# csvkit Documentation

***Release 0.5.0 (beta)***

**Christopher Groskopf**

June 28, 2013

# CONTENTS

CHAPTER

ONE

# ABOUT

csvkit is a suite of utilities for converting to and working with CSV, the king of tabular file formats.

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

Important links:

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

CHAPTER

TWO

# PRINCIPLES

csvkit is to tabular data what the standard Unix text processing suite (grep, sed, cut, sort) is 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 formally defined CSV format, csvkit encourages well-known formatting standards:

- Output favors compatability with the widest range of applications. This means that quoting is done with double-quotes and only when necessary, columns are separated with commas, and lines are terminated with unix style line endings (“\n”).
- Data that is modified or generated will prefer consistency over brevity. Floats always include at least one decimal place, even if they are round. Dates and times are written in ISO8601 format.

CHAPTER

THREE

# INSTALLATION

For users:

```
pip install csvkit
```

For developers:

```
git clone git://github.com/onyxfish/csvkit.git
cd csvkit
mkvirtualenv --no-site-packages csvkit
pip install -r requirements.txt
nosetests
```

**Note:** csvkit is routinely tested on OSX, somewhat less frequently on Linux and once in a while on Windows. All platforms are supported. It is tested against Python 2.6, 2.7 and PyPy. Neither Python < 2.6 nor Python >= 3.0 are supported at this time.

CHAPTER

FOUR

# TUTORIAL

The csvkit tutorial walks through processing and analyzing a real dataset from data.gov. It is divided into several parts for easier reading:

## 4.1 Getting started

### 4.1.1 Description

There is no better way to learn how to use a new tool than to see it applied in a real world situation. This tutorial will explain the workings of most of the csvkit utilities (including some nifty tricks) in the context of analyzing a real dataset from data.gov.

The dataset that I've chosen to work with is recipients of United States Department of Veteran Affairs education benefits, by state and year. For those who are unfamiliar, these are individuals whom the US government is paying to attend school as a benefit of their having served in the military (or, in some case, having had a close relative who served). We will be working with 2009 and 2010 data.

If you have never done much data processing work this tutorial should double as a reasonable introduction to that as well.

### 4.1.2 Following along

I strongly encourage anyone reading this tutorial to work through the examples, however, if you really just want to see how of the tools are applied then you can read through the complete bash script for the entire tutorial.

### 4.1.3 Getting the data

Let's start by creating a clean workspace:

```
$ mkdir va_benefits
$ cd va_benefits
```

Now let's fetch the 2009 data file:

```
$ wget -O 2009.csv http://www.data.gov/download/4029/csv
```

At first glance this may appear to have worked. You will end up with a `2009.csv` file in your working directory. However, when I said this tutorial would tackle real-world problems, I meant it. Let's take a look at the contents of that file:

```
$ cat 2009.csv
<html><head><title>Request Rejected</title></head><body>The requested URL was rejected. <br><br>Pleas
```

It seems data.gov is redirecting our request to the VA's website and they don't like people fetching down files from the command-line. Fortunantely for us, this is a terribly naive thing to do, and easy to work around.

Let's pretend our request is coming from Google Chrome instead of wget:

```
$ wget -O 2009.csv -U "Mozilla/5.0 (X11; U; Linux x86_64; en-US) AppleWebKit/534.16 (KHTML, like Geck
```

And use the Unix text-processing utility `head` to check the first five lines and make sure the file looks right this time:

```
$ head -n 5 2009.csv
State Name,State Abbreviate,Code,Montgomery GI Bill-Active Duty,Montgomery GI Bill- Selective Reserve
ALABAMA,AL,01,"6,718","1,728","2,703","1,269",8,"12,426",
ALASKA,AK,02,776,154,166,60,2,"1,158",
ARIZONA,AZ,04,"26,822","2,005","3,137","2,011",11,"33,986",
ARKANSAS,AR,05,"2,061",988,"1,575",886,3,"5,513",
```

That looks better so let's fetch the 2010 data using the same trick:

```
$ wget -O 2010.csv -U "Mozilla/5.0 (X11; U; Linux x86_64; en-US) AppleWebKit/534.16 (KHTML, like Geck
```

If you're working from a copy of the csvkit source code, you can also find these files in the `examples/realdata` folder with their default names, `FY09_EDU_Recipients_by_State.csv` and `Datagov_FY10_EDU_recp_by_State.csv`, but getting them this way was a lot more fun, right?

### 4.1.4 Fixing the files with sed

Nothing is ever easy when you're working with government data. We've got one more problem before we can get down to brass tacks and start hacking these files. The first file looked fine, but let's check out the `head` of that second file:

```
$ head -n 5 2010.csv
,,,,,,,,
Number of Beneficaries (Students) Who Recieved VA Education Benefit By State During FY 2010,,,,,,,,
State Name,State Abbreviate,Post-9/11GI Bill Program,Montgomery GI Bill - Active Duty,Montgomery GI B
ALABAMA,AL,7738,5779,2075,3102,883,5,"19,582"
ALASKA,AK,1781,561,170,164,28,1,"2,705"
```

As you can see, this multiple header lines. We need to modify this file to fix the issue, but as a matter of best practice let's backup our originals first:

```
$ cp 2009.csv 2009_original.csv
$ cp 2010.csv 2010_original.csv
```

With that done let's use the old hacker standby `sed` to kill those first two header lines:

```
$ cat 2010_original.csv | sed "1,2d" > 2010.csv
```

`sed` is an abbreviation for "stream editor", which is a useful phrase to keep in mind if you're not used to working with these tools. csvkit and the other tools introduced in the next chapter all operate on streams of data, processing them one line at a time. The `sed` command we just used translates to, "Select lines 1 and 2 of the input, (d)elete them."

The previous command also introduces a couple other concepts that are much more important than the indiosyncrancies of `sed`.

### 4.1.5 Piping

The first is piping. If you haven't spent too much time in the terminal you may be unfamiliar with the `|` (pipe). The pipe means, "take the output of the former command and use it as input to the latter." So in this case the output of `cat` (which simply prints a file) is being piped into `sed`.

### 4.1.6 Output redirection

The second interesting thing is output redirection. The > character means, send output from this command to a file. If I had used >> it would have been appended to the end of the file rather than overwriting it. This is important because by default `sed` will simply send its output to the console, which is great for piping, but not very useful if you want to save your results.

### 4.1.7 Putting it together

All the csvkit utilities work with the concepts of piping and output redirection. The output of any utility can be piped into another and into another and then at some point down the road 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*.

## 4.2 Examining the data

### 4.2.1 Cutting up the data with csvcut

Now let's start investigating this dataset. The first thing we might want to know is what columns are included. To figure that out let's use *csvcut*:

```
$ csvcut -n 2009.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:
```

`csvcut` is the beating heart of csvkit and was the original inspiration for building the rest of the tools. However, in this case we aren't using its full power. When the -n flag is specified, `csvcut` simply prints the column numbers and names and exits. We'll see the rest of its capabilities shortly.

This tells us a few things about the dataset, including that the final column doesn't have a name. However, it doesn't tell us anything about what sort of data is in each column. Let's try peeking at a couple columns to try to figure that out:

```
$ csvcut -c 2,3 2009.csv | head -n 5
State Name,Code
AL,01
AK,02
```

```
AZ,04
AR,05
```

Now we see the power of `csvcut`. Using the -c flag we are able to specify columns we want to extract from the input. We also use Unix utility `head` to look at just the first five rows of output. From the look of things the “Code” column is a identifier that is unique to each state. It also appears that the dataset is alphabetical by state.

### 4.2.2 Statistics on demand with csvstat

Another utility included with csvkit is *csvstat*, which mimics the summary() function from the computational statistics programming language “R”.

Let’s use what we learned in the previous section to look at just a slice of the data:

```
$ csvcut -c 1,4,9,10 2009.csv | csvstat
  1. State Name
    <type 'unicode'>
    Nulls: Yes
    Unique values: 53
    Max length: 17
    Samples: "FLORIDA", "IDAHO", "ARIZONA", "OHIO", "IOWA"
  2. Montgomery GI Bill-Active Duty
    <type 'int'>
    Nulls: Yes
    Min: 435
    Max: 34942
    Mean: 6263
    Median: 3548.0
    Unique values: 53
  3. TOTAL
    <type 'int'>
    Nulls: Yes
    Min: 768
    Max: 46897
    Mean: 9748
    Median: 6520.0
    Unique values: 53
  4.
        Empty column
```

Like `csvcut`, `csvstat` lists columns with their numbers (in this case the column numbers are those of the CSV file *output* by `csvcut`). However, `csvstat` also performs type inference on the columns and computes relevant statistics based on their types. In this case we have a *unicode* column (internally csvkit uses unicode exclusively to represent text), and two *int* (integer) columns. For the unicode column we know it contains nulls (blanks), that it has 53 unique values, that the longest value is 17 characters, and we also have five examples of data from that column.

From the statistics on the integer columns we can see that the median number of indviduals exercising VA benefits across the states is 6520, of which 3548 is the median number exercising the GI Bill while on active duty.

We also see that the final column of the original CSV not only lacks a header, but is entirely empty. (Which is the same thing as saying that every row in this file included a trailing comma.)

If this dataset had included a column of dates or times, `csvstat` would have displayed the range and other details relevant to time-sequences.

### 4.2.3 Searching for rows with csvgrep

After reviewing the summary statistics you might wonder where your home state falls in the order. 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:

```
$ csvcut -c 1,"TOTAL" 2009.csv | csvgrep -c 1 -m ILLINOIS
State Name,TOTAL
ILLINOIS,"21,964"
```

In this case we are searching for the value "ILLINOIS" in the first column of the input. We can also build a more-powerful and less-verbose search by using the regular expressions flag:

```
$ csvcut -c 1,"TOTAL" 2009.csv | csvgrep -c 1 -r "^I"
State Name,TOTAL
ILLINOIS,"21,964"
```

Here we have found all the states that start with the letter "I".

What if we wanted to know where Illinois ranks amongst the states with individuals claiming VA benefits? In order to answer that we need to learn a few more tricks.

### 4.2.4 Flipping column order with csvcut

*(Note: In the next few sections we will repeat some commands to show how you can build up a complex operation as a sequence of simple ones.)*

Returning for a moment to *csvcut*, we can use its column selection logic as a powertool for reordering columns. Let's pare back the number of columns and make the column we are most interested in be first:

```
$ csvcut -c 9,1 2009.csv | head -n 5
TOTAL,State Name
12426,ALABAMA
1158,ALASKA
33986,ARIZONA
5513,ARKANSAS
```

### 4.2.5 Sorting with csvsort

Now we can use *csvsort* to sort the rows by the first column:

```
$ csvcut -c 9,1 2009.csv | csvsort -r | head -n 5
TOTAL,State Name
46897,CALIFORNIA
40402,TEXAS
36394,FLORIDA
33986,ARIZONA
```

The -r tells `csvsort` to sort in descending order.

We can now see that Illinois ranks fifth for individuals claiming VA benefits, behind mostly larger states, although Arizona is a surprising name to appear in the top five, given its relative size.

This works well for finding Illinois' rank as its in the top five, but if it had been further down the list we would have had to count rows to determine its rank. That's inefficient and there is a better way.

### 4.2.6 Using line numbers as proxy for rank

The `-l` flag is a special flag that can be passed to any csvkit utility in order to add a column of line numbers to its output. Since this data is being sorted we can use those line numbers as a proxy for rank:

```
$ csvcut -c 9,1 2009.csv | csvsort -r -l | head -n 11
line_number,TOTAL,State Name
1,46897,CALIFORNIA
2,40402,TEXAS
3,36394,FLORIDA
4,33986,ARIZONA
5,21964,ILLINOIS
6,20541,VIRGINIA
7,18236,GEORGIA
8,15730,NORTH CAROLINA
9,13967,NEW YORK
10,13962,MISSOURI
```

Missouri had the tenth largest population of individuals claiming veterans education benefits.

If we were to join this data up with a table of state population's we could see how much of an outlier state's like Arizona and Missouri are. In future sections we'll present tools for doing just that, however, this specific question is left as an experiment for the reader.

### 4.2.7 Reading through data with csvlook and less

You may notice in the previous output that starting on line ten the total numbers cease to line up correctly. This problem would be worse if we hadn't reordered the columns to put the number first. For this reason CSV is often somewhat difficult to work with in the terminal. To mitigate this problem we can use *csvlook* to display the data in a fixed-width table:

```
$ csvcut -c 9,1 2009.csv | csvsort -r -l | csvlook
---------------------------------------------
|  line_number | TOTAL | State Name         |
---------------------------------------------
|  1           | 46897 | CALIFORNIA         |
|  2           | 40402 | TEXAS              |
|  3           | 36394 | FLORIDA            |
|  4           | 33986 | ARIZONA            |
|  5           | 21964 | ILLINOIS           |
|  6           | 20541 | VIRGINIA           |
|  7           | 18236 | GEORGIA            |
|  8           | 15730 | NORTH CAROLINA     |
|  9           | 13967 | NEW YORK           |
|  10          | 13962 | MISSOURI           |
[...]
```

*Hint: If your table doesn't render like this one, try making you terminal window wider.*

Isn't that better? You may still find it annoying it to have the entire contents of the table get printed to your terminal window. To better manage the output try piping it to the unix utility `less` or, if you're just glancing at it, `more`.

### 4.2.8 Saving your work

The complete ranking might be a useful thing to have around. Rather than computing it every time, let's use output redirection to save a copy of it:

```
$ csvcut -c 9,1 2009.csv | csvsort -r -l > 2009_ranking.csv
```

### 4.2.9 Onward to merging

At this point you should be comfortable with the analytical capabilities of csvkit.

Next up: *Adding another year of data*.

## 4.3 Adding another year of data

Coming soon..

## 4.4 Wrapping up

Coming soon...

CHAPTER

# FIVE

# USAGE

csvkit is comprised of a number of individual command line utilities that be loosely divided into a few major categories: Input, Processing, and Output. Documentation and examples for each utility are described on the following pages.

*Input*

## 5.1 in2csv

### 5.1.1 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] [-e ENCODING] [-f FORMAT] [-s SCHEMA]
              [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 FORMAT, --format FORMAT
                        The format of the input file. If not specified will be
                        inferred from the file type. Supported formats: csv,
                        dbf, fixed, geojson, json, xls, xlsx.
  -s SCHEMA, --schema SCHEMA
                        Specifies a CSV-formatted schema file for converting
                        fixed-width files. See documentation for details.
  -k KEY, --key KEY     Specifies a top-level key to use look within for a
                        list of objects to be converted when processing JSON.
  -y SNIFFLIMIT, --snifflimit SNIFFLIMIT
                        Limit CSV dialect sniffing to the specified number of
                        bytes.
  --sheet SHEET         The name of the XLSX sheet to operate on.
```

Also see: *Arguments common to all utilities*.

### 5.1.2 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/r
```

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

https://github.com/onyxfish/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/onyxfish/csvkit/issues?state=open | in2csv -f json -v > issues.cs
```

Convert a DBase DBF file to an equivalent CSV:

```
$ in2csv examples/testdbf.dbf > testdbf_converted.csv
```

Fetch the ten most recent robberies in Oakland, convert the GeoJSON response into a CSV and write it to a file:

```
$ curl "http://oakland.crimespotting.org/crime-data?format=json&type=robbery&count=10" | in2csv -f ge
```

*Processing*

## 5.2 csvclean

### 5.2.1 Description

Cleans a CSV file of common syntax errors. 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] [-e ENCODING] [-n]
                [FILE]

Fix common syntax 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         If this argument is present, no output will be
                        created. Information about what would have been done
                        will be printed to STDERR.
```

Also see: *Arguments common to all utilities*.

### 5.2.2 Examples

Test a file with known bad rows:

```
$ csvclean -n examples/bad.csv

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

## 5.3 csvcut

### 5.3.1 Description

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

```
usage: csvcut [-h] [-d DELIMITER] [-t] [-q QUOTECHAR] [-u {0,1,2,3}] [-b]
              [-p` ESCAPECHAR] [-e ENCODING] [-n] [-c COLUMNS] [-s] [-l]
              [FILE]

Filter and truncate CSV files. Like 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.
  -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.
```

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

Also see: *Arguments common to all utilities*.

### 5.3.2 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
```

## 5.4 csvgrep

### 5.4.1 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] [-e ENCODING] [-l] [-n] [-c COLUMNS] [-r]
               [FILE] [PATTERN]

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.
```

Also see: *Arguments common to all utilities*.

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

### 5.4.2 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
```

## 5.5 csvjoin

### 5.5.1 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-seperated 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] [-e ENCODING] [-j JOIN] [--outer] [--left]
               [--right]
               FILES [FILES ...]

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

positional arguments:
  FILES                 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.
```

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

Also see: *Arguments common to all utilities*.

### 5.5.2 Examples

Join examples coming soon...

## 5.6 csvsort

### 5.6.1 Description

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

```
usage: csvsort [-h] [-d DELIMITER] [-t] [-q QUOTECHAR] [-u {0,1,2,3}] [-b]
               [-p` ESCAPECHAR] [-e ENCODING] [-n] [-c COLUMNS] [-r]
               [FILE]

Sort CSV files. Like 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
  -y SNIFFLIMIT, --snifflimit SNIFFLIMIT
                        Limit CSV dialect sniffing to the specified number of
                        bytes.
                        Specify the encoding the input file.
  -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.
  -r, --reverse         Sort in descending order.
```

Also see: *Arguments common to all utilities*.

### 5.6.2 Examples

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

```
$ cat examples/realdata/FY09_EDU_Recipients_by_State.csv | csvsort -c 9
```

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

```
$ cat examples/realdata/FY09_EDU_Recipients_by_State.csv | csvcut -c 1,9 | csvsort -r -c 2 | head -n
```

## 5.7 csvstack

### 5.7.1 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] [-e ENCODING] [-g GROUPS] [-n GROUP_NAME]
                FILES [FILES ...]

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

positional arguments:
  FILES

optional arguments:
  -h, --help            show this help message and exit
  -g GROUPS, --groups GROUPS
                        A comma-seperated 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.
```

Also see: *Arguments common to all utilities*.

### 5.7.2 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_
```

## 5.8 csvpys

### 5.8.1 Description

Run arbitrary python expression on each CSV row to compute value of a new column. It is possible to compute many new columns at once using `-s` option multiple times:

```
usage: csvpys [-h] [-d DELIMITER] [-t] [-q QUOTECHAR] [-u {0,1,2,3}] [-b]
          [-p ESCAPECHAR] [-z MAXFIELDSIZE] [-e ENCODING] [-v] [-l]
          [--zero] [-n] [-s NEW_COLUMN PY_SCRIPT]
          [FILE]

Python scripting in CSV files. Run arbitrary python expression on each CSV row
to compute value of a new column.

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

```
                        input on STDIN.

optional arguments:

  -n, --names           Display column names and indices from the input CSV
                        and exit.
  -s NEW_COLUMN PY_SCRIPT, --script NEW_COLUMN PY_SCRIPT
                        New column name and a python script.". Inside script
                        use c[0] or ch['header'] locals.
```

Also see: *Arguments common to all utilities*.

### 5.8.2 Scripting language

There is no custom scripting language. Just use plain python expressions. Internally expressions are evaluated using python standard `eval()` function (they are also precompiled for efficiency). Python expression can use following locals and globals:

- **Locals references two variables representing current row:**
  - `c` - list of column values indexed with column id (default: 1-based indexing, with `--zero` enabled indexing is 0-based),
  - `ch` - dict with values accessible via columns name.
- **Globals are supplied with following modules:**
  - re and Rex (rex is regular expressions for humans, specially written by experiences using csvkit scripting),
  - math,
  - random,
  - collections.

**Note:** Remember that both `c` and `ch` variables have unicode type. Need to be converted if used in other contexts than strings.

### 5.8.3 Examples

For following data:

```
$ csvcut -c 1,4,5 examples/realdata/FY09_EDU_Recipients_by_State.csv | csvlook
|--------------------+-------------------------------+----------------------------------------|
|  State Name        | Montgomery GI Bill-Active Duty | Montgomery GI Bill- Selective Reserve  |
|--------------------+-------------------------------+----------------------------------------|
|  ALABAMA           | 6,718                          | 1,728                                  |
|  ALASKA            | 776                            | 154                                    |
|  ARIZONA           | 26,822                         | 2,005                                  |
|  ARKANSAS          | 2,061                          | 988                                    |
|  CALIFORNIA        | 34,942                         | 2,987                                  |
|  COLORADO          | 10,389                         | 914                                    |
|  CONNECTICUT       | 1,771                          | 490                                    |

    ...                ...                              ...
```

```
|  WYOMING           | 686                              | 212                                       |
|  PUERTO RICO       | 822                              | 1,107                                     |
|                    |                                  |                                           |
|--------------------+----------------------------------+-------------------------------------------|
```

Let's sum values for both Montgomery GI Bill:

```
$ csvcut -c 1,4,5 examples/realdata/FY09_EDU_Recipients_by_State.csv | csvpys -s TOTAL "int(c[2].repl
|--------------------+----------------------------------+-------------------------------------------+------
|  State Name        | Montgomery GI Bill-Active Duty | Montgomery GI Bill- Selective Reserve | TOTAL
|--------------------+----------------------------------+-------------------------------------------+------
|  ALABAMA           | 6,718                            | 1,728                                     | 8446
|  ALASKA            | 776                              | 154                                       | 930
|  ARIZONA           | 26,822                           | 2,005                                     | 28827
|  ARKANSAS          | 2,061                            | 988                                       | 3049
|  CALIFORNIA        | 34,942                           | 2,987                                     | 37929
|  COLORADO          | 10,389                           | 914                                       | 11303
|  CONNECTICUT       | 1,771                            | 490                                       | 2261

    ...                 ...                                ...

|  WYOMING           | 686                              | 212                                       | 898
|  PUERTO RICO       | 822                              | 1,107                                     | 1929
|                    |                                  |                                           | 0
|--------------------+----------------------------------+-------------------------------------------+------
```

**Note:** Expression `int(c[2].replace(',', '') if c[2] != '' else 0)` converts a string e.g. `"4,156"` or empty string to a proper int value (`4156`), by removing semicolon and casting to int or returning 0 on empty string.

The same using column names:

```
$ csvcut -c 1,4,5 examples/realdata/FY09_EDU_Recipients_by_State.csv | csvpys -s TOTAL "int(ch['Montg
```

Other example, let's play with data:

```
$ csvcut -c 1,8 examples/realdata/FY09_EDU_Recipients_by_State.csv | csvlook
|--------------------+------------------------------------------------------------|
|  State Name        | Post-Vietnam Era Veteran's Educational Assistance Program  |
|--------------------+------------------------------------------------------------|
|  ALABAMA           | 8                                                          |
|  ALASKA            | 2                                                          |
|  ARIZONA           | 11                                                         |
|  ARKANSAS          | 3                                                          |
|  CALIFORNIA        | 48                                                         |
|  COLORADO          | 10                                                         |
|  CONNECTICUT       | 4                                                          |

    ...                 ...

|  WYOMING           | 1                                                          |
|  PUERTO RICO       | 3                                                          |
|                    |                                                            |
|--------------------+------------------------------------------------------------|
```

The task is to classify as True all states that have a value greater or equal than 10 in second column:

```
$ csvcut -c 1,8 examples/realdata/FY09_EDU_Recipients_by_State.csv | csvpys -s Classify "bool(int(c[2
|--------------------+-----------------------------------------------------------+-----------|
|  State Name        | Post-Vietnam Era Veteran’s Educational Assistance Program | Classify  |
|--------------------+-----------------------------------------------------------+-----------|
|  ALABAMA           | 8                                                         | False     |
|  ALASKA            | 2                                                         | False     |
|  ARIZONA           | 11                                                        | True      |
|  ARKANSAS          | 3                                                         | False     |
|  CALIFORNIA        | 48                                                        | True      |
|  COLORADO          | 10                                                        | True      |
|  CONNECTICUT       | 4                                                         | False     |

    ...                 ...                                                         ...

|  WYOMING           | 1                                                         | False     |
|  PUERTO RICO       | 3                                                         | False     |
|                    |                                                           |           |
|--------------------+-----------------------------------------------------------+-----------|
```

**Note:** If statement is only needed because we need to deal with last line which has empty string ″.

OK, within the last example we will calculate number of A’s in state names:

```
$ csvcut -c 1 examples/realdata/FY09_EDU_Recipients_by_State.csv | csvpys -s "A letter count" "collec
|--------------------+-----------------|
|  State Name        | A letter count  |
|--------------------+-----------------|
|  ALABAMA           | 4               |
|  ALASKA            | 3               |
|  ARIZONA           | 2               |
|  ARKANSAS          | 3               |
|  CALIFORNIA        | 2               |
|  COLORADO          | 1               |
|  CONNECTICUT       | 0               |

    ...                   ...

|  WYOMING           | 0               |
|  PUERTO RICO       | 0               |
|                    | 0               |
|--------------------+-----------------|
```

Regular expressions can also be very useful in scripting. You can use very simple module named Rex, because standard python re implementation is not very continence to use in online expressions. For more information on Rex please refer to documentation.

Let’s use already nice regular expressions support of csvkit, and grep columns to leave only states with started with “NEW”:

```
$ csvcut -c 1-3 examples/realdata/FY09_EDU_Recipients_by_State.csv | csvgrep -c 1 -r "^NEW" | csvlook
|----------------+------------------+-------|
|  State Name    | State Abbreviate | Code  |
|----------------+------------------+-------|
|  NEW HAMPSHIRE | NH               | 33    |
|  NEW JERSEY    | NJ               | 34    |
|  NEW MEXICO    | NM               | 35    |
|  NEW YORK      | NY               | 36    |
|----------------+------------------+-------|
```

Let's say we would like to extract the second part of the state name and concatenate it with the state abbreviate (e.g. for `NEW YORK` => `YORK (NY)`). Of course it could be done in plenty of ways, but we would like to use regular expressions for that:

```
$ csvcut -c 1-3 examples/realdata/FY09_EDU_Recipients_by_State.csv | csvgrep -c 1 -r "^NEW" | csvpys
|----------------+------------------+------+-----------------|
|  State Name    | State Abbreviate | Code | New name        |
|----------------+------------------+------+-----------------|
|  NEW HAMPSHIRE | NH               | 33   | HAMPSHIRE (NH)  |
|  NEW JERSEY    | NJ               | 34   | JERSEY (NJ)     |
|  NEW MEXICO    | NM               | 35   | MEXICO (NM)     |
|  NEW YORK      | NY               | 36   | YORK (NY)       |
|----------------+------------------+------+-----------------|
```

Yes, it's that simple and powerful!

## 5.9 csvgroup

### 5.9.1 Description

Perform a SQL-like group by operation on a CSV file on a specified column or columns. Operation results in printing columns that were aggregation key (it will be distinct in every row) and appending all columns from aggregation functions values:

```
usage: csvgroup [-h] [-d DELIMITER] [-t] [-q QUOTECHAR] [-u {0,1,2,3}] [-b]
            [-p ESCAPECHAR] [-z MAXFIELDSIZE] [-e ENCODING] [-v] [-l]
            [--zero] [-c COLUMNS] [-a FUNCTION COLUMNS] [-n]
            [FILE]

Execute a SQL-like group by on specified column or columns

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

optional arguments:
  -c COLUMNS, --columns COLUMNS
                        The column name(s) on which to group by. Should be
                        either one name (or index) or a comma-separated list.
                        May also be left unspecified, in which case none
                        columns will be used
  -a FUNCTION COLUMNS, --aggregation FUNCTION COLUMNS
                        Aggregate column values using max function
  -n, --names           Display column names and indices from the input CSV
                        and exit.
```

Also see: *Arguments common to all utilities*.

### 5.9.2 Aggregation functions

- max
- min

- sum
- count - count every row
- countA - count non zero rows
- common - returns most common value (handy with text fields aggregation)

### 5.9.3 Examples

Having a data:

```
$ csvlook examples/test_group.csv
|-----+----+----+----+----+-----|
|  h1 | h2 | h3 | h4 | h5 | h6  |
|-----+----+----+----+----+-----|
|  a  | a  | b  | 1  | 2  | 3   |
|  c  | b  | b  | 0  | 0  | 0   |
|  d  | b  | b  | 6  | 7  | 1   |
|  b  | a  | b  | 3  | 2  | 1   |
|-----+----+----+----+----+-----|
```

**Warning:** Before grouping by any number of columns (see `-c` option) you should always sort data using the same columns identifiers ( e.g. use `csvsort -c 1,2,3 | csvgroup -c 1,2,3`). csvgroup pre assumes that data are already sorted using aggregation key. This is very similar to Linux `sort` and `uniq` idiom.
Sorting is however not necessary for grouping without specifying *-c* parameter (without aggregation key).

Lets aggregate by column h2 using min, max and count of columns h4, h5, h6:

```
$ csvsort -c h2 examples/test_group.csv | csvgroup -c h2 -a min h4 -a max h5 -a count h6 | csvlook
|-----+---------+---------+------------|
|  h2 | min(h4) | max(h5) | count(h6)  |
|-----+---------+---------+------------|
|  a  | 1       | 2       | 2          |
|  b  | 0       | 7       | 2          |
|-----+---------+---------+------------|
```

We can also define many columns for one aggregate:

```
$ csvsort -c h2 examples/test_group.csv | csvgroup -c h2 -a max 4-6 | csvlook
|-----+---------+---------+----------|
|  h2 | max(h4) | max(h5) | max(h6)  |
|-----+---------+---------+----------|
|  a  | 3       | 2       | 3        |
|  b  | 6       | 7       | 1        |
|-----+---------+---------+----------|
```

Aggregating by two columns:

```
$ csvsort -c h2,h3 examples/test_group.csv | csvgroup -c h2,h3 -a max 4-6 | csvlook
|-----+----+---------+---------+----------|
|  h2 | h3 | max(h4) | max(h5) | max(h6)  |
|-----+----+---------+---------+----------|
|  a  | b  | 3       | 2       | 3        |
|  b  | b  | 6       | 7       | 1        |
|-----+----+---------+---------+----------|
```

And by all rows:

```
$ csvgroup -a max 4-6 examples/test_group.csv | csvlook
|----------+---------+----------|
|  max(h4) | max(h5) | max(h6)  |
|----------+---------+----------|
|  6       | 7       | 3        |
|----------+---------+----------|
```

**Note:** Notice that aggregation by all rows does not require any kind of sorting, because every row is treated as unique.

Get most common value of every column:

```
$ csvgroup -a common 1-6 examples/test_group.csv | csvlook
|-----+----+----+----+----+-----|
|  h1 | h2 | h3 | h4 | h5 | h6  |
|-----+----+----+----+----+-----|
|  a  | a  | b  | 1  | 2  | 1   |
|-----+----+----+----+----+-----|
```

*Output (and Analysis)*

## 5.10 csvjson

### 5.10.1 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 MAXFIELDSIZE] [-e ENCODING] [-v] [-l]
               [--zero] [-i INDENT] [-k KEY] [--lat LAT] [--lon LON]
               [--crs CRS]
               [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:
  -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.
```

Also see: *Arguments common to all utilities*.

### 5.10.2 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":
    {
        "": "",
         "Code": "53",
         "Reserve Educational Assistance Program": "549",
         "Dependents' Educational Assistance": "2,192",
         "Montgomery GI Bill-Active Duty": "7,969",
         "State Name": "WASHINGTON",
         "Montgomery GI Bill- Selective Reserve": "769",
         "State Abbreviate": "WA",
         "Post-Vietnam Era Veteran's Educational Assistance Program": "13",
         "TOTAL": "11,492"
    },
    [...]
}
```

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": [
        {
            "geometry": {
                "type": "Point",
                "coordinates": [
                    -95.30181,
                    32.35066
                ]
            },
            "type": "Feature",
            "id": "dcl",
            "properties": {
                "photo_credit": "",
                "description": "In addition to being the only coffee shop in downtown Tyler, DCL also
                "artist": "",
                "title": "Downtown Coffee Lounge",
                "install_date": "",
                "address": "200 West Erwin Street",
                "last_seen_date": "3/30/12",
                "type": "Gallery",
                "photo_url": ""
            }
        },
    [...]
    ]
}
```

## 5.11 csvlook

### 5.11.1 Description

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

```
usage: csvlook [-h] [-d DELIMITER] [-t] [-q QUOTECHAR] [-u {0,1,2,3}] [-b]
               [-p` ESCAPECHAR] [-e ENCODING]
               [FILE]

Render a CSV file in the console as a 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
```

If a table is too wide to display properly try truncating it using *csvcut*.

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

Also see: *Arguments common to all utilities*.

### 5.11.2 Examples

Basic use:

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

This utility is especially useful as a final operation when piping through other utilities:

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

## 5.12 csvpy

### 5.12.1 Description

Loads a CSV file into a `csvkit.CSVKitReader` 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 MAXFIELDSIZE] [-e ENCODING] [-v]
             FILE

Load a CSV file into a CSVKitReader object and then drops into a Python shell.

positional arguments:
  FILE                  The CSV file to operate on.

optional arguments:
  -h, --help            show this help message and exit
  --dict                Use CSVKitDictReader instead of CSVKitReader.
```

This utility 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.

Also see: *Arguments common to all utilities*.

### 5.12.2 Examples

Basic use:

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

As a dictionary:

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

## 5.13 csvsql

### 5.13.1 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 MAXFIELDSIZE] [-e ENCODING] [-v]
              [-y SNIFFLIMIT]
              [-i {access,sybase,sqlite,informix,firebird,mysql,oracle,maxdb,postgresql,mssql}]
              [--db CONNECTION_STRING] [--insert]
              [FILE]

Generate SQL statements for a CSV file or create execute those statements
directly on a database.

Generate a SQL CREATE TABLE statement for a CSV file.

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
  -y SNIFFLIMIT, --snifflimit SNIFFLIMIT
                        Limit CSV dialect sniffing to the specified number of
                        bytes.
  -i {access,sybase,sqlite,informix,firebird,mysql,oracle,maxdb,postgresql,mssql}, --dialect {access,
                        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.
  --insert              In addition to creating the table, also insert the
                        data into the table. Only valid when --db is
                        specified.
  --table TABLE_NAME    Specify a name for the table to be created. If
                        omitted, the filename (minus extension) will be used.
  --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.
  --blanks              Do not coerce empty strings to NULL values.
```

Also see: *Arguments common to all utilities*.

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

### 5.13.2 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 Postgres:

```
$ createdb test
$ csvsql --db postgresql:///test --table fy09 --insert examples/realdata/FY09_EDU_Recipients_by_State
```

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 --table fy0
```

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

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

## 5.14 csvstat

### 5.14.1 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] [-e ENCODING]
                  [FILE]

Print descriptive statistics for all columns 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
  -y SNIFFLIMIT, --snifflimit SNIFFLIMIT
                        Limit CSV dialect sniffing to the specified number of
                        bytes.
  -c COLUMNS, --columns COLUMNS
                        A comma separated list of column indices or names to
                        be examined. Defaults to all columns.
  --max                 Only output max.
  --min                 Only output min.
  --sum                 Only output sum.
  --mean                Only output mean.
  --median              Only output median.
  --stdev               Only output standard deviation.
  --nulls               Only output whether column contains nulls.
  --unique              Only output unique values.
  --freq                Only output frequent values.
  --len                 Only output max value length.
```

Also see: *Arguments common to all utilities*.

### 5.14.2 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 --freq examples/realdata/FY09_EDU_Recipients_by_State.csv

  1. State Name: None
  2. State Abbreviate: None
  3. Code: None
  4. Montgomery GI Bill-Active Duty: 3548.0
  5. Montgomery GI Bill- Selective Reserve: 1019.0
  6. Dependents' Educational Assistance: 1261.0
  7. Reserve Educational Assistance Program: 715.0
  8. Post-Vietnam Era Veteran's Educational Assistance Program: 6.0
  9. TOTAL: 6520.0
 10. _unnamed: None
```

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

```
$ csvstat -c 4 --freq examples/realdata/FY09_EDU_Recipients_by_State.csv

3548.0
```

*Appendices*

## 5.15 Arguments common to all utilities

### 5.15.1 Description

All utilities 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            Specifies 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, --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 is
                      set to "Quote None" and the quotechar if doublequote
                      is not specified.
-z MAXFIELDSIZE, --maxfieldsize MAXFIELDSIZE
                      Maximum length of a single field in the input CSV
                      file.
-e ENCODING, --encoding ENCODING
-v, --verbose         Print detailed tracebacks when errors occur.
                      Specify the encoding the input file.
-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.
```

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.

Note that the output of csvkit's utilities 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.)

### 5.15.2 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/r
```

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

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

CHAPTER

SIX

# DEVELOPMENT

csvkit is designed to augment or supercede much of Python's `csv` module. Important parts of the API are documented here:

## 6.1 csvkit

This module contains csvkit's superpowered reader and writer. The most improvement over the standard library versions is that these versions are completely unicode aware and can support any encoding by simply passing in the its name at the time they are created.

We recommend you use these as a replacement for `csv.reader()` and `csv.writer()`.

**class** `csvkit.`**`CSVKitReader`**(*f*, *encoding='utf-8'*, *maxfieldsize=None*, ***kwargs*)

A unicode-aware CSV reader. Currently adds nothing to `csvkit.unicsv.UnicodeCSVReader`, but might someday.

**`line_num`**

**`next`**()

**class** `csvkit.`**`CSVKitWriter`**(*f*, *encoding='utf-8'*, *line_numbers=False*, ***kwargs*)

A unicode-aware CSV writer with some additional features.

**`writerow`**(*row*)

**`writerows`**(*rows*)

**class** `csvkit.`**`CSVKitDictReader`**(*f*, *fieldnames=None*, *restkey=None*, *restval=None*, **args*, ***kwargs*)

A unicode-aware CSV DictReader. Currently adds nothing to `csvkit.unicsv.UnicodeCSVWriter`, but might someday.

**`fieldnames`**

**`next`**()

**class** `csvkit.`**`CSVKitDictWriter`**(*f*, *encoding='utf-8'*, *line_numbers=False*, ***kwargs*)

A unicode-aware CSV DictWriter with some additional features.

**`writerow`**(*row*)

**`writerows`**(*rows*)

**`writeheader`**()

## 6.2 csvkit.unicsv

This module contains unicode aware replacements for `csv.reader()` and `csv.writer()`. The implementations are largely copied from examples in the csv module documentation.

**class** `csvkit.unicsv.`**`UTF8Recoder`**(*f*, *encoding*)
  Iterator that reads an encoded stream and reencodes the input to UTF-8.

  **`next`**()

**class** `csvkit.unicsv.`**`UnicodeCSVReader`**(*f*, *encoding='utf-8'*, *maxfieldsize=None*, ***kwargs*)
  A CSV reader which will read rows from a file in a given encoding.

  **`next`**()

  **`line_num`**

**class** `csvkit.unicsv.`**`UnicodeCSVWriter`**(*f*, *encoding='utf-8'*, ***kwargs*)
  A CSV writer which will write rows to a file in the specified encoding.

  **NB: Optimized so that eight-bit encodings skip re-encoding. See:** https://github.com/onyxfish/csvkit/issues/175

  **`writerow`**(*row*)

  **`writerows`**(*rows*)

**class** `csvkit.unicsv.`**`UnicodeCSVDictReader`**(*f*, *fieldnames=None*, *restkey=None*, *restval=None*, **args*, ***kwargs*)
  Defer almost all implementation to `csv.DictReader`, but wraps our unicode reader instead of `csv.reader()`.

  **`fieldnames`**

  **`next`**()

**class** `csvkit.unicsv.`**`UnicodeCSVDictWriter`**(*f*, *fieldnames*, *writeheader=False*, *restval='‘*, *extrasaction='raise'*, **args*, ***kwds*)
  Defer almost all implementation to `csv.DictWriter`, but wraps our unicode writer instead of `csv.writer()`.

  **`writeheader`**()

  **`writerow`**(*rowdict*)

  **`writerows`**(*rowdicts*)

## 6.3 csvkit.sniffer

`csvkit.sniffer.`**`sniff_dialect`**(*sample*)
  A functional version of `csv.Sniffer().sniff`, that extends the list of possible delimiters to include some seen in the wild.

CHAPTER

SEVEN

# CONTRIBUTING

Want to hack on csvkit? Here's how:

## 7.1 Contributing to csvkit

### 7.1.1 Welcome!

Thanks for your interest in contributing to csvkit. There is work be done by developers of all skill levels.

### 7.1.2 Process for contributing code

Contributors should use the following roadmap to guide them through the process of submitting a contribution:

1. Fork the project on Github.
2. Check out the issue tracker and find a task that needs to be done and is of a scope you can realistically expect to complete in a few days. Don't worry about the priority of the issues at first, but try to choose something you'll enjoy. You're much more likely to finish something to the point it can be merged if it's something you really enjoy hacking on.
3. Comment on the ticket letting everyone know you're going to be hacking on it so that nobody duplicates your effort. It's also good practice to provide some general idea of how you plan on resolving the issue so that other developers can make suggestions.
4. Write tests for the feature you're building. Follow the format of the existing tests in the test directory to see how this works. You can run all the tests with the command `nosetests`. The one exception to testing is command-line scripts. These don't need unit test, though all reusable components should be factored into library modules.
5. Write the code. Try to stay consistent with the style and organization of the existing codebase. A good patch won't be refused for stylistic reasons, but large parts of it may be rewritten and nobody wants that.
6. As your coding, periodically merge in work from the master branch and verify you haven't broken anything by running the test suite.
7. Write documentation for user-facing features (and library features once the API has stabilized).
8. Once it works, is tested, and has documentation, submit a pull request on Github.
9. Wait for it to either be merged or to recieve a comment about what needs to be fixed.
10. Rejoice.

### 7.1.3 Legalese

To the extent that they care, contributors 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 code 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 code contributions (and no others), however, the released version of the code can never be retracted or relicensed.

# CHAPTER EIGHT

# 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

- Krzysztof Dorosz

CHAPTER

NINE

# LICENSE

CHAPTER

TEN

# CHANGELOG

## 10.1 0.5.1

- Added Chris Rosenthal to AUTHORS.
- Fixed multi-file input to csvsql. (#193)
- Added csvpys command (python scripting). (#214, #215)
- Added csvgroup command (equivalent to sql group by). (#215)
- Added Krzysztof Dorosz to AUTHORS.

## 10.2 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)

## 10.3 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)

## 10.4 0.4.3

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

CHAPTER

ELEVEN

# INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

# PYTHON MODULE INDEX

### c

csvkit, **??**
csvkit.sniffer, **??**
csvkit.unicsv, **??**