How to Clean Messy Data in Python

DS 6001: Practice and Applications of Data Science

Getting Yourself Unstuck

- Online Communities
- Using Python's built-in help documentation
- Good old Google
- Stack Overflow
- Interacting with other Python users on PySlackers
- Live chats with Python users on Freenode
- Python Mailing lists

Loading CSV and ASCII Data into Python

- Electronic data files
- Changing the working directory
- Loading standard CSV files
- Looking at the data to see if it loaded correctly
- Loading messy CSV and other ASCII files
- Writing CSV and ASCII files

Loading Other Kinds of Electronic Data Files

- Loading fixed width files
- Loading Excel files
- Loading SAS, Stata, and SPSS files

A note before we begin

We will be **updating these slides over the next several weeks**, and posting the most recent versions on Collab before each class.

These slides are intended to be a guide on data management in Python, but not on the basics of Python programming.

We need you to be comfortable with

- Using a Python interface (Jupyter, Spyder, etc.)
- Defining and working with objects
- Saving scripts/notebooks
- Using functions

If you are not comfortable with these skills, that's fine, but speak to us after class so we can help you get these skills.

Managing Data

At most companies, data scientists spend about 80% of their time managing data (also called wrangling, cleaning, engineering, or organizing data).

If you want to get to the fun stuff, like **visualizing, modeling, and forecasting**, you need to be able to clean data very quickly and accurately.

Among other skills, you will need to

- Load electronic data into Python
- Store data in a database
- Query the database
- Manage the rows and columns of a dataset
- Reshape and merge datasets

And you need to be able to perform these tasks instinctively, without having to think about it too much.

Managing Data

In other words, you will all become Python Ninjas:

But before we can teach you all those ninja skills, we have to talk about the most important programming skill of all, which is . . .

Getting Yourself Unstuck

The single most important skill to master with a programming language is knowing what to do when you get stuck.

Everyone, even people with years of Python programming experience, uses help resources CONSTANTLY. So **don't** feel like it is cheating or a judgment on your skills to look for help.

Here are at least six places to go for help:

1. Python documentation
2. Google
3. Stack Overflow
4. PySlackers
5. Internet relay chat (IRC) rooms with other Python users
6. Various Python mailing lists

Online Communities

These options involve becoming a responsible, respectful member of the **worldwide community of Python users** (See python.org/community/).

Open-source platforms like Python and R depend on a community of volunteers who develop and maintain the tools that we use. All these people working for the common good. That's a beautiful thing.

Slack, Stack Overflow, the Freenode IRC, and mailing lists are also online communities for Python users.

BUT, like any online community, there's the potential for a **toxic culture** to destroy everything.

Toxic Online Communities

What is a toxic culture? How do you know one when you see one?

Toxic cultures are more likely when

- members are allowed to be **anonymous** (Lapidot-Lefler and Barak 2011),
- and members **up-vote and down-vote** and **comment on** each other's contributions (Massanari 2015).

A culture can be either actively or passively toxic.

Actively toxic communities are easy to identify. They encourage and are characterized by **overt** sexism, racism, bigotry, and calls for violence or other aggression against individuals.

Toxic Online Communities

Most of the toxicity you will encounter in online programming and data science communities is not actively, but passively toxic.

Passive toxicity is characterized by **gate-keeping**. Subtle behaviors that discourage people with less experience, or with some social anxiety, from participating.

Stack overflow, IRCs, and mailing lists are notorious for passive toxic behavior.

Passive toxicity is a bigger problem for us than active toxicity because

- actively toxic behavior is usually explicitly banned by codes of conduct,
- and **individuals are often unaware of when they are acting in a passively toxic way**.

Toxic Online Communities

Examples of passive toxic behavior:

Condescending language: “obviously”, “clearly”, “actually”, “just”, “that should be easy”, “google it”, “read the manual”, “RTFM”. An example.

Shaming: implying that the solution is easy and that someone is an idiot for not knowing it.

- “Learn to debug your own code”
- “If you don’t get this, you have no business being a data scientist”

Downvotes without explanation: this can be very upsetting to anyone, especially to people with less experience

Toxic Online Communities

Examples of passive toxic behavior:

Virtue signaling: implying that people are superior/inferior because of the language, software, or methods they use.

- "Real programmers don't use for loops"
- "You still use SAS?"
- So many memes:

HATING EXCEL
R USERS
PYTHON USERS

Toxic Online Communities

Examples of passive toxic behavior:

Authoritarianism: Abusing people for failing to follow all of a community's rules for asking questions.

- Ignore the content of the question but comment "provide an example"
- Editing a user's post to remove where they wrote "Hi everyone" and "thanks"

Overzealous curation: Being very quick to tag a question as a "duplicate" without checking to see nuanced ways in which the question comes from a new situation.

Toxic Online Communities

The result of passive toxicity is that many **potential community members** choose not to participate because

1. Initial experiences made new members feel ashamed, confused, or belittled.

2. Other potential new members *observe* negative interactions involving other members, and choose to disengage.

Passive toxicity **shrinks** the community and makes it **more homogeneous**.

Across society, small, homogeneous communities are much more likely to exclude or discriminate against people based on sex, race, class, language and other factors. And that leads to many ethical problems.

Toxic Online Communities

Under no circumstances are you to contribute to an active or passive toxic culture in any community, online or otherwise.

Please keep the behaviors we discussed in mind when you engage in online communities, and avoid them. Don't be afraid to call out other people who behave in these ways.

Everyone, please **rise**, raise your right hand, place your left hand over your computer, and repeat after me:

I promise
not to contribute to the destruction of society
by being an asshole online

Python's Built-in Help Documentation

Packages, modules, classes and functions in Python have built-in documentation that you can display directly in the console or in the output of a notebook.

This is the first place to look for help with specific, pre-built Python code.

There are three parts of Python's documentation for an object:

1. The signature – if the object is a function, the complete function syntax, including all arguments and their default values
2. The docstring – text explaining how to use the object, in detail (we'll go over this next)
3. The type – what kind of object is this?

Python's Built-in Help Documentation

There are a few ways to display the help documentation for an object:

`help(x)` – displays the docstring for object `x`

`x?` – displays an **abbreviated** docstring for `x`, as well as the signature and type

`x??` – same as `x?`, but shows the internal code of `x` if `x` is a Python function (not just a call to a C function)

The most important skill is to know how to read the docstring to quickly find the information you need.

Python's Built-in Help Documentation

To understand how to read the docstring, call up the docstring for a linear regression class object from the `sklearn` package:

```
import sklearn.linear_model
help(sklearn.linear_model.LinearRegression)
```

Docstrings often have **sections** that convey particular information.

1. The header

```
Help on class LinearRegression in module
sklearn.linear_model.base:

class LinearRegression(LinearModel,
sklearn.base.RegressorMixin, sklearn.base.MultiOutputMixin)
```

The header tells us that the `LinearRegression` object is a class, stored in the `linear_model.base` module within the `sklearn` package.

Python's Built-in Help Documentation

2. The signature

```
LinearRegression(fit_intercept=True, normalize=False,
copy_X=True, n_jobs=None)
```

Some docstrings list the signature, although the signature can be accessed elsewhere. The signature lists all of the parameters of a function.

The value each parameter is set equal to is the **default**. If the user doesn't specify the parameter, it's set to the default.

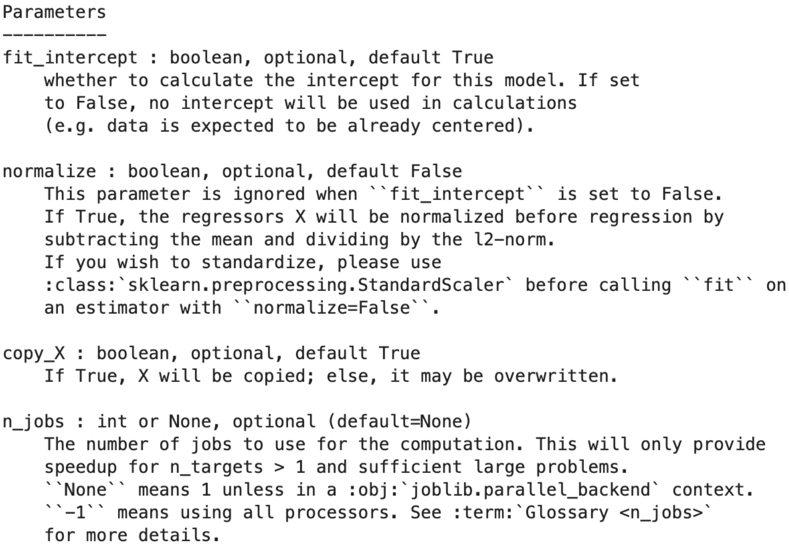
3. The short description

```
Ordinary least squares Linear Regression.
```

A one-or-two sentence summary of what the function does.

Python's Built-in Help Documentation

4. The parameters section is the most useful for learning how to use a function:



```
Parameters
----------
fit_intercept : boolean, optional, default True
    whether to calculate the intercept for this model. If set
    to False, no intercept will be used in calculations
    (e.g. data is expected to be already centered).

normalize : boolean, optional, default False
    This parameter is ignored when ``fit_intercept`` is set to False.
    If True, the regressors X will be normalized before regression by
    subtracting the mean and dividing by the l2-norm.
    If you wish to standardize, please use
    :class:`sklearn.preprocessing.StandardScaler` before calling ``fit`` on
    an estimator with ``normalize=False``.

copy_X : boolean, optional, default True
    If True, X will be copied; else, it may be overwritten.

n_jobs : int or None, optional (default=None)
    The number of jobs to use for the computation. This will only provide
    speedup for n_targets > 1 and sufficient large problems.
    ``None`` means 1 unless in a :obj:`joblib.parallel_backend` context.
    ``-1`` means using all processors. See :term:`Glossary <n_jobs>`
    for more details.
```

Python's Built-in Help Documentation

The parameters section lists the parameters, in the order in which they appear in the signature of the function, along with information about each parameter.

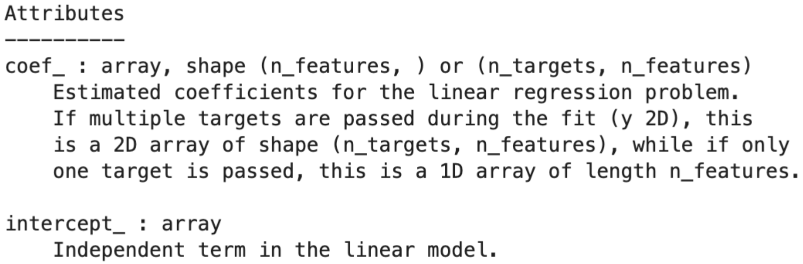
Each parameter has a **type**: in this case, the first three parameters are boolean, which means they can be set to either True or False. The fourth parameter is an integer.

Each parameter is noted as either required or optional in a call to the function.

Each parameter is described in a sentence or two to explain what the parameter does.

Python's Built-in Help Documentation

5. The attributes



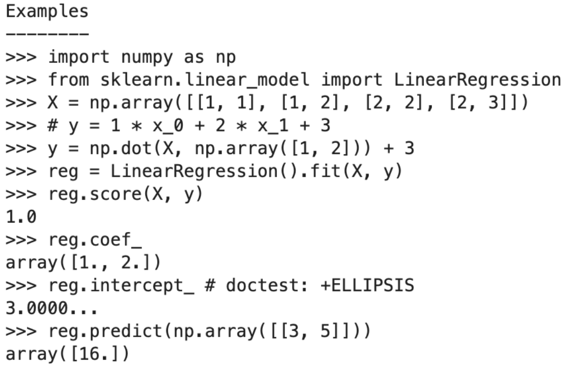
```
Attributes
----------
coef_ : array, shape (n_features, ) or (n_targets, n_features)
    Estimated coefficients for the linear regression problem.
    If multiple targets are passed during the fit (y 2D), this
    is a 2D array of shape (n_targets, n_features), while if only
    one target is passed, this is a 1D array of length n_features.

intercept_ : array
    Independent term in the linear model.
```

Attributes are components of the the output of the function.

If the output is saved in an object named `regress`, to access the coefficients, type `regress.coef_`, and to access the intercept, type `regress.intercept_`.

Python's Built-in Help Documentation

6. The examples

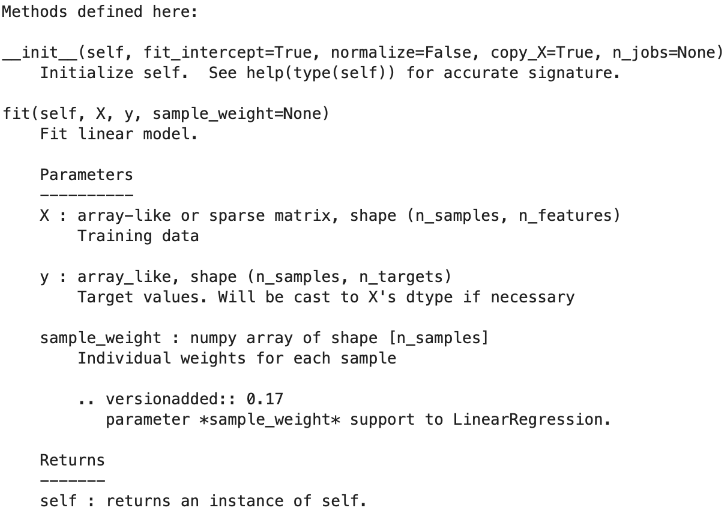


```
Examples
--------
>>> import numpy as np
>>> from sklearn.linear_model import LinearRegression
>>> X = np.array([[1, 1], [1, 2], [2, 2], [2, 3]])
>>> # y = 1 * x_0 + 2 * x_1 + 3
>>> y = np.dot(X, np.array([1, 2])) + 3
>>> reg = LinearRegression().fit(X, y)
>>> reg.score(X, y)
1.0
>>> reg.coef_
array([1., 2.])
>>> reg.intercept_ # doctest: +ELLIPSIS
3.0000...
>>> reg.predict(np.array([[3, 5]]))
array([16.])
```

Examples are meant to be run, not just looked at. Copy-and-paste the examples into your notebook or script, run the code. See if you can do more things with the given objects than the examples do.

Python's Built-in Help Documentation

7. The related methods defines methods that expand the functionality of the one you are looking at, along with their own documentation:



```
Methods defined here:

__init__(self, fit_intercept=True, normalize=False, copy_X=True, n_jobs=None)
    Initialize self.  See help(type(self)) for accurate signature.

fit(self, X, y, sample_weight=None)
    Fit linear model.

    Parameters
    ----------
    X : array-like or sparse matrix, shape (n_samples, n_features)
        Training data

    y : array_like, shape (n_samples, n_targets)
        Target values. Will be cast to X's dtype if necessary

    sample_weight : numpy array of shape [n_samples]
        Individual weights for each sample

        .. versionadded:: 0.17
           parameter *sample_weight* support to LinearRegression.

    Returns
    -------
    self : returns an instance of self.
```

Using Google

If you know the functions you need more information about, using the built-in documentation the best habit.

Don't go to Google first! There are way too many presentations of any one topic to sift through. It can take a lot longer if you use Google primarily.

If the built-in documentation doesn't give you the information you need, the best Google search is

Python *the function you want help with additional details*

Google will often take you to **Stack Overflow**.

Stack Overflow

Stack Overflow is the most popular and most useful website for help with programming of all kinds. Google searching a Python problem will usually lead to a Stack Overflow post on the same issue.

Python is now the **most frequent** tag for posts on Stack Overflow: see the video embedded on this blog post.

Finding a Stack Overflow post that's relevant to your problem can give you both the code and intuition to solve your problem.

Or maybe not! Small differences in the situation can make the solution irrelevant to you. **Be cautious** and don't treat a Stack Overflow post as automatically a definitive answer.

How Stack Overflow Works

1. Someone asks a question

2. Other people comment on and provide answers to the question

3. The person who asked the question replies to the comments, and can choose an answer to mark as “accepted”

4. People with reputation scores higher than 15 can **upvote or downvote** questions and answers.

5. Reputation points are awarded for asking questions or giving answers that other people upvote, or for having an answer accepted. Points are taken away for downvotes or spam or offensive posts.

Going for reputation is an **entirely optional** activity. If you don’t want to worry about it, don’t.

Asking a Question on Stack Overflow

Okay, so you're stuck. You've combed through the Python documentation, Google, and old Stack Overflow posts, but you haven't found a solution.

It's time to consider writing a **new question** on Stack Overflow.

Asking a Question on Stack Overflow

This can be frightening. A lot of the time, people answering questions on Stack Overflow can be, well . . . huge assholes that cause real suffering.

You might choose to avoid posting to Stack Overflow, so as not to support a website that has harbored and even encouraged abuse. That's completely fair.

If you do post to Stack Overflow, you are likely to get some very useful responses if you follow some guidelines. There's **a strategy for getting good responses**: stackoverflow.com/help/how-to-ask

Asking a Question on Stack Overflow

You are more likely to get a good response if you follow these steps:

Step 1: Search Stack Overflow and Google to see if the question has already been answered. Commenters dislike if the same question is asked repeatedly. This poor guy got roasted for posing a “duplicate” question.

(An aside: *Why?* There’s an idea that Stack Overflow should be a central repository of knowledge. That means there should be one **canonical** answer to one question. But people often take this much too far. There are kinder ways to point to an existing answer.)

So spend a significant amount of time digging through the internet. If there’s something similar, but not quite what you need, you can say so in your post.

Asking a Question on Stack Overflow

Step 2: Write a good title for your post.

A good title is specific about the problem, and also succinct:

Bad: Problem with matplotlib (not specific)

Bad: How do I place the labels of cars in a scatterplot of the weight and miles per gallon of cars onto the points in the scatterplot using matplotlob 3.3.1 on Python 3.7.4 on Mac OSX 10.14.5? (not succinct)

Good: How to place labels on top of points in a matplotlib scatterplot?

Getting Help: Asking a Question on Stack Overflow

Step 3: Start the post with a paragraph describing the problem in more detail.

Some good things to include in this paragraph:

- The context of the problem – how did you come across the problem? Describe the **overall goal**, not the just the buggy step
- What you've already tried to solve the problem, and what happened
- What is the expected output? What do you see instead?
- You can write the version of Python you are using, the version of the modules, and the operating system on your computer, in case the problem turns out to be specific to one of those

Asking a Question on Stack Overflow

Step 4: If possible, include code that reproduces the problem. The code SHOULD NOT simply be the code in your script that isn't working. It needs to be able to work on someone else's computer.

That means the code should not depend on any **specific data files**, and should not contain file addresses that refer to a location on your computer. Only use modules that are easy to get.

If the code needs to run on data, can you use something pre-loaded in Python that everyone can access? (There are example datasets included with `psykitlearn`, for example.)

Make the code as short as possible, and use comments, to help people understand the code more quickly.

A few additional things to keep in mind

Be courteous and respectful. Respond to and thank everyone who comments.

Post a follow-up once the problem is solved so that people who come across this page in the future with the same problem know the solution.

Don't ask people to write code for you. It's better to request help with code your provide.

Don't claim you found a bug in Python or in a module. It's a bit rude to the people who programmed the code (who don't get paid).

Don't ask about **homework problems**. (Here's an example of someone getting called out on this)

Interacting with other Python users on PySlackers

The main slack page for the global community of Python users is **Pyslackers**: https://pyslackers.com/web

To join, just go to the URL and click ”join the community”.

Some useful channels:

- data_science
- python_
- job_advice

Live chats with Python users on Freenode

The Python user community is world-wide, and for the most part, very supportive. There are active internet relay chat (IRC) networks where you can post a question to members who are also logged in, to possibly **get an answer right away**.

The most active Python IRC is the #python channel on Freenode (https://webchat.freenode.net/). When I logged in while writing this slide, there were 1,778 people logged on.

Internet chatrooms can be rough places, but the #python channel claims to enforce this Code of Conduct: https://www.python.org/psf/codeofconduct/.

Live chats with Python users on Freenode

Getting started on Freenode can be tricky, but it's easier if follow these steps:

1. Go to (https://webchat.freenode.net/). Choose a **nickname**. Make it professional (you're a UVA student after all!) and unique.

2. Don't write anything under channel. Prove you are not a robot by selecting pictures of motorcycles or something. Then, once your humanity has been established, click **Start**.

3. To use the #python channel, you need to register your nickname. To check if your nickname is unique, click on the "freenode" tab on the left-hand sidebar. A text box will appear on the bottom of the screen. Type:

```
/msg NickServ info
```

Live chats with Python users on Freenode

4. Step 3 will open a new tab. Switch to that tab. If no one else already has your nickname, you will see

```
NickServ: (notice) <nickname> is not registered.
```

If you see something else, it means someone already has your nickname. You can change your nickname right here by typing `/nick` followed by another nickname. Then type `/msg NickServ info` again. Repeat until you see the message listed above.

Live chats with Python users on Freenode

Important note: DON'T use a password here that you use for important things like **email, bank accounts, etc.**

We shouldn't have the same faith in the security of Freenode's servers as we can have in Google's.

Also, this is the kind of platform that tends to attract hackers. And for people used to a graphical user interface, it might be easy to mistype in a way that accidentally displays your password in the chat.

Use a unique, throwaway password!

Live chats with Python users on Freenode

5. To register this nickname, type

```
/msg NickServ register <password> <email-address>
```

where `<password>` is a password you will use in the future, and `<email address>` is the email you want associated with this account.

Live chats with Python users on Freenode

6. Check your email for a confirmation code. Be patient. It can take up to 20 minutes for the email to go through.

7. Once you have the code, paste it and your nickname into this code, and submit it:

```
/msg NickServ VERIFY REGISTER <nickname> <secret-code>
```

8. You are now registered! Return to https://webchat.freenode.net/ and log-in with your nickname and password. Type `#python` under channel.

You are free to chat away. Pay attention to the guidelines that appear as links on the top of the screen.

Python mailing lists and message boards

Usenet – a distributed discussion system (no central server) – was invented in 1979, and is still in use today. The Python Usenet message boards are at https://mail.python.org/archives. The `comp.lang.python` board is for general discussions and questions about Python.

The tutor mailing list (https://mail.python.org/mailman/listinfo/tutor) is for users who want to ask questions about learning computer programming with Python.

If you have a question for the Python core development team, send an email to help@python.org. The team is pretty busy, so be sure to check other resources and lists for an answer first.

Many ways to do the same thing in Python

Stata, SAS, SPSS, and Excel are carefully curated software. There is often only one way to perform a task. It's easier to memorize how to do a task.

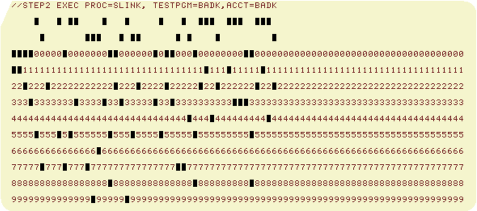
In Python (and R), there are usually many, many ways to do the same thing using different functions and packages.

What follows is a set of *guidelines and suggestions*. NOT a definitive list of how to do things.

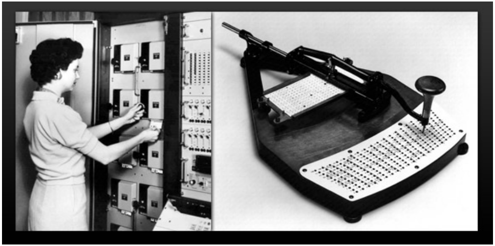
It's OKAY to mix styles, packages, and approaches. Use whatever works, but **keep track of what you do**.

Electronic Data Files

Through the 1970s, data was stored on punch cards and fed directly to a mainframe computer capable of regression analysis.



```
//STEP2 EXEC PROC=SLINK, TESTPGM=BADK,ACCT=BADK
```



Electronic Data Files

In the late 1970s and through the 1980s, it became possible to store data directly on a computer hard drive. But space was very limited.

In order to store data as efficiently as possible, universal standards were adopted.

ASCII – American Standard Code for Information Interchange
pronounced "As-Key"

- Defined 128 characters to be "legal" in data files
- Text files. Messy, but we can deal with them.
- Designed to be as small and as universally portable as possible.
- Data points usually delimited by commas, spaces, or tabs. Might require a data dictionary to read.

Electronic Data Files

Today there are two common ways to access electronic data.

1. People can share individual data files through websites, email, or hard storage. These files are often in ASCII format, but can be stored in other (sometimes proprietary) formats.

2. Through a local or remote relational database – a collection of many individual datasets – managed using SQL.

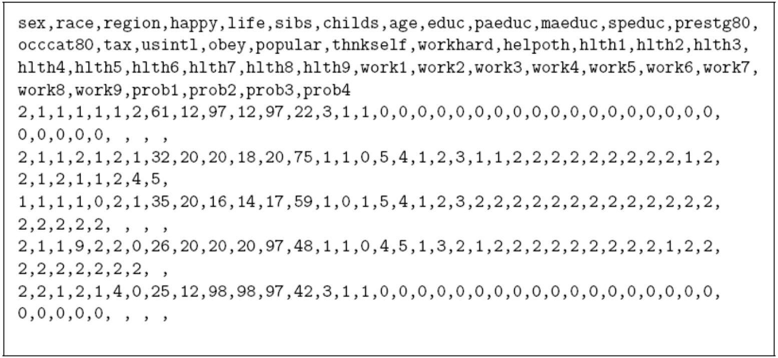
It's important to be very comfortable **working with both** methods of sharing data. To build a database, we often have to collect and clean individual data files.

We will go over individual data files today, and databases soon.

Kinds of ASCII files

Our task: to load ASCII data into Python, identify the ways in which it is messy, and create tidy data.

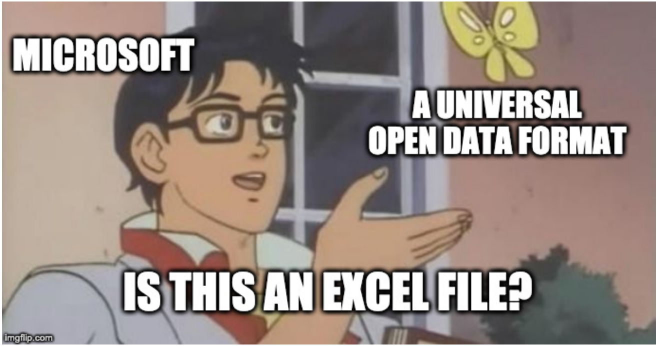
A comma-separated values (CSV) file:



```
sex,race,region,happy,life,sibs,childs,age,educ,paeduc,maeduc,speduc,prestg80,
occcat80,tax,usintl,obey,popular,thnkself,workhard,helpoth,hlth1,hlth2,hlth3,
hlth4,hlth5,hlth6,hlth7,hlth8,hlth9,work1,work2,work3,work4,work5,work6,work7,
work8,work9,prob1,prob2,prob3,prob4
2,1,1,1,1,1,2,61,12,97,12,97,22,3,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0, , , ,
2,1,1,2,1,2,1,32,20,20,18,20,75,1,1,0,5,4,1,2,3,1,1,2,2,2,2,2,2,2,2,2,1,2,
2,1,2,1,1,2,4,5,
1,1,1,1,0,2,1,35,20,16,14,17,59,1,0,1,5,4,1,2,3,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2, , , ,
2,1,1,9,2,2,0,26,20,20,20,97,48,1,1,0,4,5,1,3,2,1,2,2,2,2,2,2,2,2,2,1,2,2,
2,2,2,2,2,2,2, ,
2,2,1,2,1,4,0,25,12,98,98,97,42,3,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0, , , ,
```

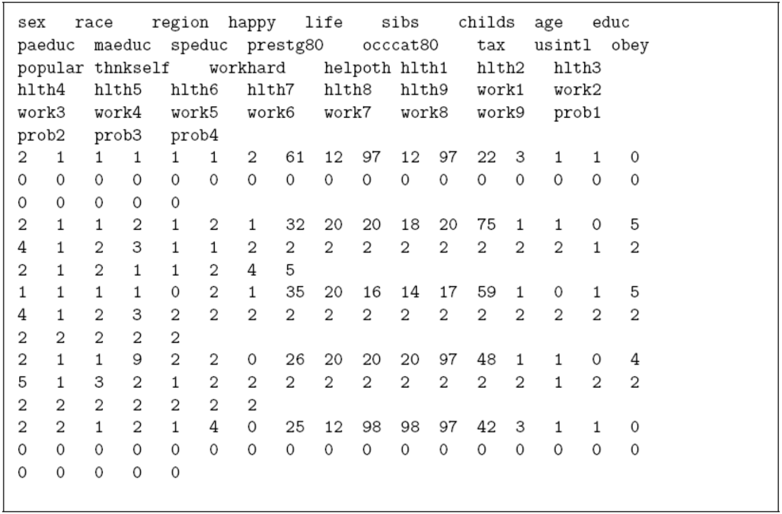
Kinds of ASCII files

Note: Although the CSV format is universal, Excel sometimes opens by default when you double-click on the CSV file. But, CSV files are NOT exclusive to Excel.



Kinds of ASCII files

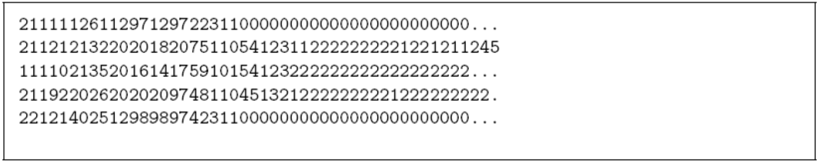
A tab delimited file:



```
sex     race     region  happy   life    sibs    childs  age     educ
paeduc  maeduc   speduc  prestg80        occcat80        tax     usintl  obey
popular thnkself         workhard        helpoth hlth1   hlth2   hlth3
hlth4   hlth5    hlth6   hlth7   hlth8   hlth9   work1   work2
work3   work4    work5   work6   work7   work8   work9   prob1
prob2   prob3    prob4
2    1    1    1    1    1    2    61   12   97   12   97   22   3    1    1    0
0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
0    0    0    0    0
2    1    1    2    1    2    1    32   20   20   18   20   75   1    1    0    5
4    1    2    3    1    1    2    2    2    2    2    2    2    2    2    1    2
2    1    2    1    1    2    4    5
1    1    1    1    0    2    1    35   20   16   14   17   59   1    0    1    5
4    1    2    3    2    2    2    2    2    2    2    2    2    2    2    2    2
2    2    2    2    2
2    1    1    9    2    2    0    26   20   20   20   97   48   1    1    0    4
5    1    3    2    1    2    2    2    2    2    2    2    2    2    1    2    2
2    2    2    2    2    2    2
2    2    1    2    1    4    0    25   12   98   98   97   42   3    1    1    0
0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
0    0    0    0    0
```

Kinds of ASCII files

A fixed-width ASCII file with no delimination. Files like these minimize memory (no need to store a bunch of commas), but require a dictionary file to read them.



```
211111261129712972231100000000000000000000000...
211212132202018207511054123112222222221221211245
111102135201614175910154123222222222222222222...
21192202620202097481104513212222222222122222222.
221214025129898974231100000000000000000000000...
```

Dictionary:

- Variable 1: sex, column 1
- Variable 2: race, column 2
- . . .
- Variable 8: age, columns 8-9

Changing the working directory

Before we go over the functions, it is useful to set the working directory at the start of your script or notebook.

This **sets the default folder** where Python looks for files. If all of your files are in the same folder, setting the working directory means you don't have to write out the paths each time you load or save a file.

To set the working directory:

- Load the os package: `import os`
- Type the folder's address into `os.chdir("folder")`

Changing the working directory

To **check** on the path Python is currently using as a default, type `os.getcwd()` into the console.

If you want to change the working directory back after you've run the relevant code:

```
import os
oldpath = os.getcwd()
os.chdir("folder")

#(Your code goes here)

os.chdir(oldpath)
```

Loading CSV files

We will be using the Pandas package:

```
import pandas as pd
```

The main function for loading an ASCII data file is `pd.read_csv()` . There are lots of parameters, and we'll go over a few important ones, starting with this one:

```
pd.read_csv(filepath_or_buffer)
```

`filepath_or_buffer` – (string) one of three things:

1. The **full file address and file name** of the data file
2. Just the file name of the data file if you've already set the working directory to the folder where the file exist
3. The URL of a data file that's accessible online

Example: 2016 American National Election Study (ANES)

The ANES is a large survey, conducted every 4 years after the presidential election, that has 1000s of variables on topics no poll gets into. See https://electionstudies.org/

I put several versions of the ANES data on our class GitHub page.

You can load the `anes_example.csv` data by either **downloading and unzipping** the file, or by using the URL:

https://raw.githubusercontent.com/NovaVolunteer/Practice_Application_DS/master/Week%205/anes_example.csv

Example: 2016 American National Election Study (ANES)

If you download and unzip the ANES data, and you've already changed your working directory, then to load the ANES data, type

```
anes = pd.read_csv("anes_example.csv")
```

If you want to load the data directly from the URL, save the URL as a separate object, then pass this to the function:

```
url = "https://raw.githubusercontent.com/NovaVolunteer/
    Practice_Application_DS/master/Week%205/
    anes_example.csv"
anes = pd.read_csv(url)
```

Looking at the data to see if it loaded correctly

Before we get to the other parameters of the `pd.read_csv()` function, let's talk about the workflow of loading data.

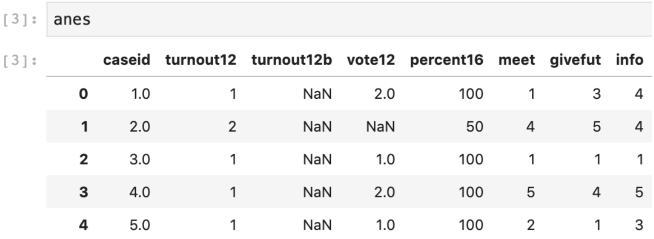
The steps are

1. Run code to load a data file
2. **Examine the loaded dataframe** object to make sure the data was correctly read
3. If you catch anything weird, return to 1. and try different parameters for `pd.read_csv()`

There's an important set of functions in Python that let you quickly explore a dataframe.

Looking at the data to see if it loaded correctly

If you are using a Jupyter Notebook, typing the name of the data frame **in its own cell** will produce a good-looking HTML table illustrating the data frame.



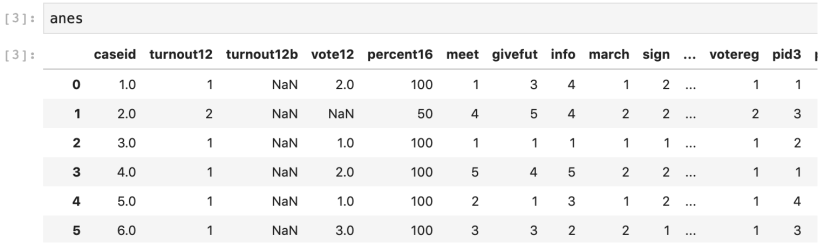
```
anes
```

	caseid	turnout12	turnout12b	vote12	percent16	meet	givefut	info
0	1.0	1	NaN	2.0	100	1	3	4
1	2.0	2	NaN	NaN	50	4	5	4
2	3.0	1	NaN	1.0	100	1	1	1
3	4.0	1	NaN	2.0	100	5	4	5
4	5.0	1	NaN	1.0	100	2	1	3

If you are using Spyder, look in the upper-right window and select the “Variable explorer” tab. Clicking on the data frame will open a separate window for viewing the data.

Looking at the data to see if it loaded correctly

One annoying thing about Jupyter's interactive viewer is that it omits the columns in the middle for data frames with more than about 20 columns:



```
[3]: anes
```

[3]:

	caseid	turnout12	turnout12b	vote12	percent16	meet	givefut	info	march	sign	...	votereg	pid3
0	1.0	1	NaN	2.0	100	1	3	4	1	2	...	1	1
1	2.0	2	NaN	NaN	50	4	5	4	2	2	...	2	3
2	3.0	1	NaN	1.0	100	1	1	1	1	1	...	1	2
3	4.0	1	NaN	2.0	100	5	4	5	2	2	...	1	1
4	5.0	1	NaN	1.0	100	2	1	3	1	2	...	1	4
5	6.0	1	NaN	3.0	100	3	3	2	2	1	...	1	3

The columns it skipped (about **148** in this case) are replaced by a column of dots.

To keep Python from skipping columns, you can change this behavior **globally** (for all subsequent code) or **locally** (for each line of code individually).

Looking at the data to see if it loaded correctly

To always display all of the columns, type

```
pd.set_option('display.max_columns', None)
```

To always display all of the rows, type

```
pd.set_option('display.max_rows', None)
```

Caution: If you are working with large dataframes, it's probably not a good idea to always display ALL of the rows and columns.

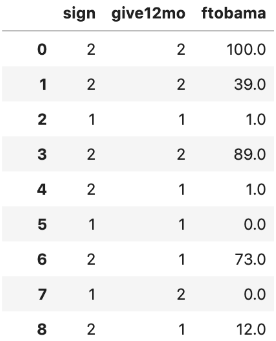
To keep a specific line of code from skipping variables, use the `anes.loc` and `anes.iloc` functions. (Replace "anes" with the name of your dataframe object.)

Looking at the data to see if it loaded correctly

`anes.loc` allows you to select columns of a data frame by name, and `anes.iloc` allows you to select columns by column number.

To see the “sign”, “give12mo”, and “ftobama” variables, type

```
anes.loc[:, ['sign', 'give12mo', 'ftobama']]
```

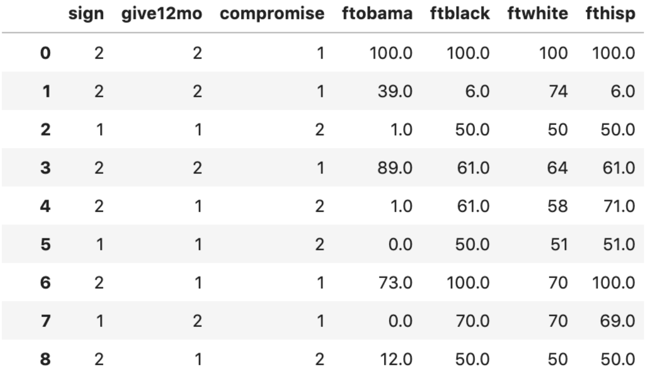


	sign	give12mo	ftobama
0	2	2	100.0
1	2	2	39.0
2	1	1	1.0
3	2	2	89.0
4	2	1	1.0
5	1	1	0.0
6	2	1	73.0
7	1	2	0.0
8	2	1	12.0

Looking at the data to see if it loaded correctly

To see all variables in between “sign”, and “fthisp”, type

```
anes.loc[:, 'sign':'fthisp']
```



	sign	give12mo	compromise	ftobama	ftblack	ftwhite	fthisp
0	2	2	1	100.0	100.0	100	100.0
1	2	2	1	39.0	6.0	74	6.0
2	1	1	2	1.0	50.0	50	50.0
3	2	2	1	89.0	61.0	64	61.0
4	2	1	2	1.0	61.0	58	71.0
5	1	1	2	0.0	50.0	51	51.0
6	2	1	1	73.0	100.0	70	100.0
7	1	2	1	0.0	70.0	70	69.0
8	2	1	2	12.0	50.0	50	50.0

Looking at the data to see if it loaded correctly

To select columns and rows numerically, use `anes.iloc`. To see rows 254 through 262 and all columns, type

```
anes.iloc[254:262, :]
```

To see all rows, columns 21 through 30, type

```
anes.iloc[:, 21:30]
```

To see rows 254 through 262 , columns 21 through 30, type

```
anes.iloc[254:262, 21:30]
```

Looking at the data to see if it loaded correctly

To see only the first 10 rows of the data, type `anes.head(10)`. Replace 10 with however many rows you want to see.

To see only the last 10 rows of the data, type `anes.tail(10)`.

Typing `anes.info()` tells us the dimensions of the data, the number of variables of each type, and the size of the dataframe in memory:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1200 entries, 0 to 1199
Columns: 168 entries, caseid to ever_vs_12mo_rand
dtypes: float64(76), int64(86), object(6)
memory usage: 1.5+ MB
```

Looking at the data to see if it loaded correctly

`anes.columns` lists all the variable names.

If there are too many variables, Python will abbreviate the list with “. . .” To see the omitted items, change the maximum number of items that can display in a list with:

```
pd.set_option('display.max_seq_items', None)
```

(Again, be careful about removing this limit for data frames with a large number of columns)

`anes.dtypes` lists the variables along with their types (int64 for integers, float64 for numbers with decimals, object for variables that might be either categorical or string).

Looking at the data to see if it loaded correctly

`anes.describe()` shows basic summary statistics for every variable in the dataframe.

There are different summary statistics for different types of variables. By default, `anes.describe()` displays stats only for the `float` and `int` types:

- `count` – number of non-missing observations
- `mean` – the sample mean
- `std` – the sample standard deviation
- `min` – the minimum value
- 25% – the 25th percentile
- 50% – the median value
- 75% – the 75th percentile
- `max` – the maximum value

Looking at the data to see if it loaded correctly

Use the `percentiles` argument to display different percentiles. To see the 20th, 37.5th, and 74.23th percentiles, type

```
anes.describe(percentiles = [.20, .375, .7423])
```

To see just the `int` variables, type `anes.describe(include = "int")`, and to see just the `float` variables, type `anes.describe(include = "float")`.

To see `object` variables, type `anes.describe(include = "object")`. These variables have different stats:

- `count` – number of non-missing observations
- `unique` – number of unique observations
- `top` – the most frequent value
- `freq` – the frequency of the top value

Looking at the data to see if it loaded correctly

To see all of the variables, type `anes.describe(include = "all")`, but this will result in `NA` values for stats that aren't relevant to the variable.

Use these tools a lot! After loading the data, you need to quickly be able to see if there were any problems with loading the data. Ask:

- Are the dimensions what I expect?
- Are the variable names set to what they are supposed to be?
- Are there any bizarrely high/low means or other stats?

There are many reasons why a load might have failed. Fortunately, there are parameters within the `pd.read_csv()` function to deal with many of these issues.

Loading messy CSV and other ASCII files

```
pd.read_csv(filepath_or_buffer, sep, header)
```

`sep` or `delimiter` – (string) The symbol that is used in the file to separate one datapoint from the next on the same row. By default, it looks for commas.

- For tab-delimited, use `sep="\t"`
- For semi-colon delimited, use `sep=";"`

`header` – (integer or string) Where to look for **variable names**.

- The default is `header=0`, which uses the first row as variable names
- `header=None` assumes there are no variable names and that the first row is data. It labels the columns with numbers, but if you also type `prefix="X"` the variables will be X0, X1, . . .
- `header=j` uses the j_{th} row for variable names, and deletes all higher rows

Loading messy CSV and other ASCII files

```
pd.read_csv(filepath_or_buffer, sep, header, usecols)
```

`usecols` – (a list of strings or integers) Use this if you only want some of the columns to be loaded from the outset:

- `usecols = [0, 3, 5]` only loads the 1st, 4th, and 6th columns (**note that Python always starts at 0, making all indices off-by-one**)
- `usecols = ["caseid", "vote12", "meet"]` only loads the variables named "caseid", "vote12", and "meet", as recognized by whatever Python thinks is the header

In general, don't use this parameter unless the data file is too large to load in its entirety. You can delete columns later.

Loading messy CSV and other ASCII files

```
pd.read_csv(filepath_or_buffer, sep, header, usecols,
skiprows, skipfooter, nrows )
```

`skiprows` – (integer, or a list of integers) Likewise, which rows to skip when loading the data:

- `skiprows=3` skips the first three rows of the data. If `header` is left to its default, the 4th row is assumed to contain the variable names
- `skiprows=[0,3,5]` skips the 1st, 4th, and 6th rows

`skipfooter` – same as `skiprows` but counts up from the bottom row

`nrows` – (integer) only loads the first several rows, as specified by the user

Loading messy CSV and other ASCII files

```
pd.read_csv(filepath_or_buffer, sep, header, usecols,
skiprows, skipfooter, nrows, na_values )
```

`na_values` – (list of strings or numeric) Sometimes data authors use codes other than `NA` to indicate a missing value.

Example: the American National Election Study (ANES) data uses -7, -8, -9, and 998, as well as blank cells and `NA` to represent missing values.

To replace all these values with `NA` across the whole data frame, type `na_values = [-7, -8, -9, 998]` .

Caution: Only specify missing codes in the `pd.read_csv()` function if the code ALWAYS means a missing value. If 998 is a valid datapoint for some variables, you can replace the missing codes for relevant variables later.

Loading messy CSV and other ASCII files

```
pd.read_csv(filepath_or_buffer, sep, header, usecols,
skiprows, skipfooter, nrows, na_values, comment )
```

`comment` – (string) If there are comments in the data file itself (it shouldn't happen but it does!), what character to read as indicating a commented-out row.

If the data authors wrote "# Collected on Mon 9/23" before some rows, then "# Collected on Tues 9/24" further down, you can ignore these by typing `comment="#"` .

Careful: if the comment-symbol appears ANYWHERE on the row, the remainder of the row is not read. That's a problem if, for example, the data contain tweets and one tweet reads "UVA is #1!".

Writing CSV and ASCII files

Once the data are loaded into Python, there are many tools, techniques, and functions to know to get the data into a clean state.

We'll go over all of that in detail soon. But after having cleaned the data, you might want to **save the cleaned dataframe as a CSV** or as a different ASCII file.

Suppose the `anes` object contains a cleaned dataframe. To save it as a CSV, use `anes.to_csv()`. There are several parameters, you can see with `help(anes.to_csv)`.

Let's talk about two important parameters:

```
anes.to_csv(path_or_buf, sep)
```

Writing CSV and ASCII files

```
anes.to_csv(path_or_buf, sep)
```

path_or_buf – (string) the name of the file to save, with the appropriate file extension (.csv, .txt, etc.)

You can write an entire file path here if you want. But if you **set the working directory**, and write the file name alone, it will save in the working directory.

sep – (string) the character to use as a delimiter. A comma by default. Use sep="\t" for a tab-delimited file.

To save the anes dataframe as a standard CSV file, type:

```
anes.to_csv("anes_cleaned.csv", sep=",")
```

Loading fixed width files

A fixed-width file contains no delimiters. Instead, it aligns all of the data for one variable in the **same position** on each row. These files might use less memory than CSV.

But that makes the data impossible to parse without an external list of which variable is stored where. The first and most important step is to get this list.

Example: the National Journal conducted a public opinion poll and saved the data in fixed-width format. I saved the codebook on GitHub, and the data here:

https://raw.githubusercontent.com/NovaVolunteer/Practice_Application_DS/master/Week%205/njcc33850.dat

Loading fixed width files

In this codebook, find the variable names and save them in a list, for example:

```
datanames = ['psraid', 'sample', 'int_date', 'area',
'state', 'cregion', 'density', 'usr', 'cc1', 'cc1a',
'cc2', 'cc3', 'cc4', 'cc5', 'cc6', 'cc7', 'ql1', 'ql1a',
'qc1', 'hh1', 'employ', 'par', 'sex', 'age', 'educ2',
'hisp', 'race', 'inc', 'income', 'reg', 'party',
'partyln', 'iphoneus', 'hphoneus', 'recage', 'receduc',
'racethn', 'standwt', 'raceos']
```

There are **two ways** to proceed next:

Method 1: If you know how many characters each variable takes, at maximum, save these widths as a list:

```
datawidths = [6, 1, 6, 3, 2, 1, 1, 3, 1, 1,
            1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
            1, 1, 1, 2, 1, 1, 1, 2, 1, 1,
            1, 1, 1, 1, 1, 1, 1, 4, 30]
```

Loading fixed width files

Method 2: if you know the starting and ending position of each variable, create a **list of length 2** for each variable, where

- the first element is the column the previous variable ends on (or 0 for the first variable)
- and the second element is the column the current variable ends on.

For example, if a variable occupies columns 34, 35, and 36, its list of length 2 is `[33,36]`.

Create a list-of-lists, which can look like:

```
datapos = [[0,6], [6,7], [7,13], [13,16], [16,18],
    [18,19], [19,20], [20,23], [23,24], [24,25],
    [25,26], [26,27], [27,28], [28,29], [29,30],
    [30,31], [31,32], [32,33], [33,34], [34,35],
    [35,36], [36,37], [37,38], [38,40], [40,41],
    [41,42], [42,43], [43,45], [45,46], [46,47],
    [47,48], [48,49], [49,50], [50,51], [51,52],
    [52,53], [53,54], [54,58], [58,88]]
```

Loading fixed width files

To read the fixed-width file, use the `pd.read_fwf()` function.

To parse the data using variable widths, save the URL, the variable names, and widths in separate objects (as on the previous slides), and type:

```
njcc = pd.read_fwf(url, widths=datawidths,
    header=None, names=datanames)
```

To parse the data using variable positions, save the URL, the variable names, and positions in separate objects, and type:

```
njcc = pd.read_fwf(url, colspecs=datapos,
    header=None, names=datanames)
```

Loading Excel files

Like it or not, you will be working with Excel files as a professional data scientist. Excel is **ubiquitous**, and at some point, you will have to load Excel cleanly into Python.

My usual strategy: Open Excel, select the sheet I want, and save it as a CSV before loading it in Python.

But that's not a good strategy because it requires me to have access to Excel. To work entirely with Python, use the `pd.read_excel()` function.

Many of the parameters that work for `pd.read_csv()` work for `pd.read_excel()` too, including: `header`, `names`, `usecols`, `skiprows`, `skipfooter`, `nrows`, `na_values`, and `comment`.

Loading Excel files

There are two arguments we should go over:
`pd.read_excel(io, sheet_name)`

`io` – (string) This argument is the same as the `filepath_or_buffer` parameter for `pd.read_csv()`. It can be a

- path and filename,
- filename alone (if you've set the working directory),
- or a URL where the Excel file is stored online.

`sheet_name` – (string, int, or list) If the Excel file has sheets with names, you can type the name of the sheet here. Or type a number: **0 refers to the first sheet, 1 to the second**, etc.

If you specify a list, `pd.read_excel()` will produce a list of dataframes, one for each sheet you specify. Typing `sheet_name = None` produces a list with all of the sheets.

Loading Excel files

Example: I saved an Excel sheet on GitHub with NBA statistics, here:

https://github.com/NovaVolunteer/Practice_Application_DS/blob/master/Week%205/NBA-Team-Sample-BoxScore-Dataset.xlsx?raw=true

This Excel file has four sheets:

- **NBA-TEAM-SAMPLE** has team stats for every game last season;
- **METADATA** defines variables;
- **TEAMS** provides team names and locations;
- **PROVIDE DATE FORMAT** has information abouty date formats.

Loading Excel files

I save the URL as an object. Then, to load the **NBA-TEAM-SAMPLE** sheet, I type one of these lines:

```
nba = pd.read_excel(url, sheet_name="NBA-TEAM-SAMPLE")
nba = pd.read_excel(url, sheet_name=0)
```

To load the **TEAMS** sheet, I type one of these lines:

```
nba = pd.read_excel(url, sheet_name="TEAMS")
nba = pd.read_excel(url, sheet_name=2)
```

To load both sheets:

```
nba = pd.read_excel(url,
    sheet_name=["NBA-TEAM-SAMPLE","TEAMS"])
nba = pd.read_excel(url, sheet_name=[0,2])
```

Loading SAS, Stata, and SPSS files

We're living in a moment when open-source programming has become the standard for data science.

BUT until recently most data science was conducted using proprietary software: SAS, Stata, or SPSS. Many researchers still use these platforms. So **you will likely have to work with these files.**

Like with Excel, opening SAS/Stata/SPSS and saving as CSV is a bad solution because you need the software to do that, and the software is expensive.

Loading SAS, Stata, and SPSS files

Regular **SAS** files have the extension .sas7bdat, and compressed SAS files ("transport files") have the extension .xport. We'll work with a dataset on inflation, here:

https://github.com/NovaVolunteer/Practice_Application_DS/blob/master/Week%205/inflation.sas7bdat?raw=true

Stata files all have the extension .dta. We'll work with a CBS news poll, here:

https://github.com/NovaVolunteer/Practice_Application_DS/blob/master/Week%205/cbspoll.dta?raw=true

SPSS files have the extension .sav, or .zsav for compressed files. We'll work with the ANES in SPSS format:

https://github.com/NovaVolunteer/Practice_Application_DS/blob/master/Week%205/anes_timeseries_2016.sav?raw=true

Loading SAS, Stata, and SPSS files

You can load SAS and Stata files with Pandas using the `pd.read_sas()` and `pd.read_stata()` functions.

To load an SPSS file, you need to install the `pyreadstat` package,

```
pip install pyreadstat
```

and import this package

```
import pyreadstat
```

Then you can use the `pyreadstat.read_sav()` function.

Loading SAS, Stata, and SPSS files

These functions are very similar to `pd.read_csv()`, but one important difference is they can't read a URL. So you have to download a **local copy** of the files.

To load the SAS inflation data:

```
inflation = pd.read_sas("inflation.sas7bdat")
```

To load the Stata CBS poll data:

```
cbspoll = pd.read_stata("cbspoll.dta")
```

Loading SPSS data is trickier. You have to define **two objects**, separated by a comma. The first object will contain the dataframe, and the second object will contain the SPSS metadata:

```
anes_spss, anes_spss_meta =
    pyreadstat.read_sav("anes_timeseries_2016.sav")
```