csvkit

Release 1.0.2

Jun 29, 2017

Contents

1 About **1**

2 Why csvkit? **3**

3 Table of contents **5**

3.1 Tutorial 5
3.1.1 Getting started 5
3.1.2 Examining the data 8
3.1.3 Power tools 10
3.1.4 Going elsewhere with your data 15
3.2 Reference 16
3.2.1 Input 16
3.2.2 Processing 19
3.2.3 Output and Analysis 25
3.2.4 Common arguments 34
3.3 Tips and Troubleshooting 35
3.3.1 Tips 35
3.3.2 Troubleshooting 36
3.4 Contributing to csvkit 38
3.4.1 Getting Started 38
3.4.2 Principles of development 38
3.4.3 How to contribute 39
3.4.4 A note on new tools 39
3.4.5 Streaming versus buffering 39
3.4.6 Legalese 40
3.5 Release process 40
3.6 License 41
3.7 Changelog 41
3.7.1 1.0.2 - April 28, 2017 41
3.7.2 1.0.1 - December 29, 2016 42
3.7.3 1.0.0 - December 27, 2016 42
3.7.4 0.9.1 - March 31, 2015 44
3.7.5 0.9.0 44
3.7.6 0.8.0 45
3.7.7 0.7.3 45
3.7.8 0.7.2 46
3.7.9 0.7.1 46

3.7.10 0.7.0 46
3.7.11 0.6.1 46
3.7.12 0.5.0 47
3.7.13 0.4.4 47
3.7.14 0.4.3 47

4 Citation **49**

5 Authors **51**

6 Indices and tables **55**

CHAPTER 1

About

csvkit is a suite of command-line tools for converting to and working with CSV, the king of tabular file formats.

It is inspired by pdftk, gdal and the original csvcut tool by Joe Germuska and Aaron Bycoffe.

If you need to do more complex data analysis than csvkit can handle, use agate.

Important links:

- Repository: https://github.com/wireservice/csvkit
- Issues: https://github.com/wireservice/csvkit/issues
- Documentation: http://csvkit.rtfd.org/
- Schemas: https://github.com/wireservice/ffs
- Buildbot: https://travis-ci.org/wireservice/csvkit

CHAPTER 2

Why csvkit?

Because it makes your life easier.

Convert Excel to CSV:

```
in2csv data.xls > data.csv
```

Convert JSON to CSV:

```
in2csv data.json > data.csv
```

Print column names:

```
csvcut -n data.csv
```

Select a subset of columns:

```
csvcut -c column_a,column_c data.csv > new.csv
```

Reorder columns:

```
csvcut -c column_c,column_a data.csv > new.csv
```

Find rows with matching cells:

```
csvgrep -c phone_number -r "555-555-\d{4}" data.csv > new.csv
```

Convert to JSON:

```
csvjson data.csv > data.json
```

Generate summary statistics:

```
csvstat data.csv
```

Query with SQL:

```
csvsql --query "select name from data where age > 30" data.csv > new.csv
```

Import into PostgreSQL:

```
csvsql --db postgresql:///database --insert data.csv
```

Extract data from PostgreSQL:

```
sql2csv --db postgresql:///database --query "select * from data" > new.csv
```

And much more...

CHAPTER 3

Table of contents

Tutorial

The csvkit tutorial walks through processing and analyzing a real dataset:

Getting started

About this tutorial

There is no better way to learn how to use a new tool than to see it applied in a real world situation. To that end, this tutorial explains how to use csvkit tools by analyzing a real dataset.

The data we will be using is a subset of the United States Defense Logistic Agency Law Enforcement Support Office's (LESO) 1033 Program dataset, which describes how surplus military arms have been distributed to local police forces. This data was widely cited in the aftermath of the Ferguson, Missouri protests. The particular data we are using comes from an NPR report analyzing the data.

This tutorial assumes you have some basic familiarity with the command line. If you don't have much experience, fear not! This has been written with beginners in mind. No prior experience with data processing or analysis is assumed.

Installing csvkit

Installing csvkit is easy:

```
sudo pip install csvkit
```

If you have problems installing, look for help in the *Tips and Troubleshooting* section of the documentation.

Note: If you're familiar with virtualenv, it is better to install csvkit in its own environment. If you are doing this, then you should leave off the `sudo` in the previous command.

Getting the data

Let's start by creating a clean workspace:

```
mkdir csvkit_tutorial
cd csvkit_tutorial
```

Now let's fetch the data:

```
curl -L -O https://raw.githubusercontent.com/wireservice/csvkit/master/examples/
↪realdata/ne_1033_data.xlsx
```

in2csv: the Excel killer

For purposes of this tutorial, I've converted this data to Excel format. (NPR published it in CSV format.) If you have Excel you can open the file and take a look at it, but really, who wants to wait for Excel to load? Instead, let's convert it to a CSV:

```
in2csv ne_1033_data.xlsx
```

You should see a CSV version of the data dumped into your terminal. All csvkit tools write to the terminal output, called "standard out", by default. This isn't very useful, so let's write it to a file instead:

```
in2csv ne_1033_data.xlsx > data.csv
```

`data.csv` will now contain a CSV version of our original file. If you aren't familiar with the > syntax, it means "redirect standard out to a file". If that's hard to remember it may be more convenient to think of it as "save to".

We can verify the that the data is saved to the new file by using the `cat` command to print it:

```
cat data.csv
```

in2csv can convert a variety of common file formats to CSV, including both `.xls` and `.xlsx` Excel files, JSON files, and fixed-width formatted files.

csvlook: data periscope

Now that we have some data, we probably want to get some idea of what's in it. We could open it in Excel or Google Docs, but wouldn't it be nice if we could just take a look in the command line? To do that, we can use *csvlook*:

```
csvlook data.csv
```

At first the output of *csvlook* isn't going to appear very promising. You'll see a mess of data, pipe character and dashes. That's because this dataset has many columns and they won't all fit in the terminal at once. You have two options:

1. Pipe the output to `less -S` to display the lines without wrapping and use the arrow keys to scroll left and right:

```
csvlook data.csv | less -S
```

2. Reduce which columns of our dataset are displayed before we look at it. This is what will do in the next section.

csvcut: data scalpel

csvcut is the original csvkit tool. It inspired the rest. With it, we can select, delete and reorder the columns in our CSV. First, let's just see what columns are in our data:

```
csvcut -n data.csv
```

```
 1: state
 2: county
 3: fips
 4: nsn
 5: item_name
 6: quantity
 7: ui
 8: acquisition_cost
 9: total_cost
10: ship_date
11: federal_supply_category
12: federal_supply_category_name
13: federal_supply_class
14: federal_supply_class_name
```

As you'll can see, our dataset has fourteen columns. Let's take a look at just columns `2`, `5` and `6`:

```
csvcut -c 2,5,6 data.csv
```

Now we've reduced our output CSV to only three columns.

We can also refer to columns by their names to make our lives easier:

```
csvcut -c county,item_name,quantity data.csv
```

Putting it together with pipes

Now that we understand *in2csv*, *csvlook* and *csvcut* we can demonstrate the power of csvkit's when combined with the standard command-line "pipe". Try this command:

```
csvcut -c county,item_name,quantity data.csv | csvlook | head
```

In addition to specifying filenames, all csvkit tools accept an input file via "standard in". This means that, using the `|` ("pipe") character we can use the output of one csvkit tool as the input of the next.

In the example above, the output of `csvcut` becomes the input to `csvlook`. This also allow us to pipe output to standard Unix commands such as `head`, which prints only the first ten lines of its input. Here, the output of `csvlook` becomes the input of `head`.

Piping is a core feature of csvkit. Of course, you can always write the output of each command to a file using `>`. However, it's often faster and more convenient to use pipes to chain several commands together.

We can also pipe `in2csv`, allowing us to combine all our previous operations into one:

```
in2csv ne_1033_data.xlsx | csvcut -c county,item_name,quantity | csvlook | head
```

Summing up

All the csvkit tools work with standard input and output. Any tool can be piped into another and into another. The output of any tool can be redirected to a file. In this way they form a data processing "pipeline" of sorts, allowing you to do non-trivial, repeatable work without creating dozens of intermediary files.

Make sense? If you think you've got it figured out, you can move on to *Examining the data*.

Examining the data

csvstat: statistics without code

In the previous section we saw how we could use csvlook and csvcut to view slices of our data. This is a good tool for exploring a dataset, but in practice we usually need to get the broadest possible view before we can start diving into specifics.

csvstat is designed to give us just such a broad understanding of our data. Inspired by the `summary()` function from the computational statistics programming language "R", `csvstat` will generate summary statistics for all the data in a CSV file.

Let's examine summary statistics for a few columns from our dataset. As we learned in the last section, we can use `csvcut` and a pipe to pick out the columns we want:

```
csvcut -c county,acquisition_cost,ship_date data.csv | csvstat
```

```
1. county
  Text
  Nulls: False
  Unique values: 35
  Max length: 10
  5 most frequent values:
      DOUGLAS:      760
      DAKOTA:       42
      CASS: 37
      HALL: 23
      LANCASTER:    18
2. acquisition_cost
  Number
  Nulls: False
  Min: 0.0
  Max: 412000.0
  Sum: 5430787.55
  Mean: 5242.0729247104247104247104250425
  Median: 6000.0
  Standard Deviation: 13368.07836799839045093904423
  Unique values: 75
  5 most frequent values:
      6800.0:       304
      10747.0:      195
      6000.0:       105
      499.0:        98
      0.0:  81
3. ship_date
  Date
  Nulls: False
  Min: 2006-03-07
  Max: 2014-01-30
  Unique values: 84
  5 most frequent values:
      2013-04-25:   495
      2013-04-26:   160
      2008-05-20:   28
      2012-04-16:   26
      2006-11-17:   20

Row count: 1036
```

csvstat infers the type of data in each column and then performs basic statistics on it. The particular statistics computed depend on the type of the column (numbers, text, dates, etc).

In this example the first column, `county` was identified as type `Text`. We see that there are `35` counties represented in the dataset and that `DOUGLAS` is far and away the most frequently occurring. A quick Google search shows that there are `93` counties in Nebraska, so we know that either not every county received equipment or that the data is incomplete. We can also find out that Douglas county contains Omaha, the state's largest city by far.

The `acquisition_cost` column is type `Number`. We see that the largest individual cost was `412000.0`. (Probably dollars, but let's not presume.) Total acquisition costs were `5430787.55`.

Lastly, the `ship_date` column (type `Date`) shows us that the earliest data is from `2006` and the latest from `2014`. We may also note that an unusually large amount of equipment was shipped in April, 2013.

As a journalist, this quick glance at the data gave me a tremendous amount of information about the dataset. Although we have to be careful about assuming to much from this quick glance (always double-check the numbers mean what you think they mean!) it can be an invaluable way to familiarize yourself with a new dataset.

csvgrep: find the data you need

After reviewing the summary statistics you might wonder what equipment was received by a particular county. To get a simple answer to the question we can use *csvgrep* to search for the state's name amongst the rows. Let's also use `csvcut` to just look at the columns we care about and `csvlook` to format the output:

```
csvcut -c county,item_name,total_cost data.csv | csvgrep -c county -m LANCASTER |␣
↪csvlook
```

```
| county    | item_name                      | total_cost |
| --------- | ------------------------------ | ---------- |
| LANCASTER | RIFLE,5.56 MILLIMETER          |        120 |
| LANCASTER | RIFLE,5.56 MILLIMETER          |        120 |
| LANCASTER | RIFLE,5.56 MILLIMETER          |        120 |
| LANCASTER | RIFLE,5.56 MILLIMETER          |        120 |
| LANCASTER | RIFLE,5.56 MILLIMETER          |        120 |
| LANCASTER | RIFLE,5.56 MILLIMETER          |        120 |
| LANCASTER | RIFLE,5.56 MILLIMETER          |        120 |
| LANCASTER | RIFLE,5.56 MILLIMETER          |        120 |
| LANCASTER | RIFLE,5.56 MILLIMETER          |        120 |
| LANCASTER | RIFLE,5.56 MILLIMETER          |        120 |
| LANCASTER | LIGHT ARMORED VEHICLE          |          0 |
| LANCASTER | LIGHT ARMORED VEHICLE          |          0 |
| LANCASTER | LIGHT ARMORED VEHICLE          |          0 |
| LANCASTER | MINE RESISTANT VEHICLE         |    412,000 |
| LANCASTER | IMAGE INTENSIFIER,NIGHT VISION |      6,800 |
| LANCASTER | IMAGE INTENSIFIER,NIGHT VISION |      6,800 |
| LANCASTER | IMAGE INTENSIFIER,NIGHT VISION |      6,800 |
| LANCASTER | IMAGE INTENSIFIER,NIGHT VISION |      6,800 |
```

LANCASTER county contains Lincoln, Nebraska, the capital of the state and its second-largest city. The -m flag means "match" and will find text anywhere in a given column–in this case the county column. For those who need a more powerful search you can also use -r to search for a regular expression.

csvsort: order matters

Now let's use *csvsort* to sort the rows by the total_cost column, in reverse (descending) order:

```
csvcut -c county,item_name,total_cost data.csv | csvgrep -c county -m LANCASTER |␣
↪csvsort -c total_cost -r | csvlook
```

```
| county    | item_name                      | total_cost |
| --------- | ------------------------------ | ---------- |
| LANCASTER | MINE RESISTANT VEHICLE         |    412,000 |
| LANCASTER | IMAGE INTENSIFIER,NIGHT VISION |      6,800 |
| LANCASTER | IMAGE INTENSIFIER,NIGHT VISION |      6,800 |
| LANCASTER | IMAGE INTENSIFIER,NIGHT VISION |      6,800 |
| LANCASTER | IMAGE INTENSIFIER,NIGHT VISION |      6,800 |
| LANCASTER | RIFLE,5.56 MILLIMETER          |        120 |
| LANCASTER | RIFLE,5.56 MILLIMETER          |        120 |
| LANCASTER | RIFLE,5.56 MILLIMETER          |        120 |
| LANCASTER | RIFLE,5.56 MILLIMETER          |        120 |
| LANCASTER | RIFLE,5.56 MILLIMETER          |        120 |
| LANCASTER | RIFLE,5.56 MILLIMETER          |        120 |
| LANCASTER | RIFLE,5.56 MILLIMETER          |        120 |
| LANCASTER | RIFLE,5.56 MILLIMETER          |        120 |
| LANCASTER | RIFLE,5.56 MILLIMETER          |        120 |
| LANCASTER | RIFLE,5.56 MILLIMETER          |        120 |
| LANCASTER | LIGHT ARMORED VEHICLE          |          0 |
| LANCASTER | LIGHT ARMORED VEHICLE          |          0 |
| LANCASTER | LIGHT ARMORED VEHICLE          |          0 |
```

Two interesting things should jump out about this sorted data: that LANCASTER county got a very expensive MINE RESISTANT VEHICLE and that it also go three other LIGHT ARMORED VEHICLE.

What commands would you use to figure out if other counties also received large numbers of vehicles?

Summing up

At this point you should be able to use csvkit to investigate the basic properties of a dataset. If you understand this section, you should be ready to move onto *Power tools*.

Power tools

csvjoin: merging related data

One of the most common operations that we need to perform on data is "joining" it to other, related data. For instance, given a dataset about equipment supplied to counties in Nebraska, one might reasonably want to merge that with a dataset containing the population of each county. *csvjoin* allows us to take two those two datasets (equipment and population) and merge them, much like you might do with a SQL JOIN query. In order to demonstrate this, let's grab a second dataset:

```
curl -L -O https://raw.githubusercontent.com/wireservice/csvkit/master/examples/
↪realdata/acs2012_5yr_population.csv
```

Now let's see what's in there:

```
csvstat acs2012_5yr_population.csv
```

```
1. fips
  Number
  Nulls: False
  Min: 31001
  Max: 31185
  Sum: 2891649
  Mean: 31093
  Median: 31093
  Standard Deviation: 53.98147830506311726900562525
  Unique values: 93
  5 most frequent values:
      31001:        1
      31003:        1
      31005:        1
      31007:        1
      31009:        1
2. name
  Text
  Nulls: False
  Unique values: 93
  Max length: 23
  5 most frequent values:
      Adams County, NE:     1
      Antelope County, NE:  1
      Arthur County, NE:    1
      Banner County, NE:    1
      Blaine County, NE:    1
3. total_population
  Number
  Nulls: False
  Min: 348
  Max: 518271
  Sum: 1827306
  Mean: 19648.45161290322580645161290
  Median: 6294
  Standard Deviation: 62501.00530730896711321285542
  Unique values: 93
  5 most frequent values:
      31299:        1
      6655: 1
      490:  1
      778:  1
      584:  1
4. margin_of_error
  Number
  Nulls: False
  Min: 0
  Max: 115
  Sum: 1800
  Mean: 19.35483870967741935483870968
  Median: 0
  Standard Deviation: 37.89707031274211909117708454
  Unique values: 15
  5 most frequent values:
      0:    73
      73:   2
      114:  2
      97:   2
      99:   2

Row count: 93
```

As you can see, this data file contains population estimates for each county in Nebraska from the 2012 5-year ACS estimates. This data was retrieved from Census Reporter and reformatted slightly for this example. Let's join it to our equipment data:

```
csvjoin -c fips data.csv acs2012_5yr_population.csv > joined.csv
```

Since both files contain a fips column, we can use that to join the two. In our output you should see the population data appended at the end of each row of data. Let's combine this with what we've learned before to answer the question "What was the lowest population county to receive equipment?":

```
csvcut -c county,item_name,total_population joined.csv | csvsort -c total_population␣
↪| csvlook | head
```

```
| county     | item_name                                                      | total_
↪population |
| ---------- | -------------------------------------------------------------- | ------
↪---------- |
| MCPHERSON  | RIFLE,5.56 MILLIMETER                                          |       ␣
↪       348 |
| WHEELER    | RIFLE,5.56 MILLIMETER                                          |       ␣
↪       725 |
| GREELEY    | RIFLE,7.62 MILLIMETER                                          |       ␣
↪     2,515 |
| GREELEY    | RIFLE,7.62 MILLIMETER                                          |       ␣
↪     2,515 |
| GREELEY    | RIFLE,7.62 MILLIMETER                                          |       ␣
↪     2,515 |
| NANCE      | RIFLE,5.56 MILLIMETER                                          |       ␣
↪     3,730 |
| NANCE      | RIFLE,7.62 MILLIMETER                                          |       ␣
↪     3,730 |
| NANCE      | RIFLE,7.62 MILLIMETER                                          |       ␣
↪     3,730 |
```

Two counties with fewer than one-thousand residents were the recipients of 5.56 millimeter assault rifles. This simple example demonstrates the power of joining datasets. Although SQL will always be a more flexible option, `csvjoin` will often get you where you need to go faster.

csvstack: combining subsets

Frequently large datasets are distributed in many small files. At some point you will probably want to merge those files for bulk analysis. *csvstack* allows you to "stack" the rows from CSV files with the same columns (and identical column names). To demonstrate, let's imagine we've decided that Nebraska and Kansas form a "region" and that it would be useful to analyze them in a single dataset. Let's grab the Kansas data:

```
curl -L -O https://raw.githubusercontent.com/wireservice/csvkit/master/examples/
↪realdata/ks_1033_data.csv
```

Back in *Getting started*, we had used in2csv to convert our Nebraska data from XLSX to CSV. However, we named our output *data.csv* for simplicity at the time. Now that we are going to be stacking multiple states, we should re-convert our Nebraska data using a file naming convention matching our Kansas data:

```
in2csv ne_1033_data.xlsx > ne_1033_data.csv
```

Now let's stack these two data files:

```
csvstack ne_1033_data.csv ks_1033_data.csv > region.csv
```

Using csvstat we can see that our `region.csv` contains both datasets:

```
csvstat -c state,acquisition_cost region.csv
```

```
1. state
  Text
  Nulls: False
  Values: NE, KS
  Max length: 2
  5 most frequent values:
      KS:   1575
      NE:   1036
8. acquisition_cost
  Number
  Nulls: False
  Min: 0.0
  Max: 658000
  Sum: 9440445.91
  Mean: 3615.643780160857908847184987
  Median: 138
  Standard Deviation: 23730.63142202547205726466358
  Unique values: 127
  5 most frequent values:
      120.0:        649
      499.0:        449
      138.0:        311
      6800.0:       304
      58.71:        218

Row count: 2611
```

If you supply the `-g` flag then `csvstack` can also add a "grouping column" to each row, so that you can tell which file each row came from. In this case we don't need this, but you can imagine a situation in which instead of having a `county` column each of this datasets had simply been named `nebraska.csv` and `kansas.csv`. In that case, using a grouping column would prevent us from losing information when we stacked them.

csvsql and sql2csv: ultimate power

Sometimes (almost always), the command-line isn't enough. It would be crazy to try to do all your analysis using command-line tools. Often times, the correct tool for data analysis is SQL. *csvsql* and *sql2csv* form a bridge that eases migrating your data into and out of a SQL database. For smaller datasets `csvsql` can also leverage sqlite to allow execution of ad hoc SQL queries without ever touching a database.

By default, `csvsql` will generate a create table statement for your data. You can specify what sort of database you are using with the `-i` flag:

```
csvsql -i sqlite joined.csv
```

```
CREATE TABLE joined (
        state VARCHAR(2) NOT NULL,
        county VARCHAR(10) NOT NULL,
        fips DECIMAL NOT NULL,
        nsn VARCHAR(16) NOT NULL,
        item_name VARCHAR(62),
        quantity DECIMAL NOT NULL,
        ui VARCHAR(7) NOT NULL,
        acquisition_cost DECIMAL NOT NULL,
        total_cost DECIMAL NOT NULL,
        ship_date DATE NOT NULL,
        federal_supply_category DECIMAL NOT NULL,
        federal_supply_category_name VARCHAR(35) NOT NULL,
        federal_supply_class DECIMAL NOT NULL,
        federal_supply_class_name VARCHAR(63) NOT NULL,
        name VARCHAR(21) NOT NULL,
        total_population DECIMAL NOT NULL,
        margin_of_error DECIMAL NOT NULL
);
```

Here we have the sqlite "create table" statement for our joined data. You'll see that, like `csvstat`, `csvsql` has done its best to infer the column types.

Often you won't care about storing the SQL statements locally. You can also use `csvsql` to create the table directly in the database on your local machine. If you add the `--insert` option the data will also be imported:

```
csvsql --db sqlite:///leso.db --insert joined.csv
```

How can we check that our data was imported successfully? We could use the sqlite command-line interface, but rather than worry about the specifics of another tool, we can also use `sql2csv`:

```
sql2csv --db sqlite:///leso.db --query "select * from joined"
```

Note that the `--query` parameter to `sql2csv` accepts any SQL query. For example, to export Douglas county from the `joined` table from our sqlite database, we would run:

```
sql2csv --db sqlite:///leso.db --query "select * from joined where county='DOUGLAS';"␣
↪> douglas.csv
```

Sometimes, if you will only be running a single query, even constructing the database is a waste of time. For that case, you can actually skip the database entirely and `csvsql` will create one in memory for you:

```
csvsql --query "select county,item_name from joined where quantity > 5;" joined.csv |␣
↪csvlook
```

SQL queries directly on CSVs! Keep in mind when using this that you are loading the entire dataset into an in-memory database, so it is likely to be very slow for large datasets.

Summing up

`csvjoin`, `csvstack`, `csvsql` and `sql2csv` represent the power tools of csvkit. Using these tools can vastly simplify processes that would otherwise require moving data between other systems. But what about cases where these tools still don't cut it? What if you need to move your data onto the web or into a legacy database system? We've got a few solutions for those problems in our final section, *Going elsewhere with your data*.

Going elsewhere with your data

csvjson: going online

Very frequently one of the last steps in any data analysis is to get the data onto the web for display as a table, map or chart. CSV is rarely the ideal format for this. More often than not what you want is JSON and that's where *csvjson* comes in. `csvjson` takes an input CSV and outputs neatly formatted JSON. For the sake of illustration, let's use `csvcut` and `csvgrep` to convert just a small slice of our data:

```
csvcut -c county,item_name data.csv | csvgrep -c county -m "GREELEY" | csvjson --
↪indent 4
```

```
[
    {
        "county": "GREELEY",
        "item_name": "RIFLE,7.62 MILLIMETER"
    },
    {
        "county": "GREELEY",
        "item_name": "RIFLE,7.62 MILLIMETER"
    },
    {
        "county": "GREELEY",
        "item_name": "RIFLE,7.62 MILLIMETER"
    }
]
```

A common usage of turning a CSV into a JSON file is for usage as a lookup table in the browser. This can be illustrated with the ACS data we looked at earlier, which contains a unique `fips` code for each county:

```
csvjson --indent 4 --key fips acs2012_5yr_population.csv | head
```

```
{
    "31001": {
        "fips": "31001",
        "name": "Adams County, NE",
        "total_population": "31299",
        "margin_of_error": "0"
    },
    "31003": {
        "fips": "31003",
        "name": "Antelope County, NE",
        "...": "..."
    }
}
```

For making maps, `csvjson` can also output GeoJSON, see its *csvjson* for more details.

csvpy: going into code

For the programmers out there, the command line is rarely as functional as just writing a little bit of code. *csvpy* exists just to make a programmer's life easier. Invoking it simply launches a Python interactive terminal, with the data preloaded into a CSV reader:

```
csvpy data.csv
```

```
Welcome! "data.csv" has been loaded in a reader object named "reader".
>>> print len(list(reader))
1037
>>> quit()
```

In addition to being a time-saver, because this uses agate, the reader is Unicode aware.

csvformat: for legacy systems

It is a foundational principle of csvkit that it always outputs cleanly formatted CSV data. None of the normal csvkit tools can be forced to produce pipe or tab-delimited output, despite these being common formats. This principle is what allows the csvkit tools to chain together so easily and hopefully also reduces the amount of crummy, non-standard CSV files in the world. However, sometimes a legacy system just has to have a pipe-delimited file and it would be crazy to make you use another tool to create it. That's why we've got *csvformat*.

Pipe-delimited:

```
csvformat -D \| data.csv
```

Tab-delimited:

```
csvformat -T data.csv
```

Quote every cell:

```
csvformat -U 1 data.csv
```

Ampersand-delimited, dollar-signs for quotes, quote all strings, and asterisk for line endings:

```
csvformat -D \& -Q \$ -U 2 -M \* data.csv
```

You get the picture.

Summing up

Thus concludes the csvkit tutorial. At this point, I hope, you have a sense a breadth of possibilities these tools open up with a relatively small number of command-line tools. Of course, this tutorial has only scratched the surface of the available options, so remember to check the *Reference* documentation for each tool as well.

So armed, go forth and expand the empire of the king of tabular file formats.

Reference

csvkit is composed of command-line tools that can be divided into three major categories: Input, Processing, and Output. Documentation and examples for each tool are described on the following pages.

Input

in2csv

Description

Converts various tabular data formats into CSV.

Converting fixed width requires that you provide a schema file with the "-s" option. The schema file should have the following format:

```
column,start,length
name,0,30
birthday,30,10
age,40,3
```

The header line is required though the columns may be in any order:

```
usage: in2csv [-h] [-d DELIMITER] [-t] [-q QUOTECHAR] [-u {0,1,2,3}] [-b]
              [-p ESCAPECHAR] [-z FIELD_SIZE_LIMIT] [-e ENCODING] [-S] [-H]
              [-v] [-l] [--zero] [-V] [-f FILETYPE] [-s SCHEMA] [-k KEY] [-n]
              [--sheet SHEET] [-y SNIFF_LIMIT] [-I]
              [FILE]

Convert common, but less awesome, tabular data formats to CSV.

positional arguments:
  FILE                  The CSV file to operate on. If omitted, will accept
                        input on STDIN.

optional arguments:
  -h, --help            show this help message and exit
  -f FILETYPE, --format FILETYPE
                        The format of the input file. If not specified will be
                        inferred from the file type. Supported formats: csv,
                        dbf, fixed, geojson, json, ndjson, xls, xlsx.
  -s SCHEMA, --schema SCHEMA
                        Specify a CSV-formatted schema file for converting
                        fixed-width files. See documentation for details.
  -k KEY, --key KEY     Specify a top-level key to use look within for a list
                        of objects to be converted when processing JSON.
  -n, --names           Display sheet names from the input Excel file.
  --sheet SHEET         The name of the Excel sheet to operate on.
  --write-sheets WRITE_SHEETS
                        The names of the Excel sheets to write to files, or
                        "-" to write all sheets.
  -y SNIFF_LIMIT, --snifflimit SNIFF_LIMIT
                        Limit CSV dialect sniffing to the specified number of
                        bytes. Specify "0" to disable sniffing entirely.
  -I, --no-inference    Disable type inference when parsing CSV input.

Some command-line flags only pertain to specific input formats.
```

See also: *Arguments common to all tools*.

Note: The "ndjson" format refers to "newline delimited JSON", as used by many streaming APIs.

Note: If an XLS looks identical to an XLSX when viewed in Excel, they may not be identical as CSV. For example, XLSX has an integer type, but XLS doesn't. Numbers that look like integers from an XLS will have decimals in CSV, but those from an XLSX won't.

Note: To convert from HTML, consider messytables.

Examples

Convert the 2000 census geo headers file from fixed-width to CSV and from latin-1 encoding to utf8:

```
in2csv -e iso-8859-1 -f fixed -s examples/realdata/census_2000/census2000_geo_schema.
↪csv examples/realdata/census_2000/usgeo_excerpt.upl
```

Note: A library of fixed-width schemas is maintained in the `ffs` project:

https://github.com/wireservice/ffs

Convert an Excel .xls file:

```
in2csv examples/test.xls
```

Standardize the formatting of a CSV file (quoting, line endings, etc.):

```
in2csv examples/realdata/FY09_EDU_Recipients_by_State.csv
```

Fetch csvkit's open issues from the GitHub API, convert the JSON response into a CSV and write it to a file:

```
curl https://api.github.com/repos/wireservice/csvkit/issues?state=open | in2csv -f␣
↪json -v
```

Convert a DBase DBF file to an equivalent CSV:

```
in2csv examples/testdbf.dbf
```

sql2csv

Description

Executes arbitrary commands against a SQL database and outputs the results as a CSV:

```
usage: sql2csv [-h] [-v] [-l] [-V] [--db CONNECTION_STRING] [--query QUERY]
               [-e ENCODING] [-H]
               [FILE]

Execute an SQL query on a database and output the result to a CSV file.

positional arguments:
  FILE                  The file to use as SQL query. If both FILE and QUERY
                        are omitted, query will be read from STDIN.

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         Print detailed tracebacks when errors occur.
  -l, --linenumbers     Insert a column of line numbers at the front of the
                        output. Useful when piping to grep or as a simple
                        primary key.
  -V, --version         Display version information and exit.
  --db CONNECTION_STRING
                        An sqlalchemy connection string to connect to a
                        database.
  --query QUERY         The SQL query to execute. If specified, it overrides
                        FILE and STDIN.
  -e ENCODING, --encoding ENCODING
                        Specify the encoding of the input query file.
  -H, --no-header-row   Do not output column names.
```

Examples

Load sample data into a table using *csvsql* and then query it using *sql2csv*:

```
csvsql --db "sqlite:///dummy.db" --tables "test" --insert examples/dummy.csv
sql2csv --db "sqlite:///dummy.db" --query "select * from test"
```

Load data about financial aid recipients into PostgreSQL. Then find the three states that received the most, while also filtering out empty rows:

```
createdb recipients
csvsql --db "postgresql:///recipients" --tables "fy09" --insert examples/realdata/
↪FY09_EDU_Recipients_by_State.csv
sql2csv --db "postgresql:///recipients" --query "select * from fy09 where \"State␣
↪Name\" != '' order by fy09.\"TOTAL\" limit 3"
```

You can even use it as a simple SQL calculator (in this example an in-memory SQLite database is used as the default):

```
sql2csv --query "select 300 * 47 % 14 * 27 + 7000"
```

The connection string accepts parameters. For example, to set the encoding of a MySQL database:

```
sql2csv --db 'mysql://user:pass@host/database?charset=utf8' --query "SELECT myfield␣
↪FROM mytable"
```

Processing

csvclean

Description

Cleans a CSV file of common syntax errors:

- reports rows that have a different number of columns than the header row
- attempts to correct the CSV by joining short rows into a single row

Note that every csvkit tool does the following:

- removes optional quote characters, unless the *–quoting* (*-u*) option is set to change this behavior
- changes the field delimiter to a comma, if the input delimiter is set with the *–delimiter* (*-d*) or *–tabs* (*-t*) options

- changes the record delimiter to a line feed
- changes the quote character to a double-quotation mark, if the character is set with the –*quotechar* (*-q*) option
- changes the character encoding to UTF-8, if the input encoding is set with the –*encoding* (*-e*) option

Outputs [basename]_out.csv and [basename]_err.csv, the former containing all valid rows and the latter containing all error rows along with line numbers and descriptions:

```
usage: csvclean [-h] [-d DELIMITER] [-t] [-q QUOTECHAR] [-u {0,1,2,3}] [-b]
                [-p ESCAPECHAR] [-z FIELD_SIZE_LIMIT] [-e ENCODING] [-S] [-v]
                [-l] [--zero] [-V] [-n]
                [FILE]

Fix common errors in a CSV file.

positional arguments:
  FILE                  The CSV file to operate on. If omitted, will accept
                        input on STDIN.

optional arguments:
  -h, --help            show this help message and exit
  -n, --dry-run         Do not create output files. Information about what
                        would have been done will be printed to STDERR.
```

See also: *Arguments common to all tools*.

Examples

Test a file with known bad rows:

```
csvclean -n examples/bad.csv

Line 1: Expected 3 columns, found 4 columns
Line 2: Expected 3 columns, found 2 columns
```

csvcut

Description

Filters and truncates CSV files. Like the Unix “cut” command, but for tabular data:

```
usage: csvcut [-h] [-d DELIMITER] [-t] [-q QUOTECHAR] [-u {0,1,2,3}] [-b]
              [-p ESCAPECHAR] [-z FIELD_SIZE_LIMIT] [-e ENCODING] [-S] [-H]
              [-v] [-l] [--zero] [-V] [-n] [-c COLUMNS] [-C NOT_COLUMNS] [-x]
              [FILE]

Filter and truncate CSV files. Like the Unix "cut" command, but for tabular
data.

positional arguments:
  FILE                  The CSV file to operate on. If omitted, will accept
                        input on STDIN.

optional arguments:
  -h, --help            show this help message and exit
  -n, --names           Display column names and indices from the input CSV
                        and exit.
  -c COLUMNS, --columns COLUMNS
                        A comma separated list of column indices or names to
                        be extracted. Defaults to all columns.
  -C NOT_COLUMNS, --not-columns NOT_COLUMNS
                        A comma separated list of column indices or names to
                        be excluded. Defaults to no columns.
  -x, --delete-empty-rows
                        After cutting, delete rows which are completely empty.
```

See also: *Arguments common to all tools*.

Note: csvcut does not implement row filtering, for this you should pipe data to *csvgrep*.

Examples

Print the indices and names of all columns:

```
csvcut -n examples/realdata/FY09_EDU_Recipients_by_State.csv
  1: State Name
  2: State Abbreviate
  3: Code
  4: Montgomery GI Bill-Active Duty
  5: Montgomery GI Bill- Selective Reserve
  6: Dependents' Educational Assistance
  7: Reserve Educational Assistance Program
  8: Post-Vietnam Era Veteran's Educational Assistance Program
  9: TOTAL
 10:
```

Extract the first and third columns:

```
csvcut -c 1,3 examples/realdata/FY09_EDU_Recipients_by_State.csv
```

Extract columns named “TOTAL” and “State Name” (in that order):

```
csvcut -c TOTAL,"State Name" examples/realdata/FY09_EDU_Recipients_by_State.csv
```

Add line numbers to a file, making no other changes:

```
csvcut -l examples/realdata/FY09_EDU_Recipients_by_State.csv
```

Extract a column that may not exist in all files:

```
echo d, | csvjoin examples/dummy.csv - | csvcut -c d
echo d, | csvjoin examples/join_no_header_row.csv - | csvcut -c d
```

Display a column’s unique values:

```
csvcut -c 1 examples/realdata/FY09_EDU_Recipients_by_State.csv | sed 1d | sort | uniq
```

csvgrep

Description

Filter tabular data to only those rows where certain columns contain a given value or match a regular expression:

```
usage: csvgrep [-h] [-d DELIMITER] [-t] [-q QUOTECHAR] [-u {0,1,2,3}] [-b]
               [-p ESCAPECHAR] [-z FIELD_SIZE_LIMIT] [-e ENCODING] [-S] [-H]
               [-v] [-l] [--zero] [-V] [-n] [-c COLUMNS] [-m PATTERN]
               [-r REGEX] [-f MATCHFILE] [-i]
               [FILE]

Search CSV files. Like the Unix "grep" command, but for tabular data.

positional arguments:
  FILE                  The CSV file to operate on. If omitted, will accept
                        input on STDIN.

optional arguments:
  -h, --help            show this help message and exit
  -n, --names           Display column names and indices from the input CSV
                        and exit.
  -c COLUMNS, --columns COLUMNS
                        A comma separated list of column indices or names to
                        be searched.
  -m PATTERN, --match PATTERN
                        The string to search for.
  -r REGEX, --regex REGEX
                        If specified, must be followed by a regular expression
                        which will be tested against the specified columns.
  -f MATCHFILE, --file MATCHFILE
                        If specified, must be the path to a file. For each
                        tested row, if any line in the file (stripped of line
                        separators) is an exact match for the cell value, the
                        row will pass.
  -i, --invert-match    If specified, select non-matching instead of matching
                        rows.
```

See also: *Arguments common to all tools*.

NOTE: Even though '-m', '-r', and '-f' are listed as "optional" arguments, you must specify one of them.

Examples

Search for the row relating to Illinois:

```
csvgrep -c 1 -m ILLINOIS examples/realdata/FY09_EDU_Recipients_by_State.csv
```

Search for rows relating to states with names beginning with the letter "I":

```
csvgrep -c 1 -r "^I" examples/realdata/FY09_EDU_Recipients_by_State.csv
```

Search for rows that do not contain an empty state cell:

```
csvgrep -c 1 -r "^$" -i examples/realdata/FY09_EDU_Recipients_by_State.csv
```

csvjoin

Description

Merges two or more CSV tables together using a method analogous to SQL JOIN operation. By default it performs an inner join, but full outer, left outer, and right outer are also available via flags. Key columns are specified with the -c flag (either a single column which exists in all tables, or a comma-separated list of columns with one corresponding to each). If the columns flag is not provided then the tables will be merged “sequentially”, that is they will be merged in row order with no filtering:

```
usage: csvjoin [-h] [-d DELIMITER] [-t] [-q QUOTECHAR] [-u {0,1,2,3}] [-b]
               [-p ESCAPECHAR] [-z FIELD_SIZE_LIMIT] [-e ENCODING] [-S] [-H]
               [-v] [-l] [--zero] [-V] [-c COLUMNS] [--outer] [--left]
               [--right] [-y SNIFF_LIMIT] [-I]
               [FILE [FILE ...]]

Execute a SQL-like join to merge CSV files on a specified column or columns.

positional arguments:
  FILE                  The CSV files to operate on. If only one is specified,
                        it will be copied to STDOUT.

optional arguments:
  -h, --help            show this help message and exit
  -c COLUMNS, --columns COLUMNS
                        The column name(s) on which to join. Should be either
                        one name (or index) or a comma-separated list with one
                        name (or index) for each file, in the same order that
                        the files were specified. May also be left
                        unspecified, in which case the two files will be
                        joined sequentially without performing any matching.
  --outer               Perform a full outer join, rather than the default
                        inner join.
  --left                Perform a left outer join, rather than the default
                        inner join. If more than two files are provided this
                        will be executed as a sequence of left outer joins,
                        starting at the left.
  --right               Perform a right outer join, rather than the default
                        inner join. If more than two files are provided this
                        will be executed as a sequence of right outer joins,
                        starting at the right.
  -y SNIFF_LIMIT, --snifflimit SNIFF_LIMIT
                        Limit CSV dialect sniffing to the specified number of
                        bytes. Specify "0" to disable sniffing entirely.
  -I, --no-inference    Disable type inference when parsing CSV input.

Note that the join operation requires reading all files into memory. Don't try
this on very large files.
```

See also: *Arguments common to all tools*.

Examples

```
csvjoin -c 1 examples/join_a.csv examples/join_b.csv
```

This command says you have two files to outer join, file1.csv and file2.csv. The key column in file1.csv is ColumnKey, the key column in file2.csv is Column Key.

Add two empty columns to the right of a CSV:

```
echo "," | csvjoin examples/dummy.csv -
```

csvsort

Description

Sort CSV files. Like the Unix "sort" command, but for tabular data:

```
usage: csvsort [-h] [-d DELIMITER] [-t] [-q QUOTECHAR] [-u {0,1,2,3}] [-b]
               [-p ESCAPECHAR] [-z FIELD_SIZE_LIMIT] [-e ENCODING] [-S] [-H]
               [-v] [-l] [--zero] [-V] [-n] [-c COLUMNS] [-r] [-y SNIFF_LIMIT]
               [-I]
               [FILE]

Sort CSV files. Like the Unix "sort" command, but for tabular data.

positional arguments:
  FILE                  The CSV file to operate on. If omitted, will accept
                        input on STDIN.

optional arguments:
  -h, --help            show this help message and exit
  -n, --names           Display column names and indices from the input CSV
                        and exit.
  -c COLUMNS, --columns COLUMNS
                        A comma separated list of column indices or names to
                        sort by. Defaults to all columns.
  -r, --reverse         Sort in descending order.
  -y SNIFF_LIMIT, --snifflimit SNIFF_LIMIT
                        Limit CSV dialect sniffing to the specified number of
                        bytes. Specify "0" to disable sniffing entirely.
  -I, --no-inference    Disable type inference when parsing the input.
```

See also: *Arguments common to all tools*.

Note: If your file is large, try `sort -t, file.csv` instead.

Examples

Sort the veteran's education benefits table by the "TOTAL" column:

```
csvsort -c 9 examples/realdata/FY09_EDU_Recipients_by_State.csv
```

View the five states with the most individuals claiming veteran's education benefits:

```
csvcut -c 1,9 examples/realdata/FY09_EDU_Recipients_by_State.csv | csvsort -r -c 2 |␣
↪head -n 5
```

csvstack

Description

Stack up the rows from multiple CSV files, optionally adding a grouping value to each row:

```
usage: csvstack [-h] [-d DELIMITER] [-t] [-q QUOTECHAR] [-u {0,1,2,3}] [-b]
                [-p ESCAPECHAR] [-z FIELD_SIZE_LIMIT] [-e ENCODING] [-S] [-H]
                [-v] [-l] [--zero] [-V] [-g GROUPS] [-n GROUP_NAME]
                [--filenames]
                FILE [FILE ...]

Stack up the rows from multiple CSV files, optionally adding a grouping value.

positional arguments:
  FILE                  The CSV file(s) to operate on. If omitted, will accept
                        input on STDIN.

optional arguments:
  -h, --help            show this help message and exit
  -g GROUPS, --groups GROUPS
                        A comma-separated list of values to add as "grouping
                        factors", one for each CSV being stacked. These will
                        be added to the stacked CSV as a new column. You may
                        specify a name for the grouping column using the -n
                        flag.
  -n GROUP_NAME, --group-name GROUP_NAME
                        A name for the grouping column, e.g. "year". Only used
                        when also specifying -g.
  --filenames           Use the filename of each input file as its grouping
                        value. When specified, -g will be ignored.
```

See also: *Arguments common to all tools*.

Examples

Contrived example: joining a set of homogoenous files for different years:

```
csvstack -g 2009,2010 examples/realdata/FY09_EDU_Recipients_by_State.csv examples/
↪realdata/Datagov_FY10_EDU_recp_by_State.csv
```

Output and Analysis

csvformat

Description

Convert a CSV file to a custom output format.:

```
usage: csvformat [-h] [-d DELIMITER] [-t] [-q QUOTECHAR] [-u {0,1,2,3}] [-b]
                 [-p ESCAPECHAR] [-z FIELD_SIZE_LIMIT] [-e ENCODING] [-S] [-v]
                 [-V] [-D OUT_DELIMITER] [-T] [-Q OUT_QUOTECHAR]
                 [-U {0,1,2,3}] [-B] [-P OUT_ESCAPECHAR]
                 [-M OUT_LINETERMINATOR]
```

```
                  [FILE]

Convert a CSV file to a custom output format.

positional arguments:
  FILE                  The CSV file to operate on. If omitted, will accept
                        input on STDIN.

optional arguments:
  -h, --help            show this help message and exit
  -D OUT_DELIMITER, --out-delimiter OUT_DELIMITER
                        Delimiting character of the output CSV file.
  -T, --out-tabs        Specify that the output CSV file is delimited with
                        tabs. Overrides "-D".
  -Q OUT_QUOTECHAR, --out-quotechar OUT_QUOTECHAR
                        Character used to quote strings in the output CSV
                        file.
  -U {0,1,2,3}, --out-quoting {0,1,2,3}
                        Quoting style used in the output CSV file. 0 = Quote
                        Minimal, 1 = Quote All, 2 = Quote Non-numeric, 3 =
                        Quote None.
  -B, --out-no-doublequote
                        Whether or not double quotes are doubled in the output
                        CSV file.
  -P OUT_ESCAPECHAR, --out-escapechar OUT_ESCAPECHAR
                        Character used to escape the delimiter in the output
                        CSV file if --quoting 3 ("Quote None") is specified
                        and to escape the QUOTECHAR if --no-doublequote is
                        specified.
  -M OUT_LINETERMINATOR, --out-lineterminator OUT_LINETERMINATOR
                        Character used to terminate lines in the output CSV
                        file.
```

See also: *Arguments common to all tools*.

Examples

Convert a comma-separated file to a pipe-delimited file:

```
csvformat -D "|" examples/dummy.csv
```

Convert to carriage return line-endings:

```
csvformat -M $'\r' examples/dummy.csv
```

csvjson

Description

Converts a CSV file into JSON or GeoJSON (depending on flags):

```
usage: csvjson [-h] [-d DELIMITER] [-t] [-q QUOTECHAR] [-u {0,1,2,3}] [-b]
               [-p ESCAPECHAR] [-z FIELD_SIZE_LIMIT] [-e ENCODING] [-S] [-v]
               [-l] [--zero] [-V] [-i INDENT] [-k KEY] [--lat LAT] [--lon LON]
```

```
                [--crs CRS] [--stream] [-y SNIFF_LIMIT] [-I]
                [FILE]

Convert a CSV file into JSON (or GeoJSON).

positional arguments:
  FILE                  The CSV file to operate on. If omitted, will accept
                        input on STDIN.

optional arguments:
  -h, --help            show this help message and exit
  -i INDENT, --indent INDENT
                        Indent the output JSON this many spaces. Disabled by
                        default.
  -k KEY, --key KEY     Output JSON as an array of objects keyed by a given
                        column, KEY, rather than as a list. All values in the
                        column must be unique. If --lat and --lon are also
                        specified, this column will be used as GeoJSON Feature
                        ID.
  --lat LAT             A column index or name containing a latitude. Output
                        will be GeoJSON instead of JSON. Only valid if --lon
                        is also specified.
  --lon LON             A column index or name containing a longitude. Output
                        will be GeoJSON instead of JSON. Only valid if --lat
                        is also specified.
  --crs CRS             A coordinate reference system string to be included
                        with GeoJSON output. Only valid if --lat and --lon are
                        also specified.
  --stream              Output JSON as a stream of newline-separated objects,
                        rather than an as an array.
  -y SNIFF_LIMIT, --snifflimit SNIFF_LIMIT
                        Limit CSV dialect sniffing to the specified number of
                        bytes. Specify "0" to disable sniffing entirely.
  -I, --no-inference    Disable type inference when parsing CSV input.
```

See also: *Arguments common to all tools*.

Examples

Convert veteran’s education dataset to JSON keyed by state abbreviation:

```
csvjson -k "State Abbreviate" -i 4 examples/realdata/FY09_EDU_Recipients_by_State.csv
```

Results in a JSON document like:

```
{
    [...]
    "WA": {
        "State Name": "WASHINGTON",
        "State Abbreviate": "WA",
        "Code": 53.0,
        "Montgomery GI Bill-Active Duty": 7969.0,
        "Montgomery GI Bill- Selective Reserve": 769.0,
        "Dependents' Educational Assistance": 2192.0,
        "Reserve Educational Assistance Program": 549.0,
        "Post-Vietnam Era Veteran's Educational Assistance Program": 13.0,
        "TOTAL": 11492.0,
        "": null
    },
    [...]
}
```

Converting locations of public art into GeoJSON:

```
csvjson --lat latitude --lon longitude --k slug --crs EPSG:4269 -i 4 examples/test_
↪geo.csv
```

Results in a GeoJSON document like:

```
{
    "type": "FeatureCollection",
    "bbox": [
        -95.334619,
        32.299076986939205,
        -95.250699,
        32.351434
    ],
    "crs": {
        "type": "name",
        "properties": {
            "name": "EPSG:4269"
        }
    },
    "features": [
        {
            "type": "Feature",
            "id": "dcl",
            "geometry": {
                "type": "Point",
                "coordinates": [
                    -95.30181,
                    32.35066
                ]
            },
            "properties": {
                "title": "Downtown Coffee Lounge",
                "artist": null,
                "description": "In addition to being the only coffee shop in downtown␣
↪Tyler, DCL also features regular exhibitions of work by local artists.",
                "install_date": null,
                "address": "200 West Erwin Street",
                "type": "Gallery",
                "photo_url": null,
                "photo_credit": null,
                "last_seen_date": "2012-03-30"
            }
        },
        [...]
    ],
    "crs": {
        "type": "name",
        "properties": {
            "name": "EPSG:4269"
        }
    }
}
```

csvlook

Description

Renders a CSV to the command line in a Markdown-compatible, fixed-width format:

```
usage: csvlook [-h] [-d DELIMITER] [-t] [-q QUOTECHAR] [-u {0,1,2,3}] [-b]
               [-p ESCAPECHAR] [-z FIELD_SIZE_LIMIT] [-e ENCODING] [-S] [-H]
               [-v] [-l] [--zero] [-V] [--max-rows MAX_ROWS]
               [--max-columns MAX_COLUMNS]
               [--max-column-width MAX_COLUMN_WIDTH] [-y SNIFF_LIMIT] [-I]
               [FILE]

Render a CSV file in the console as a Markdown-compatible, fixed-width table.

positional arguments:
  FILE                  The CSV file to operate on. If omitted, will accept
                        input on STDIN.

optional arguments:
  -h, --help            show this help message and exit
  --max-rows MAX_ROWS   The maximum number of rows to display before
                        truncating the data.
  --max-columns MAX_COLUMNS
                        The maximum number of columns to display before
                        truncating the data.
  --max-column-width MAX_COLUMN_WIDTH
                        Truncate all columns to at most this width. The
                        remainder will be replaced with ellipsis.
  -y SNIFF_LIMIT, --snifflimit SNIFF_LIMIT
                        Limit CSV dialect sniffing to the specified number of
                        bytes. Specify "0" to disable sniffing entirely.
  -I, --no-inference    Disable type inference when parsing the input.
```

If a table is too wide to display properly try piping the output to `less -S` or truncating it using *csvcut*.

If the table is too long, try filtering it down with grep or piping the output to `less`.

See also: *Arguments common to all tools*.

Examples

Basic use:

```
csvlook examples/testfixed_converted.csv
```

This tool is especially useful as a final operation when piping through other tools:

```
csvcut -c 9,1 examples/realdata/FY09_EDU_Recipients_by_State.csv | csvlook
```

csvpy

Description

Loads a CSV file into a `agate.csv.Reader` object and then drops into a Python shell so the user can inspect the data however they see fit:

```
usage: csvpy [-h] [-d DELIMITER] [-t] [-q QUOTECHAR] [-u {0,1,2,3}] [-b]
             [-p ESCAPECHAR] [-z FIELD_SIZE_LIMIT] [-e ENCODING] [-S] [-v]
             [-V] [--dict] [--agate]
             [FILE]

Load a CSV file into a CSV reader and then drop into a Python shell.

positional arguments:
  FILE                  The CSV file to operate on. If omitted, will accept
                        input on STDIN.

optional arguments:
  -h, --help            show this help message and exit
  --dict                Load the CSV file into a DictReader.
  --agate               Load the CSV file into an agate table.
```

This tool will automatically use the IPython shell if it is installed, otherwise it will use the running Python shell.

Note: Due to platform limitations, csvpy does not accept file input on STDIN.

See also: *Arguments common to all tools*.

Examples

Basic use:

```
csvpy examples/dummy.csv
Welcome! "examples/dummy.csv" has been loaded in a reader object named "reader".
>>> reader.next()
[u'a', u'b', u'c']
```

As a dictionary:

```
csvpy --dict examples/dummy.csv
Welcome! "examples/dummy.csv" has been loaded in a DictReader object named "reader".
>>> reader.next()
{u'a': u'1', u'c': u'3', u'b': u'2'}
```

As an agate table:

```
csvpy --agate examples/dummy.csv
Welcome! "examples/dummy.csv" has been loaded in a from_csv object named "reader".
>>> reader.print_table()
|    a | b | c |
| ---- | - | - |
| True | 2 | 3 |
```

csvsql

Description

Generate SQL statements for a CSV file or execute those statements directly on a database. In the latter case supports both creating tables and inserting data:

```
usage: csvsql [-h] [-d DELIMITER] [-t] [-q QUOTECHAR] [-u {0,1,2,3}] [-b]
              [-p ESCAPECHAR] [-z FIELD_SIZE_LIMIT] [-e ENCODING] [-S] [-H]
              [-v] [--zero] [-V]
              [-i {firebird,mssql,mysql,oracle,postgresql,sqlite,sybase}]
              [--db CONNECTION_STRING] [--query QUERY] [--insert]
              [--tables TABLE_NAMES] [--no-constraints] [--no-create]
              [--blanks] [--db-schema DB_SCHEMA] [-y SNIFF_LIMIT] [-I]
              [FILE [FILE ...]]

Generate SQL statements for one or more CSV files, or execute those statements
directly on a database, and execute one or more SQL queries.

positional arguments:
  FILE                  The CSV file(s) to operate on. If omitted, will accept
                        input on STDIN.

optional arguments:
  -h, --help            show this help message and exit
  -i {firebird,mssql,mysql,oracle,postgresql,sqlite,sybase}, --dialect {firebird,
↪mssql,mysql,oracle,postgresql,sqlite,sybase}
                        Dialect of SQL to generate. Only valid when --db is
                        not specified.
  --db CONNECTION_STRING
                        If present, a sqlalchemy connection string to use to
                        directly execute generated SQL on a database.
  --query QUERY         Execute one or more SQL queries delimited by ";" and
                        output the result of the last query as CSV. QUERY
                        may be a filename.
  --insert              In addition to creating the table, also insert the
                        data into the table. Only valid when --db is
                        specified.
  --prefix PREFIX       Add an expression following the INSERT keyword, like
                        IGNORE or REPLACE.
  --tables TABLE_NAMES  Specify the names of the tables to be created. By
                        default, the tables will be named after the filenames
                        without extensions or "stdin".
  --no-constraints      Generate a schema without length limits or null
                        checks. Useful when sampling big tables.
  --no-create           Skip creating a table. Only valid when --insert is
                        specified.
  --overwrite           Drop the table before creating.
  --db-schema DB_SCHEMA
                        Optional name of database schema to create table(s)
                        in.
  -y SNIFF_LIMIT, --snifflimit SNIFF_LIMIT
                        Limit CSV dialect sniffing to the specified number of
                        bytes. Specify "0" to disable sniffing entirely.
  -I, --no-inference    Disable type inference when parsing the input.
```

See also: *Arguments common to all tools*.

For information on connection strings and supported dialects refer to the SQLAlchemy documentation.

If you prefer not to enter your password in the connection string, store the password securely in a PostgreSQL Password File, a MySQL Options File or similar files for other systems.

Note: Using the `--query` option may cause rounding (in Python 2) or introduce [Python floating point issues](https://docs.python.org/3.4/tutorial/floatingpoint.html) (in Python 3).

Examples

Generate a statement in the PostgreSQL dialect:

```
csvsql -i postgresql examples/realdata/FY09_EDU_Recipients_by_State.csv
```

Create a table and import data from the CSV directly into PostgreSQL:

```
createdb test
csvsql --db postgresql:///test --tables fy09 --insert examples/realdata/FY09_EDU_
↪Recipients_by_State.csv
```

For large tables it may not be practical to process the entire table. One solution to this is to analyze a sample of the table. In this case it can be useful to turn off length limits and null checks with the `no-constraints` option:

```
head -n 20 examples/realdata/FY09_EDU_Recipients_by_State.csv | csvsql --no-
↪constraints --tables fy09
```

Create tables for an entire folder of CSVs and import data from those files directly into PostgreSQL:

```
createdb test
csvsql --db postgresql:///test --insert examples/*_converted.csv
```

If those CSVs have identical headers, you can import them into the same table by using *csvstack* first:

```
createdb test
csvstack examples/dummy?.csv | csvsql --db postgresql:///test --insert
```

Group rows by one column:

```
csvsql --query "select * from 'dummy3' group by a" examples/dummy3.csv
```

You can also use CSVSQL to “directly” query one or more CSV files. Please note that this will create an in-memory SQL database, so it won’t be very fast:

```
csvsql --query  "select m.usda_id, avg(i.sepal_length) as mean_sepal_length from iris␣
↪as i join irismeta as m on (i.species = m.species) group by m.species" examples/
↪iris.csv examples/irismeta.csv
```

csvstat

Description

Prints descriptive statistics for all columns in a CSV file. Will intelligently determine the type of each column and then print analysis relevant to that type (ranges for dates, mean and median for integers, etc.):

```
usage: csvstat [-h] [-d DELIMITER] [-t] [-q QUOTECHAR] [-u {0,1,2,3}] [-b]
               [-p ESCAPECHAR] [-z FIELD_SIZE_LIMIT] [-e ENCODING] [-S] [-H]
               [-v] [--zero] [-V] [--csv] [-n] [-c COLUMNS] [--type] [--nulls]
               [--unique] [--min] [--max] [--sum] [--mean] [--median]
               [--stdev] [--len] [--freq] [--freq-count FREQ_COUNT] [--count]
               [-y SNIFF_LIMIT]
               [FILE]

Print descriptive statistics for each column in a CSV file.

positional arguments:
  FILE                  The CSV file to operate on. If omitted, will accept
                        input on STDIN.

optional arguments:
  -h, --help            show this help message and exit
  --csv                 Output results as a CSV, rather than text.
  -n, --names           Display column names and indices from the input CSV
                        and exit.
  -c COLUMNS, --columns COLUMNS
                        A comma separated list of column indices or names to
                        be examined. Defaults to all columns.
  --type                Only output data type.
  --nulls               Only output whether columns contains nulls.
  --unique              Only output counts of unique values.
  --min                 Only output smallest values.
  --max                 Only output largest values.
  --sum                 Only output sums.
  --mean                Only output means.
  --median              Only output medians.
  --stdev               Only output standard deviations.
  --len                 Only output the length of the longest values.
  --freq                Only output lists of frequent values.
  --freq-count FREQ_COUNT
                        The maximum number of frequent values to display.
  --count               Only output total row count.
  -y SNIFF_LIMIT, --snifflimit SNIFF_LIMIT
                        Limit CSV dialect sniffing to the specified number of
                        bytes. Specify "0" to disable sniffing entirely.
```

See also: *Arguments common to all tools*.

Examples

Basic use:

```
csvstat examples/realdata/FY09_EDU_Recipients_by_State.csv
```

When an statistic name is passed, only that stat will be printed:

```
csvstat --min examples/realdata/FY09_EDU_Recipients_by_State.csv

    1. State Name: None
    2. State Abbreviate: None
    3. Code: 1
    4. Montgomery GI Bill-Active Duty: 435
    5. Montgomery GI Bill- Selective Reserve: 48
    6. Dependents' Educational Assistance: 118
    7. Reserve Educational Assistance Program: 60
    8. Post-Vietnam Era Veteran's Educational Assistance Program: 1
    9. TOTAL: 768
   10. j: None
```

If a single stat *and* a single column are requested, only a value will be returned:

```
csvstat -c 4 --mean examples/realdata/FY09_EDU_Recipients_by_State.csv

6,263.904
```

To diff CSVs, consider daff. An alternative to csvsql is q.

Common arguments

Arguments common to all tools

All tools which accept CSV as input share a set of common command-line arguments:

```
-d DELIMITER, --delimiter DELIMITER
                      Delimiting character of the input CSV file.
-t, --tabs            Specify that the input CSV file is delimited with
                      tabs. Overrides "-d".
-q QUOTECHAR, --quotechar QUOTECHAR
                      Character used to quote strings in the input CSV file.
-u {0,1,2,3}, --quoting {0,1,2,3}
                      Quoting style used in the input CSV file. 0 = Quote
                      Minimal, 1 = Quote All, 2 = Quote Non-numeric, 3 =
                      Quote None.
-b, --no-doublequote  Whether or not double quotes are doubled in the input
                      CSV file.
-p ESCAPECHAR, --escapechar ESCAPECHAR
                      Character used to escape the delimiter if --quoting 3
                      ("Quote None") is specified and to escape the
                      QUOTECHAR if --no-doublequote is specified.
-z FIELD_SIZE_LIMIT, --maxfieldsize FIELD_SIZE_LIMIT
                      Maximum length of a single field in the input CSV
                      file.
-e ENCODING, --encoding ENCODING
                      Specify the encoding of the input CSV file.
-L LOCALE, --locale LOCALE
                      Specify the locale (en_US) of any formatted numbers.
-S, --skipinitialspace
                      Ignore whitespace immediately following the delimiter.
--blanks              Do not coerce empty, "na", "n/a", "none", "null", "."
                      strings to NULL values.
--date-format DATE_FORMAT
                      Specify a strptime date format string like "%m/%d/%Y".
--datetime-format DATETIME_FORMAT
                      Specify a strptime datetime format string like
                      "%m/%d/%Y %I:%M %p".
-H, --no-header-row   Specify that the input CSV file has no header row.
                      Will create default headers (A,B,C,...).
-L SKIP_LINES, --skip-lines SKIP_LINES
```

```
                      Specify the number of initial lines to skip (e.g.
                      comments, copyright notices, empty rows).
-v, --verbose         Print detailed tracebacks when errors occur.
-l, --linenumbers     Insert a column of line numbers at the front of the
                      output. Useful when piping to grep or as a simple
                      primary key.
--zero                When interpreting or displaying column numbers, use
                      zero-based numbering instead of the default 1-based
                      numbering.
-V, --version         Display version information and exit.
```

These arguments may be used to override csvkit's default "smart" parsing of CSV files. This is frequently necessary if the input file uses a particularly unusual style of quoting or is an encoding that is not compatible with utf-8. Not every command is supported by every tool, but the majority of them are.

Note that the output of csvkit's tools is always formatted with "default" formatting options. This means that when executing multiple csvkit commands (either with a pipe or via intermediary files) it is only ever necessary to specify formatting arguments the first time. (And doing so for subsequent commands will likely cause them to fail.)

Tips and Troubleshooting

Tips

Reading compressed CSVs

csvkit has builtin support for reading `gzip` or `bz2` compressed input files. This is automatically detected based on the file extension. For example:

```
csvstat examples/dummy.csv.gz
csvstat examples/dummy.csv.bz2
```

Please note, the files are decompressed in memory, so this is a convenience, not an optimization.

Reading a CSV with a byte-order mark (BOM)

Set the encoding to `utf-8-sig`, for example:

```
csvcut -e utf-8-sig -c column1 csv-with-bom.csv
```

Specifying STDIN as a file

Most tools use `STDIN` as input if no filename is given, but tools that accept multiple inputs like *csvjoin* and *csvstack* don't. To use `STDIN` as an input to these tools, use `-` as the filename. For example, these three commands produce the same output:

```
csvstat examples/dummy.csv
cat examples/dummy.csv | csvstat
cat examples/dummy.csv | csvstat -
```

`csvstack` can take a filename and `STDIN` as input, for example:

```
cat examples/dummy.csv | csvstack examples/dummy3.csv -
```

Alternately, you can pipe in multiple inputs like so:

```
csvjoin -c id <(csvcut -c 2,5,6 a.csv) <(csvcut -c 1,7 b.csv)
```

Troubleshooting

Installation

csvkit is supported on:

- Python 2.7+
- Python 3.3+
- PyPy

It is tested on OS X, and has also been used on Linux and Windows.

If installing on Ubuntu, you may need to install Python's development headers first:

```
sudo apt-get install python-dev python-pip python-setuptools build-essential
pip install csvkit
```

If the installation is successful but csvkit's tools fail, you may need to update Python's setuptools package first:

```
pip install --upgrade setuptools
pip install --upgrade csvkit
```

On OS X, if you see *OSError: [Errno 1] Operation not permitted*, try:

```
sudo pip install --ignore-installed csvkit
```

If you use Python 2 and have a recent version of pip, you may need to run pip with `--allow-external argparse`.

If you use Python 2 on FreeBSD, you may need to install py-sqlite3.

Note: Need more speed? If you use Python 2, `pip install cdecimal` for a boost.

CSV formatting and parsing

- Are values appearing in incorrect columns?
- Does the output combine multiple fields into a single column with double-quotes?
- Does the outplit split a single field into multiple columns?
- Are *csvstat -c 1* and *csvstat –count* reporting inconsistent row counts?

These may be symptoms of CSV sniffing gone wrong. As there is no single, standard CSV format, csvkit uses Python's csv.Sniffer to deduce the format of a CSV file: that is, the field delimiter and quote character. By default, the entire file is sent for sniffing, which can be slow. You can send a small sample with the `--snifflimit` option. If you're encountering any cases above, you can try setting `--snifflimit 0` to disable sniffing and set the `--delimiter` and `--quotechar` options yourself.

Although these issues are annoying, in most cases, CSV sniffing Just Works™. Disabling sniffing by default would produce a lot more issues than enabling it by default.

CSV data interpretation

- Are the numbers `1` and `0` being interepted as `True` and `False`?
- Are phone numbers changing to integers and losing their leading `+` or `0`?
- Is the Italian comune of "None" being treated as a null value?

These may be symptoms of csvkit's type inference being too aggressive for your data. CSV is a text format, but it may contain text representing numbers, dates, booleans or other types. csvkit attempts to reverse engineer that text into proper data types—a process called "type inference".

For some data, type inference can be error prone. If necessary you can disable it with the To `--no-inference` switch. This will force all columns to be treated as regular text.

Slow performance

csvkit's tools fall into two categories: Those that load an entire CSV into memory (e.g. *csvstat*) and those that only read data one row at a time (e.g. *csvcut*). Those that stream results will generally be very fast. For those that buffer the entire file, the slowest part of that process is typically the "type inference" described in the previous section.

If a tool is too slow to be practical for your data try setting the `--snifflimit` option or using the `--no-inference`.

Database errors

Are you seeing this error message, even after running `pip install psycopg2` or `pip install MySQL-python`?

```
You don't appear to have the necessary database backend installed for connection␣
↪string you're trying to use. Available backends include:

Postgresql: pip install psycopg2
MySQL:      pip install MySQL-python

For details on connection strings and other backends, please see the SQLAlchemy␣
↪documentation on dialects at:

http://www.sqlalchemy.org/docs/dialects/
```

First, make sure that you can open a `python` interpreter and run `import psycopg2`. If you see an error containing `mach-o, but wrong architecture`, you may need to reinstall `psycopg2` with `export ARCHFLAGS="-arch i386" pip install --upgrade psycopg2` (source). If you see another error, you may be able to find a solution on StackOverflow.

Python standard output encoding errors

If, when running a command like `csvlook dummy.csv | less` you get an error like:

```
'ascii' codec can't encode character u'\u0105' in position 2: ordinal not in␣
↪range(128)
```

The simplest option is to set the encoding that Python uses for standard streams, using the PYTHONIOENCODING environment variable:

```
PYTHONIOENCODING=utf8 csvlook dummy.csv | less
```

Contributing to csvkit

csvkit actively encourages contributions from people of all genders, races, ethnicities, ages, creeds, nationalities, persuasions, alignments, sizes, shapes, and journalistic affiliations. You are welcome here.

We seek contributions from developers and non-developers of all skill levels. We will typically accept bug fixes and documentation updates with minimal fuss. If you want to work on a larger feature—great! The maintainers will be happy to provide feedback and code review on your implementation.

Before making any changes or additions to csvkit, please be sure to read the rest of this document, especially the "Principles of development" section.

Getting Started

Set up your environment for development:

```
git clone git://github.com/wireservice/csvkit.git
cd csvkit
mkvirtualenv csvkit

# If running Python 2:
pip install -r requirements-py2.txt

# If running Python 3:
pip install -r requirements-py3.txt

python setup.py develop
tox
```

Principles of development

csvkit is to tables as Unix text processing commands (cut, grep, cat, sort) are to text. As such, csvkit adheres to the Unix philosophy.

1. Small is beautiful.
2. Make each program do one thing well.
3. Build a prototype as soon as possible.
4. Choose portability over efficiency.
5. Store data in flat text files.
6. Use software leverage to your advantage.
7. Use shell scripts to increase leverage and portability.
8. Avoid captive user interfaces.
9. Make every program a filter.

As there is no single, standard CSV format, csvkit encourages popular formatting options:

- Output targets broad compatibility: Quoting is done with double-quotes and only when required. Fields are delimited with commas. Rows are terminated with Unix line endings ("\n").
- Output favors consistency over brevity: Numbers always include at least one decimal place, even if they are round. Dates and times are output in ISO 8601 format. Null values are rendered as empty strings.

How to contribute

1. Fork the project on GitHub.
2. Look through the open issues for a task that you can realistically expect to complete in a few days. Don't worry about the issue's priority; instead, choose something you'll enjoy. You're more likely to finish something if you enjoy hacking on it.
3. Comment on the issue to let people know you're going to work on it so that no one duplicates your effort. It's good practice to provide a general idea of how you plan to resolve the issue so that others can make suggestions.
4. Write tests for any changes to the code's behavior. Follow the format of the tests in the `tests/` directory to see how this works. You can run all the tests with the command `tox`, or just your Python version's with `nosetests` (faster).
5. Write the code. Try to be consistent with the style and organization of the existing code. A good contribution won't be refused for stylistic reasons, but large parts of it may be rewritten and nobody wants that.
6. As you're working, periodically merge in changes from the upstream master branch to avoid having to resolve large merge conflicts. Check that you haven't broken anything by running the tests.
7. Write documentation for any user-facing features.
8. Once it works, is tested, and is documented, submit a pull request on GitHub.
9. Wait for it to be merged or for a comment about what needs to be changed.
10. Rejoice.

A note on new tools

As a general rule, csvkit is no longer adding new tools. This is the result of limited maintenance time as well as a desire to keep the toolkit focused on the most common use cases. Exceptions may be made to this rule on a case-by-case basis. We, of course, welcome patches to improve existing tools or add useful features.

If you decide to build your own tool, we encourage you to create and maintain it as a separate Python package. You will probably want to use the agate library, which csvkit uses for most of its CSV reading and writing. Doing so will safe time and make it easier to maintain common behavior with csvkit's core tools.

Streaming versus buffering

csvkit tools operate in one of two fashions: Some, such as *csvsort*, buffer their entire input into memory before writing any output. Other tools—those that can operate on individual records—write write a row immediately after reading a row. Records are "streamed" through the tool. Streaming tools produce output faster and require less memory than buffering tools.

For performance reasons tools should always offer streaming when possible. If a new feature would undermine streaming functionality it must be balanced against the utility of having a tool that can efficiently operate over large datasets.

Currently, the following tools stream:

- *csvclean*
- *csvcut*
- *csvformat*
- *csvgrep*
- *csvstack*
- *sql2csv*

Currently, the following tools buffer:

- *csvjoin*
- *csvjson* unless `--stream` and `--no-inference` are set but `--skip-lines` is not
- *csvlook*
- *csvpy*
- *csvsort*
- *csvsql*
- *csvstat*
- *in2csv* unless `--format` is `ndjson` and `--no-inference` is set, or `--format` is `csv` and `--no-inference` is set but `--skip-lines` is not

Legalese

To the extent that contributors care, they should keep the following legal mumbo-jumbo in mind:

The source of csvkit and therefore of any contributions are licensed under the permissive MIT license. By submitting a patch or pull request you are agreeing to release your contribution under this license. You will be acknowledged in the AUTHORS file. As the owner of your specific contributions you retain the right to privately relicense your specific contributions (and no others), however, the released version of the code can never be retracted or relicensed.

Release process

1. Verify no high priority issues are outstanding.
2. Run the full test suite with fresh environments for all versions: `tox -r` (Everything MUST pass.)
3. **Ensure these files all have the correct version number:**
    - CHANGELOG
    - setup.py
    - csvkit/cli.py
    - docs/conf.py
4. Tag the release: `git tag -a x.y.z; git push --tags`
5. Roll out to PyPI: `python setup.py sdist upload`
6. Iterate the version number in all files where it is specified. (see list above)

7. Flag the new version for building on Read the Docs.
8. Wait for the documentation build to finish.
9. Flag the new release as the default documentation version.
10. Announce the release on Twitter, etc.

License

Changelog

1.0.2 - April 28, 2017

Improvements:

- Add a `--version` flag.
- Add a `--skip-lines` option to skip initial lines (e.g. comments, copyright notices, empty rows).
- Add a `--locale` option to set the locale of any formatted numbers.
- Add a `--date-format` option to set a strptime date format string.
- Add a `--datetime-format` option to set a strptime datetime format string.
- Make `--blanks` a common argument across all tools.
- `-I` is the short option for `--no-inference`.
- *csvclean*, *csvjson*, *csvpy* support `--no-header-row`.
- *csvclean* is faster and no longer requires exponential time in the worst case.
- *csvjoin* supports `--snifflimit` and `--no-inference`.
- *csvpy* supports `--linenumbers` (no-op) and `--zero` (no-op).
- *csvsql* adds a `--prefix` option to add expressions like OR IGNORE or OR REPLACE following the INSERT keyword.

- *csvsql* adds a `--overwrite` flag to drop any existing table with the same name before creating.
- *csvsql* accepts a file name for the `--query` option.
- *csvsql* supports `--linenumbers` (no-op).
- *csvsql* adds a `--create-if-not-exists` flag to not abort if the table already exists
- *csvstat* adds a `--freq-count` option to set the maximum number of frequent values to display.
- *csvstat* supports `--linenumbers` (no-op).
- *in2csv* adds a `--names` flag to print Excel sheet names.
- *in2csv* adds a `--write-sheets` option to write the named Excel sheets to files.
- *sql2csv* adds an `--encoding` option to specify the encoding of the input query file.

Fixes:

- *csvgrep* no longer ignores common arguments if `--linenumbers` is set.
- *csvjson* supports Decimal.
- *csvpy* again supports IPython.
- *csvsql* restores support for `--no-constraints` and `--db-schema`.
- *csvstat* will no longer crash when `--freq` is set.
- *in2csv* restores support for `--no-inference` for Excel files.
- *in2csv* restores support for converting Excel files from standard input.
- *in2csv* accepts utf-8 arguments to the `--sheet` option in Python 2.

1.0.1 - December 29, 2016

This is a minor release which fixes several bugs reported in the `1.0.0` release earlier this week. It also significantly improves the output of *csvstat* and adds a `--csv` output option to that command.

- *csvstat* will no longer crash when a `Number` column has `None` as a frequent value. (#738)
- *csvlook* docs now note that output tables are Markdown-compatible. (#734)
- *csvstat* now supports a `--csv` flag for tabular output. (#584)
- *csvstat* output is now easier to read. (#714)
- *csvpy* now has a better description when using the `--agate` flag. (#729)
- Fix a Python 2.6 bug preventing *csvjson* from parsing utf-8 files. (#732)
- Update required version of unittest to latest. (#727)

1.0.0 - December 27, 2016

This is the first major release of csvkit in a very long time. The entire backend has been rewritten to leverage the agate data analysis library, which was itself inspired by csvkit. The new backend provides better type detection accuracy, as well as some new features.

Because of the long and complex cycle behind this release, the list of changes should not be considered exhaustive. In particular, the output format of some tools may have changed in small ways. Any existing data pipelines using csvkit should be tested as part of the upgrade.

Much of the credit for this release goes to James McKinney, who has almost single-handedly kept the csvkit fire burning for a year. Thanks, James!

Backwards-incompatible changes:

- *csvjoin* now renames duplicate columns with integer suffixes to prevent collisions in output.
- *csvsql* now generates `DateTime` columns instead of `Time` columns.
- *csvsql* now generates `Decimal` columns instead of `Integer`, `BigInteger`, and `Float` columns.
- *csvsql* no longer generates max-length constraints for text columns.
- The `--doublequote` long flag is gone, and the `-b` short flag is now an alias for `--no-doublequote`.
- When using the `--columns` or `--not-columns` options, you must not have spaces around the comma-separated values, unless the column names contain spaces.
- When sorting, null values are now greater than other values instead of less than.
- `CSVKitReader`, `CSVKitWriter`, `CSVKitDictReader`, and `CSVKitDictWriter` have been removed. Use `agate.csv.reader`, `agate.csv.writer`, `agate.csv.DictReader` and `agate.csv.DictWriter`.
- Dropped support for older versions of PyPy.
- Dropped Python 2.6 support.
- If `--no-header-row` is set, the output will have column names `a`, `b`, `c`, etc. instead of `column1`, `column2`, `column3`, etc.
- csvlook renders a simpler, markdown-compatible table.

Improvements:

- csvkit is now tested against Python 3.6. (#702)
- `import csvkit as csv` will now defer to agate readers/writers.
- *csvgrep* supports `--no-header-row`.
- *csvjoin* supports `--no-header-row`.
- *csvjson* streams input and output if the `--stream` and `--no-inference` flags are set.
- *csvjson* supports `--snifflimit` and `--no-inference`.
- *csvlook* adds `--max-rows`, `--max-columns` and `--max-column-width` options.
- *csvlook* supports `--snifflimit` and `--no-inference`.
- *csvpy* supports `--agate` to read a CSV file into an agate table.
- `csvsql` supports custom SQLAlchemy dialects.
- *csvstat* supports `--names`.
- *in2csv* CSV-to-CSV conversion streams input and output if the `--no-inference` flag is set.
- *in2csv* CSV-to-CSV conversion uses `agate.Table`.
- *in2csv* GeoJSON conversion adds columns for geometry type, longitude and latitude.
- Documentation: Update tool usage, remove shell prompts, document connection string, correct typos.

Fixes:

- Fixed numerous instances of open files not being closed before utilities exit.
- Change `-b`, `--doublequote` to `--no-doublequote`, as doublequote is True by default.

- *in2csv* DBF conversion works with Python 3.
- *in2csv* correctly guesses format when file has an uppercase extension.
- *in2csv* correctly interprets `--no-inference`.
- *in2csv* again supports nested JSON objects (fixes regression).
- *in2csv* with `--format geojson` will print a JSON object instead of `OrderedDict([(...)])`.
- *csvclean* with standard input works on Windows.
- *csvgrep* returns the input file's line numbers if the `--linenumbers` flag is set.
- *csvgrep* can match multiline values.
- *csvgrep* correctly operates on ragged rows.
- *csvsql* correctly escapes `%`` characters in SQL queries.
- *csvsql* adds standard input only if explicitly requested.
- *csvstack* supports stacking a single file.
- *csvstat* always reports frequencies.
- The `any_match` argument of `FilteringCSVReader` now works correctly.
- All tools handle empty files without error.

0.9.1 - March 31, 2015

- Add Antonio Lima to AUTHORS.
- Add support for ndjson. (#329)
- Add missing docs for csvcut -C. (#227)
- Reorganize docs so TOC works better. (#339)
- Render docs locally with RTD theme.
- Fix header in "tricks" docs.
- Add install instructions to tutorial. (#331)
- Add killer examples to doc index. (#328)
- Reorganize doc index
- Fix broken csvkit module documentation. (#327)
- Fix version of openpyxl to work around encoding issue. (#391, #288)

0.9.0

- Write missing sections of the tutorial. (#32)
- Remove -q arg from sql2csv (conflicts with common flag).
- Fix csvjoin in case where left dataset rows without all columns.
- Rewrote tutorial based on LESO data. (#324)
- Don't error in csvjson if lat/lon columns are null. (#326)
- Maintain field order in output of csvjson.

- Add unit test for json in2csv. (#77)
- Maintain key order when converting JSON into CSV. (#325.)
- Upgrade python-dateutil to version 2.2 (#304)
- Fix sorting of columns with null values. (#302)
- Added release documentation.
- Fill out short rows with null values. (#313)
- Fix unicode output for csvlook and csvstat. (#315)
- Add documentation for –zero. (#323)
- Fix Integrity error when inserting zero rows in database with csvsql. (#299)
- Add Michael Mior to AUTHORS. (#305)
- Add –count option to CSVStat.
- Implement csvformat.
- Fix bug causing CSVKitDictWriter to output ‘utf-8’ for blank fields.

0.8.0

- Add pnaimoli to AUTHORS.
- Fix column specification in csvstat. (#236)
- Added “Tips and Tricks” documentation. (#297, #298)
- Add Espartaco Palma to AUTHORS.
- Remove unnecessary enumerate calls. (#292)
- Deprecated DBF support for Python 3+.
- Add support for Python 3.3 and 3.4 (#239)

0.7.3

- Fix date handling with openpyxl > 2.0 (#285)
- Add Kristina Durivage to AUTHORS. (#243)
- Added Richard Low to AUTHORS.
- Support SQL queries “directly” on CSV files. (#276)
- Add Tasneem Raja to AUTHORS.
- Fix off-by-one error in open ended column ranges. (#238)
- Add Matt Pettis to AUTHORS.
- Add line numbers flag to csvlook (#244)
- Only install argparse for Python < 2.7. (#224)
- Add Diego Rabatone Oliveira to AUTHORS.
- Add Ryan Murphy to AUTHORS.
- Fix DBF dependency. (#270)

0.7.2

- Fix CHANGELOG for release.

0.7.1

- Fix homepage url in setup.py.

0.7.0

- Fix XLSX datetime normalization bug. (#223)
- Add raistlin7447 to AUTHORS.
- Merged sql2csv utility (#259).
- Add Jeroen Janssens to AUTHORS.
- Validate csvsql DB connections before parsing CSVs. (#257)
- Clarify install process for Ubuntu. (#249)
- Clarify docs for –escapechar. (#242)
- Make `import csvkit` API compatible with `import csv`.
- Update Travis CI link. (#258)
- Add Sébastien Fievet to AUTHORS.
- Use case-sensitive name for SQLAlchemy (#237)
- Add Travis Swicegood to AUTHORS.

0.6.1

- Add Chris Rosenthal to AUTHORS.
- Fix multi-file input to csvsql. (#193)
- Passing –snifflimit=0 to disable dialect sniffing. (#190)
- Add aarcro to the AUTHORS file.
- Improve performance of csvgrep. (#204)
- Add Matt Dudys to AUTHORS.
- Add support for –skipinitialspace. (#201)
- Add Joakim Lundborg to AUTHORS.
- Add –no-inference option to in2csv and csvsql. (#206)
- Add Federico Scrinzi to AUTHORS file.
- Add –no-header-row to all tools. (#189)
- Fix csvstack blowing up on empty files. (#209)
- Add Chris Rosenthal to AUTHORS file.
- Add –db-schema option to csvsql. (#216)

- Add Shane StClair to AUTHORS file.
- Add –no-inference support to csvsort. (#222)

0.5.0

- Implement geojson support in csvjson. (#159)
- Optimize writing of eight bit codecs. (#175)
- Created csvpy. (#44)
- Support –not-columns for excluding columns. (#137)
- Add Jan Schulz to AUTHORS file.
- Add Windows scripts. (#111, #176)
- csvjoin, csvsql and csvstack will no longer hold open all files. (#178)
- Added Noah Hoffman to AUTHORS.
- Make csvlook output compatible with emacs table markup. (#174)

0.4.4

- Add Derek Wilson to AUTHORS.
- Add Kevin Schaul to AUTHORS.
- Add DBF support to in2csv. (#11, #160)
- Support –zero option for zero-based column indexing. (#144)
- Support mixing nulls and blanks in string columns.
- Add –blanks option to csvsql. (#149)
- Add multi-file (glob) support to csvsql. (#146)
- Add Gregory Temchenko to AUTHORS.
- Add –no-create option to csvsql. (#148)
- Add Anton Ian Sipos to AUTHORS.
- Fix broken pipe errors. (#150)

0.4.3

- Begin CHANGELOG (a bit late, I'll admit).

CHAPTER 4

Citation

When citing csvkit in publications, you may use this BibTeX entry:

```
@Manual{,
  title = {csvkit},
  author = {Christopher Groskopf and contributors},
  year = 2016,
  url = {https://csvkit.readthedocs.org/}
}
```

CHAPTER 5

Authors

The following individuals have contributed code to csvkit:

- Christopher Groskopf
- Joe Germuska
- Aaron Bycoffe
- Travis Mehlinger
- Alejandro Companioni
- Benjamin Wilson
- Bryan Silverthorn
- Evan Wheeler
- Matt Bone
- Ryan Pitts
- Hari Dara
- Jeff Larson
- Jim Thaxton
- Miguel Gonzalez
- Anton Ian Sipos
- Gregory Temchenko
- Kevin Schaul
- Marc Abramowitz
- Noah Hoffman
- Jan Schulz
- Derek Wilson

- Chris Rosenthal
- Davide Setti
- Gabi Davar
- Sriram Karra
- James McKinney
- Aaron McMillin
- Matt Dudys
- Joakim Lundborg
- Federico Scrinzi
- Chris Rosenthal
- Shane StClair
- raistlin7447
- Alex Dergachev
- Jeff Paine
- Jeroen Janssens
- Sébastien Fievet
- Travis Swicegood
- Ryan Murphy
- Diego Rabatone Oliveira
- Matt Pettis
- Tasneem Raja
- Richard Low
- Kristina Durivage
- Espartaco Palma
- pnaimoli
- Michael Mior
- Jennifer Smith
- Antonio Lima
- Dave Stanton
- Pedrow
- Neal McBurnett
- Anthony DeBarros
- Baptiste Mispelon
- James Seppi
- Karrie Kehoe
- Geert Barentsen

- Cathy Deng
- Eric Bréchemier
- Neil Freeman
- Fede Isas
- Patricia Lipp
- Kev++
- edwardros
- Martin Burch
- Pedro Silva
- hydrosIII
- Tim Wisniewski
- Santiago Castro
- Dan Davison
- Éric Araujo
- Sam Stuck
- Edward Betts
- Jake Zimmerman

CHAPTER 6

Indices and tables

- genindex
- modindex
- search