week 03: data exploration and visualization

reading for this week

- David Donoho, *50 years of data science*, unpublished manuscript (2015)
- Hadley Wickham, *Tidy data*, Journal of Statistical Software (2014)
- Tracey Weissgerber *et al.*, *Beyond bar and line graphs: time for a new data presentation paradigm*, PLOS Biology (2015)

on data science

The term "data science" seems to have been introduced in its current sense by W. S. Cleveland in 2001. David Donoho's *50 years of data science* is an excellent summary of what "data science" means, especially from the perspective of how academic disciplines are organized, and how they change over time.

W.S. Cleveland, *Data science: an action plan for expanding the technical areas of the field of statistics*, International Statistical Review, 2001.

At first glance (and subsequent glances too) the term "data science" is a bit silly, a tautology. All sciences do data analysis. Even if you're talking 'big data', Pierre Laplace was doing large scale data analysis on Paris and London census records in *1781* (which we'll discuss next week). As the statistician Karl Broman has asked, when a physicist does a mathematical analysis, does she say she's doing "number science"?

Donoho makes a case that "data science" is a rebranding exercise, and his piece gives interesting insight into disciplinary fad and fashion. But since we biologists are still in in the midst of "systems biology", "integrative biology", and "quantitative biology" (and a whole lot of -omes) we won't throw too many stones in the glass house of branding "new" subdisciplines.

There must be something interesting and real about the idea of "data science", or it wouldn't be so high in our collective consciousness. Donoho argues that data science is a rebranding of statistics by *outsiders*, and that statistics departments opened the door to this insurgency by downplaying and avoiding the messy practicalities of real-world data analyses. Donoho repeats a famous quote that 80% of the effort in data analysis is just being able to deal with the messiness of data at all: reformatting, exploring, cleaning, visualizing. Computer scientists, in particular, have taken up those practical challenges by necessity and opportunity, and re-invigorated an old area. Even Donoho turns up his nose a bit at computational technologies that seem to him like mere "coping skills for dealing with organizational artifacts of large-scale cluster computing", but being able to compute effectively on terabytes or petabytes of data does present serious, expensive, and interesting practical challenges.

In biology, that adage about 80% of the effort in data analysis might even be an understatement. Biology is particularly ready for more emphasis on the key ideas of data science: the practicalities of data exploration, data cleaning, and data visualization, and the coding skills to be able to do it. This week, we're going to focus on some of those practicalities.

a data paranoia toolkit

The first thing you do with a new data file is look at it. But, you say, come on, my file is too big to read the whole thing. Here's a litany of basic techniques for dealing with big data files, aiming to get big unmanageable data sets down to little data sets that human, intuitive biological superpowers can work on:

- slice: look at the start and the end
- select: look at known cases
- sample: look at a random sample
- sort: look for outliers
- summarize: look for more outliers
- validate: the whole file, once you think you understand it.

We'll focus here on simple line-oriented data files, but the same ideas apply to more complex formats with multiple lines per record.

look at the start, end

The best place to start is at the beginning. The start of the file tends to include helpful information like column headers, comments, and other documentation. If you're going to parse the file, this might help -- plus your parser might have to know to skip or specially deal with header material.

The easiest way to look at the start of a text file is on the command line:

```
% head -10 w03-data.tbl
```

shows you the first 10 lines of the file. (Obviously, you can use this to get however many you want to read.)

Worth checking the tail end of the file too. It might end in a special kind of line that you'd need to parse. A corrupted file (in a truncated download) might end abruptly in an obvious way. Some files don't end in a newline character, and that might screw up your parser if you aren't expecting it. Again, the command line way is:

```
% tail -20 w03-data.tbl
```

which shows you the last 20 lines.

Of course, if the file is small enough that you could just look at the whole thing, for example using `more` on the command line:

```
% more w03-data.tbl
```

select known cases

The superpower of a biologist looking at a fresh new data set: 'let's look at my favorite gene... oh wait, that's not right, oh good lord.'

Look at a few cases where you know the right answer. For instance, in a cell type specific RNA-seq data set, look at a transcript that's supposed to be on in that cell type, and look at one that's supposed to be off. If you're going to worry about cross-contamination in the data, look at highly expressed genes in other nearby cell types (the most likely cross-contaminants). If you're going to worry about sensitivity of detecting rarer transcripts, find a low-expressed transcript to look at.

For pulling specific lines out of a file, `grep` is a go-to. This will print all lines in the file `w03-data.tbl` that contain the string `coriander`:

```
% grep coriander w03-data.tbl
```

take a random sample

Many data files are arranged in a nonrandom order, which means that just looking at the start and the end might not reveal some horrific systematic problem. Conversely, looking at the head and tail can make it look like the data file's worse than it actually is: genome sequence files, ordered by chromosome coordinate, can look terrible when they start and end with great swaths of N's, which is just placeholding for the difficult-to-sequence telomere repeats.

For small to medium sized files, if you don't care about efficiency, there are plenty of ways to randomly sample *N* lines, One quick incantation (when you have a modern GNU-ish `sort` program installed) is to sort the file randomly (the `-R` option) and take the first *N* lines:

```
    % sort -R w03-data.tbl | head -10
```

For large files, where efficiency matters, we're going to introduce the *reservoir sampling* algorithm in just a moment.

There's a great Feynman story about the power of random sampling. Feynman was called in to consult on the design of a big nuclear plant, looking over engineering blueprints that he didn't understand at all, surrounded by a crowd of actual engineers. Just trying to start somewhere and understand something small, he points randomly to a symbol and says "what happens if that valve gets stuck?". Not even sure that he's pointing to a valve! The engineers look at it, and then at each other, and say "well, if *that* valve gets stuck... uh... you're absolutely right, sir" and they pick up the blueprints and walk out, thinking he's a genius.

I wonder how many other problems there were in that design...

sort to find outliers

In a tabular file, you can sort on each column to look for weird values and outliers (and in other data formats, you can script to generate tabular summaries that you can study this way too). For example, this will sort the data table on the values in the third column (`-k3`), numerically (`-n`), and print the first ten:

```
    % sort -n -k3 w03-data.tbl | head
```

`-r` reverses the order of the sort, to look at the last ten. `sort` has many other useful options too.

Bad values tend to sort to the edges, making them easier to spot.

summarize to find more outliers

You can also find ways to put all the values in a column through some sort of summarization. Here your go-to is the venerable `awk`, which makes it easy to pull single fields out of the lines of tabular files: `awk '{print $3}' myfile`, the world's simplest script, prints the third field on every one of `myfile`.

I have a little Perl script of my own called `avg` that calculates a whole bunch of statistics (mean, std.dev., range...) on any input that consists of one number per line, and I'll do something like:

```
% awk '{print $3}' w03-data.tbl | avg
```

You don't have my `avg` script, but you can write your own in Python.

If the values in a column are categorical (in a defined vocabulary), then you can summarize all the values that are present in the column:

```
% awk '{print $2}' myfile | sort | uniq -c
```

where the `uniq` utility collapses identical adjacent lines into one, and the `-c` option says to also print the count, how many times each value was seen.

For something sequential, like DNA or protein sequences, you can whip up a quick script to count character composition; you may be surprised by the non-DNA characters that many DNA sequence files contain.

Another trick is to count the length of things in a data record (like sequence lengths), sort or summarize on that, and look for unreasonable outliers.

validate

By now you have a good mental picture of what's supposed to be in the file. The pathologically compulsive, which of course includes all careful scientists among us, might now write a validation script: a script that looks automatically over the whole file, making sure that each field is what it's supposed to be (number, string; length, range; whatever).

Of course, you're not going to do all of these paranoid steps on every data file you encounter. The point is that you *can*, when the situation calls for it. It only takes basic scripting and command line skills to do some very powerful and systematic things at scale, enabling you (as a biologist) to wade straight into a large dataset and wrap your mind around it.

reservoir sampling algorithm

Suppose you're trying to take a random sample of a really large data set. Suppose the data set is so large that it doesn't fit in your computer's memory. Suppose you don't even know how large it is; you're going to read one element at a time in a very long stream, until you run out of elements. Is it possible to sample m elements from n total elements, where n is unknown (until the end), such that each element is sampled uniformly with probability $p = \frac{m}{n}$?

Algorithm: reservoir sampling

- allocate a *reservoir* to hold up to m elements, with "slots" $1 \ldots m$.
- put the first m elements in reservoir slots $1 \ldots m$.
- for each subsequent element k, starting with $k = m + 1$, up to the last one $k = n$:
    - choose a random number $r = [1 \ldots k]$
    - if $r <= m$, swap element k into reservoir slot r, discarding the element that was there. (That is, each element *k*

initially gets put in the reservoir with probability $\frac{m}{k}$).

Our first complete algorithm in the course, I think! One of the things we want you to learn is how to read a pseudocode algorithm and implement it in your own Python code. You'll find that one detail you always need to keep track of (obsessively) is that pseudocode algorithms often count with indexes starting from 1, but in Python, lists are indexed starting from 0: "off-by-one" errors are easy to make.

Reservoir sampling is efficient in both time and memory. It becomes the preferred sampling algorithm when files get large. Writing your own script gives you the ability to ignore non-data parts of the file (here, comment lines) and to parse the file into data records any way you need to (not just one line at a time). Knowing how to write a quick reservoir sampling script gives you the superpower of being able to randomly downsample any data file, no matter how large the file, and no matter the format of the data.

proof that it works

We prove it in two cases. First consider the case of element $i \leq m$, which gets into the original sample automatically. At each iteration $k = m + 1$ to n, you have a chance to knock it out of the reservoir; the probability that it survives at iteration k is $\frac{k-1}{k}$. To make it into the final sample, it has to survive every round, so:

$$\begin{aligned} P(i \text{ survives}) &= \prod_{k=m+1}^{n} \frac{k-1}{k} \\ &= \frac{m}{m+1} \cdot \frac{m+1}{m+2} \cdot \frac{m+2}{m+3} \dots \frac{n-1}{n} \\ &= \frac{m}{n} \end{aligned}$$

which is what we want: the probability that *i* is sampled is $\frac{m}{n}$. Obviously the key thing here is how every numerator neatly cancels its preceding denominator.

Now consider the case of an element $i > m$. The probability that it gets into the reservoir when its turn comes is $\frac{m}{i}$. Then it has to survive getting knocked out by any subsequent element $k = i + 1..n$, so

$$\begin{aligned}P(i \text{ selected and survives}) &= \frac{m}{i} \prod_{k=i+1}^{n} \frac{k-1}{k} \\ &= \frac{m}{i} \cdot \left[\frac{i}{i+1} \cdot \frac{i+1}{i+2} \cdot \frac{i+2}{i+3} \ldots \frac{n-1}{n}\right] \\ &= \frac{m}{n}\end{aligned}$$

The same cancellation, but from a different initial condition.

Two points to make about this proof:

- if you don't remember anything about the proof, you'll probably have to look up the reservoir sampling algorithm every time you implement it, because it's easy to get muddled up on off-by-one issues. But if you even remember the outline of the proof, you'll remember that the final sample n has to be sampled with probability $\frac{m}{n}$, and that will remind you that the recursive rule is to sample each element i with probability $\frac{m}{i}$.

- I looked this up on Google to double check that I had the proof right, and the first m pages I looked at were complicated, confusing, and/or wrong.

computational complexity - big O notation

How many steps does the reservoir sampling algorithm take? It does a couple of operations for each element n (sample a random number;

compare it to m; maybe swap element k into the reservoir). If we sample from a data set that's 10x larger, it will take us 10x more operations. We say that the algorithm is $O(n)$ time, "order n in time".

How much memory does it take? We only need to hold the reservoir's m elements in memory, plus one current element k. We say the algorithm is *O(m)* memory, "order m in memory".

We saw that another way to take a random sample is to sort the n elements randomly and take the first m of them. In-memory sorting algorithms are typically $O(n \log n)$ time and $O(n)$ memory. ($\log n$ terms often arise in algorithms that are doing some sort of binary divide-and-conquer, like recursively sorting elements to the left or right of some middle point.) Which algorithm should we prefer?

The big-O of these algorithms tells us that at some point, for large enough $n >> m$, reservoir sampling is the preferred algorithm in both time and memory, because $O(n)$ is better than $O(n \log n)$ time, and $O(m)$ is better than $O(n)$ memory.

But: Big-O notation is only about *relative* scaling as the problem size n increases; it ignores constant factors. If $O(n)$ algorithm A happens to require 1000000 operations per element and $O(n \log n)$ algorithm B does 1 operation per element, algorithm B is actually faster in practice until *n* gets really big.

It can be difficult to predict how long an individual operation takes, especially on a modern computer where CPU operations are blazingly fast and random memory accesses are like sending a pigeon to the moon. Algorithms can be analyzed theoretically to determine their big-O, but implementations also must be analyzed empirically to determine how long they actually take, on a given computer.

Big-O notation has a more formal definition in computer science. The most important thing to know about the formalism is that big-O reflects *worst case* performance, an asymptotic upper bound: "O(g(n)) means there exists a constant *c* such that the time required is $\leq c \cdot g(n)$ for all *n*". There are also ways to talk about *best case* behavior $\Omega()$ and *average case* behavior $\Theta()$. Some algorithms, including reservoir sampling, always take the same number of steps. A good example where it makes a difference is a popular sorting algorithm called *quicksort*, which almost always takes $\Theta(n \log n)$ time but is worst case $O(n^2)$. Nonetheless, you will see people (including me) using $O()$ notation for average case behavior sometimes. Technically this is sloppy and colloquial; they're literally meaning "this calculation is going to take on the order of $n \log n$ time", rather than meaning to state a formal upper bound like a rigorous computer scientist would prefer.

displaying data in graphs and tables

switching gears...

You're reading the Weissgerber *et al.* paper this week because it makes important points about data presentation in graphs and tables. In particular:

> *Summary statistics are only meaningful when there are enough data to summarize.*

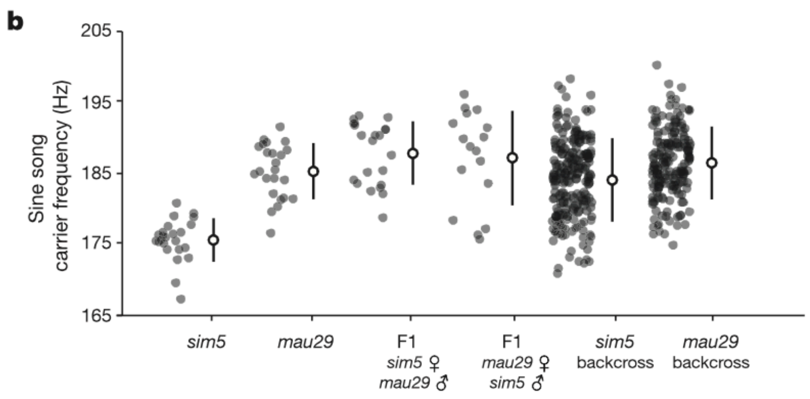
It is extremely common to see bar graphs in biology, displaying means and standard errors. But we've often only done a small number of replicates (maybe *n* = 3-10), and we could just as easily just show the raw data points. Why don't we?

Plotting mean and standard error implicitly assumes that the data can be summarized with just those two numbers, which is often not true. You're basically assuming that your data were normally distributed, and maybe they weren't. Looking at the raw data points shows you the actual distribution - maybe it was multimodal, skewed, had outliers, or whatever.

The other thing is the business of plotting the *standard error* (i.e. the expected error of the mean), not the *sample standard deviation*. If we're visualizing the *data*, we're first primarily interested in the variability of the *data*: if we do the experiment again, if we take another independent sample, where is it likely to be? That's the sample variance, and the square root of that is the standard deviation. Maybe once we're digging into our analysis a little deeper -- we're convinced that the data are distributed normally, and we want to parameterize where the mean is likely to be -- then we might start worrying about how good our estimated mean is likely to be (the standard error) but that's usually not what someone's graph of their samples is aiming to represent, or implying that it represents.

swarm plots and jittered strip plots

An admirable, exemplary data figure is shown below: Figure 1B from [Ding et al, Nature 2016], who measured the frequency of courtship songs in different species of *Drosophila*.



The raw data points themselves are plotted, alongside a summary of the mean and +/- one standard deviation. Not standard error, because what we're most interested in is the variability in the sample data, not the variability we expect in the estimate of the mean.

This business of plotting your actual raw data has become trendy. There's even been a Twitter hashtag, #barbarplots. Packages for graphing data are still catching up. A buzzword is a **swarm plot** or a **beeswarm plot**. We'll use the python Seaborn module in this week's homework because it has the ability to make swarm plots. You can also do an almost identical thing with a **strip plot** by adding random jitter (`jitter=True`, in Seaborn). And you can emulate all this easily enough in a **scatter plot** by plotting your data points along one axis while adding a small amount of random jitter in the other.

I ran across a member of the Microsoft Excel development team replying to a community request for swarm plots by saying "why would anyone want to do that?" Sigh.

two other things we all learn in high school, then forget

- **Significant figures:** state your numbers to a non-obscene precision. Just because a computer outputs a floating point real number as '2.745970945707204' doesn't mean you need to report it that way; in biological data we've often only got numbers nailed down to a digit or two. '2.7' is easier to think about anyway, and remember, Fermi estimation.

- **Always label your axes:** seriously, always label your axes. Few people will stop you in a talk and ask you "um, what's on the x-axis?" because it's hard to ask that without it sounding like "hey, do you even know how to make graphs?". But that's what they'll be *thinking.*

tidy data

Hadley Wickham's influential paper *Tidy data* offers a systematic way of organizing tabular data. Wickham distinguishes 'tidy data' from everything else ('messy data'). He observes that tabular data consist of **values**, **variables**, and **observations**.

A **variable** is something that can take on different values, either because we set it as a condition of our experiment (a **fixed variable**), or because we're measuring it (a **measured variable**). For example, a fixed variable "gene" (which takes on 20,000 different values for different gene names), or a fixed variable "sex" ('M' or 'F'), or a measured variable "TPM" (a number for an estimated expression level). A **value** is the actual number, string, or whatever that's assigned to a variable.

An **observation** is one measurement or experiment, done under some certain conditions that we might be varying, resulting in observed values for one or more things we measured in that same experiment.

In a tidy data set,

- each variable forms a column
- each observation forms a row
- each type of observational unit forms a table

Wickham says it's "usually easy to figure out what are observations and what are variables", but at least for RNA-seq data I'm not so sure I agree. I don't think it's entirely clear whether it's better to treat genes as observations or variables. That is, it seems to me that this is tidy:

gene	mouse	cell_type	sex	TPM
carrot	1	neuron	M	10.5
carrot	2	glia	F	9.8
kohlrabi	1	neuron	M	53.4
kohlrabi	2	glia	F	62.1
cabbage	1	neuron	M	0.4

cabbage	2	glia	F	0.3

but so is this:

mouse	**cell_type**	**sex**	**carrot**	**kohlrabi**	**cabbage**
1	neuron	M	10.5	53.4	0.4
2	glia	F	9.8	62.1	0.3

Wickham says:

> *it is easier to describe functional relationships between variables (e.g., z is a linear combination of x and y, density is the ratio of weight to volume) than between rows, and it is easier to make comparisons between groups of observations (e.g., average of group a vs. average of group b) than between groups of columns.*

which seems to argue most strongly for the second version above, with genes as columns. An "observation" is an RNA-seq experiment on one sample. I get measurements for 20,000 genes in that experiment, and they're linked. I might want to study how one or more genes covary with others across samples (either for real, or because of some artifact like a batch effect). But that's a *really* wide table, 20,000 columns, and you'll have trouble looking at it; some text editors might even choke on lines that long.

We'll see in the homework that it'll be easier to deal with using Seaborn to visualize some example data if we use the first version instead. We can transform the second form into the first by **melting** the data table, in Wickham's nomenclature, and Pandas gives us a method to automate that.

Clearly what I don't want in a tidy RNA-seq data set, though, is a bunch of fixed-variable values, representing my experimental conditions masquerading as column headers, like the 'wt' vs 'mut' and 'M' vs. 'F' columns in this week's homework. That's #1 in Wickham's list of most common problems in messy data:

- when column headers are values, not variable names
- when there's multiple variables in one column
- when variables are stored in both rows and columns
- when multiple types of observational units are in the same table
- when a single observational unit is spread across multiple tables

and he names the procedures for fixing these problems:

- **melt:** turn columns to rows, convert column header values to values of variables
- **split:** convert a column to 2+ columns
- **casting:** merge rows and create new columns
- **normalization:** getting one observational unit per table

Wickham's paper gives nice examples of each.

practicalities: a lightning intro to Pandas

One of our goals this week is to introduce you to Pandas (if you haven't already been using it). Among other things, Pandas gives you powerful abilities to manipulate 2D tables of data.

See also:

- online Pandas documentation
- The book *Python for Data Analysis*, by Wes McKinney, the author of Pandas. You want the 2nd edition, which is for Python 3.

The usual import aliases Pandas to `pd`:

```
import pandas as pd
```

creating a pandas DataFrame

A Pandas DataFrame is a 2D table with rows and columns. The rows and columns can have names, or they can just be numbered $0..r-1, 0..c-1$.

You can create a DataFrame from a 2D array (a Python list of lists, or a numpy array), with default numbering of rows and columns, like so:

```
# create a data table with 4 rows, 3 columns:
D1   =  [[ 12.0,  7.0,  5.0 ],
         [ 16.0, 21.0, 14.0 ],
         [  4.0, 14.0, 20.0 ],
         [  8.0, 28.0, 10.0 ]]
df1  = pd.DataFrame(D1)
```

Or you can create one from a dict of lists, where the dict keys are the *column names*, and the list for each key is a *column*. Here I'm also naming the rows `sample1` etc.

```
D2 = { 'tamarind': [ 12.0, 16.0,  4.0, 8.0 ],
       'caraway':  [  7.0, 21.0, 14.0, 28.0],
       'kohlrabi': [  5.0, 25.0, 20.0, 10.0] }
df2 = pd.DataFrame(D2)
```

Pandas calls the row names an *index*, and it calls the column names, the, uh, *column names*. You can set these column and row numbers with lists either when you create the DataFrame, or by assignment any time after. For example:

```
df1 = pd.DataFrame(D1,
                   index=['sample1', 'sample2', 'samp
                   columns=['tamarind','caraway','koh

df2 = pd.DataFrame(D2,
                   index=['sample1', 'sample2', 'samp

df1.index   = ['sample1', 'sample2', 'sample3', 'samp
df1.columns = ['tamarind','caraway','kohlrabi']
```

Try these out in a Jupyter notebook page - type `df1` or `df2` on its own in a cell and Jupyter will show you the contents of the object you've created.

reading tabular data from files

The `pd.read_table()` function parses data in a file:

```
df = pd.read_table('file.tbl')
```

By default, `read_table()` assumes tab-delimited fields. You will also encounter comma-delimited and whitespace-delimited files commonly. You specify an alternative single-character field separator with a `sep='<c>'` argument, and you can specific whitespace-delimited with a `delim_whitespace` argument.

```
df = pd.read_table('file.tbl', sep=',')
df = pd.read_table('file.tbl', delim_whitespace)
```

Column names: by default, the first row is assumed to be a header containing the column names, but you can customize:

```
df = pd.read_table('file.tbl')
df = pd.read_table('file.tbl', header=None)
df = pd.read_table('file.tbl', header=5)
df = pd.read_table('file.tbl', name=['col1', 'col2'
```

Yeah, now the argument for setting your own list of column names is `name=[]`, not `columns=[]`. Pandas is not great on consistency.

Row names: by default, assumes there are no row names (there are n fields per line, for n column names), but if the first field is a name for each row (i.e. there are $n + 1$ fields on each data line, one name and n named columns), use `index_col=0`:

```
df = pd.read_table('file.tbl', index_col=0)     # Fi
```

To get Pandas to skip comments in the file, use the `comment=` option to tell it what the comment character is:

```
df = pd.read_table('file.tbl', comment='#')     # Sk
```

peeking at a DataFrame

```
    (nrows, ncols) = df.shape()   # Returns 'shape' c
    df.head(5)                    # Returns first 5 r
    df.sample(20)                 # Returns 20 random
    df.sample(20, axis=1)         # Returns 20 random
```

indexing rows, columns, cells in a DataFrame

The best way to learn is to play with example DataFrames in a Jupyter notebook page. I find Pandas to be inconsistent and confusing (sometimes you access a DataFrame as if it's column-major, and sometimes like it's row-major, depending on what you're doing) but it's powerful once you get used to it.

To extract single columns, or a list of several columns, you can index into the DataFrame as if it's in column-major form:

```
    df['caraway']                 # Gets 'caraway' col
    df[['caraway','kohlrabi']]    # Gets two columns,
```

As a convenience, you can also use slicing operators to get a subset of rows; with this, you index into the DataFrame as if it's in row-major form, and the slice operator behaves as you'd expect from Python (i.e. as a half-open interval):

```
    df[2:4]      # gets rows 2..3; i.e. a half-open sli
    df[2:3]      # gets just row 2

    df[2]        # DOESN'T WORK: it will try to get a *
```

To access data by row/column **labels**, use the `.loc[]` method. `.loc` takes two arguments, for which rows and columns you want to access, in that order (now we're row-major again). These arguments are the row/column **labels**: either a single label, or a list. Note that `.loc` takes arguments in square brackets, not parentheses! Also note that when you use slicing syntax with `.loc`, it takes a *closed* interval, in contrast to everywhere else in Python where you'd expect a half-open interval.

```
## When rows are labeled:
df.loc['sample1']                          # Extrac
df.loc[['sample1','sample2']]              # Extrac

## When rows are numbered, not labeled:
df.loc[2:3]                                # Extrac
df.loc[2:3, ['caraway', 'kohlrabi']]       # Extrac
df.loc[2,'A2M']                            # Get a
```

To access data by row/column **positions**, use the `.iloc[]` method. `.iloc[]` has square brackets around its row/col arguments just like `.loc[]` but the arguments are an integer, a list of integers, or a range like 2:5. Ranges in `.iloc[]` are half-open intervals; compare:

```
df.loc[2:3]      # By label, closed interval: get
df.iloc[2:3]     # By index, half-open interval:
```

Did I mention yet, Pandas is great and super-useful and all, but well-designed and consistent? Well, you can't have everything. Just get used to it and keep good crib notes.

getting, or iterating over row and column labels

```
df.index               # Gets row labels, as an inc
df.columns             # Gets column labels, as an
list(df)               # Gets column labels as a li
[c for c in df]        # Example of iterating over
[r for r in df.index]  #   ...or of iterating over
```

sorting data tables

The `sort_values()` method gives you a wide range of sorting abilities.

The `by=` option tells it what field to sort on (which column label, usually). The argument to `by=` is a single column label (a string), or a list of them. If it's a list, it sorts first by the first one, then second by the second, etc.

By default, it sorts rows, by some column's value. If you choose 'axis=1', it sorts columns, by some row's value.

By default, it sorts in ascending order; specify `ascending=False` to get descending order.

```
df_sorted = df.sort_values(by='coriander', ascendi
```

plotting example

To scatter plot two columns against each other
(i.e. each row is a point on the scatter plot):

```
    ax = df.plot(kind='scatter', x='caraway', y='coria
```