          Katy Perry
1             Rihanna
2       Justin Bieber
3              Drake
4             Rihanna
5        Taylor Swift
6              Adele
7              Adele
8        Taylor Swift
9       Justin Bieber
...
1395       Katy Perry
dtype: object
```

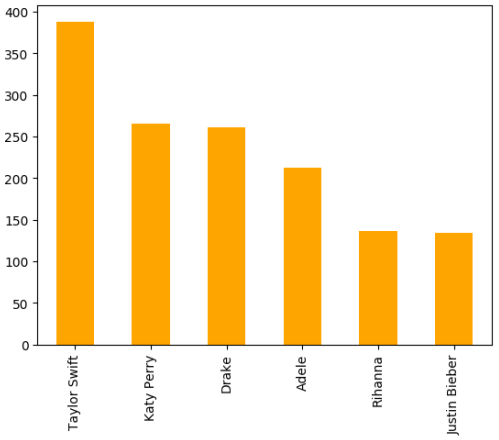
It would be useful to see an infographic on how popular each celebrity is, and combining `.value_counts()` and `.plot()` makes it a snap:

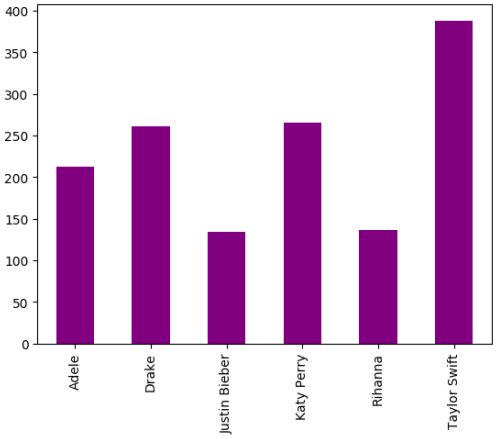
```
faves.value_counts().plot(kind='bar',color="orange")
```

The `.sort_values()` method wasn't needed here, because our friend `.value_counts()` already returns its answer in decreasing numerical order. If we wanted the bars in alphabetical order instead, we'd just sort the `Series` by index before plotting:

```
faves.value_counts().sort_index().plot(kind='bar',
    color="purple")
```

These long lines with lots of strung-together methods are concise, but can also be confusing. It's always an option to use temporary variables to store the intermediate results instead:






```
counts = faves.value_counts()
alphbetical_counts = counts.sort_index()
alphbetical_counts.plot(kind='bar',color="purple")
```

Just a matter of preference.

15.5 Numerical data: histograms

As I mentioned on p. 154, sometimes we don't actually care about the labels in a `Series`, only the values. This is when we're trying to size up how *often* values of various magnitudes appear, irrespective of which specific objects of study those values go with.

My favorite plot is the **histogram**. It's super powerful if you know how to read it, but underused because few people seem to know how. The idea is that we take a numeric, univariate data set, and divide it up into **bins**. Bins are sort of the reverse of quantiles: all bins have the *same* size range, but a *different* number of data points fall into each one.

Suppose we had data on the entire history of a particular NCAA football conference. A `Series` called "`pts`" has the number of points scored by each team in all that conference's games. It looks like this:

```
print(pts)
```

```
0        7
1       35
2       40
3       17
4       10
...
399     14
dtype: int64
```

Some basic summary statistics of interest include:

```
print("min: {}".format(pts.quantile(0)))
print(".25-quantile: {}".format(pts.quantile(.25)))
print(".5-quantile: {}".format(pts.quantile(.5)))
print(".75-quantile: {}".format(pts.quantile(.75)))
print("max: {}".format(pts.quantile(1)))
print("mean: {}".format(pts.mean()))
```

```
min: 0.0
.25-quantile: 17.0
.5-quantile: 25.0
.75-quantile: 32.0
max: 55.0
mean: 23.755
```

Looks like a typical score is in the 20's, with the conference record being a whopping 55 points in one game. The IQR is 32 – 17, or 15 points.

We can plot a histogram of this `Series` with this code:

```
pts.plot(kind='hist')
```

The result is in Figure 15.1. Stare hard at it. Python has divided up the points into ranges: 0 through 5 points, 6 through 11 points, 12 through 17, *etc.* Each of these ranges is a bin. The height of each blue bar on the plot is simply the number of games in which a team scored in that range.

Now what do we learn from this? Lots, if we know how to read it. For one thing, it looks like the vast majority of games have teams scoring between 12 and 38 points. A few teams have managed to eke out 40 or more, and there have been a modest number of single-digit scores or shutouts. Moreover, it appears that scores between 24 and 38 are considerably more common than those between 12 and 24. Finally, this data shows some evidence of being "bell-curvy" in the sense that values in the middle of the range are more common than values at either end, and it is (very roughly) symmetrical on both sides of the median.

This is even more precise information than the quantiles gave us. We get an entire birds-eye view of the data set. Whenever I'm looking at a numerical, univariate data set, pretty much the first thing I do is throw a histogram up on the screen and spend at least a couple minutes staring at it. It's almost the best diagnostic tool available.

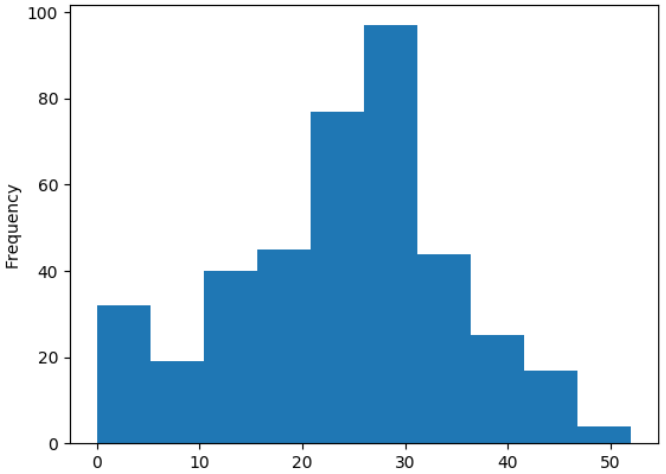


Figure 15.1: A histogram of the historical points-per-game for teams in a certain NCAA football conference.

Bin size

Now one idiosyncrasy with histograms is that a lot depends on the bin size and placement. Python made its best guess at a decent bin size here by choosing ranges of 6 points each. But we can control this by passing a second parameter to the `.plot()` function, called "`bins`":

```
pts.plot(kind='hist', bins=30)
```

Here we specifically asked for *thirty* bins in total, and we get the result in Figure 15.2. Now each bin is only two points wide, and as you can see there's a lot more detail in the plot.

Whether that amount of detail is a good thing or not takes some practice to decide. Make your bins too large and you don't get much precision in your histogram. Make them too small and the trees can overwhelm the forest. In this case, I'd say that Figure 15.2

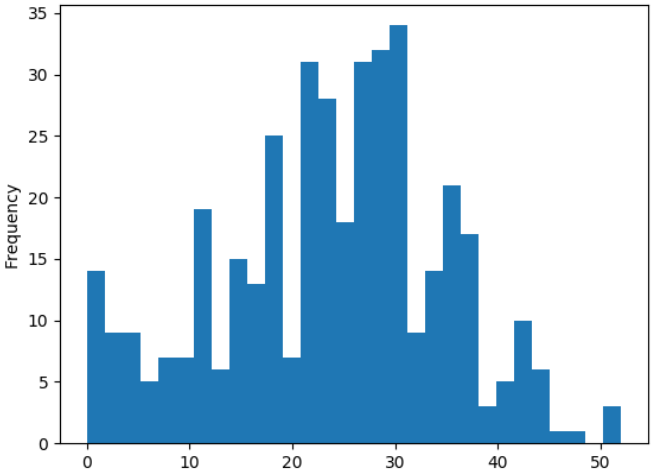


Figure 15.2: The same data set as in Figure 15.1, but with more (and smaller) bins.

is good in that it tells us something not apparent from Figure 15.1: there are quite a few shutouts (zero-point performances), not merely games with six-points-or-less. Whether the trough between 22 and 24 points is meaningful is another matter, and my guess is that part is obscuring the more general features apparent in the first plot.

The rule is: whenever you create a histogram, *take a few minutes to experiment with different bin sizes.* Often you'll find a "sweet spot" where the amount of detail is just right, and you'll get great insight into the data. But you do have to work at it a little bit.

Non-bell-curvy data

Let's return again to the YouTube example. We had some surprises when we looked at the quantiles and saw that the 1-quantile (max) was astronomically higher than the .9-quantile was. Let's see what happens when we plot a histogram (show in Figure 15.3):

```
num_plays.plot(kind='hist', color="red")
```

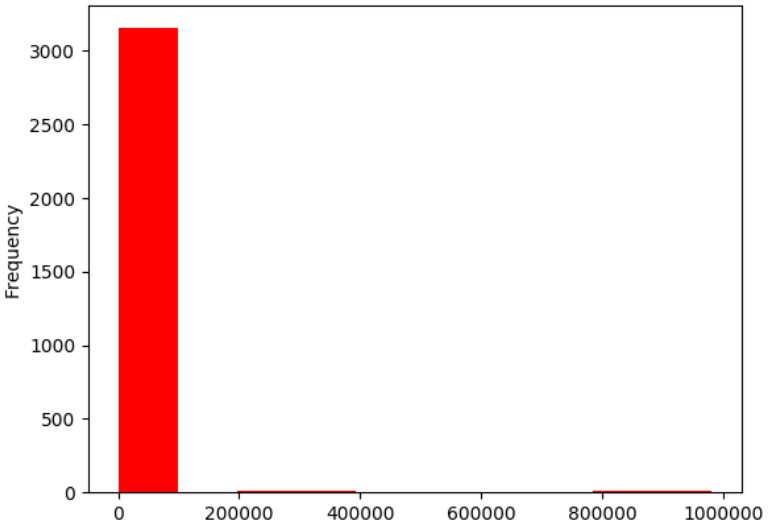


Figure 15.3: A first attempt at plotting the YouTube `num_plays` data set.

Huh?? Wait, where are all the bars of varying heights? We seem to have got only a single one.

But they're there! They're just so small you can't see them. If you stare at the x-axis – and your eyesight is good – you might see tiny signs of life at higher values. But the overall picture is clear: the vast, vast majority of videos in this set have between 0 and 100,000 plays.

Let's see if we can get more detail by increasing the number of bins (say, to 1000):

```
num_plays.plot(kind='hist', bins=1000, color="red")
```

We now get the left-hand side of Figure 15.4. It didn't really help much. Turns out the masses aren't merely crammed below a hundred thousand plays; they're crammed below *one* thousand. We need another approach if we're going to see any detail on the low-play videos.

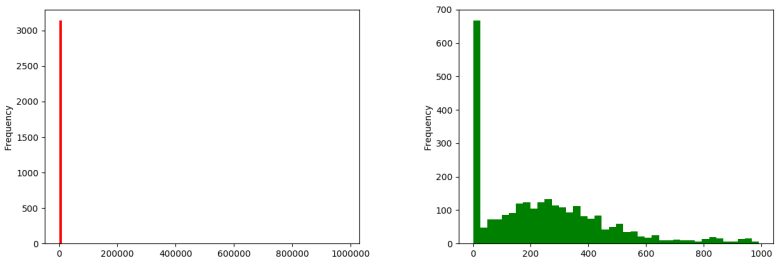


Figure 15.4: Further attempts at plotting the YouTube `num_plays` data set. On the left side, we decreased the bin size to no avail. On the right side, we gave up on plotting the popular videos and concentrated only on the unpopular ones, which does illuminate the lower end somewhat. (Don't miss the x-axis ranges!)

The only way to really see the distribution on the low end is to *only* plot the low end. Let's use a query (recall section 13.1 from p. 124) to filter out *only* the videos with 1000 plays or fewer, and then plot a histogram of that:

```
unpopular_video_plays = num_plays[num_plays <= 1000]
unpopular_video_plays.plot(kind='hist', color="green")
```

This gives the right-hand side of Figure 15.4. Now we can at least see what's going on. Looks like our `Series` has a crap-ton of videos that have never been viewed at all (recall our .1-quantile epiphany for this data set on p.151) plus a chunk that are in the 500-views-or-fewer range.

The takeaway here is that not all data sets (by a long shot!) are bell-curvy. Statistics courses often present nice, symmetric data sets on physical phenomena like bridge lengths or actor heights or

free throw percentages, which have nice bell curves and are nicely summarized by means and standard deviations. But for many social phenomena (like salaries, numbers of likes/followers/plays, lengths of Broadway show runs, *etc.*) the data looks more like this YouTube example. A few extremely large values dominate everything else by their sheer magnitude, which makes it more difficult to wrap your head around.

It also makes it more challenging to answer the question, "what's the *typical* value for this variable?" It ain't the mean, that's for sure. If you asked me for the "typical" number of plays of one of these YouTube videos, I'd probably say "zero" since that's an extremely common value. Another reasonable answer would be "somewhere in the low hundreds," since there are quite a few videos in that range, as illustrated by the right-hand-side of Figure 15.4. But you'd be hard-pressed to try and sum up the entire data set with a single typical value. There just isn't one for stuff like this.

15.6 Numerical data: box plots

Let's talk about one more type of plot in this chapter, even though it's really most useful when dealing with bivariate data, as we'll address in chapter 20. It's called the **box plot** (also known as a "**box-and-whisker" plot**). We can create one by passing "`kind="box"`" to the `.plot()` method (here for the NCAA football data):

```
pts.plot(kind="box")
```

The result is shown in Figure 15.5, along with some annotations in red so you can figure out what's going on.

For now, don't worry about the mysterious word "`None`" at the bottom. (This indicates which "group" the box represents, and will feature prominently in our bivariate data chapter.) For a univariate data set like this one, the x-axis has no meaning. The y-axis, on the other hand, is easy to understand: it's the number of points per football game.

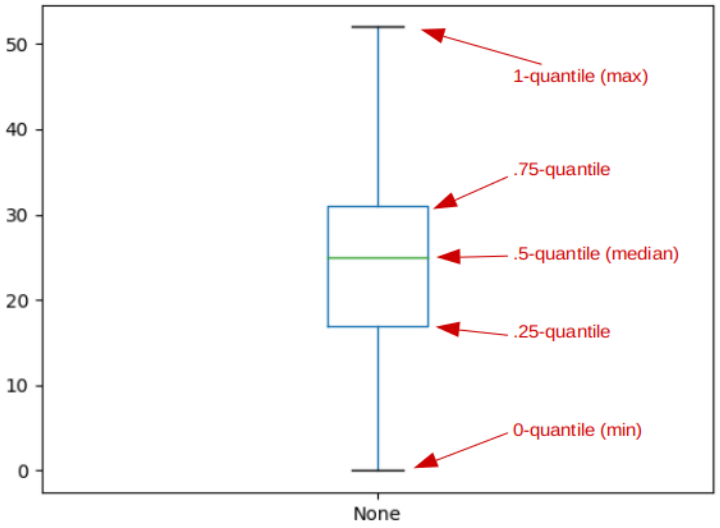


Figure 15.5: A box plot of the NCAA points data.

Now the thing to realize about box plots is that they're essentially just a graphical way of showing quartiles; or, put another way, a graphical way of showing these five quantiles:

- The 0-quantile (the minimum value) is the y-value of the bottom "whisker."
- The .25-quantile is the y-value of the bottom of the "box."
- The .5-quantile (the median) is the y-value of the horizontal line within the box.
- The .75-quantile is the y-value of the top of the "box."
- The 1-quantile (the maximum value) is the y-value of the top "whisker."

Using your quantile knowledge from section 15.2, you'll realize the following fact: *the box alone contains exactly half the data points.* This is a key insight. While the whiskers show the entire range of the data, the box shows the middle 50% of it. (And the height of the box is precisely the IQR.) This makes it very easy to grasp where the bulk of the data lies, and it reinforces the lesson we learned from the histogram on this data set (Figure 15.1 on page 161): a

big chunk of the time, teams score in the 20's.

You might object to showing an entire plot for this, since I've just revealed that it's merely a fancy way to show five numbers. And you're right, in a way. However, when we show multiple *groups* of data side-by-side, each with their own box, it becomes a particularly powerful tool. Stay tuned for that.

Outliers

What happens if we show our head-scratching YouTube data set as a box plot? You get the monstrosity in Figure 15.6.

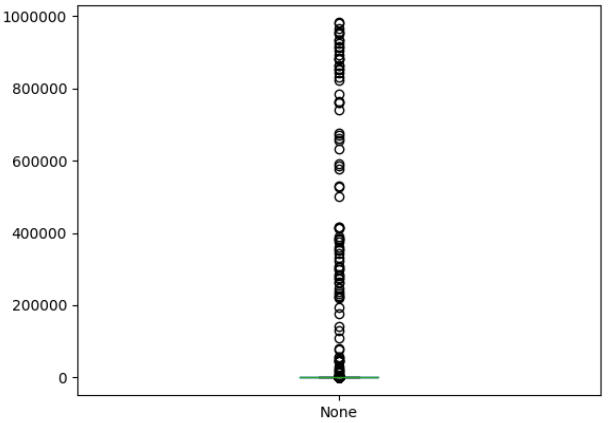


Figure 15.6: A box plot of a non-bell-curvy data set.

Geez Louise, does that look wacky. The little circles (which to me always looked like bubbles from fish breath) represent **outliers**, an important concept in data science. An outlier is basically any data point that's so far out of the normal range that it seems strange. Python is essentially flagging it for us, so we can judge for ourselves whether it was a data entry error or just a strange data point. In this case, these aren't errors – there's just a handful of videos that have been played a ton of times. And this makes the whole box plot look weird.

Notice from Figure 15.6 that *the entire box and both whiskers* have gotten smooshed at the bottom of the figure, as if crushed by the gravity of a black hole. You'll see that the top whisker doesn't *really* mean "maximum," since it's way down there in thousand-land despite the fact that we have videos with almost a million views. The top whisker truly means "the maximum *reasonable-looking* data point in the `Series`," where "reasonable-looking" is something Pandas is trying to make an educated guess about. There are ways to tweak what counts as an outlier, but my purpose here is just to get you to realize that when you have a highly skewed data set (like YouTube), prepare to see lots of things that are considered "outliers," and prepare to comb through all the mess on your box plots to try and discern the true meaning it's trying to convey.

Chapter 16

Tables in Python (1 of 3)

The third of our three aggregate data types from waaaay back in Chapter 7 was the **table**. Don't worry: we haven't forgotten about him. In this chapter, we'll implement him by means of the Pandas `DataFrame`, the most important data type in this entire book.

16.1 Reading a `DataFrame` from a `.csv` file

Unlike NumPy arrays and Pandas `Series`es, which we learned several different ways to create, we're only going to learn one way to create a `DataFrame`. That's because `DataFrame`s are normally big enough that it's just too tedious to ever type them in manually. Instead, we'll read them from an external source; a `.csv` file.

We'll actually use the same `read_csv()` function that we used in section 11.1 (p. 107), although oddly, this time we won't need to specify as many arguments. Let's say we have a "`davieses.csv`" file with these contents:

```
person,age,gender,height,instrument
Dad,50,M,73,piano
Mom,49,F,66,flute
Lizzy,21,F,63,guitar
TJ,20,M,71,trombone
Johnny,17,M,72,euphonium
```

We can read it into a `DataFrame` with this code:

```
my_first_df = pd.read_csv("davieses.csv").set_index('person')
print(my_first_df)
```

```
         age gender  height instrument
person
Dad       51      M      73      piano
Mom       49      F      66      flute
Lizzy     21      F      63     guitar
TJ        20      M      71   trombone
Johnny    17      M      72  euphonium
```

A couple things. First, you may have noticed that the `davieses.csv` file had a "header" row. This means that the first line of the file is not like the others: instead of containing information on a specific family member, it contains the *kind* of information for *every* family member. It looked like this:

```
person,age,gender,height,instrument
```

and you'll notice that these words (except for the first one; more on that in a moment) became the *column names* when we imported the data. This sort of information, by the way, is called "**metadata**," a geeky-sounding word that basically means "data about data." If "Lizzy plays the guitar" is a piece of data, then "family members play instruments" is a piece of *meta*data.

Second, don't miss the ending I tacked on to the `read_csv()` line, where I called the `.set_index()` method on the `DataFrame`. This tells Pandas that one of the columns in the `DataFrame` should be designated as the **index** (or the **keys**).

Back on p. 57 I asserted that unlike associative arrays, tables didn't have keys. And that's true of the general "table" concept. But Pandas designed their `DataFrame`s to behave in the same way as their `Series`es: one uniquely-valued column will be used to identify each row.

This choice is usually easy; if you glance back to Figure 7.3 (p. 57), we'd probably want to choose the screenname as the index (although a case could be made for the real name column instead). For the table in Figure 7.4 (p. 59), it would be the item column. In the `DataFrame` we just created above, obviously `person` is the correct choice – it's the only one sure to be unique.[1]

Anyway, designating a column as the index in this way sort of removes it from the other, "ordinary" columns. In the output, above, you may notice that the word "`person`" is printed somewhat lower than the other column names are. It turns out that if we want to talk about the index column specifically, we'll need to use a slightly different technique than we do for the other columns. More on that next chapter.

Finally, note that calling `.set_index()` is optional. It's perfectly fine to just call `pd.read_csv()` and leave it at that. In that case, Pandas will use integers (starting with 0, of course) as the index/keys.

16.2 Missing values

Let's change the example to a different family, and a slightly bigger `DataFrame`. The "`simpsons.csv`" file is reproduced below. Do you notice anything odd about it?

```
name,species,age,gender,fave,IQ,hair,salary
Homer,human,36,M,beer,74,,52000
Marge,human,34,F,helping others,120,stacked tall,
Bart,human,10,M,skateboard,90,buzz,
Lisa,human,8,F,saxophone,200,curly,
Maggie,human,1,F,pacifier,100,curly,
SLH,dog,4,M,,,shaggy,
```

What I mean is the positioning of some of the commas. The sharp-eyed reader will see a "double comma" in Homer's row. Even a

[1] With apologies to boxing legend George Foreman, who named all four of his sons "George."

dull-eyed reader will notice several commas in a row in SLH's[2] row. And nearly *every* row (the exception being Homer's) *ends* with a comma, which just looks messed up.

This weird punctuation implies the existence of **missing values**, which means just what it sounds like: there's simply no data for certain columns of certain rows. Homer doesn't have a "`hair`" value, no one *but* Homer has a "`salary`" value, and SLH is missing all kinds of stuff.

When we read this into a Pandas `DataFrame` a la:

```
simpsons = pd.read_csv("simpsons.csv").set_index('name')
```

the result looks like this:

```
        species  age gender            fave     IQ          hair   salary
name
Homer     human   36      M            beer   74.0           NaN  52000.0
Marge     human   34      F  helping others  120.0  stacked tall      NaN
Bart      human   10      M      skateboard   90.0          buzz      NaN
Lisa      human    8      F       saxophone  200.0         curly      NaN
Maggie    human    1      F        pacifier  100.0         curly      NaN
SLH         dog    4      M             NaN    NaN        shaggy      NaN
```

The missing values come up as `NaN`'s, the same value you may remember from p. 114. The monker "not a number" makes sense for the `salary` case, although I think it's a bit weird for Homer's `hair` (not a number? is hair *supposed* to be a number?...) At any rate, we can expect that this will be the case for many real-world data sets.

"Missing" can mean quite a few subtly different things, actually. Maybe it means that the value for that object of study was collected, but lost. Maybe it means it was never collected at all. Maybe it means that variable doesn't really make *sense* for that object, as in the case of a dog's IQ. Ultimately, if we want to use the other values in that row, we'll have to come to terms with what the missing

[2]The Simpson's dog was named "Santa's Little Helper."

values *mean*. For now, let's just learn a couple of coarse ways of dealing with them.

One (sometimes) handy method is `.dropna()`. If you call it, it will return a modified copy of the `DataFrame` in which any row with an `NaN` is removed. This turns out to be overkill in the Simpson's case, though:

```
print(simpsons.dropna())
```

```
Empty DataFrame
Columns: [species, age, gender, fave, IQ, hair, salary]
Index: []
```

In other words, nothing's left. (Every row had at least one `NaN` in it, so nothing survived.)

We could pass an optional argument to `.dropna()` called "`how`", and set it equal to `"all"`: in this case only rows with *all* `NaN` values are removed. Sometimes that's "underkill," as in our Simpson's example: after all, none of the rows are *entirely* `NaN`'s, so calling `.dropna(how="all")` would leave everything intact.

Another option is the `.fillna()` method, which takes a "default value" argument: any `NaN` value is replaced with the default in the modified copy returned. Let's try it with the string `"none"` as the default value:

```
print(simpsons.fillna("none"))
```

```
        species  age gender            fave    IQ          hair salary
name
Homer     human   36      M            beer    74          none  52000
Marge     human   34      F  helping others   120  stacked tall   none
Bart      human   10      M      skateboard    90          buzz   none
Lisa      human    8      F       saxophone   200         curly   none
Maggie    human    1      F        pacifier   100         curly   none
SLH         dog    4      M            none  none        shaggy   none
```

This is possibly useful, but in this case it's not a perfect fit because different columns call for different defaults. The `fave` and `hair` columns could well have "`none`" (indicating no favorite thing, and no hair, respectively) but we might want the default `salary` to be 0. The way to accomplish that is to change the individual columns of the `DataFrame`. Here goes:

```
simpsons['salary'] = simpsons['salary'].fillna(0)
simpsons['IQ'] = simpsons['IQ'].fillna(100)
simpsons['hair'] = simpsons['hair'].fillna("none")
print(simpsons)
```

```
        species  age gender            fave     IQ          hair   salary
name
Homer     human   36      M            beer   74.0          none  52000.0
Marge     human   34      F  helping others  120.0  stacked tall      0.0
Bart      human   10      M      skateboard   90.0          buzz      0.0
Lisa      human    8      F       saxophone  200.0         curly      0.0
Maggie    human    1      F        pacifier  100.0         curly      0.0
SLH         dog    4      M             NaN  100.0        shaggy      0.0
```

Here we've assumed that the default IQ, for someone who hasn't taken the test, is 100 (the average). I left the `NaN` in `fave` as is, since that seemed appropriate.

By the way, that code is actually more than it may appear at first. When we execute a line like:

```
simpsons['salary'] = simpsons['salary'].fillna(0)
```

we're really saying "please *replace* the `salary` column of the `simpsons DataFrame` with a new column. That new column should be – wait for it – the existing `salary` column but with zeros replacing the `NaN`'s."

We'll see many more cases of changing `DataFrame` columns wholesale in the following chapters.

16.3 Removing rows/columns

Finally, after reading a `.csv` file into a `DataFrame`, there are times when you want to manually delete certain rows and/or columns that are not going to be of interest.

The easiest syntax for deleting a row (say, Santa's Little Helper) is:

```
simpsons = simpsons.drop('SLH')
```

The `.drop()` method takes the index of the undesired row as an argument, Like most of the methods we've seen so far, it returns a modified copy of the `DataFrame` it's called on, so you have to reassign this to the original variable (or use the `inplace=True` argument).

You can even delete multiple rows at the same time by enclosing the undesired indices in boxies:

```
simpsons = simpsons.drop(['Homer','Marge','SLH'])
```

Deleting a column is even more common, since many tables "in the wild" have many, many columns, only a few of which you may care about in your analysis. You can whack one entirely with the `del` operator, just like we did for `Series`es (p. 111):

```
del simpsons['IQ']
```

Chapter 17

Tables in Python (2 of 3)

It's easy to get tripped up on Pandas' syntax for accessing the individual bits of `DataFrame`s. First, let's talk about rows and columns, and then we'll talk about the atomic elements ("cells") themselves.

17.1 Accessing individual rows and columns

Suppose you have a `DataFrame` called `df`. Here's how you can extract particular rows and columns:

- `df.loc[`*i*`]` – access the **row** with **index** *i*
- `df.iloc[`*n*`]` – access **row** number *n*
- `df[`*c*`]` – access **column** *c*

The second of these is reminiscent of the `.iloc` syntax we learned for `Series`es on p. 112. With it, we specify the *number* we want, rather than the index/key/label. That's not super common to do, but it happens. More common is the first form: we specify the row we want by its index.

The last one is tricky, because everyone (including me, several times a week, it seems) assumes that just typing (say) "`df['Bart']`" would give you `Bart`'s row. This is probably how it *ought* to work, since `Series`es worked that way. Alas, no: if you specify neither `.loc` nor `.iloc`, you're asking for a *column*, not a row.

Yet another odd thing is how a single row is presented on the screen. Let's go back to the `simpsons` data set (bottom of p. 174), and access the `Bart` row the proper way (with `.loc`):

```
print(simpsons.loc['Bart'])
```

```
species          human
age                 10
gender               M
fave        skateboard
IQ                   90
hair               buzz
salary                0
Name: Bart, dtype: object
```

This bugs the heck out of me. Bart, like all other Simpsons, was a *row* in the original `DataFrame`, but here, it presents Bart's information vertically instead of horizontally. I find it visually jarring. The reason Pandas does it this way is that *each row of a* ***`DataFrame`*** *is a* ***`Series`***, and the way Pandas displays `Series`es is vertically. We'll deal somehow.

Btw, for any of the three options, you can provide a *list* with multiple things you want, instead of just one thing. You do so by using *double* boxies:

- `df.loc[[`*i1,i2,i3,...*`]]` – access the row**s** with indices *i1, i2, i3, etc.*
- `df.iloc[[`*n1,n2,n3,...*`]]` – access the row**s** numbered *n1, n2, n3, etc.*
- `df[[`*c1,c2,c3,...*`]]` – access the column**s** names *c1, c2, c3, etc.*

Examples

To test your understanding of all of the above, confirm that you understand the following examples:

```
print(simpsons.iloc[3])
```

```
species        human
age                8
gender             F
fave       saxophone
IQ               200
hair           curly
salary             0
Name: Lisa, dtype: object
```

```
print(simpsons['age'])
```

```
name
Homer     36
Marge     34
Bart      10
Lisa       8
Maggie     1
SLH        4
Name: age, dtype: int64
```

```
print(simpsons.loc[['Lisa','Maggie','Bart']])
```

```
        species  age gender        fave     IQ   hair  salary
name
Lisa     human    8      F   saxophone  200.0  curly     0.0
Maggie   human    1      F    pacifier  100.0  curly     0.0
Bart     human   10      M  skateboard   90.0   buzz     0.0
```

```
print(simpsons.iloc[[1,3,4]])
```

```
        species  age gender            fave     IQ          hair  salary
name
Marge    human   34      F  helping others  120.0  stacked tall     0.0
Lisa     human    8      F       saxophone  200.0         curly     0.0
Maggie   human    1      F        pacifier  100.0         curly     0.0
```

```
print(simpsons[['age','fave','IQ']])
```

```
         age            fave     IQ
name
Homer     36            beer   74.0
Marge     34  helping others  120.0
Bart      10      skateboard   90.0
Lisa       8       saxophone  200.0
Maggie     1        pacifier  100.0
SLH        4             NaN   30.0
```

Incidentally, you'll notice how the `name` values are treated differently from all the other columns, since `name` is the `DataFrame`'s index. For one thing, `name` *always* appears, even though it's not included among the columns we asked for. For another, it's listed at the bottom of the single-row `Series` listings rather than up with the other values in that row.

17.2 Accessing individual elements

I mentioned above the eternal truth that *each row of a `DataFrame` is a `Series`*. Once you grasp this, you'll realize that you can access an individual "cell" of a `DataFrame` simply by getting the row you want, and then getting the specific value from that. A two-step process for doing this would be:

```
lisas_row = simpsons.loc['Lisa']
lisas_iq = lisas_row['IQ']
print(lisas_iq)
```

```
200.0
```

But a shorter, one-stepper just combines these two operations on the same line:

```
lisas_iq = simpsons.loc['Lisa']['IQ']
print(lisas_iq)
```

```
200.0
```

17.3 Accessing a DataFrame's metadata

We can get some meta-information about a `DataFrame` without even looking at individual rows. If we want to know what the index values themselves are, we use `.index`:

```
print(simpsons.index)
```

```
Index(['Homer', 'Marge', 'Bart', 'Lisa', 'Maggie', 'SLH'],
    dtype='object', name='name')
```

That weird-looking output tells us several things. First, the index of this `DataFrame` consists of `string`s (remember from p. 71 that's what "`dtype='object'`" means). Second, the *name* of the index column is, ironically, "`name`". (It could be named anything at all, of course.) Third, the actual index values are `Homer`, `Marge`, and all the rest.

That's the index, or the "row names," if you will. To get the column names, we use `.columns`:

```
print(simpsons.columns)
```

```
Index(['species', 'age', 'gender', 'fave', 'IQ', 'hair',
    'salary'], dtype='object')
```

Interestingly, this too is an "`Index`" beast, also comprised of `strings`. Pandas treats both "axes" of a `DataFrame` similarly, in that both of them are the same type of thing (an "`Index`"). Notice that `name` is not present in the column names list, because as the `DataFrame`'s index it's a different sort of thing.

How many rows does a `DataFrame` have? This is answerable by using the `len()` function again:

```
print(len(simpsons))
```

```
6
```

This is our third use of the word `len()`: it can be used to find the number of characters in a `string`, the number of key/value pairs of a `Series`, and (here) the number of rows of a `DataFrame`.

Finally, we often want to get a quick sense of how large a `DataFrame` is, both in terms of rows and columns. The `.shape` syntax is handy here:

```
print(simpsons.shape)
```

```
(6, 7)
```

This tells us that `simpsons` has six rows and seven columns. As I mentioned previously (p. 56) this is definitely not the typical case: most `DataFrames` will have many more rows (thousands or even millions) than columns (at most, dozens).

17.4 Sorting `DataFrames`

Sorting a `DataFrame` is largely like sorting a `Series`, except we have more choices: instead of just the keys and the values, we have the index and potentially *many* different columns.

The `.sort_index()` method works just like it did for `Series`es:

```
print(simpsons.sort_index())
```

```
        species  age gender            fave     IQ          hair   salary
name
Bart      human   10      M      skateboard   90.0          buzz      0.0
Homer     human   36      M            beer   74.0          none  52000.0
Lisa      human    8      F       saxophone  200.0         curly      0.0
Maggie    human    1      F        pacifier  100.0         curly      0.0
Marge     human   34      F  helping others  120.0  stacked tall      0.0
SLH         dog    4      M             NaN   30.0        shaggy      0.0
```

The result is rows sorted alphabetically by name. And I hate to keep repeating myself, but remember that `.sort_index()` returns a modified copy, unless you pass the `inplace=True` argument. The `ascending=False` argument is also allowed, and will sort by the index highest-to-lowest instead of lowest-to-highest.

To sort by one of the columns, we call `.sort_values()` and pass it the column name:

```
print(simpsons.sort_values('IQ')
```

```
        species  age gender            fave     IQ          hair   salary
name
SLH         dog    4      M             NaN   30.0        shaggy      0.0
Homer     human   36      M            beer   74.0          none  52000.0
Bart      human   10      M      skateboard   90.0          buzz      0.0
Maggie    human    1      F        pacifier  100.0         curly      0.0
Marge     human   34      F  helping others  120.0  stacked tall      0.0
Lisa      human    8      F       saxophone  200.0         curly      0.0
```

Sometimes we want to include more than one column in the sort. Why? As a tie-breaker. Consider sorting a roster for a student club, which has `first_name` and `last_name` columns, among other things. We might want to sort the list alphabetically by last name, but for students with the same last name, we should go to the first name as a tie-breaker (so that `Angela Smith` shows up after `Velma Patterson` but before `Brad Smith`).

To do this, we pass a list of columns, instead of a single column:

```
print(simpsons.sort_values(['gender','hair','IQ']))
```

```
        species  age gender            fave     IQ          hair   salary
name
Maggie   human    1      F        pacifier  100.0         curly      0.0
Lisa     human    8      F       saxophone  200.0         curly      0.0
Marge    human   34      F  helping others  120.0  stacked tall      0.0
Bart     human   10      M      skateboard   90.0          buzz      0.0
Homer    human   36      M            beer   74.0          none  52000.0
SLH        dog    4      M             NaN   30.0        shaggy      0.0
```

Here, we said "sort the rows alphabetically by `gender`. For rows with the same `gender`, use `hair` as a tie-breaker. And for rows with the same `gender` *and* the same `hair`, use `IQ` as a second tie-breaker." Glance at that output and convince yourself that it's correct.

We control the "ascendingness" of the multi-column sort by specifying a list of *each* `ascending` value, one for each column we're sorting by. Consider this:

```
print(simpsons.sort_values(['gender','hair','IQ'],
    ascending=[False,True,False]))
```

```
        species  age gender            fave     IQ          hair   salary
name
Bart     human   10      M      skateboard   90.0          buzz      0.0
Homer    human   36      M            beer   74.0          none  52000.0
SLH        dog    4      M             NaN   30.0        shaggy      0.0
Lisa     human    8      F       saxophone  200.0         curly      0.0
Maggie   human    1      F        pacifier  100.0         curly      0.0
Marge    human   34      F  helping others  120.0  stacked tall      0.0
```

Now we're saying "sort *reverse* alphabetically by `gender`, breaking ties by comparing `hair` alphabetically, and breaking further ties by `reverse` sorted order by IQ."

Oh, and the `inplace=True` argument works for all these examples as well.

17.5 Summary statistics for DataFrames

Summary statistics like the mean, median, minimum/maximum, and the like, can of course all be computed on individual columns of a `DataFrame`, because each column is a `Series`:

```
print(simpsons['IQ'].median())
```

```
95.0
```

```
print(simpsons['salary'].sum())
```

```
52000.0
```

You can also, believe it or not, compute the sum/mean/max/etc on the entire `DataFrame`. This computes it on every column individually:

```
print(simpsons.mean())
```

```
age         15.500000
IQ         102.333333
salary    8666.666667
dtype: float64
```

Pandas left out the non-numeric columns (`species`, `gender`, *etc.*) and computed the mean of each of the others, giving us a `Series` containing their values.

Finally, I often find the `.describe()` method useful:

```
print(simpsons.describe())
```

```
       count   6.000000    6.000000      6.000000
       mean   15.500000  102.333333   8666.666667
       std    15.436969   56.645094  21228.911104
       min     1.000000   30.000000      0.000000
       25%     5.000000   78.000000      0.000000
       50%     9.000000   95.000000      0.000000
       75%    28.000000  115.000000      0.000000
       max    36.000000  200.000000  52000.000000
```

Neat! We get the number of values, the mean, the standard deviation, and all the quartiles for each of the numeric columns. Lots of dashboard information at a glance!

Chapter 18

Tables in Python (3 of 3)

18.1 Queries

Back in section 13.1 (p. 124), we learned how to write simple **queries** to selectively match only certain elements of a `Series`. The same technique is available to us with `DataFrame`s, only it's more powerful since there are more columns to work with at a time.

Let's return to the Simpsons example from p. 174, which is reproduced here:

```
        species  age gender            fave     IQ          hair   salary
name
Homer     human   36      M            beer   74.0          none  52000.0
Marge     human   34      F  helping others  120.0  stacked tall      0.0
Bart      human   10      M      skateboard   90.0          buzz      0.0
Lisa      human    8      F       saxophone  200.0         curly      0.0
Maggie    human    1      F        pacifier  100.0         curly      0.0
SLH         dog    4      M             NaN  100.0        shaggy      0.0
```

We can filter this on certain rows by including a query in boxies:

```
adults = simpsons[simpsons.age > 18]
```

```
       species  age gender            fave     IQ          hair   salary
name
Homer    human   36      M            beer   74.0          none  52000.0
Marge    human   34      F  helping others  120.0  stacked tall      0.0
```

As with Seriesses, we can't forget to repeat the name of the variable ("simpsons") before giving the query criteria ("> 18"). Unlike with Seriesses, we also specify a column name (".age") that we want to query.

We can also provide compound conditions in just the same way as before (section 13.1, p. 128). If we want only human children, we say:

```
kids = simpsons[(simpsons.age <= 18) & (simpsons.species == "human")]
```

```
        species  age gender        fave     IQ   hair  salary
name
Bart      human   10      M  skateboard   90.0   buzz     0.0
Lisa      human    8      F   saxophone  200.0  curly     0.0
Maggie    human    1      F    pacifier  100.0  curly     0.0
```

whereas if we want everybody who's smart and/or old, we say:

```
old_andor_wise = simpsons[(simpsons.IQ > 100) | (simpsons.age > 30)]
```

```
       species  age gender            fave     IQ          hair   salary
name
Homer    human   36      M            beer   74.0          none  52000.0
Marge    human   34      F  helping others  120.0  stacked tall      0.0
Lisa     human    8      F       saxophone  200.0         curly      0.0
```

To narrow it down to only specific columns, we can combine our query with the syntax from section 17.1 (p. 177). You see, our query gave us another (shorter) DataFrame as a result, which has the same rights and privileges as any other DataFrame. So tacking on another pair of boxies gives us just a column:

```
print(simpsons[simpsons.age > 18]['fave'])
```

```
name
Homer               beer
Marge    helping others
Name: fave, dtype: object
```

while tacking on *double* boxies gives us columns:

```
print(simpsons[simpsons.age > 18][['fave','gender','IQ']])
```

```
                  fave gender     IQ
name
Homer             beer      M   74.0
Marge  helping others      F  120.0
```

Note that in the first of these cases, we got a *Series* back, whereas in the second (with the double boxies) we got a DataFrame with multiple columns.

Combining all these operations takes practice, but lets you slice and dice a DataFrame up in innumerable different ways.

18.2 The .groupby() method

One of the most useful methods in the whole DataFrame repertoire is .groupby(). It applies when you want summary statistics (mean, quantile, max/min, *etc.*) not for the *whole* data set, but for each *subset* of the data set, where the subsets split on the values of one of the variables.

Here's an example in action. It's old news that we could find, say, the median IQ of the Simpsons family overall:

```
print(simpsons['IQ'].median())
```

```
95.0
```

But it's new news that we can do this for each gender separately, via:

```
print(simpsons.groupby('gender')['IQ'].median())
```

```
gender
F    120.0
M     74.0
Name: IQ, dtype: float64
```

We give a *categorical* variable as the argument to `.groupby()`, and specify a *numeric* variable as the column we wish to analyze. Finally, we choose the summary statistic we want (`.median()` in the above case).

All this produces a resulting ***`Series`***. Think hard: the *keys* of the resulting `Series` are the *values* of the categorical variable (in the original `Series`) that we grouped by; and the *values* of the resulting `Series`, are the results of applying the summary statistic function to each of the subsets separately.

So now, in addition to knowing that the overall Simpson family median IQ is 95, we also know that among Simpson boys and men, it's only 74, whereas among girls and women, it's an impressive 120.

Another example: let's find the maximum age for each hair style:

```
print(simpsons.groupby('hair')['age'].max())
```

```
hair
buzz            10
curly            8
none            36
shaggy           4
stacked tall    34
Name: age, dtype: int64
```

(Since there's so many different hairstyles present, Maggie turns out to be the only one whose age is not represented here.)

18.3 Looping with `DataFrames`

Just as we wrote `for` loops to iterate through the elements of a NumPy array (section 14.3) and a Pandas `Series` (section 14.4), so we can iterate through the rows of a `DataFrame`. We'll do so using the weirdly-named `.itertuples()` method.

By using `.itertuples()` in the loop header, we have available to us in the loop body a `Series` representing *the current row.* ("Current row" just means "the row we're on as we successively go through all the rows in sequence.") We can access individual elements of it by using column names and the dot (or boxie) syntax, as follows:

```
for row in simpsons.itertuples():
    print("A certain {}-year old has {} hair.".format(row.age,
        row.hair))
```

```
A certain 36-year old has none hair.
A certain 34-year old has stacked tall hair.
A certain 10-year old has buzz hair.
A certain 8-year old has curly hair.
A certain 1-year old has curly hair.
A certain 4-year old has shaggy hair.
```

I called the loop variable "`row`" because it represents a row of the `DataFrame` (duh), although you can call it anything you want to. (I could have named it "`family_member`" or "`Simpson`" instead.)

This is very intuitive. Slightly less intuitive is that if we want the index column (in `simpsons`, it's called "`name`," remember) we can't use the name of the index column. Instead, we have to literally say "`.Index`," and yes that's a *capital* '`I`'.

To illustrate:

```
for family_member in simpsons.itertuples():
    print("{} Simpson, {} years of age, has {} hair.".format(
        family_member.Index, family_member.age, family_member.hair))
```

```
Homer Simpson, 36 years of age, has none hair.
Marge Simpson, 34 years of age, has stacked tall hair.
Bart Simpson, 10 years of age, has buzz hair.
Lisa Simpson, 8 years of age, has curly hair.
Maggie Simpson, 1 years of age, has curly hair.
SLH Simpson, 4 years of age, has shaggy hair.
```

Chapter 19

Exploratory Data Analysis: bivariate (1 of 2)

In this chapter, we'll extend our EDA repertoire to cover **bivariate data**, which means studying the relationships between *pairs* of variables, rather than focusing only on one variable at a time. This is where most of the action is: you'll be awed and impressed by how much more we can dig out of a data set in this chapter.

Bivariate data analysis is especially suited to the **table**s (in Python, `DataFrame`s) from section 7.1 and chapters 16–18. This is because each column of a table is a variable that matches one-for-one with every *other* column in the table.

In the Simpsons example (p. 174), the fourth `species` value corresponds to Lisa, as does the fourth `age` value, the fourth `fave` value, the fourth `gender` value, the fourth `fave` value, the fourth `IQ` value, the fourth `hair` value, and the fourth `salary` value. This means that if we examine any two columns, we know that matching indices go together (*i.e.*, represent the same person). This implicit connection is what allows us to meaningfully examine a pair of variables.

19.1 The concept of statistical significance

Before we get to the details, we need to face head-on what is probably the single most important concept in statistics, that of **statistical significance** (or "stat sig," for short). It is so immensely important that I'm going to ask you to put down whatever snack you're eating right now, fold your hands, and pay very close attention.

All forms of bivariate analysis are variations on a single theme, namely: discovering whether or not an *association* exists between two variables. Recall from section 10.2 (p. 91) that an association means that two variables are correlated in some way: that certain values of one tend to go more often with certain values of the other. To make it concrete, let's say one of our variables is `sex` (at birth, male or female) and the other is `height` (in inches, say). We want to know: "are taller people more often male, and shorter people more often female, or is there no connection between `sex` and `height`?"

Now the first thing you think of doing, of course, is getting a **sample** (recruiting volunteers, say) of both males and females, measuring their heights, and taking the average (mean). Let's say you do that, and you come up with the following numbers:

Females – average height: 65.5 inches
Males – average height: 69.3 inches

Clearly, in your *sample*, males were on average somewhat taller – 3.8 inches taller, in fact. A careless thinker would immediately conclude: "aha! My hypothesis is confirmed. I scientifically carried out my study, and mathematically computed the results, and now here is some hard data proving the conclusion that generally speaking, men tend to be taller than women."

Are you convinced by that reasoning?

I hope you're not. Here's why. Let's change the example, and suppose that instead of height, we measured our volunteers' IQ. Taking the averages as before, we come up with these numbers:

Females – average IQ: 102.4
Males – average IQ: 98.6

In this case, the average of the females in the sample was higher than the males was. Shall we conclude that in general, women tend to be smarter than men?

Confirmation bias

If you're like most people, you'll accept that first finding as confirmation of men's tallness, and you'll reject the second finding as just a fluke of the sample. Undoubtedly, this is because you went into the question already having an opinion about the matter. You just *know in your heart* that men *do* tend to be taller than women (you've observed thousands of both sexes, in fact, and have in fact noticed that trend) whereas you know in your heart that neither sex has an advantage in intelligence (ditto). This leads you to reason as follows:

1. "Well of course my male volunteers were taller than the female ones. I've known all along that males tend to be taller in general, and this just confirms it!"
2. "Aw, c'mon, we only sampled a few people and measured their IQs. Sure, these particular women might have been a bit smarter than these particular men, but if I ran the experiment again on different volunteers, it might just as easily go the other way. It'd be silly to draw a grand conclusion from that."

Psychologists call this fallacy of reasoning "**confirmation bias**." We have a natural tendency to interpret information in a way that affirms our prior beliefs. Data that seems to contradict it, we simply talk our way out of.

Confirmation bias is one of the most insidious enemies of humankind. It leads to wrong reasoning, the entrenchment of beliefs, dangerous overconfidence, polarization, and in the worst cases, **groupthink**. When a group of people succumbs to groupthink, "orthodox" viewpoints are encouraged, while alternative viewpoints are dismissed and suppressed. Every piece of evidence that conforms

to the group's consensus belief is hailed as evidence confirming it, and evidence that contradicts it is chalked up to mere statistical anomaly.

One of many examples of this phenomenon was the CIA circa 2002: from the top to the bottom, nearly every member of the organization was *certain* that Sadaam Hussein's terrible regime in Iraq possessed weapons of mass destruction (WMDs). Later, when it was inexplicably discovered that this "fact" wasn't true after all (*after* we had made irreversible decisions based on it), analysts pored over the CIA's decision-making process to try and make sense of it. Confirmation bias was perhaps the key ingredient.

Be aware of it in your own thinking, and at all costs steer yourself away from it!

The perils of eyeballing it

Okay. Let's suppose that we've freed ourselves from confirmation bias, and we're actually looking at the numbers objectively. The first question still remains: *does* an association exist between the two variables?

Men were an average of 3.8 inches taller in our sample. That's a difference...but is it *enough* of a difference? Women were smarter by 3.8 IQ points. That's a difference...but is it *enough*?

To clarify, when we say "enough," what we mean is: "*enough of a difference to generalize our findings to the population at large.*" Here's a paradox: what we have is a sample; yet oddly, it's never the sample we actually care about. We care about what the sample tells us *about the* ***population***.

Think about it. Nobody cares whether the 14 females we surveyed are smarter on average than the 17 males we surveyed. But if we make the claim: "*in general* women are smarter than men," suddenly everybody cares. To use another example, nobody cares that 58% of the 2,000 people in our phone sample said they intend to vote for Elizabeth Warren, and only 39% said Donald Trump. But if we say "*across the country*, we predict more Warren votes than Trump votes by such-and-such margin," this is big news.

Now the first law to beat into your head is that *you absolutely cannot reliably eyeball it.* This is what everyone who hasn't taken a Data Science or Statistics class tries to do. They squint at the difference (3.8 IQ points, *e.g.*) and bite their lip and mutter, "well, that sure *seems* (or *doesn't seem*) like a pretty big difference. I'll bet this says (or doesn't say) something about intelligence among the sexes in general."

Stop. You cannot. People are demonstrably very bad at judging whether or not a difference between groups is "enough." Part of the problem is that the answer to the question turns on three separate things: how big the difference is, how large your sample size is, and – importantly – how variable the data is (meaning, how widely the points you sample differ from each other). All three of these factors need to be mixed into a soup in just the right way in order to properly judge, and human intuition is just flat terrible at doing that.

So eyeballing is a non-starter. But happily, it turns out that statistics provides us an iron-clad, dependable, quantitative, take-it-to-the-bank method for determining whether the pattern in a data set is "enough" to justifiably claim an association between variables. And that is the concept of statistical significance.

A "statistically significant difference"

The correct way to determine whether a difference is "enough" is to use the appropriate **statistical test**. A statistical test is a standard procedure for incorporating all three factors I previously mentioned (the degree of difference, the sample size, and the amount of variance among data points) to come up with a principled, defensible answer to this question: is the pattern I see in my sample a *statistically significant* one? Can we be reasonably confident that it will also be true of the population as a whole?

All the statistical tests we'll learn (in the next chapter) have a common output: a ***p*-value**. That's nice because you don't have to memorize a lot of different rules for interpreting a lot of different tests.

So what the heck is a "p-value?" There have been controversies galore[1], and even entire books[2] devoted to the subject, which means that whatever I choose to write here can be nitpicked by statisticians a dozen different ways.

That's okay. I'm going to write it anyway, and this will serve you very well. Here goes:

A p-value is a number between 0 and 1. If you run a statistical test on your bivariate data, and the p-value is ***less than*** your α ("alpha") value (normally, .05), then there ***is*** a statistically significant association between the variables. Otherwise, there isn't. End of story.

Recall from section 10.5 (p. 102) that α is "where to set the bar" to detect a meaningful association. It's essentially how often we're willing to draw a false conclusion. For social science data (that is, data involving humans), you should always choose .05 to avoid controversy. For physical science data, you should always choose .01.

The bottom line is this: if you spot a possible relationship between two of your variables (like gender and IQ), run the appropriate statistical test (see next chapter) and look at the p-value. If it's less than α, then the difference you thought you saw officially *is* "enough." You can therefore declare "yep, these two variables *are* associated, to a confidence level of α." If it's not less than α, then even though you thought you saw a meaningful tendency in the data, you can officially say, "nope, it's not a stat sig diff. This is very likely just an artifact of the particular data points I collected in my sample. Pay of no mind."

[1] For instance:

- Colquhoun D (2017). "p-values." *Royal Society Open Science.* **4**(12): 171085.
- Murtaugh, Paul A. (2014). "In defense of p-values." *Ecology.* **95**(3): 611–617
- Wasserstein, Ronald L.; Lazar, Nicole A. (2016). "The ASA's Statement on p-Values: Context, Process, and Purpose." *The American Statistician*, **70**(2): 129–133

[2] Vickers, A. (2009). *What is a p-value anyway?* Boston: Pearson.

19.2 Moving on

Which statistical test is appropriate depends on your two variables' scales of measure: in particular, whether they are categorical or numeric. There are three scenarios for bivariate analysis: two categorical variables, two numeric variables, or one of each. In the next chapter, in addition to learning how to meaningfully plot all three cases, we'll learn how to run and interpret the statistical test applicable to each case, in order to determine once and for all whether "enough" is *enough*.

Chapter 20

Exploratory Data Analysis: bivariate (2 of 2)

20.1 Three bivariate scenarios

As we saw with univariate data in chapter 15, different kinds of plots and statistics are appropriate depending on the variable's scale of measure – categorical or numeric. There are thus three different cases for bivariate analysis:

- Two categorical variables
- One categorical variable and one numeric variable
- Two numeric variables

We'll consider each case in turn. Throughout all the remaining sections, we'll use this fictitious data set, called `people`:

```
   gender  salary   color  followers
0    male   54.94  purple         26
1  female   72.48  purple         22
2    male    9.47    blue         27
3   other   60.08     red         22
4    male   37.62     red         13
                .
                .
```

Each row represents one fictional person we interviewed, and in-

cludes their `gender`, their `salary` (in thousands of dollars per year), their favorite `color`, and the number of `followers` they have on some unspecified social media website.

The `DataFrame` has 5000 rows, and no special "index" variable: none of the columns that we collected are unique, so we just let Pandas default to indexing the rows by number, 0 through 4,999.

20.2 Importing `scipy.stats`

All of the statistical tests we'll demonstrate in this chapter come from the **SciPy** Python package (pronounced "sigh pie.") SciPy is huge, and has several different parts; for the time being, we'll only be using the "`stats`" component. Therefore, we need one additional import statement:

```
import scipy.stats
```

You can include this in a cell at the top of your Jupyter Notebook just like your `numpy` and `pandas` imports.

20.3 Two categorical variables

Okay. Let's return our attention to the `people` `DataFrame`, and begin with a bivariate analysis of the `gender` and `color` columns. The first thing we should do, of course, is inspect each one individually, using `.value_counts()` and perhaps a bar chart from sections 15.1 and 15.4. Let's say we've done that.

The next obvious question: is there an *association* between the two variables? In other words, are there particular values of one that tend to go with particular values of the other? In still other words, do people of different genders tend to have different favorite colors?

Contingency tables

The first tool to get at this question is called a **contingency table**. This is very much like `.value_counts()`, but for two variables instead of one. Our function is `crosstab()` from the Pandas package: if we give it two columns as arguments, it computes the complete set of counts from all possible combinations of variables. Here's what it looks like:

```
pd.crosstab(people.gender, people.color)
```

```
color   blue  green  pink  purple  red  yellow
gender
female   240    402   665     644  289     378
male    1403      0     0     248  463     258
other      1      2     2       2    1       2
```

Interpreting this is straightforward. Every cell in the matrix tells us how many people had a particular gender and a particular favorite color. For instance, there were 378 females who named `yellow` as their favorite color, and no males at all chose `green`.

Plotting two categorical variables

So now we have a table of counts – how to turn this into a pretty and informative plot?

Unfortunately, there doesn't seem to be any great way to do this. There's something called a "mosaic plot" which attempts it, but they're not very easy to visually interpret. Another option is a "heat map," which essentially reproduces the above table as squares in a grid, with each square color coded on a continuum by its height (for instance, low numbers might be dark blue and high numbers bright yellow, with a rainbow spectrum of number in between). That's sort of okay, but to be honest I prefer to just look at the numbers.

The χ^2 test

The statistical test to use for two categorical variables is called the χ^2 **test** (pronounced "kai-squared," not "chai-squared," by the way). To run it, it's convenient to first store the contingency table itself as a variable. I'll call it `gender_color` since it's a table of the genders of people and their favorite colors:

```
gender_color = pd.crosstab(people.gender, people.color)
```

Now, we run the test by calling the `chi2_contingency()` function from SciPy:

```
scipy.stats.chi2_contingency(gender_color)
```

```
(2125.8933435, 0.0, 10, array([[8.60798e+02, 2.11534e+02,
    3.49241e+02, 4.68098e+02, 3.94270e+02, 3.34056e+02],
   [7.79913e+02, 1.91657e+02, 3.16424e+02, 4.24113e+02,
    3.57223e+02, 3.02667e+02],
   [3.28800e+00, 8.08000e-01, 1.33400e+00, 1.78800e+00,
    1.50600e+00, 1.27600e+00]]))
```

I know, I know: that output is downright hideous. Here's the deal, though: all you have to do is look at *the second number* in that long, banana-and-boxie-laden thing. **The second number is the p-value.** It is 0.0. This is obviously lower than .05 (our α), and therefore, *we can conclude that `gender` and `color` are associated.*

All the other stuff in that output are fine-grained details that statisticians like to pore over. For us, the only thing we need to see from a χ^2 (or any other) test is the p-value.

20.4 One categorical and one numeric variable

Next let's consider the case where we want to test for an association between one categorical and one numeric variable. This is the "`gender` vs. `IQ`" scenario from the last chapter. In the `people` example, we might look at `gender` vs. `salary` to whether different genders earn different amounts of money on average.

Grouped box plots

The best plot for this scenario (IMHO) is the **grouped** box plot. It's the same as the chapter 15 box plots, except that we draw a different box (and pair of "whiskers") for each group.

Here's the command in Pandas:

```
people.boxplot('salary',by='gender')
```

This produces the plot on the left-hand side of Figure 20.1. Refer back to section 15.6 (p. 165) for instructions on how to interpret each part of the box and whiskers. From the plot, it doesn't look like there's much difference between the `male`s and `female`s, although those identifying with neither gender look perhaps to be somewhat of a salary disadvantage.

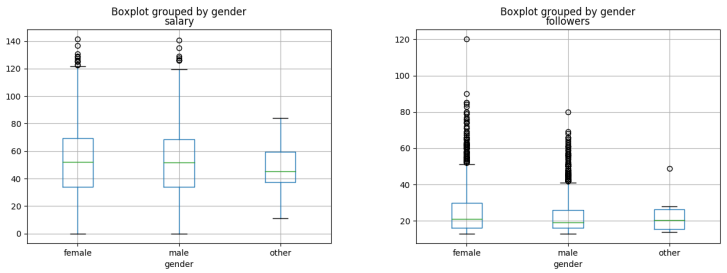


Figure 20.1: Grouped box plots of salary (left) and number of social media followers (right), grouped by gender.

Similarly, we get the plot on the right-hand side with this code:

```
people.boxplot('salary',by='followers')
```

This looks more skewed (`female`s appear to perhaps have more followers on average than `male`s), but of course we won't know for sure until we run the right statistical test.

The *t*-test

The test we'll use for significance here is called the ***t*-test** (sometimes "Student's *t*-test") and is used to determine whether the *means* of two groups are significantly different.[1] Remember, we can get the mean salary for each of the groups by using the `.groupby()` method:

```
people.groupby('gender')['salary'].mean()
```

```
gender
female    52.031283
male      51.659983
other     48.757000
```

Females have the edge over males, 52.03 to 51.66. Our question is: is this "enough" of a difference to justify generalizing to the population?

To run the *t*-test, we first need a `Series` with just the `male` salaries, and a different `Series` with just the `female` salaries. These two `Series`es are *not* usually the same size. Let's use a query to get those:

[1] Strictly speaking, the *t*-test assumes that the data sets you're comparing are "bell curvy" (or "normally distributed," to be precise) and we haven't checked for that here. However, since we're doing *exploratory* data analysis (not drawing up and documenting final conclusions) it's common to use a *t*-test as a quick-and-dirty just to see what's worth investigating.

```
female_salaries = people[people.gender=="female"]['salary']
male_salaries = people[people.gender=="male"]['salary']
```

and then we can feed those as arguments to the `ttest_ind()` function:

```
scipy.stats.ttest_ind(female_salaries, male_salaries)
```

```
Ttest_indResult(statistic=0.52411385896, pvalue=0.60022263724)
```

This output is a bit more readable than the χ^2 was. The second number in that output is labeled "`pvalue`", which is over .05, and therefore we conclude that *there is no evidence that average salary differs between males and females.*

Just to complete the thought, let's run this on the `followers` variable instead:

```
female_followers = people[people.gender=="female"]['followers']
male_followers = people[people.gender=="male"]['followers']
scipy.stats.ttest_ind(female_salaries, male_salaries)
```

```
Ttest_indResult(statistic=9.8614730266, pvalue=9.8573024317e-23)
```

Warning! When you first look at that p-value, you may be tempted to say "9.857 is *waaay* greater than .05, so I guess this is a 'no evidence' result as well." Not so fast! If you look at the entire number – including the ending – you see:

`9.857302431746571e-23`

that sneaky little "`e-23`" at the end is the kicker. This is how Python displays numbers in **scientific notation** The "`e`" means "times-ten-to-the." In mathematics, we'd write that number as:

$$9.857302431746571 \times 10^{-23}$$

which is:

.00000000000000000000000009857302431746571

Wow! That's clearly waaay *less than* .05, and so we can say *the average number of followers does depend significantly on the gender.*

Be careful with this. It's an easy mistake to make, and can lead to embarrassingly wrong slides in presentations. ☺

More than two groups: ANOVA

By the way, the t-test is only appropriate when your categorical variable has *two* values (male vs. female, for example, or vaccinated vs. non-vaccinated). If there are more than two, the appropriate test to run is called an "ANOVA" (ANalysis Of VAriance). It's beyond the scope of this text, but it's described in any intro stats textbook and is eminently Googleable.

20.5 Two numeric variables

Finally, we have the case where both variables are numeric. The `salary` and `followers` columns are this case. Are they associated?

Scatter plots

The correct plot to visualize this is the **scatter plot**. It has an axis for each numeric variable, and plots one dot (or other marker) for each object of study: its x/y coordinates depend on that object's value for each variable.

The Pandas code is as follows:

```
people.plot.scatter(x='followers',y='salary')
```

which produces Figure 20.2. Interestingly, there appear to be a lot of people pegged at zero salary, and also at 10-ish followers. (These observations would have shown up in a univariate analysis as well.) There's no super obvious connection between the two variables, but

if you squint at the plot it (maybe) looks like there's a slight up-and-to-the-right trend, which would indicate that having more followers is modestly associated with earning more money.

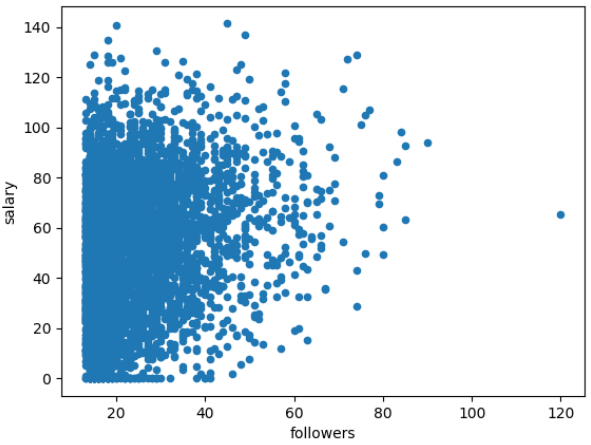


Figure 20.2: A scatter plot of `followers` vs. `salary`. Each point in the plot represents one person, with the x and y coordinates corresponding to his/her/their number of followers and salary.

20.6 Pearson correlation coefficient

The test we'll use to see whether this pattern is significant is the **Pearson correlation coefficient** (also called "Pearson's r"). To run it, we call SciPy's `pearsonr()` function and pass it the two columns:

```
scipy.stats.pearsonr(people.salary, people.followers)
```

```
(0.2007815176819964, 1.2285885030618397e-46)
```

We're given two numbers as output. The *second* of these is the p-value, and remembering our pitfall from above, we're savvy enough to notice the `e-46` at the end and declare it significant. So we can

say *we have high confidence that a person's salary is associated with their number of social media followers.*

Now for the first number, which is the actual "correlation coefficient." *If* the second number is below α and therefore significant (as it was here), you then look at the first number and see whether it's positive or negative. Positive numbers indicate **positive correlations**: an increase in one of the variables corresponds to an *increase* in the other. Negative numbers indicate **negative correlations**: an increase in one of the variables corresponds to a *decrease* in the other. Here, we have a positive number, which means that *having more followers tends to go with a higher salary.*

As an example of the second (negative) case, suppose two of our variables in a data set of sailboat races were `length` (the length of the sailboat, from bow to stern) and `finish_time` (the number of minutes the boat took to complete the race). We're likely to see a negative correlation in this case, because physics tells us that *longer* boats can travel through the water *faster* (and therefore have *lower* finish times). These two variables would thus be correlated, but in a negative way: a high value for one would typically indicate a *low* value for the other.

Chapter 21

Branching

In this chapter, we'll learn our next programming trick: how to execute code *conditionally*. This is called **branching**. It's another variant of non-linear programming, like the loops from chapter 14: it enables something other than the strict, start-to-finish, line-by-line execution of a program. In particular, branching allow us to designate certain lines of code to be executed "only sometimes."

21.1 The `if` statement

The main branching statement in Python and most languages is the `if` **statement**. Here's a couple of them in action:

```
 1: name = "Horace"
 2: cash_on_hand = 100000
 3: IQ = 90
 4: print("Nice to meet you, {}!".format(name))
 5: if cash_on_hand > 5000:
 6:     print("Wow, you're rich! Gimme a fiver.")
 7:     cash_on_hand = cash_on_hand - 5
 8: if IQ > 100:
 9:     print("Wow, you're smart! Read a book.")
10:     IQ = IQ + 5
11: print("{}'s IQ is {} and he has ${}.".format(name,
12:     IQ, cash_on_hand))
```

Even without any explanation, you might be able to figure out that the output of the code snippet above is:

```
Nice to meet you, Horace!
Wow, you're rich! Gimme a fiver.
Horace's IQ is 90 and he has $99995.
```

If not, stay tuned.

Just like a `for` loop, every `if` statement has a header and a body. And just like a `for` loop, the determining factor of which lines constitute the body depends on the indentation:

- ☞ The first `if` statement's header is line **5**.
- ☞ The first `if` statement's body is lines **6 and 7**.
- ☞ The second `if` statement's header is line **8**.
- ☞ The second `if` statement's body is lines **9 and 10**.

When an `if` statement is reached, its **condition** is evaluated; in the first case, the condition "`cash_on_hand > 500`" is evaluated to `True`, and in the second case, "`IQ > 100`" is determined to be `False`. Then, *only if* the condition is true will the body of the `if` statement execute. Otherwise, it'll be skipped over.

Thus, the lines of the above program execute in this order: 1, 2, 3, 4, 5, 6, 7, 8, 11/12. Lines 9 and 10 are skipped entirely, since Horace's IQ wasn't above average. Observe that the `cash_on_hand` variable was updated inside the body of the first `if` statement, but that `IQ` was not.

Compound conditions

Conditions can be more complicated than the ones above; just as with queries (p. 128) they can contain more than one component:

```
if cash_on_hand > 10000 and IQ < 50:
    print("Wow, some dumb people are sure rich!")
```

You might have been surprised to see the word "`and`" in that `if` statement instead of the character "`&`". I feel you. It's totally inconsistent, but nevertheless true: although in a query, you must use the symbols `&`, `|`, and `~`, in an `if` condition, you must use the words `and`, `or`, and `not`. (In other news, the bananas around the components of an `if` condition aren't necessary, but you can include them if you want.)

For your convenience, the `if` condition operators are listed in Figure 21.1. (**Remember** the double-equals!!)

21.2 The `if/else` statement

The above examples execute the `if` body as long as the condition is true, and do nothing otherwise. It's common to want to do something else in the "otherwise" case instead, and for that, we have the `if/else` statement.

```
name = "Gladys"
cash_on_hand = 2000
IQ = 120
print("Nice to meet you, {}!".format(name))
if cash_on_hand > 5000:
    print("Wow, you're rich! Gimme a fiver.")
    cash_on_hand = cash_on_hand - 5
else:
    print("I wish you well!")
if IQ > 100:
    print("Wow, you're smart! Read a book.")
    IQ = IQ + 5
else:
    print("You're currently not that smart. Read a book!")
    IQ = IQ + 10
print("{}'s IQ is {} and she has ${}.".format(name, IQ,
    cash_on_hand))
```

```
Nice to meet you, Gladys!
I wish you well!
Wow, you're smart! Read a book.
Gladys's IQ is 125 and she has $2000.
```

You can see that "`I wish you well!`" was printed. This is because `cash_on_hand` was *not* greater than 5000 (as required by the `if` condition). Also, the "...`you're smart!`..." message was printed but not the "...`not that smart`..." one. Both the `if` part and the `else` part have an indented body, although only the `if` part has a condition.

And that brings up another point. Although it hardly seems worth mentioning, let me nevertheless emphasize this oft-overlooked truth:

Whenever an `if`/`else` statement is reached, ***either* the `if` body *or* the `else` body will always be executed. It's never both, and it's never neither one.**

This is always, always true, because of the nature of things. The reason the `else` header doesn't have a condition is because its condition is implicitly *the exact opposite* of the `if` condition. Period. In *any* case where the `if` condition isn't true – and *only* in such a case – will the **else** condition be executed.

To test whether you fully understand this point, see if you can predict the output of the following program, which 99% of beginning programmers get wrong:

```
name = "Javier"
lang = "French"

if lang == "Spanish":
    print("Hola, {}!".format(name))
if lang == "French":
    print("Bonjour, {}!".format(name))
if lang == "Chinese":
    print("Ni hao, {}!".format(name))
else:
    print("Hello, {}!".format(name))
```

Seriously, don't feel bad if you miss this one. The answer (*drum roll please*) is:

```
Bonjour, Javier!
Hello, Javier!
```

Wait...wut? Why did it print two messages? Surely if `Javier`'s preferred language is `French`, it ought to say "`Bonjour`" and skip all the other options?

To understand this behavior, you have to realize that an `if`/`else` statement is a *single* entity. This multi-lingual greeting program has three components:

1. an `if` statement
2. an `if` statement
3. an `if`/`else` statement

And you must remember our golden rule in the shaded box: ***either* the `if` body *or* the `else` body will always be executed: never both, and never neither one.** Therefore, the above program does this:

1. If the language is Spanish, print an "Hola" message. (Otherwise, do nothing.)
2. If the language is French, print a "Bonjour" message. (Otherwise, do nothing.)
3. If the language is Chinese print a "Ni hao" message. Otherwise, print a "Hello" message.

Once you recognize that structure, you'll realize that when step 3 is encountered, the program *must* print either "Ni hao" or "Hello." It can't print both, and it can't print neither. An `if`/`else` just doesn't work any other way.

21.3 The `if`/`elif`/`else` statement

The problem with the previous example is that we really want our four languages to be **mutually exclusive** options. If `lang` is `"Spanish"`, we want it to print "`Hola`" and skip all the rest. The easiest way to get this behavior is to use "`elif`" (a horrid abbreviation of the phrase "else if").

Operator	Meaning
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to
`!=`	*not* equal to
`==`	equal to
`and`	and
`or`	or
`not`	not

Figure 21.1: Operators in `if` statements: simple and compound.

Squint hard at this program until you see the differences between it and the previous one:

```
name = "Javier"
lang = "French"

if lang == "Spanish":
    print("Hola, {}!".format(name))
elif lang == "French":
    print("Bonjour, {}!".format(name))
elif lang == "Chinese":
    print("Ni hao, {}!".format(name))
else:
    print("Hello, {}!".format(name))
```

It's identical except that we replaced the second two `if`'s with `elif`'s. This tells Python: only if the language is *not* Spanish should you then consider whether or not it's French. And only if it's *not* French (and *not* Spanish) should you consider whether or

not it's Chinese. And only if it's not Chinese (and not French (and not Spanish)) should you print "Hello."

Realize, too, that an entire `if`/`elif`/`elif`/.../`elif`/`else` chain is a *single* statement, no matter how many conditions it has. You can't just have an "`elif`" (or an "`else`," for that matter) floating out in the void without an initial `if` to anchor it. This may help you to understand how the `elif` structure acts, and why it will only ever execute *one* of the bodies. It's because:

Whenever an `if`/`elif`/.../`elif`/`else` statement is reached, **exactly *one* of the bodies will be executed. It's never more than one, and it's never none.**

Whether to use sequential `if`s or a chain of `elif`s isn't always an easy question to answer. Neither choice is always right: you have to think rigorously logically about how the program should act. Ask yourself: "do I want Python to consider *all* of these conditions – and execute the appropriate `if` bodies – no matter what? Or do I want it to bail out as soon as it finds one that's true?" Like most things, it takes practice to get right.

21.4 Nesting

Now as if this weren't complex enough, let me inform you that the body of an `if` (or `else`, or `elif`) statement can *itself* contain other `if` statements! This is actually quite common. Consider this example:

```
first_name = "Emma"
last_name = "Watson"
gender = "female"
marital_status = "single"
degree = "BA"

if degree == "PhD" or degree == "MD":
    print("Why hello, Dr. {}!".format(last_name))
elif gender == "male":
    print("Why hello, Mr. {}!".format(last_name))
elif gender == "female":
    if marital_status == "married":
        print("Why hello, Mrs. {}!".format(last_name))
    elif marital_status == "single":
        print("Why hello, Miss {}!".format(last_name))
    else:
        print("Why hello, Ms. {}!".format(last_name))
else:
    print("Why hello, Mx. {}!".format(last_name))
```

```
Why hello, Miss Watson!
```

This program implements the quite convoluted social norms for salutations in Western culture. Consider it carefully. If a person is either a Ph.D. or a Medical Doctor, that trumps everything, and we use "Dr." as their form of address. This is to indicate just how studly these people are.

If they don't have such a college degree – and *only* if they don't (notice the `elif`) – will we then consider their gender. For men, it's simple: a plain old "Mr." will do. For women, it's complicated: their marital status now comes into play. This is the *nested* part of the structure: we have another `if` statement (a whole `if`/`elif`/`else` chain, actually) *inside* the "`gender == "female"`" condition. If the person in question identifies as neither `male` nor `female`, all that Mrs./Miss/Ms. stuff will be skipped, and we'll drop straight to the `Mx.` message.

Pay careful attention to the indentation in these examples, since it's the key to discerning the structure of the program.

21.5 Combining branching with loops

And now for the really important application of branching in data science programs: combining it with *loops*.

Just as we can have an `if` statement (or `if`/`else`, or `if`/`elif`/`else`) inside an `if` body, so we can have an `if` statement (and friends) inside a *loop* body. This is where we're going to get a lot of mileage.

Let's return to Springfield. Our `simpsons DataFrame` from p. 174 looked like this:

```
         species  age gender            fave     IQ          hair   salary
name
Homer      human   36      M            beer   74.0          none  52000.0
Marge      human   34      F  helping others  120.0  stacked tall      0.0
Bart       human   10      M      skateboard   90.0          buzz      0.0
Lisa       human    8      F       saxophone  200.0         curly      0.0
Maggie     human    1      F        pacifier  100.0         curly      0.0
SLH          dog    4      M             NaN  100.0        shaggy      0.0
```

Now an ordinary loop could print (say) the name and favorite things of all the characters:

```
for row in simpsons.itertuples():
    print("{} likes {}.".format(row.Index, row.fave))
```

```
Homer likes beer.
Marge likes helping others.
Bart likes skateboard.
Lisa likes saxophone.
Maggie likes pacifier.
SLH likes nan.
```

But combining this with branching techniques gives us more power. We could, for example, print only the females:

```
for row in simpsons.itertuples():
    if row.gender == "F":
        print("{} likes {}.".format(row.Index, row.fave))
```

```
Marge likes helping others.
Lisa likes saxophone.
Maggie likes pacifier.
```

or give different messages for different age ranges:

```
for row in simpsons.itertuples():
    if row.age >= 18:
        print("{} earns ${} outside the home.".format(row.Index,
            row.salary))
    else:
        print("Aw, {}'s just a kid.".format(row.Index))
```

```
Homer earns $52000.0 outside the home.
Marge earns $0.0 outside the home.
Aw, Bart's just a kid.
Aw, Lisa's just a kid.
Aw, Maggie's just a kid.
Aw, SLH's just a kid.
```

or combine these things in innumerable ways:

```
for row in simpsons.itertuples():
   if row.species == "human":
      if row.IQ > 115:
         print("We'd like to nominate {} for a Nobel prize.".format(
            row.Index))
      elif row.IQ >= 90:
         print("You can trust {} with a {}, or even a knife.".format(
            row.Index, row.fave))
      else:
         print("Hmm. No comment.")
      if row.salary > 0:
         print("The {}-year-old {} is gainfully employed.".format(
            row.age, row.Index))
   else:
      print("Hey...{} is some kind of animal!".format(row.Index))
```

```
Hmm. No comment.
The 36-year-old Homer is gainfully employed.
We'd like to nominate Marge for a Nobel prize.
You can trust Bart with a skateboard, or even a knife.
We'd like to nominate Lisa for a Nobel prize.
You can trust Maggie with a pacifier, or even a knife.
Hey...SLH is some kind of animal!
```

You get the idea. Using a loop, we can successively consider each element of an array/`Series` or the rows of a `DataFrame`. Using `if` and friends, we can treat each one differently depending on its characteristics. The possibilities are endless!

Chapter 22

Functions (1 of 2)

And now for the very last "pure programming" lesson of the book: writing **function**s. This is more or less the final tool in the programmer's toolkit, and as I've learned over my years of teaching, it often causes the most trouble.

Now you might be thinking, "hey waitaminit, we've known about functions since all the way back on p. 20. This is something new?" Yes it is. Previously in this book, we've done a lot of *calling* functions – from simple ones like `len()` and `np.append()` to complex ones like `pd.read_csv()` to `scipy.stats.chi2_contingency()` – that someone else has written for us. By contrast, in this chapter, we look behind the curtain and join the production staff: we write our *own* functions.

22.1 Why do all this

It turns out there's a lot of syntactic nonsense involved to get all the wiring right when you do this. It can cause students to pull their hair out. So it's worth asking at the outset: what do we get for all this pain?

The answer is subtle, and can seem underwhelming at first, but it's crucial. It essentially boils down to this lesson: any complex creative work (including a computer program) should be **modular** in its design. This means that it should be composed of smaller

building blocks, which are in turn composed of still smaller building blocks. The entire thing should comprise an organized whole.

Any other way of doing it leads to madness.

Think of a car engine. When a mechanic opens up the hood, he or she doesn't see just one big monolithic thing called "The Engine," but rather piston assemblies, spark plugs, water pumps, drive shafts, and lots of other subsystems. It's what allows piece by piece investigation of problems, and piece by piece replacement of bad parts.

Or think of a rock 'n' roll tune. It's not just an impenetrable mass of sound. Instead, it's a collection of recognizable bass lines, drum sequences, vocal patterns, and variations on common guitar riffs. I don't mean to minimize the creativity involved in its orchestration; in fact, the novel combination of the myriad possible building blocks *is* the creativity. If the song were just an impermeable wall of sound, it would be noise, not music.

In the same way, once your data analysis code approaches a certain size, it really must be written in a modular way or it will become a hopelessly tangled mess, what programmers refer to as "spaghetti code." And the way to achieve this is by writing it in terms of functions that you then call at the appropriate time.

One other advantage to this approach, by the way, is that functions are **reusable**. Think of how many programmers all over the world have had reason to call `np.sort()`, or `scipy.stats.pearsonr()`! The same function becomes applicable in a variety of different contexts, so that nobody has to reinvent the wheel.

22.2 The `def` statement

Okay, down to brass tacks. The way to create (not call) a function in Python is to use the `def` statement. For our first example, let's write a function to compute an (American) football team's score in a game:

```
def football_score(num_tds, num_fgs):
    return num_tds * 7 + num_fgs * 3
```

For those not familiar with football scoring, each "touchdown" (or TD for short) a team scores is worth seven points, and each "field goal" (or FG) is worth three. (For those who *are* familiar with football scoring, please forgive the simplifications here – extra point kicks, safeties, *etc.* It's a first example.)

As you can see from the code snippet, above, the word `def` (which stands for "define," since we're "defining" – a.k.a. writing – a function) is followed by the *name* of our function, which like a variable name can be any name we choose. After the name is the list of **argument**s to the function, in bananas.

All that is the **header** of the function. The **body**, like other "bodies" we've seen (p. 137, p. 212) is *indented* underneath. The `football_score` body is just one line long, but it can be as many lines as necessary.

Finally, we see the word "`return`" on that last line. This is how we control the **return value** which is given back to the code that called our function (review section 5.3 on p. 38 if you need a refresher on this). Whenever a `return` statement is encountered during the execution of a function, the function immediately *stops* executing, and the specified value is handed back to the calling code. More on that in a minute.

22.3 Writing vs. calling

Now here's one of the most perplexing things for beginners. Consider this code:

```
team_name = "Broncos"
num_tds = 3
num_fgs = 2

def football_score(num_tds, num_fgs):
    return num_tds * 7 + num_fgs * 3
```

It surprises many to learn that this code snippet *does not compute anything*, football score or otherwise. The reason? We only *wrote* a function; we didn't actually *call* it.

This is sort of like building an impressive machine but then never pushing the "On" button. The above code says to do four things:

1. Create a `team_name` variable and set its value to the string `"Broncos"`.
2. Create a `num_tds` variable and set its value to the integer 3.
3. Create a `num_fgs` variable and set its value to the integer 2.
4. Create a function called `football_score` which, *if it is ever called in the future*, will compute and return the score of a football game.

In other words, that last step is just preparatory. It tells Python: "by the way, in case you see any code later on that calls a function named '`football_score`,' here's the code you should run in response."

To actually call your function, you have to use the same syntax we learned on p. 20, namely:

```
team_name = "Broncos"
num_tds = 3
num_fgs = 2

the_score = football_score(num_tds, num_fgs)
print("The {} scored {} points.".format(team_name,
    the_score))
```

```
The Broncos scored 27 points.
```

Follow the thread of execution closely here. First, the three variables are created, in what I'll often call "the main program." By "main," I really just mean the stuff that's all the way flush-left, and thus not inside any "`def`." It's the main program in the sense that when you execute the cell, it's what immediately happens without needing to be explicitly called.

Then, after those three variables are created, the `footbal_score()` function is called, at which point *the flow of execution is transferred to the inside of the function.* Since this simple function has only one line of code in its body (the `return` statement), executing it is really quick; but it's still important to realize that for a moment, Python isn't "in" that Broncos cell at all. Instead it jumps to the function, carries out the code inside it, and then `return`s the value...

...right back into the waiting arms of the main program, which stores that returned value (an integer 27, as it turns out) in a new variable named `the_score`. Then the flow continues, and the `print()` statement executes as normal.

Bottom line: every time you want to run your function's code – whether that's a hundred times, once, or not at all – you need to call it by typing the name of the function (with no "`def`") followed by a banana-separated list of arguments.

22.4 Naming arguments

Speaking of arguments, here's the next thing many students have trouble with. The names of the variables that your main program passes to the function are normally *not* the same as the arguments defined by the function itself.

What? Yeah.

Consider this example:

```
jets_touchdowns = 1
jets_field_goals = 3
jets_total = football_score(jets_touchdowns, jets_field_goals)

colts_tds = 1
colts_fgs = 0
colts_total = football_score(colts_tds, colts_fgs)

print("The Jets won {} to {}.".format(jets_total, colts_total))
```

```
The Jets won 16 to 7.
```

This code contains two calls to the `football_score()` function. In the first call, the variables `jets_touchdowns` (with value 2) and `jets_field_goals` (0) were passed. In the second, `colts_tds` (1) and `colts_fgs` (3) were. In *neither* case were the arguments literally named "`num_tds`" and "`num_fgs`", which were the function's own argument names.

To be crystal clear: whenever `football_score()` is called, two arguments are passed to it. The function chooses to name the first one it receives "`num_tds`" and the second one it receives "`num_fgs`". But these are *its own personal names.* They normally have nothing to do with what the calling code chooses to name them.

Why does it work this way? Perhaps this example makes clear the reason. If the main program had to name its variables exactly the same as the (indented) function did, then the function would not be reusable. In order for it to be called with different values in different contexts, there needs to be this flexible decoupling of variable names.

To reinforce the lesson, note too that you can call a function without even having variables in the main program at all:

```
x = football_score(5, 2)
print("Some mythical team scored {} points today.".format(x))
```

```
Some mythical team scored 41 points today.
```

Here we literally passed the values 5 and 2 to the function, instead of creating variables to hold them. The function doesn't mind: it just says, "hey man, whatever's given to me in slot #1, I'm going to name '`num_tds`,' and whatever's delivered through slot #2, I'm going to call '`num_fgs`.' I don't care what the outside world's variable names are...or even whether they have names at all. I just work here."

22.5 Passing aggregate data to functions

Even though the previous example involved passing atomic data to a function, you can totally pass aggregate data as well. Suppose we'd like to be able to easily compute the IQR (recall p. 150) of a univariate data set. Writing a function to do that is a snap:

```
def IQR(some_data):
    return some_data.quantile(.75) - some_data.quantile(.25)
```

We can now call it on anything we like, like our examples from p. 150 and p. 159:

```
print("The IQR of the YouTube plays data is {}".format(IQR(num_plays)))
print("The IQR of the NCAA scoring data is {}".format(IQR(pts)))
```

```
The IQR of the YouTube plays data is 412.
The IQR of the NCAA scoring data is 15.
```

Again, we named the function's own argument (`some_data`) something different than the variables it was called with (`num_plays` first, and then `pts`). This is a happy and healthy thing.

22.6 Returning text

So far, our functions have returned numeric answers. But they can certainly return text as well. Here's a function which assembles a person's full name out of his or her constituent components:

```
def full_name(last_name, first_name, middle_initial):
    return first_name + " " + middle_initial + ". " + last_name

my_full_name = full_name("Davies", "Stephen", "C")
her_full_name = full_name("Clinton", "Hillary", "R")

print("Your author's full name is: {}".format(my_full_name))
print("Another person's full name is: {}".format(her_full_name))
```

```
Your author's full name is: Stephen C. Davies
Another person's full name is: Hillary R. Clinton
```

(Recall from p. 34 that the "+" operator is used for the concatenation of strings.)

22.7 Returning `True` or `False`

It's common for a programmer to want a function which, instead of returning a number or text, tells her *whether or not something is true.* This lets her use the return value of such a function as the condition of an `if` statement.

Here's a trivial example:

```
def is_old_enough_to_vote(age):
    if number >= 18:
        return True
    else:
        return False
```

```
x = is_old_enough_to_vote(13)
if x:
    print("Yes, a 13-year-old can vote!")
else:
    print("Alas, a 13-year-old must wait.")

if is_old_enough_to_vote(19):
    print("Yes, a 19-year-old can vote!")
else:
    print("Alas, a 19-year-old must wait.")
```

```
Alas, a 13-year-old must wait.
Yes, a 19-year-old can vote!
```

The values `True` and `False` are called **boolean value**s, after the 19th-century mathematician George Boole. Note that in Python they must begin with *capital* letters.

The somewhat odd-looking "`if x:`" line is possible because `x` was set to the return value of a call to `is_old_enough_to_vote()`, and that function returned a boolean value.

22.8 Multiple return statements

Another thing the "old enough to vote" example illustrates is the presence of more than one `return` statement in a function. You might have thought this was useless, since on p. 225 I mentioned that as soon as a `return` is encountered during execution, the function is immediately completed. Why, then, would one ever have more than one – the second one could never be reached, right? Wrong. The branching nature of the `if`/`else` statement (above) means that the first `return` will be skipped over in some situations (negative arguments, *etc.*) so it's perfectly sensible to have more than one.

A more complicated example would be the "salutation" algorithm from section 21.4 (p. 217), this time embodied in a function:

```
def salutation(gender, marital_status, degree):
    if degree == "PhD" or degree == "MD":
        return "Dr."
    elif gender == "male":
        return "Mr."
    elif gender == "female":
        if marital_status == "married":
            return "Mrs."
        elif marital_status == "single":
            return "Miss"
        else:
            return "Ms."
    else:
        return "Mx."

my_salutation = salutation("male", "married", "PhD")
print("Why hello, {} Davies.".format(my_salutation))
print("And hello, {} Davies.".format(salutation("female",
    "married", "BS")))
```

```
Why hello, Dr. Davies.
And hello, Mrs. Davies.
```

Wow, all that code is in *one* function? Yeah. That's not unusual at all, although you should strive to make functions as compact as they can be. (The `salutation()` function *is* as compact as it can be, actually: there's no way to shorten it without changing what it does.)

The `salutation()` function, as you can see, has a veritable crapton of `return` statements – six in fact. But only one will ever be reached, because as soon as one *is* reached, the function is officially finished, and returns that value.

22.9 Returning nothing at all

Does it ever make sense for a function to return *nothing*? Oddly, yes: if it has a useful side effect. One side effect is printing:

```
def cheer_for(team):
    if team != "Christopher Newport":
        print("Go {} go!!".format(team))
    else:
        print("Uhh...no.")

cheer_for("Mary Washington")
cheer_for("Lady Eagles")
cheer_for("Christopher Newport")
```

```
Go Mary Washington go!!
Go Lady Eagles go!!
Uhh...no.
```

Here, instead of "`something = cheer_for(...)`" we type plain-old "`cheer_for(...)`". That's because there's no return value to capture, so there's no point in setting it equal to anything. We call the function just to enjoy its messages.

22.10 Calling a function from another function

Finally, note that we can put any code we desire inside a function's body, including one or more calls to other functions! Check out this bad boy:

```
def greet(first_n, last_n, middle_i, gender, status, deg, lang):
    sal = salutation(gender, status, deg)
    if lang == "Swedish":
        greeting = "Hej"
    elif lang == "Russian":
        greeting = "Privet"
    elif lang == "Hindi":
        greeting = "Namaste"
    else:
        greeting = "Yo"
    full = full_name(last_n, first_n, middle_i)
    print("{}, {} {}!".format(greeting, sal, full))

greet("Greta","Thunberg","F","female","single","none","Swedish")
greet("Maria","Sharapova","S","female","single","none","Russian")
greet("Garry","Kasparov","K","male","married","BA","Russian")
greet("Angela","Merkel","D","female","married","PhD","German")
```

```
Hej, Miss Greta F. Thunberg!
Privet, Miss Maria S. Sharapova!
Privet, Mr. Garry K. Kasparov!
Yo, Dr. Angela D. Merkel!
```

In the course of its duties, `greet()`'s function body calls both `salutation()` and `full_name()` for help. They each produce components of its complete solution. Good teamwork!

Chapter 23

Functions (2 of 2)

Like many things in life, writing functions is best learned by example. This chapter will feature several more of them that you can learn from and imitate.

Basketball scoring: `bb_pts()`

Continuing the sports theme, the total points a basketball player scores is related to the number of shots she makes of various kinds. Typically, the "box score" of a game (see example in Figure 23.1) reports three scoring stats: (1) the *total* number of "field goals"[1] a player made and attempted, (2) the number of these field goals, if any, that were for three points[2], and (3) the number of free throws ("easy" penalty shots) the player attempted and made.

Confusingly, (1) *includes* (2). In other words, if the first number is 4 and the second is 1, the player didn't score 4 regular two-point baskets and 1 three-pointer, but rather *3* two-point baskets and 1 three-pointer.

In Figure 23.1, the FGM-A column gives the first of these three categories, 3PM-A the second, and FTM-A the third. The PTS

[1] A "field goal" in basketball just means "a regular basket" – *i.e.*, not a free throw penalty shot.

[2] In most leagues, a basket is worth 2 points unless the shooter was farther than a certain distance from the hoop when she shot it, in which case it's worth 3.

Mary Washington

PLAYER	MIN	FGM-A	3PM-A	FTM-A	PTS
STARTERS					
50 - Tory Martin - f	25	6-10	0-0	1-4	13
10 - Faith St. Clair - g	23	1-2	1-1	0-0	3
14 - Maddie Shifflett - g	29	5-9	0-2	0-0	10
22 - Molly Sharman - g	25	5-8	1-2	2-2	13
23 - Emily Thompson - g	33	6-9	5-7	5-6	22
RESERVES					
05 - Karissa Highlander	7	1-4	0-0	0-0	2
20 - Hannah Stockman	19	2-8	0-5	0-0	4
21 - Emily Shively	10	0-2	0-1	0-0	0
32 - Bri Harper	3	1-4	1-3	0-0	3
34 - Ashley Martin	16	0-2	0-1	3-4	3
40 - Thora Gibbs	10	1-1	0-0	0-0	2
TM - TEAM					
TOTALS		28-59	8-22	11-16	75
		47.5%	36.4%	68.8%	

Figure 23.1: A basketball box score.

column gives the total number of points that player scored. (For example, Molly Sharman made 5 of her 8 attempted field goals, one of which was for three points, and she also converted both free throw attempts.)

All that took a lot longer to explain than the corresponding Python function:

```
def bb_pts(fgm, threep_fgm, ftm):
    return ((fgm - threep_fgm) * 2) + (threep_fgm * 3) + ftm

torys_pts = bb_pts(6, 0, 1)
print("Tory scored {} points.".format(torys_pts))
print("Emily scored {} points.".format(bb_pts(6,5,5)))
print("Lady Eagles scored {} points!".format(bb_pts(28,8,11)))
```

```
Tory scored 13 points.
Emily scored 22 points.
Lady Eagles scored 75 points!
```

Strictly speaking you don't need all those bananas (regular PEMDAS order-of-operations applies) but I think it's a good idea to include them for clarity and grouping.

"Exceptions": `mean_no_outliers()` and `quiz_avg()`

Sometimes we want to take the straight average of a data set, but other times we may want to filter out any strange or exceptional cases. Let's say we're computing the average age of a classroom of college students, but we want to remove any adult learners over 30 since that would skew the result. We could do this sort of thing with a function like this:

```
def mean_no_outliers(a, low_cutoff, high_cutoff):
    return a[(a >= low_cutoff) & (a <= high_cutoff)].mean()

our_class = np.array([20,18,19,18,22,21,76,20,22,22,21,18])
print("The average age (excluding outliers) is {}.".format(
    mean_no_outliers(our_class, 0, 30)))
```

```
The average age (excluding outliers) is 20.09090909090909.
```

We've provided two arguments to the function besides the data set itself: a lower and upper bound. Anything falling outside that range will be filtered out. In the example function call, we passed 0 for the `low_cutoff` since we didn't desire to filter anything at the low end. (If we wanted to, say, also remove children from the data set, we could have set that to 16 or so.)

By the way, you might find the number of decimal places printed to be unsightly. If so, we could enhance our function by rounding the result to (say) two decimals with NumPy's `round()` function:

```
def mean_no_outliers(a, low_cutoff, high_cutoff):
    return np.round(a[
        (a >= low_cutoff) & (a <= high_cutoff)].mean(),2)

print("The average age (excluding outliers) is {}.".format(
    mean_no_outliers(our_class, 0, 30)))
```

```
The average age (excluding outliers) is 20.09.
```

At this point you might think this function is getting pretty big for a one-liner. I agree. Let's split it up and use some temporary variables to make it more readable:

```
def mean_no_outliers(a, low_cutoff, high_cutoff):
    filtered_data = a[(a >= low_cutoff) & (a <= high_cutoff)]
    filtered_average = np.round(filtered_data)
    return np.round(filtered_average,2)
```

Much clearer!

A related but different example would be to remove a fixed *number* of data points from the end, instead of data points outside a specified range. For instance, in my classes, I often give students (say) eight quizzes during a semester, and drop the lowest two scores. That could be done with:

```
def quiz_avg(quizzes):
    dropped_lowest_two = np.sort(quizzes)[2:]
    return dropped_lowest_two.mean()

filberts_quizzes = np.array([7,9,10,7,0,8,4,10])
print("Filbert's avg score was {}.".format(quiz_avg(
    filberts_quizzes)))
```

```
Filbert's avg score was 8.5.
```

Filbert's 0 and 4 were dropped, leaving him with a pretty good semester score.

The trick to this implementation is *sorting* the quiz scores. Once you do that, it's easy to pick out the top six to take the average, since the lowest two scores will be at the beginning of the (sorted) array. Two notes here:

- We use the `np.sort()` function, not the `.sort()` method, since we don't want to permanently change the order of `quizzes`. We only need a temporarily sorted copy so we can omit the lowest two entries.
- That business in the boxies ("`[2:]`") is a **slice** (recall section 9.2 on p. 76) which says "only give me entries number 2 through the end of the array." And that's exactly what the doctor ordered to omit the first two.

Searching for values: `any_zeros()`

I'll end this chapter with an example which, like the "preferred language" example on p. 214, flummoxes nearly every beginning student.

Suppose students in a DATA 101 course are given labs to complete, each one worth up to 20 points. (This is purely hypothetical, as you can see.) At the midway point of the semester, the instructor would like a quick list of any students who failed to turn in one of the labs, so he can harass them for their own good.

Here's the `gradebook` `DataFrame` this professor is using:

student	Q1	Q2	Q3	Q4	lab1	lab2	lab3	lab4	lab5
Filbert	7	9	10	7	15	19	14	20	20
Jezebel	8	7	0	6	12	12	16	0	20
Betty Lou	10	10	10	10	20	20	20	20	20
Biff	3	2	6	5	10	12	0	0	16
Melvin	0	0	10	10	0	18	20	14	20

Let's write a function called `print_harass_list()` whose job is to tell this professor which students he should check up on. We'll write it as follows:

```
def print_harass_list(gradebook):
    for row in gradebook.itertuples():
        if any_zeros(np.array([row.lab1, row.lab2, row.lab3,
            row.lab4, row.lab5])):
            print("Better check up on {}.".format(row.Index))
```

Note that we've pushed some of the work on to a new function, `any_zeros()`, that we haven't written yet. This is good organizational style. Now `print_harass_list()` can do the job of iterating through the `DataFrame` rows, extracting the lab scores, and printing a message if necessary, whereas it defers to `any_zeros()` to inspect the lab scores and determine the presence of any zeros.

It doesn't work until we actually write the second function, of course, so here goes. Heads up, since this is the part that perplexes students. The following implementation of `any_zeros()` looks perfectly reasonable, yet is dead WRONG:

```
def any_zeros_WRONG(labs):
    for lab in labs:
        if lab == 0:
            return True
        else:
            return False
```

It looks so correct! And yet it is not. Check out the result:

```
print_harass_list(gradebook)
```

```
Better check up on Melvin.
```

Clearly we need to check up on `Jezebel` and `Biff` as well (look at their scores for labs 3 and 4), yet they inexplicably didn't get printed.

Here's what's WRONG with that `any_zeros()` attempt. Stare carefully at that loop and realize that the *body* of the loop is comprised of a single `if`/`else` statement. And remember our cardinal rule from the grey box on p. 214: either the `if` body or the `else` body will *always* be executed.

That means that this loop is destined to only execute exactly once! It doesn't matter how long the `labs` array is. It effectively looks

only at the *first* element, and decides based solely on that whether or not the entire array has any zeros in it!

The correct version of `any_zeros()` would look like this:

```
def any_zeros(labs):
    for lab in labs:
        if lab == 0:
            return True
    return False
```

At first glance, it may appear unchanged, but look again. First of all, there's no `else` anymore. Second of all, the "`return False`" line is indented *evenly with the word* `for`. This means that "`return False`" is *not* part of the loop at all: it will only run *after* the entire loop has executed.

That turns out to make all the difference. The function will dutifully go through *each* element of the `labs` array, inspecting each one to see whether it's zero. As soon as it finds a zero, it returns `True`, since then its job is done. Only after inspecting the *entire* array, and coming up empty on its zero quest, does this function then have the audacity to return `False`, meaning "nope! Clean as a whistle." The result:

```
Better check up on Jezebel.
Better check up on Biff.
Better check up on Melvin.
```

Postlude: thinking algorithmically

Getting tripped up on that last example is, I believe, usually a case of *thinking holistically* rather than *thinking algorithmically*. Math classes have trained people to think holistically, by which I mean looking at (say) a bunch of equations and viewing them as "all equally true, all at once." And this is the correct way to think mathematically. If I give you five simultaneous equations

that state relationships among variables, they aren't really in any order. They're just "five true things."

But programming requires you to think algorithmically. You have to execute the code in your head, step by step, and realize the consequences. The appealing symmetry of the WRONG `any_zeros()` function is appealing because you're looking at it as a whole: "it's looping (seemingly) through all the elements, with zeros being an indicator of `True`ness and non-zeros being an indicator of `False`ness. What's not to like?" The error, as you saw, is that when running through the data step-by-step, there are immense ramifications of returning early. That's only apparent if you think of the code executing sequentially as it goes. You have to pretend you're the computer, not a mathematician.

Chapter 24

Recoding and transforming

It's often the case that although a `DataFrame` contains the raw information you need, it's not exactly in the form you need for your analysis. Perhaps the data is in different units than you need – meters instead of feet; dollars instead of yen. Or perhaps you need some *combination* of available quantities – miles per gallon instead of just miles and gallons separately. Or perhaps you need to reframe a variable by binning it into meaningful subdivisions – categorizing a raw column of salaries into "high," "medium," and "low" wage earners, for instance.

In data science, these activities are known as **recoding** and/or **transforming**. There's not a sharp division between the two; usually I think of recoding as converting a single variable to one with different units (as in the dollars-to-yen and high/medium/low earners examples) and transforming as creating a new variable entirely out of a combination of columns (like miles per gallon). In both cases, though, we'll be creating and adding new columns to a `DataFrame`. These columns are sometimes called **derived columns** since they're based on (derived from) existing columns rather than containing independent information.

24.1 Recoding with simple operations

Consider the following soccer data set called `worldcup2019.csv`. Each row of this data set represents one player's performance in a particular 2019 World Cup game. Notice that we have a couple of players with more than one row (Megan Rapinoe and Rose Lavelle), and several rows for the same game (the first four rows are all from the June 28th game, for instance):

```
last,first,date,inmins,insecs,outmins,outsecs,gls,asst,tkls,shots
Morgan,Alex,28-Jun-2019,0.0,0.0,90.0,0,0,0,0,2,1
Rapinoe,Megan,28-Jun-2019,0.0,0.0,74.0,27.0,2,0,2,3
Press,Christen,28-Jun-2019,74.0,27.0,90.0,0.0,0,0,1,0
Lavelle,Rose,28-Jun-2019,0.0,0.0,90.0,0.0,0,1,3,0
Lavelle,Rose,7-Jul-2019,0.0,0.0,90.0,0.0,1,0,4,1
Rapinoe,Megan,7-Jul-2019,0.0,0.0,83.0,16.0,1,1,3,2
Lloyd,Carli,7-Jul-2019,83.0,16.0,90.0,0.0,0,0,1,0
Dunn,Crystal,23-Jun-2019,42.0,37.0,81.0,5.0,0,1,1,2
```

The data set doesn't really have a meaningful index column, since none of the columns are expected to be unique. So we'll leave off the "`.set_index()`" method call when we read it in to Python:

```
wc = pd.read_csv('worldcup2019.csv')
print(wc)
```

```
     last first   date inmins insecs outmins outsecs gls asst tkls shots
   Morgan  Alex 28-Jun      0      0      90       0   0    0    2     1
  Rapinoe Megan 28-Jun      0      0      74      27   2    0    2     3
    Press Chris 28-Jun     74     27      90       0   0    0    1     0
  Lavelle  Rose 28-Jun      0      0      90       0   0    1    3     0
  Lavelle  Rose  7-Jul      0      0      90       0   1    0    4     1
  Rapinoe Megan  7-Jul      0      0      83      16   1    1    3     2
    Lloyd Carli  7-Jul     83     16      90       0   0    0    1     0
     Dunn Cryst 23-Jun     42     37      81       5   0    1    1     2
```

Let's zero in on the columns with `mins` and `secs` in the names. These columns show us the minute and second that the player went

`in` to the game, and the minute and second that they came `out`. For example, Alex Morgan played the entire 90-minute match on June 28th. Rapinoe started that game, but came out for a substitute at the 74:27 mark. Who replaced her? Looks like Christen Press did, since she *entered* the game at exactly the same time. In most rows, the player either started the game, or ended the game or both, but the last row (Crystal Dunn's June 23rd performance) has her entering at 42:37 and exiting at 81:05.

Now the reason I bring this up is because one aspect of our analysis might be computing statistics *per minute* that each athlete played. If one player scored 3 goals in 200 minutes, for example, and another scored 3 goals in just 150 minutes, we could reasonably say that the second player was a more prolific scorer in that World Cup.

This is hard to do with the data in the form that it stands. So we'll **recode** a few of the columns. Let's collapse the minutes and seconds for each of the two clock times into a single value, in minutes. For readability, we'll also round this number to two decimal places using the `round()` function we met on p. 237:

```
wc['intime'] = np.round(wc['inmins'] + (wc['insecs']/60),2)
wc['outtime'] = np.round(wc['outmins'] + (wc['outsecs']/60),2)
```

We're taking advantage of vectorized operations here. For each row, we need to divide the `insecs` value by 60 (to convert it to minutes) and add it to the `inmins` value. Pandas makes this super easy here, since we can just write out those operations once, and it will compute it for every single row!

Let's delete the old, superfluous columns now and looksie:

```
del wc['inmins']
del wc['insecs']
del wc['outmins']
del wc['outsecs']
print(wc)
```

```
      last first   date gls asst tkls shots intime outtime
0  Morgan  Alex 28-Jun   0    0    2     1   0.00   90.00
1 Rapinoe Megan 28-Jun   2    0    2     3   0.00   74.45
2   Press Chris 28-Jun   0    0    1     0  74.45   90.00
3 Lavelle  Rose 28-Jun   0    1    3     0   0.00   90.00
4 Lavelle  Rose  7-Jul   1    0    4     1   0.00   90.00
5 Rapinoe Megan  7-Jul   1    1    3     2   0.00   83.27
6   Lloyd Carli  7-Jul   0    0    1     0  83.27   90.00
7    Dunn Cryst 23-Jun   0    1    1     2  42.62   81.08
```

This is much less unwieldy (more wieldy?) than dealing with minutes and seconds separately.

(Incidentally, notice that the technique presented here creates *new* columns (with new names) and then deletes the old columns. I strongly recommend doing it this way. If you try to change the values of an *existing* `DataFrame` column, Pandas will often give you a strange-looking message informing you of a "`SettingWithCopyWarning`". The meaning is a bit esoteric, but in layman's terms it means "your operation may not have actually worked." Avoid this problem by creating new columns instead.)

24.2 Transforming with simple operations

Now that we've converted the awkward minutes-and-seconds columns to just "`time`" columns, all we need to do to complete our analysis is **transform** this data by computing a new quantity entirely: the *total number of minutes played* for each player in each game. Again, Pandas makes this easy:

```
wc['minsplayed'] = wc.outtime - wc.intime
print(wc)
```

```
      last first   date gls asst tkls shots intime outtime minsplayed
0  Morgan  Alex 28-Jun   0    0    2     1   0.00   90.00      90.00
1 Rapinoe Megan 28-Jun   2    0    2     3   0.00   74.45      74.45
2   Press Chris 28-Jun   0    0    1     0  74.45   90.00      15.55
3 Lavelle  Rose 28-Jun   0    1    3     0   0.00   90.00      90.00
4 Lavelle  Rose  7-Jul   1    0    4     1   0.00   90.00      90.00
5 Rapinoe Megan  7-Jul   1    1    3     2   0.00   83.27      83.27
6   Lloyd Carli  7-Jul   0    0    1     0  83.27   90.00       6.73
7    Dunn Cryst 23-Jun   0    1    1     2  42.62   81.08      38.46
```

Voilà. We now have the time-on-field for each player, which gives us a whole new avenue of exploration. For example, any of the counting stats (goals, assists, *etc.*) can be converted into a "per-minute" version, showing us how productive a player was while on the field. Let's do that for `tkls` ("tackles"), and multiply by 90 to obtain a "tackles-per-90-minutes" statistic[1]:

```
wc['minsplayed'] = wc['outtime'] - wc['intime']
wc['tkl_per_90'] = np.round(wc['tkls'] /
    wc['minsplayed'] * 90,2)
del wc['tkls']
```

```
      last first   date gls asst shots intime outtime minsplayed tkl_90
0   Morgan  Alex 28-Jun   0    0     1   0.00   90.00      90.00   2.00
1 Rapinoe Megan 28-Jun   2    0     3   0.00   74.45      74.45   2.42
2    Press Chris 28-Jun   0    0     0  74.45   90.00      15.55   5.79
3 Lavelle  Rose 28-Jun   0    1     0   0.00   90.00      90.00   3.00
4 Lavelle  Rose  7-Jul   1    0     1   0.00   90.00      90.00   4.00
5 Rapinoe Megan  7-Jul   1    1     2   0.00   83.27      83.27   3.24
6    Lloyd Carli  7-Jul   0    0     0  83.27   90.00       6.73  13.37
7     Dunn Cryst 23-Jun   0    1     2  42.62   81.08      38.46   2.34
```

Transforming grouped data

The above example computed tackles-per-game all right, but it still left us with one row for every player-performance. (In other words, the results had *two* rows for Rose Lavelle, one giving her `tkl_per_90` for the June 28th game, and one giving it for the July 7th game.)

We might instead be interested in a player-by-player analysis: overall in the entire month-long World Cup, which players had the most tackles-per-game? This is easy to do with the `.groupby()` method that we first encountered in section 18.2 (p. 189). First, we group the rows by the first *two* columns (since first-and-last-names-together are needed to uniquely identify a single player):

[1] I'm choosing 90 minutes here because that's how long a regulation-length soccer match is. Therefore, our new `tkl_per_90` column gives us "number-of-tackles-per-complete-game," which is easier to interpret than "tackles-per-*minute*," which would be a miniscule number for any player.

```
grouped_wc = wc.groupby(['last','first'])
```

We then take our new, temporary `grouped_wc` variable and extract the `gls`, `asst`, `shots`, `tkls`, and `minsplayed` columns from it, **summing** each of them to produce the per-player values in the result:

```
by_player = grouped_wc[['gls','asst','shots','tkls',
    'minsplayed']].sum()
```

This yields:

		gls	asst	shots	tkls	minsplayed
last	first					
Dunn	Cryst	0	1	2	1	38.46
Lavelle	Rose	1	1	1	7	180.00
Lloyd	Carli	0	0	0	1	6.73
Morgan	Alex	0	0	1	2	90.00
Press	Chris	0	0	0	1	15.55
Rapinoe	Megan	3	1	5	5	157.72

Now, we're ready to compute a per-game analysis as before, but this time for each player's entire World Cup games:

```
by_player['tkl_per_90'] = (np.round(by_player['tkls'] /
    by_player['minsplayed'] * 90,2))
del by_player['tkls']
```

		gls	asst	shots	minsplayed	tkl_per_90
last	first					
Dunn	Cryst	0	1	2	38.46	2.34
Lavelle	Rose	1	1	1	180.00	3.50
Lloyd	Carli	0	0	0	6.73	13.37
Morgan	Alex	0	0	1	90.00	2.00
Press	Chris	0	0	0	15.55	5.79
Rapinoe	Megan	3	1	5	157.72	2.85

24.3 More complex transformations

In all the above examples, we took advantage of Pandas vectorized operations. With just a single line of code like "`wc['minsplayed'] = wc.outtime - wc.intime`", we could compute our entire new transformed column in one fell swoop.

Sometimes, we're not so lucky. In particular, if the computation of the transformed column is more complicated than just numeric operations – like, if it involves branches, loops, or calling other functions – we normally can't compute it all at once. Instead, we have to resort to a loop.

Pandas makes this procedure a bit awkward in my opinion. But once you learn the pattern, it's not hard to imitate. Here's the pattern for creating a transformed/recoded column that requires more complex operations:

1. Create a **function** that will compute the transformed value for a *single* row. Its arguments should be whatever column values are necessary to derive the new value, and its return value should be the desired transformation.
2. Create an *empty* NumPy array to hold the row-by-row results, and make sure it's the right type.
3. Write a loop that will iterate through all the rows of the original `DataFrame`. For each row, pass the appropriate values to the function, and then *append* the return value to the ever-growing NumPy array.
4. Finally, slap that NumPy array on to the `DataFrame` as a new column.

Here's a couple examples. First, suppose we want to compute a shooting percentage for each player; in other words, how many goals they scored per shot they took. Now you might think we could simply use vectorized operations:

```
wc['shots_per_goal'] = wc.gls / wc.shots
```

The problem is, for players who never attempted a shot in the game, this would result in dividing by zero, a cardinal sin. Sports convention says that if a player makes 0 goals in 0 attempts, their shooting percentage is 0.00, even though mathematically-speaking this is undefined.

Very well, following our procedure from above, we'll first define a function `shooting_perc()`:

```
def shooting_perc(gls, shots):
    if shots == 0:
        return 0.0
    else:
        return np.round(gls / shots * 100, 1)
```

Then, we create an empty NumPy array. Here's how:

```
s_perc = np.array([])
```

Looks weird, I know. But remember, the `array()` function (review p. 63) takes a boxie-enclosed list of elements. If we enclose *nothing* inside the boxies, that effectively makes it an empty list.

And why would we want to do that? Because we need to continually add to this array, one value for each row in the `DataFrame`. At the end, there must be *exactly* as many elements in `s_perc` as there are rows in `wc`, otherwise we won't be able to add it as a new column.

Here's the loop (step 3 from the shaded box):

```
for row in wc.itertuples():
    new_s_perc = shooting_perc(row.gls, row.shots)
    s_perc = np.append(s_perc, new_s_perc)
```

I've chosen to create a temporary variable here (`new_s_perc`) for readability. The first line of the loop body says to take the current `row`'s `gls` and `shots` values, and send them as arguments to the `shooting_perc()` function. That function, which we defined above, will return us a single number which is the shooting percentage for *that row*. The second line then appends that single `new_s_perc` value to the end of the ever-growing `s_perc` array.

Finally, we add this new column to the `wc DataFrame` proper:

```
wc['s_perc'] = s_perc
```

which gives us:

```
      last first   date gls asst shots intime outtime minsplayed s_perc
0  Morgan  Alex 28-Jun   0    0     1   0.00   90.00      90.00     0.0
1 Rapinoe Megan 28-Jun   2    0     3   0.00   74.45      74.45    66.7
2   Press Chris 28-Jun   0    0     0  74.45   90.00      15.55     0.0
3 Lavelle  Rose 28-Jun   0    1     0   0.00   90.00      90.00     0.0
4 Lavelle  Rose  7-Jul   1    0     1   0.00   90.00      90.00   100.0
5 Rapinoe Megan  7-Jul   1    1     2   0.00   83.27      83.27    50.0
6   Lloyd Carli  7-Jul   0    0     0  83.27   90.00       6.73     0.0
7    Dunn Cryst 23-Jun   0    1     2  42.62   81.08      38.46     0.0
```

Rose Lavelle's July 7th game was the only perfect shooting performance in this data set – who knew?

We'll complete this chapter with a slightly more complex example, but which still follows the shaded box pattern.

Say we're also interested in which players *started* which games (as opposed to being a mid-game substitute). Obviously, a starter is someone who entered the game at time 0. To create a new column

for this, we'll need our function to return the boolean value `True` if the player's `intime` value was zero, and `False` otherwise. Here's the complete code snippet for this transformation:

```
def starter_func(intime):
    if intime == 0:
        return True
    else:
        return False

starter = np.array([]).astype(bool)

for row in wc.itertuples():
    starter = np.append(starter, starter_func(row.intime))

wc['starter'] = starter
```

```
      last first   date gls asst tkls shots minsplayed s_perc starter
  0  Morgan  Alex 28-Jun   0    0    2     1       90.00    0.0    True
  1 Rapinoe Megan 28-Jun   2    0    2     3       74.45   66.7    True
  2   Press Chris 28-Jun   0    0    1     0       15.55    0.0   False
  3 Lavelle  Rose 28-Jun   0    1    3     0       90.00    0.0    True
  4 Lavelle  Rose  7-Jul   1    0    4     1       90.00  100.0    True
  5 Rapinoe Megan  7-Jul   1    1    3     2       83.27   50.0    True
  6   Lloyd Carli  7-Jul   0    0    1     0        6.73    0.0   False
  7    Dunn Cryst 23-Jun   0    1    1     2       38.46    0.0   False
```

One subtle point that is easy to miss: when we first created the empty `starter` array, we typed "`.astype(bool)`" at the end. This is because by default, the values of a new empty array will be **float**s. This worked fine for the shooting percentage example, because that's actually what we wanted, but here we want `True`/`False` values instead (for "starter" and "non-starter.")

Pretty cool, huh? The original `DataFrame` had the information we wanted, but not in the form we really needed it. What we wanted was not the entry time and exit time of each player (both in minutes and seconds) but rather the total time that player was on the pitch, and whether or not they started the game. We also wanted to convert several of the raw statistics into per-complete game numbers,

and to compute meaningful ratios like shooting percentage or fouls per assist.

Recoding and transforming turn out to be common tasks for a simple reason: *whoever collects a data set can rarely predict how an analyst will eventually use it.* We're very grateful to the author of the `.csv` file, since it contains the raw material we need to evaluate our team's performances; but how were they to know that length-of-time-on-the-field and who-started-which-game was going to be important to us? They couldn't. But thanks to recoding and transformation skills, we can cope.

Chapter 25

Machine Learning: concepts

When ordinary people hear the words "Data Science," I'll bet the first images that come to mind are of the closely-related fields of **data mining** and **machine learning (ML)**, even if they don't know those terms. After all, this is where all the sexy tech is, and the success stories too: Netflix magically knowing which movies you'll like, grocery chains using data from loyalty cards to optimally place products; the Oakland A's scouring minor league stats to build a champion team with chump change (see: *Moneyball*). There are also creepier applications of this technology: Google placing personalized eye-catching ads in front of you using data they mined from your email text, or Cambridge Analytica projecting from voter personalities to the best ways to micro-target them.

All these examples have one thing in common: they actually *make* the discoveries and predictions from the data. They're the coup de grâce. They take place after we've already acquired our data, imported it to an analysis environment (like Python), stored it in the appropriate data structures (like associative arrays or tables), recoded/transformed/pre-processed it as necessary, and explored it enough to know what we want to ask. All that stuff was mere prep work. This chapter is where we begin to really rock-and-roll.

25.1 Data mining vs. machine learning

The terms "data mining" and "ML" have a lot of sloppy overlap, but one distinction we can pick out is this. If someone says they're doing data mining, their goal is normally **inference**: deriving high-level strategic insights based on patterns in the data. Discovering that amateur pitching performances translate more reliably to the major leagues than amateur batting performances do, generally speaking, is an inference, and a potentially valuable find.

If someone says they're doing ML, on the other hand, their goal is normally **prediction**: making an educated guess about how a specific case will turn out. When we forecast how many home runs we think a college prospect will hit in his first two years in the majors, we're making a specific prediction rather than inferring a general truth – this, too, is potentially quite valuable, as it may lead us to decide to sign the player or look at different options.

25.2 Deductive vs. inductive reasoning

This chapter contains a lot of vocabulary terms. Before we dive in to the ML-specific ones, I think it's important to take a step back and make a more general point about the kind of "learning" we'll be doing. There are at least two different ways that human beings reach conclusions: **deductively** and **inductively**. Deductive reasoning is associated most prominently with Sherlock Holmes in the public mind. Through sheer application of irrefutable logic, Holmes and his companion Watson deduced new facts from known facts in their quest to catch the criminal. Their logic was seemingly air-tight, since everything they deduced followed directly and irresistibly from what came before.

There's a subdiscipline of Philosophy called Logic which covers exactly such matters. Syllogisms, *modus ponens*, first-order predicate calculus: these are all concepts you'll learn if you take an introductory course in Logic. And the nice thing about deduction is that as long as you follow the rules, *your conclusions will always be dependably correct.*

Inductive reasoning, on the other hand, does *not* always lead to 100% reliably correct conclusions. This may give you pause, and wonder why anyone would ever use it. The reason is that in the vast majority of cases, deductive reasoning simply isn't applicable to your situation, and induction is the only case.

Induction is about *reasoning from examples.* Lots of examples. Living in the world as we do, we observe plenty of examples of how people and things behave, and we start to identify certain general patterns in what we've observed. One thing I noticed long ago is that when I smile and say hi to a person, they normally smile and say hi back. But when I smile and say hi to a dog, or a bush, or a vending machine, I'm normally met with stony silence.

From this, I've **induced** the general rule that people respond to greetings but other objects don't. Now this is *not* 100% reliably true. Even in my own experience, there have been times when I've greeted someone walking down the hallway and been outright ignored. And for all I know, there may be some vending machines out there who might respond if someone talks to them – with technological advancements in voice recognition and synthesis, it's probably just a matter of time before they do. But the point is that *learning this general principle about greetings has served me very well in life.* I don't normally talk to inanimate objects, but I do to people, and this has helped me function in society. Even if a rule *isn't* accurate in absolutely every situation, it can still be very, very important.

If you do a quick scan of your brain, I believe you'll find that the vast majority of the things that you "know" about life were arrived at inductively, rather than deductively. If you ask a friend for money, he'll probably say yes; if you ask a stranger, he'll probably say no. If your friend does say yes, he'll probably expect the favor to be returned at a later point; if the stranger says yes, he probably won't. If you don't study for a test, you'll probably do poorly, and likewise if you wait until the last day to start your 5-page paper. None of these conclusions can be proven deductively, and in fact all of them have exceptions; but not to know these things is to be at a serious disadvantage in trying to make decisions.

I say all this because *everything in ML is about induction, not de-*

duction. As we'll see, the name of the game in ML is looking at lots and lots of past examples, and making future predictions based on them. It's true that "past performance is no guarantee of future success," but past performance *does* tell you *something* valuable about future possibilities, else there'd be no point in trying to learn from it. And the fact that we apply our past lessons in altering our future behavior is undeniable.

25.3 Supervised vs. unsupervised learning

Now let's dive further down to some more technical and fine-grained distinctions. There are two main categories of machine "learning": **supervised** and **unsupervised**. I think these terms are ridiculous and misleading, by the way, but they're what we're stuck with so let's learn what they mean.

In a supervised learning setting, our goal is to predict the value of some **target attribute** of some object of study. As an example, let's say we want to predict someone's *mood* based only on their facial expression and body language. "Mood" might be a categorical variable with values "happy," "angry," "bored," *etc.*

To do this prediction, we'll use a bunch of previously observed examples. These past examples, *for which the person's true mood is known*, are collectively called our **training data**. We remember seeing one person with a smile on their face and their eyes slightly squinted, and later discovered that they were happy. We remember another person with a smile but with wide open eyes, and learned that they, too, were happy. We also remember someone with clenched fists and raised eyebrows, and they turned out to be frightened. A different person with clenched fists but squinted eyes was later revealed to be angry. And so on.

Supervised machine learning is about how to extrapolate from past examples in a principled way, in order to make predictions about future examples whose true value (mood, say) is *not* known. The task is to say, "okay, there's a person down the hallway whose face is slightly flushed and whose arms are tightly crossed. Are they likely to be happy, defensive, angry, embarrassed, or something else?

Let's apply what we've learned from past examples to guess at the answer."

It's called "supervised" precisely because the "true answer" for the target attribute is known for the training data.

Now suppose we *didn't* know the true answer for our training examples. Say we've observed and recorded the eyebrow position, the mouth configuration, whether the face was flushed or pale or in between, *etc.*, for a bunch of people we've encountered in the past, but we actually never learned what their mood was. What then?

This is an unsupervised learning setting. Predicting a person's mood based on this kind of information turns out to be nearly hopeless. If we don't know what anyone else's mood was, how can we predict what this new person's mood is? But all is not lost – we may still be able to form some conclusions about what *types* of moods there are. For example, we might notice that generally speaking, raised eyebrows tend to be accompanied by certain other indicators. In many past examples, they've appeared together with an open mouth and a rigid posture. In other examples, raised eyebrows instead appeared with lips tightly-pressed together and the forehead slightly tilted forward. We don't know which moods these collections of features might correspond to, since our training data didn't have any information about moods. But we might still infer the presence of a couple of distinct raised-eyebrow moods, since they are so commonly accompanied by either one of two groups of other features.

Classification, regression, and clustering

In the supervised setting, our most common machine learning activities will be **classification** and **regression**. In each one, our job is to predict the value of the target attribute for a new object, based on the previous example objects we've seen. The only difference is the scale of measure of the target variable: if it's categorical, we're performing classification, and our goal is to build a **classifier**: an algorithm (basically, a Python function) that can **classify** future examples by guessing their target value. If the target variable is numeric, then we have regression, and our goal is to make the closest

guess we can to the true target value.

For example, if we have some census and earnings data for a region, and our goal is to predict whether or not someone in that region will be a homeowner or a renter, we're performing classification. If our goal instead is to predict their annual salary, we're doing regression.

By the way, let me make clear that the types of the *other* variables we're considering (*i.e.*, other than the target) don't play in to whether we're doing classification or regression: only the target does. If I'm using race (categorical), gender (categorical), age (numeric), and college degree (categorical) to predict *salary* (numeric), then this is a *regression* problem, even though the majority of the variables are categorical. If I were to use the same four variables to predict *political affiliation* (categorical), then it would be a classification problem, even though we had a numeric variable as a predictor.

In the unsupervised setting, the most common task is **clustering**: finding groups of related objects, with similar attribute values, in order to discern how many basic types of objects there are, and what their typical value ranges are. That's exactly what we did with the mood data, above, in the absence of information about past moods. Another example would be to look at the attributes of various movies on IMDB and discern "what basic types of films there are." We may discover that movies naturally break down into blockbuster action films, period dramas, romantic comedies, and a few other common genres. There will always be objects that defy categorization, and exist on the boundaries of defined clusters, but it's still profoundly insightful to discover the presence of common patterns that bring structure to the data.

Machine learning is a big field, and each aspect has its own techniques and deserves its own treatment. For the rest of this book, we're going to concentrate only on **supervised** learning, specifically the task of **classification**.

Chapter 26

Classification: concepts

26.1 Labeled and unlabeled examples

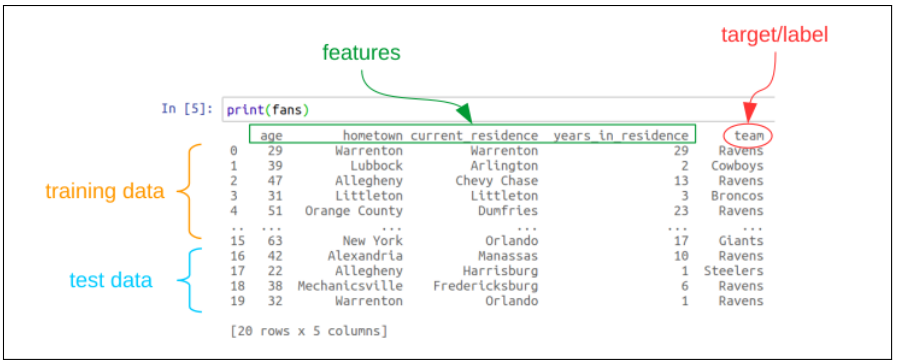
In the activity of classification, the **target** variable we aim to predict is categorical. We sometimes also call this variable the **label**. Since this is a *supervised* process, we are provided with example objects of study that have known "true answers." These are called **labeled examples**. The goal of the activity is to produce good predictions of the labels for other, **unlabeled examples**. A program that can make such predictions, after having studied the labeled examples, is called a **classifier**.

The predicted label can be seen as the "output" from our classifier. All of the other variables are essentially the inputs to our process, which we use to make our predictions. These variables are called **features** (or sometimes, **attributes**).

In terms of Pandas data structures, all these labeled examples will normally come packaged in a `DataFrame`. Each row of the `DataFrame` will be one labeled example, with its features as columns and its target/label as a column (traditionally, the rightmost one).

This is illustrated in Figure 26.1. Here we have some labeled examples for a data set on NFL fans. Each row represents one fan, and shows various features of their existence – how old they are, where they were born, where they live now, and how many years they've

lived in their current residence. The rightmost column gives the target: the team to whom this fan has sworn their allegiance. Our aim would be to predict which team a fan might root for, based on what we know about them. Lest you think this example is frivolous, consider that a sporting goods company might want to send catalogs (paper or electronic) to potential customers, and it would probably boost sales if the cover image of the catalog featured a model wearing apparel from the customer's favorite team, rather than their rival.



	age	hometown	current_residence	years_in_residence	team
0	29	Warrenton	Warrenton	29	Ravens
1	39	Lubbock	Arlington	2	Cowboys
2	47	Allegheny	Chevy Chase	13	Ravens
3	31	Littleton	Littleton	3	Broncos
4	51	Orange County	Dumfries	23	Ravens
..	...	...	...	...	...
15	63	New York	Orlando	17	Giants
16	42	Alexandria	Manassas	10	Ravens
17	22	Allegheny	Harrisburg	1	Steelers
18	38	Mechanicsville	Fredericksburg	6	Ravens
19	32	Warrenton	Orlando	1	Ravens

Figure 26.1: Some labeled examples, divided into training and test sets.

You'll also see in the figure that I've split the rows up into two groups. The first group is called the **training data**, and the second, the **test data**. (Normally we'll shuffle all the rows before assigning them, so that we don't put all the top rows of the `DataFrame` in the training set and all the bottom ones in the test set. But that's harder to show in a picture.)

26.2 Three kinds of examples

Now here's the deal. There are *three* kinds of example rows we're going to deal with:

1. **training data** – *labeled* examples which we will show to our classifier, and from which it will try to find useful patterns to make future predictions.

2. **test data** – *labeled* examples which we will *not* show to our classifier, but which we will use to measure how well it performs.
3. **new data** – *unlabeled* examples that we will get in the future, after we've deployed our classifier in the field, and which we will feed to our classifier to make predictions.

The purpose of the first group is to give the classifier useful information so it can intelligently classify.

The purpose of the second group is *to assess how good the classifier's predictions are.* Since the test set consists of labeled examples, we know the "true answer" for each one. So we can feed each of these test points to the classifier, look at its prediction, compare it to the true answer, and judge whether or not the classifier got it right. Assessing its accuracy is usually just a matter of computing the percentage of how many test points it got right.

The third group exists because after we've built and evaluated our classifier, we actually want to put it into action! These are new data points (new sporting goods customers, say) for which we don't know the "true answer" but want to predict it so we can send catalogs likely to be well-received.

Thou shalt not reuse

Now one common question – which leads to a *super* important point – is this: why can't we use *all* the labeled examples as training data? After all, if we have 1000 labeled examples we've had to work hard (or pay $$) to get, it seems silly to only use some of them to train our classifier. Shouldn't we want to give it all the labeled data possible, so it can learn the maximum amount before predicting?

The first reply is: "but then we wouldn't have any test data, and so we wouldn't know how good our classifier *was* before putting it out in the field." Clearly, before we base major business decisions on the results of our automated predictor, we need to have some idea of how accurate its predictions are.

It's then often countered: "sure, but why not then re-use that data for testing? Instead of splitting the 1000 examples into training

points and test points, why not just use all 1000 for training, and then test the classifier on all 1000 points? What's not to like?"

This is where the super important point comes in, and it's so important that I'll put it all in boldface. It turns out that **you absolutely *cannot* test your classifier on data points that you gave it to train on, because you will get an overly optimistic estimate of how good your classifier actually is.**

Here's an analogy to make this point more clear. Suppose there's a final exam coming up in your class, and your professor distributes a "sample exam" a week before exam day for you to study from. This is a reasonable thing to do. As long as the questions on the sample exam are of the same type and difficulty as the ones that will appear on the actual final, you'll learn lots about what the professor expects you to know from taking the sample exam. And you'll probably increase your actual exam score, since this will help you master exactly the right material.

But suppose the professor uses the *exact same* exam for both the sample exam and the actual final exam? Sure, the students would be ecstatic, but that's not the point. The point is: *in this case, students wouldn't even have to learn the material.* They could simply memorize the answers! And after they all get their A's back, they might be tempted to think they're really great at chemistry...but they probably aren't. They're probably just really great at memorizing and regurgitating.

Going from "the kinds of questions you may be asked" to "*exactly* the questions you *will* be asked" makes all the difference. And if you just studied the sample exam by memorization, and were then asked (surprise!) to demonstrate your understanding of the material on a *new* exam, you'd probably suck it up.

And so, the absolute iron-clad rule is this: **any data that is given to the classifier to learn from must *not* be used to test it.** The test data must be comprised of representative, but different, examples. It's the only way to assess how well the classifier **generalizes** to new data that it hasn't yet seen (which, of course, is the whole point).

Splitting the difference

Okay, so given that we have to split our precious labeled examples into two sets, one for training and one for testing, how much do we devote to each? It turns out that there are some sophisticated techniques (beyond the scope of this book, but stay tuned for Volume Two) in which we can cleverly re-use portions of the data for different purposes, and effectively make use of nearly all of it for training.

But for our introductory approach here, we'll just use a **rule of thumb: 70% for training data, and the other 30% for test data**.

As I mentioned earlier, we'll normally **shuffle** the rows randomly before dividing them into these two groups, just in case there's any pattern to the order in which they appear. For example, in our NFL fan data set, it might turn out that the data came to us sequenced in a way such that people living on the east coast were at the beginning of the `DataFrame` and those living out west were at the end. Any arrangement like this would spell doom for our classification endeavor. For one thing, we wouldn't be training on any west coast people, and so our classifier would be oblivious to what those data points looked like. For another thing, we'd only be using west coasters to *test* our classifier, meaning that whatever accuracy measure we computed is likely to be way off. **Randomizing** the data is the sure way around this.

Here's some code to create training and test sets. The `.sample()` method of a `DataFrame` lets you choose some percentage of its rows randomly. Its `frac` argument is a number between 0 and 1 and specifies what fraction of the rows you want. Using the above rule of thumb, let's choose 70% of them for our training data:

```
training = fans.sample(frac=.7)
print(training)
```

```
    age        hometown current_residence  years_in_residence      team
8    52       Arlington    Fredericksburg                  17    Ravens
15   63        New York           Orlando                  17    Giants
11   29  Fredericksburg           Seattle                  21  Seahawks
19   32       Warrenton           Orlando                   1    Ravens
2    47       Allegheny       Chevy Chase                  13    Ravens
7    31       Warrenton   Charlottesville                   6      Jets
0    29       Warrenton         Warrenton                  29    Ravens
5    32       Warrenton        Winchester                  11    Ravens
4    51   Orange County          Dumfries                  23    Ravens
9    60  Tyson's Corner      Falls Church                   4   Cowboys
10   17  Fredericksburg    Fredericksburg                  17    Ravens
1    39         Lubbock         Arlington                   2   Cowboys
13   35  Mechanicsville           Orlando                   8    Ravens
6    39        Dumfries             Miami                   5    Ravens
```

Notice that the numeric index values (far left) are in no particular order, since that's the point of taking a random sample. Also notice that there are only 14 rows in this `DataFrame` instead of the full 20 that were in `fans`.

Now, we want our test set. The trick here is to say: "give me all the rows of `fans` that were *not* selected for the `training` set." By building a query with the squiggle operator ("`~`", meaning "not") in conjunction with the "`.isin()`" method, we can create a new `DataFrame` called "`test`" that has exactly these rows:

```
test = fans[~fans.index.isin(training.index)]
print(test)
```

```
    age        hometown current_residence  years_in_residence      team
3    31       Littleton         Littleton                   3   Broncos
12   37        Richmond          Richmond                  37    Ravens
14   19        New York          New York                  19    Giants
16   42      Alexandria          Manassas                  10    Ravens
17   22       Allegheny        Harrisburg                   1  Steelers
18   38  Mechanicsville    Fredericksburg                   6    Ravens
```

That code says, in English: "create a new variable `test` that contains only those rows of `fans` whose index is *not* present in any of the `training` `DataFrame`'s indices." As you can verify through visual inspection, the result does have exactly the 6 rows that were missing from `training`.

26.3 "The prior"

One more piece of lingo before we dive into a particular classification technique next chapter. And that's known as "the **prior**" of a data set.

The term comes from something called **Bayesian reasoning**, which is a whole subject (and a super cool one!) in its own right. All you need to know here is the concept of two different quantities: the **prior**, and the **posterior**.

In common usage, the word "prior" means "beforehand," and so it does here: the prior is your best judgment about what the target value of a new example might be *before you actually look at the feature values in that example.* "Posterior," on the other hand, means "afterwards," and means your best judgment about the target value after duly taking into consideration all the feature values.

For example, you may have noticed that in my made-up data set, above, I had a lot of Ravens fans. This is because I live in the D.C. area, and happen to know a lot of Ravens fans. Out of my 20 labeled examples, a whopping twelve of them, in fact, had `Ravens` as their value in the `team` column.

Thus, consider the following question. Suppose you knew nothing about a person except that they were one of Stephen's friends. Which NFL team do you think they'd support? Assuming this data set is representative of Stephen's friends, you'd say: "I'd predict they'd be a Ravens fan, and I'd estimate that I'd have about a 60% chance of being right ($\frac{12}{20}$)." This is the *prior.* You're not taking into account anything about their age, where they were born, *etc.*; in fact, you weren't even told those things. Instead, you're just "using the prior" and treating everyone the same.

It would be a different story if I told you that this person was born in New York City. Then you might squint your eyes at my data set and realize that there are only two New Yorkers in it, and neither one is a Ravens fan: they're both Giants fans! Now you might very well move away from your prior assumption. "Sure, most of Stephen's friends are Ravens fans, so 'Ravens' is a reasonable guess,

but now that you've told me they're from NY, that very well might change my mind. Now, my guess is 'Giants'."

I keep saying "might" and "may" because different kinds of classifiers work in different ways. Some of them may choose to take advantage of some features but not others; some may just stick with the prior in certain situations. The notion of "the prior" is mainly useful as a baseline for comparison: it's the best you can do given no other possibly correlating information. The name of the game in classification, of course, is to intelligently *use* that other information to make more informed guesses, and to beat the prior. One of many ways to approach this is the decision tree classification algorithm, which we'll look at in detail next.

Chapter 27

Decision trees for classification (1 of 2)

So far our classification picture has been very general. We haven't said anything about how our classifier might actually *work*; we've just said that given values for each of the features, it will render a prediction about what the label will be.

In chapters 27 and 28, we'll study one particular algorithm for classification in machine learning: the **decision tree** algorithm. Not only does it make a good introductory technique because of its intuitive appeal, and not only can it classify pretty well in its own right, but it also serves as the basis for a more sophisticated, state-of-the-art classification method called "**random forest**" which we'll explore in Volume Two of this series.

27.1 A working example

Here's a (fictitious) domain problem that we'll use to demonstrate the principles in this chapter and the next. Say we own a videogame business, and we want to send full-color product catalogs to unsuspecting college students, so that they will buy our games and keep us in business (while meanwhile failing out of school due to playing games all the time).

Now full-color catalogs are expensive to print and ship, so we want

to be smart about this. We definitely don't want to send a bunch of catalogs to students who aren't likely buyers; that would run our business into the ground. Instead, we'd like to identify the subset of students who probably gamers, and send catalogs to only *those* students.

Suppose that through nefarious means, we have acquired the following data set:

	Major	Age	Gender	VG
0	PSYC	22	F	No
1	MATH	20	F	No
2	PSYC	19	F	No
3	CPSC	20	M	Yes
4	MATH	18	M	Yes
5	CPSC	20	F	No
6	CPSC	19	O	No
7	CPSC	17	M	Yes
8	PSYC	18	F	No
9	CPSC	20	F	No
10	MATH	18	F	No
11	CPSC	22	F	Yes
12	MATH	21	M	No
13	CPSC	23	M	Yes
14	PSYC	17	M	Yes
15	CPSC	18	F	No
16	PSYC	19	F	Yes

Each row represents one college student, with three features. The first is their major – `PSYC` (Psychology), `MATH` (Mathematics), or `CPSC` (Computer Science). (For simplicity, we'll say these are the only three possibilities, since your author happens to like them the best.) The second is their age (numeric), and the third is their gender: male, female, or other. The last column is our target: *whether or not this student is a videogamer.* Glance over this `DataFrame` for a moment.

Eyeing the prior

As you remember from section 26.3, before we even think about features, we might take a minute to just look at the target variable itself. We ask ourselves "given no other information about a student, what would be our gut feel about their videogame status?" Our pal the `.value_counts()` method is perfect to compute this:

```
print(students.VG.value_counts())
```

```
N    10
Y     7
Name: VG, dtype: int64
```

So if we're smart, we'd guess "no" for such mysterious persons, but we could only expect to be right about $\frac{10}{17}^{\text{ths}}$, or 59%, of the time. Not great, although better than a coin flip.

Sticking with categorical features

Now it turns out that decision trees work best with all categorical features, not a mix of categorical and numeric. So for now, we're going to simply classify each of our students into three buckets: "`young`" (18 or younger), "`middle`" (19-21), and "`old`" (22+).[1] For the moment, don't ask why we chose three age categories instead of two or four, and don't ask why we chose those particular split points. We just did. More on that later.

Our training data now looks like this:

[1] Believe it or not, a time will come in your life when 22 years of age does not remotely seem "old." For undergrads, though, I can see why 22 would seem on the grey side, the Taylor Swift song notwithstanding.

	Major	Age	Gender	VG
0	PSYC	old	F	No
1	MATH	middle	F	No
2	PSYC	middle	F	No
3	CPSC	middle	M	Yes
4	MATH	young	M	Yes
5	CPSC	middle	F	No
6	CPSC	middle	O	No
7	CPSC	young	M	Yes
8	PSYC	young	F	No
9	CPSC	middle	F	No
10	MATH	young	F	No
11	CPSC	old	F	Yes
12	MATH	middle	M	No
13	CPSC	old	M	Yes
14	PSYC	young	M	Yes
15	CPSC	young	F	No
16	PSYC	middle	F	Yes

and we're now officially ready to consider decision trees.

27.2 Decision Trees

First, let's get our head around what a decision tree *is*. Our inaugural example is shown in Figure 27.1. The first thing you'll notice is that it has a branching structure that branches...down. I'm not sure why Data Scientists draw trees growing *down* while the rest of the world (including trees themselves: look outside if you don't believe me) has them growing *up*, but this is the convention so we'll just deal with it. To make it even more comical, the oval at the top of the tree is called the **root** of the tree. Really.

Continuing full bore with the botany analogy, the lines connecting the various shapes are, as you might suspect, called **branches**, and the darker rectangles are called **leaves**. One non-botanic bit of lingo is the name for the other ovals: they're called **nodes**.

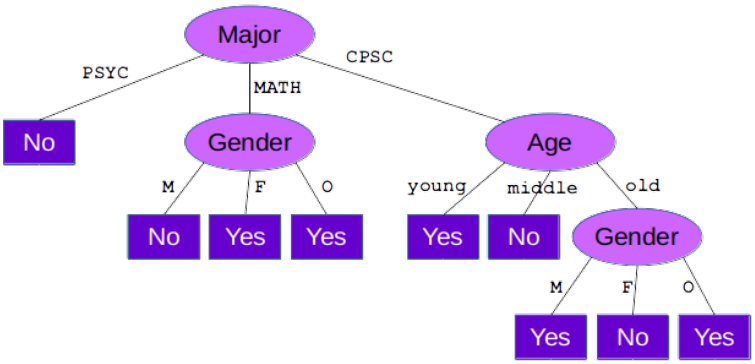


Figure 27.1: A decision tree (not a particularly good one, as it'll turn out) for the videogame data set.

Classifying with a decision tree

Okay. Now what does a decision tree "mean?" Geekily-put, it's the pictorial codification of an algorithm for classification. Not so geekily, it's a map that tells your classifier what rules to follow as it forms its prediction for an example data point.

You simply start at the root, considering feature values at each node, and following the matching branch down the tree. When you reach a leaf, the prediction you give is written on the leaf node. It's that simple.

☞ First example: suppose we have a 24-year-old male Psychology major. We want to know whether he's likely to play videogames. The decision tree in Figure 27.1 tells us to first consider his Major, since that's the root. Now because this guy's major is `PSYC`, we take the left branch and are immediately done: we've already reached a leaf. Our prediction for this guy will be `No`, he probably doesn't play videogames.

☞ Second example: we have an 18-year-old Math major who doesn't identify with either of the binary genders. Starting again at the root, we now follow the middle branch for `MATH`. Now, we look at the person's Gender. Since it is `O`, we follow

the right branch, and give a prediction of `Yes`: we predict they *do* play videogames.

☞ Third example: we now have a 22-year-old female Computer Science major. Do we think she would play videogames? The root tells us to look at her Major first, which means we go right; then we look at her Age, and since she's positively ancient we go right again; and finally, her Gender tells us to predict `No`, she's probably not a gamer.

Most students find this process very straightforward. In the next chapter, we'll look at two key questions: first, how to turn a diagram like Figure 27.1 into Python code? And second, what's the best way to make a *good* tree – *i.e.*, one that makes as many successful predictions as possible?

Chapter 28

Decision trees for classification (2 of 2)

28.1 Decision Trees in Python

Our decision tree pictures from chapter 27 were quite illustrative, but of course to actually automate something, we have to write code rather than draw pictures. What would Figure 27.1 (p. 273) look like in Python code? It's actually pretty simple, although there's a lot of nested indentation. See if you can follow the flow in Figure 28.1.

Here we're defining a function called `predict()` that takes three arguments, one for each feature value. The eye-popping set of `if`/`elif`/`else` statements looks daunting at first, but when you scrutinize it you'll realize it perfectly reflects the structure of the purple diagram. Each time we go down one level of the tree, we indent one tab to the right. The body of the "`if major == 'PSYC':`" statement is very short because the left-most branch of the tree (for Psychology) is very simple. The "`elif major == 'CPSC':`" body, by contrast, has lots of nested internal structure precisely because the right-most branch of the tree (for Computer Science) is complex. *Etc.*

```
def predict(major, age, gender):
    if major == 'PSYC':
        return 'No'
    elif major == 'MATH':
        if gender == 'M':
            return 'No'
        elif gender == 'F' or gender == 'O':
            return 'Yes'
    elif major == 'CPSC':
        if age == 'young':
            return 'Yes'
        elif age == 'middle':
            return 'No'
        elif age == 'old':
            if gender == 'M' or gender == 'O':
                return 'Yes'
            elif gender == 'F':
                return 'No'
```

Figure 28.1: A Python implementation of the decision tree in Figure 27.1.

If we call this function, it will give us exactly the same predictions we calculated by hand on p. 273:

```
print(predict('PSYC','M','old'))
print(predict('MATH','O','young'))
print(predict('CPSC','F','old'))
```

```
No
Yes
No
```

28.2 Decision Tree induction

Okay, so now we understand what a decision tree is, and even how to code one up in Python. The key question that remains is: how do we figure out what tree to build?

There are lots of different choices, even for our little videogame example. We could put any of the three features at the root. For

each branch from the root, we could put either of the other features, or we could stop with a leaf. And the leaf could be a `Yes` leaf or a `No` leaf. That's a lot of "coulds." How can we know what a *good* tree might be – *i.e.*, a tree that classifies new points more or less correctly?

The answer, of course, is to take advantage of the training data. It consists of labeled examples that are supposed to be our guide. Using the training data to "learn" a good tree is called **inducing** a decision tree. Let's see how.

"Greedy" algorithms

Our decision tree induction algorithm is going to be a **greedy** one. This means that instead of looking ahead and strategizing about future nodes far down on the tree, we're just going to grab the immediate best-looking feature at every individual step and use that. This won't by any means guarantee us the best possible tree, but it will be quick to learn one.

An illustration to help you understand greedy algorithms is to think about a strategy game like chess. If you've ever played chess, you know that the only way to play well is to think ahead several moves, and anticipate your opponent's probable responses. You can't just look at the board naïvely and say, "why look at that: if I move my rook up four squares, I'll capture my opponent's pawn! Let's do it!" Without considering the broader implications of your move, you're likely to discover that as soon as you take her pawn, she turns around and takes your rook because she's lured you into a trap.

A *greedy* algorithm for chess would do exactly that, however. It would just grab whatever morsel was in front of it without considering the fuller consequences. That may seem really dumb – and it is, for chess – but for certain other problems it turns out to be a decent approach. And decision tree induction is one of those.

The reason we resort to a greedy algorithm is that for any real-sized data set, *the number of possible trees to consider is absolutely overwhelming.* There's simply not enough time left in the universe

to look at them all – and that's not an exaggeration. So you have to find *some* way of picking a tree without actually contemplating every one, and it turns out that grabbing the immediately best-looking feature at each level is a pretty good way to do that.

Choosing "the immediate best" feature

Now what does that mean, anyway: "choosing the immediate best feature?" We're going to define it as follows: the best feature to put at any given node is *the one which, if we did no further branching from that node but instead put only leaves below it, would classify the most training points correctly.* Let's see how this works for the videogame example.

Our left-most feature in the `DataFrame` is Major, so let's consider that one first. Suppose we put Major at the root of the tree, and then made each of its branches lead to leaves. What value should we predict for each of the majors? Well, we can answer that with another clever use of `.value_counts()`, this time conjoining it with a call to `.groupby()`. Check out this primo line of code:

```
students.groupby('Major').VG.value_counts()
```

Stare hard at that code. You'll realize that all these pieces are things you already know: we're just combining them in new ways. That line of code says "take the entire `students DataFrame`, but treat each of the majors as a separate group. And what should we do with each group? Well, we count up the values of the `VG` column for the rows in that group." The result is as follows:

```
Major  VG
PSYC   No     3
       Yes    2
MATH   No     3
       Yes    1
CPSC   No     4
       Yes    4
Name: VG, dtype: int64
```

We can answer "how many would we get right?" by reading right off that chart. For the PSYC majors, there are two who play videogames and three who do not. Clearly, then, if we presented a Psychology major to this decision tree, it ought to predict 'No', and that prediction would be correct for 3 out of the 5 Psychology majors on record. For the MATH majors, we would again predict 'No', and we'd be correct 3 out of *4* times. Finally, for the CPSC majors, we have 4 Yeses and 4 Nos, so that's not much help. We essentially have to pick randomly since the training data doesn't guide us to one answer or the other. Let's choose 'Yes' for our Computer Science answer, just so it's different than the others. The best one-level decision tree that would result from putting Major at the top is therefore depicted in Figure 28.2. It gets **ten** out of the seventeen training points correct (59%). Your reaction is probably "Big whoop – we got that good a score just using the prior, and ignoring all the features!" Truth. Don't lose hope, though: Major was only one of our three choices.

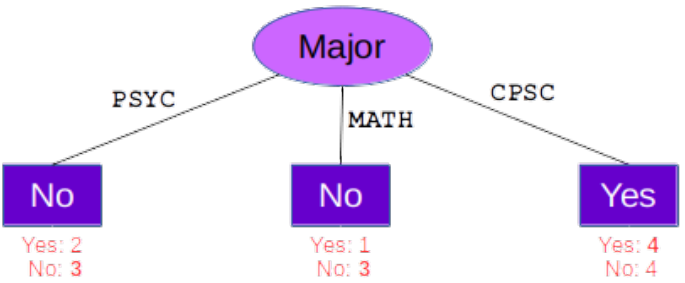


Figure 28.2: A one-level decision tree if we put the Major feature at the root – it would classify **ten** of the seventeen training points correctly.

Let's repeat this analysis for the other two features and see if either one fares any better. Here's the query for Age:

```
students.groupby('Age').VG.value_counts()
```

This yields:

```
Age      VG
middle   No     6
         Yes    2
old      Yes    2
         No     1
young    No     3
         Yes    3
Name: VG, dtype: int64
```

Making the sensible predictions at the leaves based on these values gives the tree in Figure 28.3. It gets **eleven** points right (65%) – a bit of an improvement.

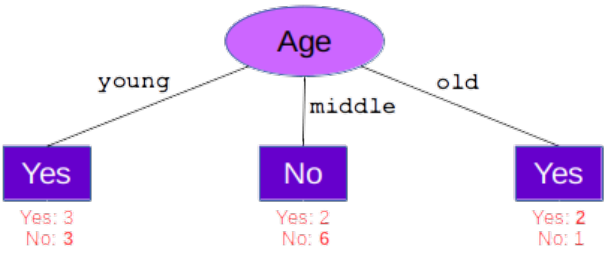


Figure 28.3: A one-level decision tree if we chose the Age feature for the root – it would classify **eleven** of the seventeen training points correctly.

Finally, we could put Gender at the root. Here's the query for it:

```
students.groupby('Gender').VG.value_counts()
```

```
Gender   VG
F        No     8
         Yes    2
M        Yes    5
         No     1
O        No     1
Name: VG, dtype: int64
```

Paydirt! Splitting on the Gender feature first, as shown in Figure 28.4, gets us a whopping **fourteen** points correct, or over 82%.

This is clearly the winner of the three. And since we're being greedy and not bothering to look further downstream anyway, we hereby elect to put Gender at the root of our tree.

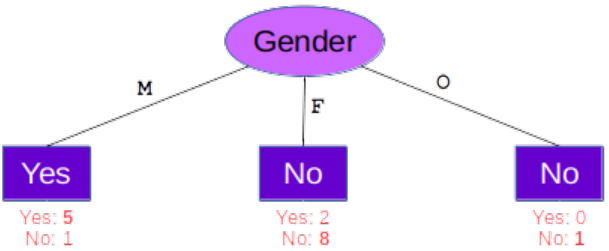


Figure 28.4: A one-level decision tree if we chose the Gender feature for the root. It would classify **fourteen** of the seventeen training points correctly – easily the best of the three choices.

It's worth taking a moment to look at those `.value_counts()` outputs and see if you can develop some intuition about why Gender worked so much better at the root than the other two features did. The reason is that for this data set, Gender split the data into groups that were more **homogeneous** than the other splits gave. "Homogeneous" here means that each group was more "pure," or put another way, more lopsided towards one of the labels. Gender gave us a 5-to-1 lopsided ratio on the `M` branch, and an even more lopsided 2-to-8 ratio on the `F` branch. Intuitively, this means that Gender really is correlated with videogame use, and this shows up in purer splits. Contrast this with the situation when we split on Major first, and we ended up with a yucky 4-to-4 ratio on the `CPSC` branch. An even split is the worst of all possible worlds: here, it means that learning someone's a Computer Science major doesn't tell you jack about their videogame use. That in turn means it's pretty useless to split on.

Lather, rinse, repeat

So far, we've done all that work just to figure out which feature to put at the root of our tree. Now, we progress down each of the branches and do the exact same thing: figure out what to put at

each branch. We'll continue on and on like this for the entire tree. It's turtles all the way down.

Let's consider the *left* branch of Figure 28.4. What do we do with males? There are now only two remaining features to split on. (It wouldn't make sense to split on Gender again, since the only people who will reach the left branch are males anyway: there'd be nothing to split on.)

Thus we could put either Major or Age at that left branch. To figure out which one is better, we'll do the same thing we did before, only with one slight change: now, we need to consider *only males* in our analysis.

We augment our primo line of code from above with a query at the beginning, so that our counts include only males:

```
students[students.Gender=="M"].groupby('Major').VG.value_counts()
```

```
Major  VG
CPSC   Yes    3
MATH   No     1
       Yes    1
PSYC   Yes    1
Name: VG, dtype: int64
```

Wow, cool: the CPSC and PSYC folks are perfectly homogeneous now. If we end up deciding to split on Major here, we can put permanent dark purple squares for each of those majors simply declaring "Yes." In all, splitting here gives us 5 out of 6 correct. The tree-in-progress we'd end up with is in Figure 28.5.

Our other choice, of course, is to split on Age instead:

```
students[students.Gender=="M"].groupby('Age').VG.value_counts()
```

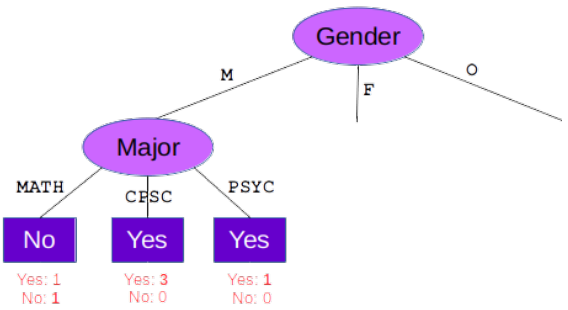


Figure 28.5: The tree-in-progress if we choose to split on Major in the male branch.

```
Age      VG
middle   No     1
         Yes    1
old      Yes    1
young    Yes    3
Name: VG, dtype: int64
```

Again, 5 out of 6 correct. Here, `middle`-aged students are the only heterogeneous group; the `old` folks and `young`-uns are clean splits. With this choice, our tree would appear as in Figure 28.6.

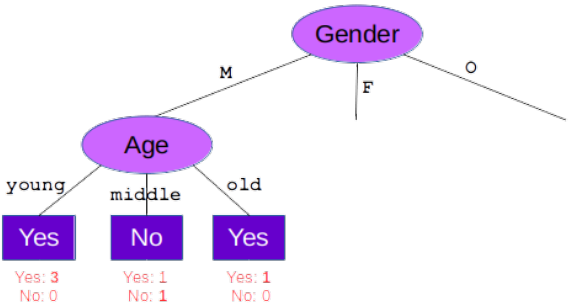


Figure 28.6: On the other hand, the tree-in-progress if we choose to split on Age in the male branch instead.

So at this point, since splitting on either feature and then stopping would give us exactly 5 out of 6 points correct, we just flip a coin. I

just flipped one, and it came out tails (for Age) – hope that's okay with you.

Finishing up the left branch

The two "Yes" leaves in Figure 28.6 are now set in stone, since every single `young` male in our training set was indeed a videogamer, as was every `old` male. Now we just have to deal with the `middle` branch.

Only one feature now remains to split on – Major – so we'll do that, and produce the result in Figure 28.7. There's exactly one `middle`-aged male `MATH` major in the original `DataFrame` (line 12, p. 272), and he's labeled "No," so we'll guess "No" in the `MATH` branch. Similarly, we have one data point to guide us for `CPSC` majors (line 3), so we'll predict "Yes" in this case. The `PSYC` branch presents a conundrum, though: our data set doesn't have *any* `middle`-aged male Psychology majors, so how do we know what to guess in this case?

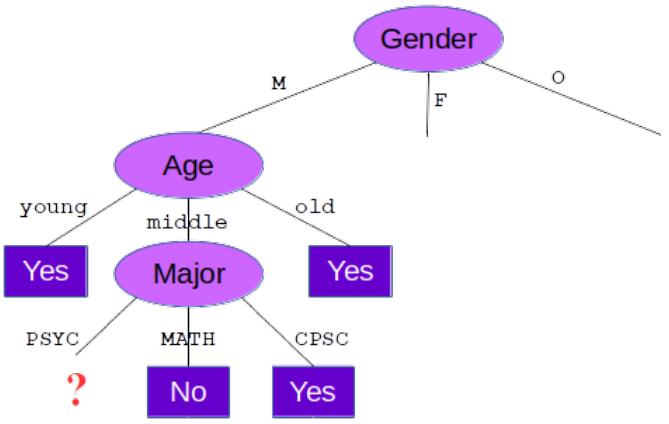


Figure 28.7: Going one level further down after splitting on Age for males. We have data for `middle`-aged `CPSC` and `MATH` males...but what to do with `middle`-aged `PSYC` males?

The best way to handle this is to fall back to a more general case where you *do* have examples. It's true that we have no training

points for `middle`-aged male Psychology majors, but we *do* have points for `middle`-aged males-in-general, and we discovered that 5 out of 6 of them were gamers. So it makes sense to default to "Yes" in the `PSYC` branch of this part of the tree, even though we don't have any data points exactly like that. So that's what we'll do. The finished left branch is depicted in Figure 28.8.

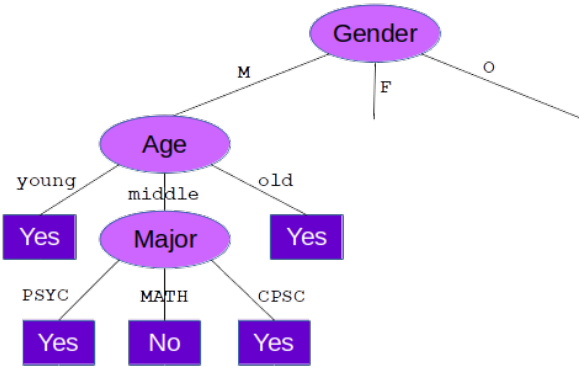


Figure 28.8: The decision tree we're in the process of inducing, with the left branch entirely completed.

Finishing up the rest of the tree

The rest of the process is just the same stuff done over and over.[1] At each branch of the tree, we take the subset of the training points that remain (*i.e.*, the training points that match the path from the root thus far, and are therefore applicable), and decide what to branch on next. When we get to a completely homogeneous group, we stop and put a leaf there. The end result of all these efforts is

[1] If you're wondering whether there's a way to automate this, the answer is a resounding **yes**! There are many packages in Python and other languages which will automatically build a decision tree from a training set; the `DecisionTreeClassifier` from the `scikit-learn` package is one of them. This exercise of learning how to build a decision tree manually is so you can understand the concepts of what's going on under the hood – kind of like you learn how to add numbers in grade school even though you'll normally use a calculator later on in life.

the final decision tree for the videogame data set, in Figure 28.9, and its Python equivalent in Figure 28.10.

One interesting aspect of our final tree is the female→`PSYC`→`middle`-aged branch. You'll see that this leaf is labeled "Yes(?)" in the diagram. Why the question mark? Because this is the one case where we have a **contradiction** in our training data. Check out lines 2 and 16 back on p. 272. They each reflect a `middle`-aged female Psychology major, but with *different* labels: the first one is not a videogame player, but the second one is.

I always thought the term "contradiction" was amusing here. Two similar people don't have exactly the same hobbies – so what? Is that really so surprising? Do all middle-aged female Psychology majors have to be identical?

Of course not. But you can also see things from the decision tree's point of view. The only things it knows about people are those three attributes, and so as far as the decision tree is concerned, the people on lines 2 and 16 really *are* indistinguishable. When contradictions occur, we have no choice but to fall back on some sort of majority-rules strategy: if out of seven otherwise-identical people, two play videogames and five do not, we'd predict "No" in that branch. In the present case, we can't even do *that* much, because we have exactly one of each. So I'll just flip a coin again. (`*flip*`) It came up heads, so we'll go with "Yes."

Notice that in this situation, the resulting tree will actually misclassify one or more *training* points. If we called our function in Figure 28.10 and passed it our person from line 2 (`'PSYC'`, `'middle'`, `'F'`), it would return `"Yes"` even though line 2 is not a gamer. Furthermore, contradictions are the *only* situation in which this will ever happen; if the data is contradiction-free, then every training point will be classified correctly by the decision tree.

Paradoxically, it turns out that's not necessarily a good thing, as we'll discover in Volume Two of this series. For now, though, we'll simply declare victory.

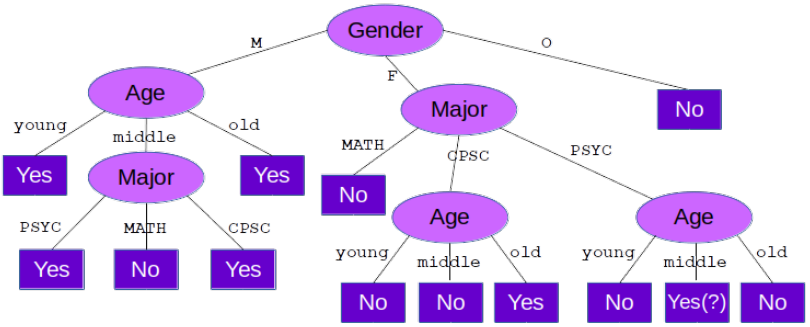


Figure 28.9: The final decision tree for the videogame data set.

```
def predict(major, age, gender):
    if gender == 'M':
        if age == 'young':
            return 'Yes'
        elif age == 'middle':
            if major == 'PSYC' or major == 'CPSC':
                return 'Yes'
            elif major == 'MATH':
                return 'No'
        elif age == 'old':
            return 'Yes'
    elif gender == 'F':
        if major == 'MATH':
            return 'No'
        elif major == 'CPSC':
            if age == 'young' or age == 'middle':
                return 'No'
            elif age == 'old':
                return 'Yes'
        elif major == 'PSYC':
            if age == 'young' or age == 'old':
                return 'No'
            elif age == 'middle':
                return 'Yes'  # Here's our "contradiction"
    elif gender == 'O':
        return 'No'
```

Figure 28.10: The final decision tree for the videogame data set, as a Python function.

Chapter 29

Evaluating a classifier

Once we've built a classifier – whether it's a decision tree or any other kind – the next step is to evaluate it to see how well it performs. This is sometimes called the classifier's **performance**. It will determine whether we deem it accurate enough to set it loose in the field, and if so, how accurate we can expect its predictions to be.

At the risk of repeating myself (recall my stern lecture from section 26.2) you must evaluate your classifier by testing it on data that was *not* used to train it. On p. 265 we learned how to randomly divide a data set into separate training and test sets.

Suppose we've done that. Suppose the `students` `DataFrame` from chapters 27 and 28 was the result of randomly choosing 70% of the labeled examples from an original data set, and that we have preserved the remaining 30% of the rows in a `DataFrame` called `students_test`, which contains:

	Major	Age	Gender	VG
0	CPSC	young	F	No
1	CPSC	old	O	No
2	MATH	old	F	No
3	CPSC	middle	M	No
4	PSYC	middle	F	Yes
5	PSYC	young	F	No
6	MATH	old	M	Yes
7	CPSC	middle	M	Yes

Our question is: "how well does our classifier do on this test data?"

29.1 What "doing well" means

The most common (and simplest) way to measure a classifier's performance is to simply count how many of the test points it correctly classifies, and divide by the total number of test points. This gives us the classification **accuracy** as a fraction between 0 and 1 (or, if we want to multiply by 100, a "percentage accuracy" from 0% to 100%.) It's possible to do this because, as you'll remember, our test data is comprised of *labeled* examples, just like our training data is. Therefore, we know the "right answer" for each test point, and we can simply compare it to our classifier's prediction.

Even though this is the most common approach, it's worth taking a moment to consider alternatives. The key assumption of this accuracy measure is that *all kinds of prediction errors are equal.* In the videogame case, we're saying that mistakenly labeling a videogamer as a non-videogamer is "just as bad" as mistakenly labeling a non-videogamer as a videogamer. And that might be just the right thing for our gaming company to do.

But consider other settings. Suppose that our classifier's inputs are features from an MRI image, and our prediction is "cancer" or "no cancer." Now, it's a much different story. Mistakenly predicting that a certain patient has cancer when they actually don't might throw a needless scare into them. That's bad. But it's far worse

in the other direction: mistakenly giving a clean bill of health to a patient who actually has early stage cancer risks losing a life. In cases like this, we would need to penalize our classifier more harshly for false negatives than for false positives.

It's also a different story when the labels aren't equally represented. Recall the NFL fan prediction problem from Figure 26.1 (p. 262). Consider if we performed fan prediction in a city like Dallas, which is comprised of (say) 99% Cowboys fans and only 1% Ravens fans. If we were to penalize a classifier equally for mistaken-Cowboy-predictions and mistaken-Ravens-predictions, a one-line classifier could earn a pretty good score:

```
def predict(age, hometown, current_residence, yrs_in_residence):
    return "Cowboys"
```

It's not even worth trying hard to ferret out the few Ravens fans if we're going to be docked a full point every time we dare to predict one. They're just too rare. The only way to get a classifier to be bold and try to identify the tiny population of Ravens fans is to penalize it more heavily for missing them than for falsely identifying them.

Anyway, for the rest of this chapter, we'll use the vanilla "count all prediction mistakes equally" approach, but it's worth remembering that this doesn't make sense in all situations.

29.2 Calculating accuracy in Python

Calculating your classifier's accuracy is actually a snap. Once your classifier's code is in a function, you just need a loop.

Return to the videogame example from last chapter, and the decision tree classifier we wrote on p. 287. We'll use a counter variable, initialized to zero, that will keep track of our number of correct predictions. We'll then loop through each row of the test set, feeding that row's features to the classifier function. If the return value

from the classifier matches the value of that row's target, ka-ching! We increment our counter to increase our score. If it doesn't, we don't. At the end, we divide by the number of test points to get our percentage. Simple!

```
count = 0
for row in students_test.itertuples():
    if predict(row.Major, row.Age, row.Gender) == row.VG:
        count += 1

accuracy = count / len(students_test) * 100
print("Our accuracy on the test set was {}%.".format(accuracy,
    count, len(students_test)))
```

```
Our accuracy on the test set was 87.5%.
```

If we want more detail, we could print a message for each prediction, and flag the incorrect ones for easy identification:

```
count = 0
for row in students_test.itertuples():
    if predict(row.Major, row.Age, row.Gender) == row.VG:
        print("  Predicted {}/{}/{} right!".format(row.Major,
            row.Age, row.Gender))
        count += 1
    else:
        print("X Predicted {}/{}/{} wrong. :(".format(row.Major,
            row.Age, row.Gender))

accuracy = count / len(students_test) * 100
print("Our accuracy on the test set was {}% ({}/{}).".format(
    accuracy, count, len(students_test)))
```

```
  Predicted CPSC/young/F right!
  Predicted CPSC/old/O right!
  Predicted MATH/old/F right!
X Predicted CPSC/middle/M wrong. :(
  Predicted PSYC/middle/F right!
  Predicted PSYC/young/F right!
  Predicted MATH/old/M right!
  Predicted CPSC/middle/M right!
Our accuracy on the test set was 87.5% (7/8).
```

Not too shabby. As you can see, the only test point we missed was the male `middle`-aged `CPSC` major, which our classifier figured would be a videogamer. Live and learn.

The data size here is laughably small so that I can fit everything on the page. But it's worth considering these three quantities anyway:

Classifier's performance on training set	94.1% (16/17)
Classifier's performance on test set	82.5% (7/8)
Just using the prior on test set	62.5% (5/8)

These three quantities will nearly always be in this order from top to bottom. When we test our classifier on the very data it was trained on, we get an inflated view of its accuracy – for decision trees, recall, it will always be 100% less any contradictions. Testing it on the data it has not yet seen gives the truer (more realistic) picture. Finally, your classifier had better outperform just using the prior (here, choosing "No" because the majority of training points were "No") or this whole thing is a pretty useless enterprise!

Index