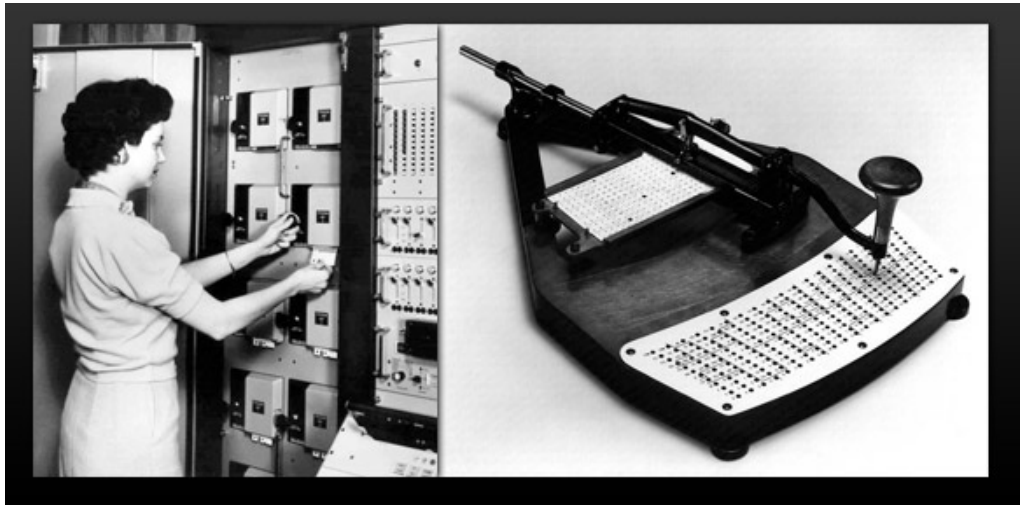
How to Load CSV, ASCII, and other data into Python

DS 6001: Practice and Applications of Data Science

Electronic Data Files

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



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.

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"

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

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.

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.

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.

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.

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.

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.

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.

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:

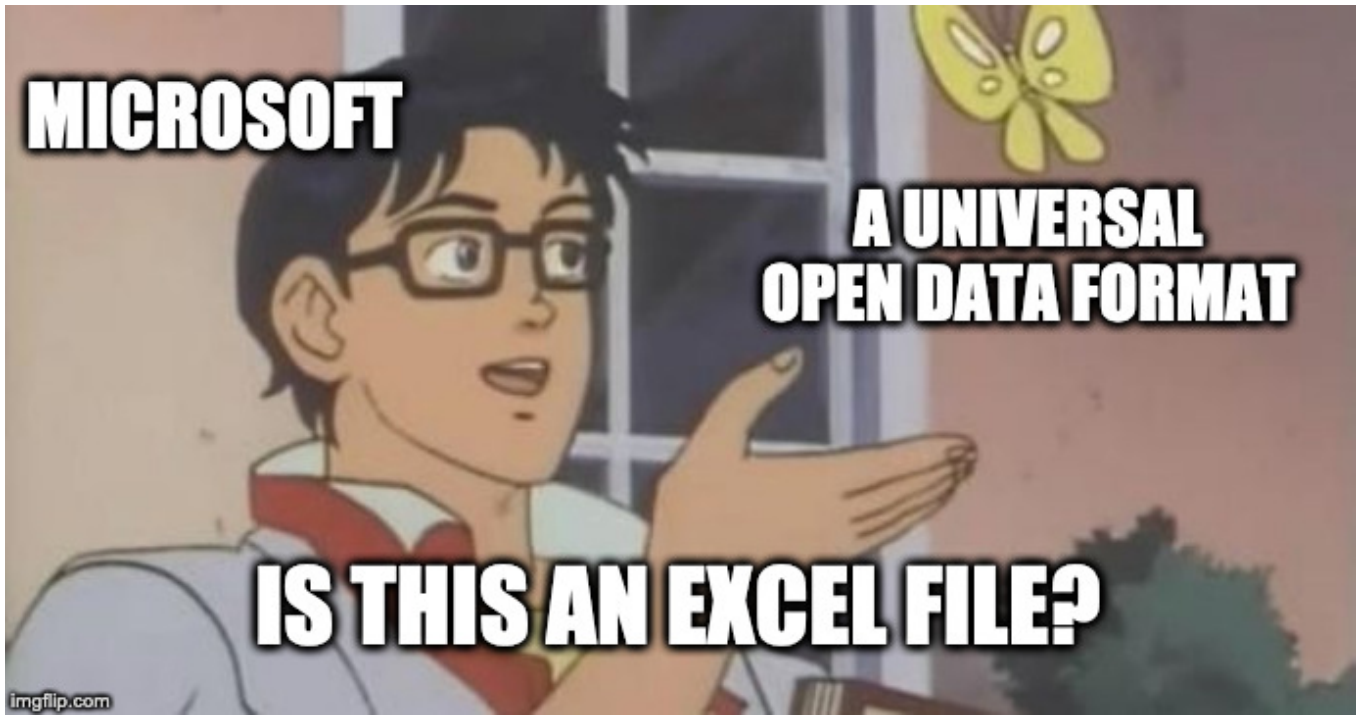
```
sex,race,region,happy,life,sibs,childs,age,educ,paeduc,maeduc,speduc,prestg80,
occcat80,tax,usintl,obey,popular,thnkself,workhard,helpoth,hlth1,hlth2,hlth3,
hlth4,hlth5,hlth6,hlth7,hlth8,hlth9,work1,work2,work3,work4,work5,work6,work7,
work8,work9,prob1,prob2,prob3,prob4
2,1,1,1,1,1,2,61,12,97,12,97,22,3,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0, , , ,
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

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.

```
21111126112971297223110000000000000000000000...
21121213220201820751105412311222222222121211245
11110213520161417591015412322222222222222222...
21192202620202097481104513212222222221222222222.
22121402512989897423110000000000000000000000...
```

Dictionary:

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

Loading CSV files

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:

Loading CSV files

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:

Loading CSV files

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

Loading CSV files

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

Loading CSV files

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

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.

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

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

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

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()`

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. Please see the notebook for a list and discussion of these functions.

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"`

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=";"`

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

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, ...

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:

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**)

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

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:

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

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

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

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.

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.

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]` .

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.

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="#"` .

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

```
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.)

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.

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.

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 `data` dataframe as a standard CSV file, type:

```
data.to_csv("data_cleaned.csv", sep=",")
```

Loading other electronic 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 generally do not store variable names, and might use less memory than CSV. But that makes the data impossible to parse without an external list of where each variable is stored.

Loading other electronic 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 generally do not store variable names, and might use less memory than CSV. But that makes the data impossible to parse without an external list of where each variable is stored.

You will also often need to work with file types from proprietary software: especially Excel, SAS, SPSS, and Stata.

Loading other electronic 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 generally do not store variable names, and might use less memory than CSV. But that makes the data impossible to parse without an external list of where each variable is stored.

You will also often need to work with file types from proprietary software: especially Excel, SAS, SPSS, and Stata.

See the notebook for the code and examples for loading each of these file types into Python.