

---

**csvkit**

*Release 0.9.1*

March 30, 2015



<b>1</b>	<b>About</b>	<b>1</b>
<b>2</b>	<b>Why csvkit?</b>	<b>3</b>
<b>3</b>	<b>Table of contents</b>	<b>5</b>
3.1	Installation . . . . .	5
3.2	Tutorial . . . . .	6
3.3	Command-Line Usage . . . . .	16
3.4	Using as a Python library . . . . .	32
3.5	Contributing to csvkit . . . . .	35
3.6	Release process . . . . .	36
<b>4</b>	<b>Authors</b>	<b>37</b>
<b>5</b>	<b>License</b>	<b>39</b>
<b>6</b>	<b>Changelog</b>	<b>41</b>
6.1	0.9.1 . . . . .	41
6.2	0.9.0 . . . . .	41
6.3	0.8.0 . . . . .	42
6.4	0.7.3 . . . . .	42
6.5	0.7.2 . . . . .	42
6.6	0.7.1 . . . . .	43
6.7	0.7.0 . . . . .	43
6.8	0.6.1 . . . . .	43
6.9	0.5.0 . . . . .	44
6.10	0.4.4 . . . . .	44
6.11	0.4.3 . . . . .	44
<b>7</b>	<b>Indices and tables</b>	<b>45</b>
	<b>Python Module Index</b>	<b>47</b>



### 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.rfd.org/>
- Schemas: <https://github.com/onyxfish/ffs>
- Buildbot: <https://travis-ci.org/onyxfish/csvkit>



---

## 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 > matching.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 > old_folks.csv
```

Import into PostgreSQL:

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

Extract data from PostgreSQL::

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

And much more...





---

## Table of contents

---

### 3.1 Installation

#### 3.1.1 Users

csvkit works on Python versions 2.6, 2.7, 3.3 and 3.4, as well as [PyPy](#). It is supported on OSX and Linux. It also works—but is tested less frequently—on Windows.

Installing csvkit is simple:

```
pip install csvkit
```

---

**Note:** If you are installing on Ubuntu you may need to install the Python development headers prior to install csvkit:

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

---

**Note:** If the installation appears to be successful but running the tools fails, try updating your version of Python setuptools:

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

---

**Note:** If you are using Python2 and have a recent version of pip, you may need to run pip with the additional arguments `--allow-external argparse`.

---

#### 3.1.2 Developers

If you are a developer that also wants to hack on csvkit, install it this way:

```
git clone git://github.com/onyxfish/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
```

Before writing any code be sure to read the documentation for *Contributing to csvkit*.

## 3.2 Tutorial

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

### 3.2.1 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. This tutorial will explain the workings of most of the csvkit utilities (including some nifty tricks) in the context of 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 are comfortable in the command line, but does not assume any prior experience doing data processing or analysis.

#### Installing csvkit

Installing csvkit is easy:

```
$ sudo pip install csvkit
```

If you have problems installing, check out the common issues described in the *Installation* section of the full documentation.

---

**Note:** If you're familiar with [virtualenv](#), it's better to install csvkit inside an env, in which case 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://github.com/onyxfish/csvkit/raw/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 make it a CSV:

```
$ in2csv ne_1033_data.xlsx
```

You should see a CSV version of the data dumped into your terminal. All csvkit utilities write to the terminal output ("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 literally means "redirect standard out to a file", but it may be more convenient to think of it as "save".

*in2csv* will convert a variety of common file formats, including xls, xlsx and fixed-width into CSV format.

### 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? Enter `csvlook`:

```
$ csvlook data.csv
```

Now 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. To fix this we need to learn how to reduce our dataset before we look at it.

### csvcut: data scalpel

*csvcut* is the original csvkit tool, the one that started the whole thing. With it, we can slice, 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 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
```

All csvkit utilities accept an input file as "standard in", in addition to as a filename. This means that we can make the output of one csvkit utility become the input of the next. In this case, the output of `csvcut` becomes the input to `csvlook`. This also means we can use this output with standard unix commands such as `head`, which prints only the first ten lines of it's input. Here, the output of `csvlook` becomes the input of `head`.

Pipeability is a core feature of csvkit. Of course, you can always write your output to a file using `>`, but many times it makes more sense to use pipes for speed and brevity.

Of course, we can also pipe `in2csv`, combining all our previous operations into one:

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

### Summing up

All the csvkit utilities work standard input and output. 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](#).

## 3.2.2 Examining the data

### csvstat: statistics without code

In the previous section we saw how we could use `csvlook` and `csvcut` to peek at slices of our data. This is a good starting place for diving into a dataset, but in practice we usually want to get the widest possible view before we start diving into specifics.

`csvstat` is designed to give us just such a broad picture of our data. It is inspired by the `summary()` function from the computational statistics programming language "R".

Let's examine summary statistics for some selected columns from our data (remember you can use `csvcut -n data.csv` to see the columns in the data):

```
$ csvcut -c county,acquisition_cost,ship_date data.csv | csvstat
1. county
   <type 'unicode'>
   Nulls: False
   Unique values: 35
   5 most frequent values:
       DOUGLAS:      760
       DAKOTA:       42
       CASS:         37
       HALL:         23
       LANCASTER:   18
   Max length: 10
```





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*.

### 3.2.3 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://github.com/onyxfish/csvkit/raw/master/examples/realdata/acs2012_5yr_population.csv
```

Now let’s see what’s in there:

```
$ csvstat acs2012_5yr_population.csv
1. fips
   <type 'int'>
   Nulls: False
   Min: 31001
   Max: 31185
   Sum: 2891649
   Mean: 31093.0
   Median: 31093
   Standard Deviation: 53.6904709112
   Unique values: 93
2. name
   <type 'unicode'>
   Nulls: False
   Unique values: 93
   Max length: 23
3. total_population
   <type 'int'>
   Nulls: False
   Min: 348
   Max: 518271
   Sum: 1827306
   Mean: 19648.4516129
   Median: 6294
   Standard Deviation: 62164.0702096
   Unique values: 93
4. margin_of_error
   <type 'int'>
   Nulls: False
   Min: 0
   Max: 115
   Sum: 1800
   Mean: 19.3548387097
   Median: 0
   Standard Deviation: 37.6927719494
```

```
Unique values: 15
5 most frequent values:
    0:      73
   115:     2
   114:     2
    99:     2
    73:     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 -n 8
```

county	item_name	total_population
MCPHERSON	RIFLE, 5.56 MILLIMETER	348
WHEELER	RIFLE, 5.56 MILLIMETER	725
GREELEY	RIFLE, 7.62 MILLIMETER	2515
GREELEY	RIFLE, 7.62 MILLIMETER	2515
GREELEY	RIFLE, 7.62 MILLIMETER	2515
NANCE	RIFLE, 5.56 MILLIMETER	3730
NANCE	RIFLE, 7.62 MILLIMETER	3730

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 aggregate analysis. `csvstack` allows you to "stack" the rows from CSV files with identical headers. 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://github.com/onyxfish/csvkit/raw/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
  <type 'unicode'>
  Nulls: False
  Values: KS, NE
8. acquisition_cost
  <type 'float'>
  Nulls: False
  Min: 0.0
  Max: 658000.0
  Sum: 9447912.36
  Mean: 3618.50339334
  Median: 138.0
  Standard Deviation: 23725.9555723
  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 INTEGER NOT NULL,
  nsn VARCHAR(16) NOT NULL,
  item_name VARCHAR(62) NOT NULL,
  quantity VARCHAR(4) NOT NULL,
  ui VARCHAR(7) NOT NULL,
  acquisition_cost FLOAT NOT NULL,
  total_cost VARCHAR(10) NOT NULL,
  ship_date DATE NOT NULL,
  federal_supply_category VARCHAR(34) NOT NULL,
  federal_supply_category_name VARCHAR(35) NOT NULL,
  federal_supply_class VARCHAR(25) NOT NULL,
  federal_supply_class_name VARCHAR(63),
  name VARCHAR(21) NOT NULL,
  total_population INTEGER NOT NULL,
```

```
margin_of_error INTEGER NOT NULL
);
```

Here we have the sqlite “create table” statement for our joined data. You’ll see that, like `csvstat`, `csvsql` has done it’s 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 this 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*.

## 3.2.4 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 those making maps, `csvjson` can also output GeoJSON, see its [documentation](#) 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 CSVKitReader object named "reader".
>>> print len(list(reader))
1037
>>> quit()

```

In addition to being a time-saver, because this uses `csvkit`'s internal `CSVKitReader` 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 relative small number of command line tools. Of course, this tutorial has only scratched the surface of the available options, so remember to check the documentation for each tool as well.

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

## 3.3 Command-Line Usage

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

### 3.3.1 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] [-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, ndjson, xls, xlsx.
-s SCHEMA, --schema SCHEMA
```

---

	Specifies a CSV-formatted schema file for converting fixed-width files. See documentation for details.
<code>-k KEY, --key KEY</code>	Specifies a top-level key to use look within for a list of objects to be converted when processing JSON.
<code>-y SNIFFLIMIT, --snifflimit SNIFFLIMIT</code>	Limit CSV dialect sniffing to the specified number of bytes. Specify "0" to disable sniffing entirely.
<code>--sheet SHEET</code>	The name of the XLSX sheet to operate on.
<code>--no-inference</code>	Disable type inference when parsing the input.

See also: *Arguments common to all utilities.*

---

**Note:** The “ndjson” format refers to “newline delimited JSON”, such as is output by the many streaming APIs.

---



---

**Note:** DBF format is only supported when running on Python 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/
```

---

**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.csv
```

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 geo
```

## sql2csv

### Description

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

```
usage: sql2csv [-h] [-v] [-l] [--db CONNECTION_STRING] [-q QUERY] [-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.
--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.
-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" --table "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" --table "fy09" --insert examples/realdata/FY09_EDU_Recipient
$ sql2csv --db "postgresql:///recipients" --query "select * from fy09 where \"State Name\" != '' orde
```

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"
```

## 3.3.2 Processing

### csvclean

#### 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.

See also: *Arguments common to all utilities.*

## 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
```

## csvcut

### 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.  
 -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 utilities.*

---

**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
```

## 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] [-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.

See also: *Arguments common to all utilities*.

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
```

## 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] [-e ENCODING] [--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,

`--right` starting at the left.  
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.

See also: *Arguments common to all utilities.*

## Examples

```
csvjoin -c "ColumnKey,Column Key" --outer file1.csv file2.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`.

## csvsort

### 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 "0" to disable sniffing entirely.  
`-n, --names` Specify the encoding the input file. 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.  
`--no-inference` Disable type inference when parsing the input.

See also: *Arguments common to all utilities.*

## 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
```

## 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] [-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-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 utilities.*

### Examples

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

```
$ csvstack -g 2009,2010 examples/realdata/FY09_EDU_Recipients_by_State.csv examples/realdata/Datagov
```

## 3.3.3 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 MAXFIELDSIZE] [-e ENCODING] [-S] [-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:**

-D OUT\_DELIMITER, --out-delimiter OUT\_DELIMITER  
                                  Delimiting character of the output CSV file.

-T, --out-tabs  
                                  Specifies 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-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 --doublequote is not specified.

-M OUT\_LINETERMINATOR, --out-lineterminator OUT\_LINETERMINATOR  
                                  Character used to terminate lines in the output CSV file.

See also: *Arguments common to all utilities.*

## Examples

Convert “standard” CSV file to a pipe-delimited one:

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

Convert to ridiculous 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 MAXFIELDSIZE] [-e ENCODING] [-H] [-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.

--stream Output JSON as a stream of newline-separated objects, rather than an as an array.

See also: *Arguments common to all utilities.*

## 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": ""  
      }  
    },  
    [...]  
  ]  
}
```

## csvlook

### 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*.

See also: *Arguments common to all utilities*.

## 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
```

## csvpy

### 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] [-H] [-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.

---

See also: *Arguments common to all utilities*.

## 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'}
```

## 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 MAXFIELDSIZE] [-e ENCODING] [-H] [-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. Specify "0" to disable sniffing entirely.

-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.

--query QUERY            Execute one or more SQL queries delimited by ";" and output the result of the last query as CSV.

--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.
--no-inference      Disable type inference when parsing the input.
--db-schema        Optional name of database schema to create table(s)
                    in.

```

See also: *Arguments common to all utilities*.

For information on connection strings and supported dialects refer to the [SQLAlchemy documentation](#).

---

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

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

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
```

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 iris as m on i.usda_id = m.usda_id"
```

## 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] [-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. Specify "0" to disable sniffing entirely.  
-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.  
--count Only output row count

See also: *Arguments common to all utilities.*

## 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
```

### 3.3.4 Appendices

#### Arguments common to all utilities

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 3
    ("Quote None") is specified and to escape the
    QUOTECHAR if --doublequote is not specified.
-z MAXFIELDSIZE, --maxfieldsize MAXFIELDSIZE
    Maximum length of a single field in the input CSV
    file.
-H, --no-header-row
    Specifies that the input CSV file has no header row.
-e ENCODING, --encoding ENCODING
-S, --skipinitialspace
    Ignore whitespace immediately following the delimiter.
-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.
--zero
    When interpreting or displaying column numbers, use
    zero-based numbering instead of the default 1-based
    numbering.
```

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 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.)

#### Tips and Tricks

##### 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.

### Specifying STDIN as a file

Most tools default to `STDIN` if no filename is specified, but tools like `csvjoin` and `csvstack` accept multiple files, so this is not possible. To work around this it is also possible to specify `STDIN` by using `-` as a filename. For example, these three commands are functionally identical:

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

This specification allows you to, for instance, `csvstack` input on `STDIN` with another file:

```
$ cat ~/src/csvkit/examples/dummy.csv | csvstack ~/src/csvkit/examples/dummy3.csv -
```

## 3.4 Using as a Python library

csvkit is designed to be used a replacement for most of Python's `csv` module. Important parts of the API are documented on the following pages.

Don't!

```
import csv
```

Do!

```
import csvkit
```

### 3.4.1 csvkit

This module contains csvkit's superpowered replacement for the builtin `csv` module. For Python 2 users, the greatest improvement over the standard library full unicode support. Python 3's `csv` module supports unicode internally, so this module is provided primarily for compatability purposes.

- Python 2: `csvkit.py2`.
- Python 3: `csvkit.py3`.

### 3.4.2 csvkit.py2

Python2-specific classes.

```
class csvkit.py2.CSVKitReader (f, encoding='utf-8', maxfieldsize=None, **kwargs)
    A unicode-aware CSV reader.
```

```
    line_num
```

```
    next ()
```

```
class csvkit.py2.CSVKitWriter (f, encoding='utf-8', line_numbers=False, **kwargs)
    A unicode-aware CSV writer.
```

```
    writerow (row)
```

```
    writerows (rows)
```

---

```
class csvkit.py2.CSVKitDictReader(f, fieldnames=None, restkey=None, restval=None, *args,
                                **kwargs)
```

A unicode-aware CSV DictReader.

```
fieldnames
```

```
next ()
```

```
class csvkit.py2.CSVKitDictWriter(f, fieldnames, encoding='utf-8', line_numbers=False,
                                **kwargs)
```

A unicode-aware CSV DictWriter.

```
writerow (row)
```

```
writerows (rows)
```

```
writeheader ()
```

```
csvkit.py2.reader (*args, **kwargs)
```

A drop-in replacement for Python's `csv.reader()` that leverages `csvkit.py2.CSVKitReader`.

```
csvkit.py2.writer (*args, **kwargs)
```

A drop-in replacement for Python's `csv.writer()` that leverages `csvkit.py2.CSVKitWriter`.

### 3.4.3 csvkit.py3

Python3-specific classes.

```
class csvkit.py3.CSVKitReader(f, **kwargs)
```

A wrapper around Python 3's builtin `csv.reader()`.

```
dialect
```

```
line_num
```

```
next ()
```

```
class csvkit.py3.CSVKitWriter(f, line_numbers=False, **kwargs)
```

A wrapper around Python 3's builtin `csv.writer()`.

```
writerow (row)
```

```
writerows (rows)
```

```
class csvkit.py3.CSVKitDictReader(f, fieldnames=None, restkey=None, restval=None, dialect='excel', *args, **kws)
```

A wrapper around Python 3's builtin `csv.DictReader`.

```
fieldnames
```

```
next ()
```

```
class csvkit.py3.CSVKitDictWriter(f, fieldnames, line_numbers=False, **kwargs)
```

A wrapper around Python 3's builtin `csv.DictWriter`.

```
writerow (row)
```

```
writerows (rows)
```

```
writeheader ()
```

```
csvkit.py3.reader (*args, **kwargs)
```

A drop-in replacement for Python's `csv.reader()` that leverages `csvkit.py3.CSVKitReader`.

```
csvkit.py3.writer (*args, **kwargs)
```

A drop-in replacement for Python's `csv.writer()` that leverages `csvkit.py3.CSVKitWriter`.

### 3.4.4 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](#).

These classes are available for Python 2 only. The Python 3 version of `csv` supports unicode internally.

---

**Note:** You probably don't want to use these classes directly. Try the `csvkit` module.

---

**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, restval='', extrasaction='raise', \*args, \*\*kws*)

Defer almost all implementation to `csv.DictWriter`, but wraps our unicode writer instead of `csv.writer()`.

`writerow` (*rowdict*)

`writerows` (*rowdicts*)

`writeheader()`

### 3.4.5 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.

## 3.5 Contributing to csvkit

### 3.5.1 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 compatibility 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.

### 3.5.2 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 `tox`.
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 you're 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.
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 receive a comment about what needs to be fixed.

10. Rejoice.

### 3.5.3 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.

## 3.6 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
  - 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.



---

**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
- aarcro
- 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

---

**License**

---

The MIT License

Copyright (c) 2014 Christopher Groskopf and contributors

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.



---

## Changelog

---

### 6.1 0.9.1

- 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)

### 6.2 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.

## 6.3 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)

## 6.4 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)

## 6.5 0.7.2

- Fix CHANGELOG for release.

## 6.6 0.7.1

- Fix homepage url in setup.py.

## 6.7 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.

## 6.8 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)

## 6.9 0.5.0

- Implement gejson 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)

## 6.10 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)

## 6.11 0.4.3

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



---

## Indices and tables

---

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



**C**

csvkit, 32  
csvkit.py2, 32  
csvkit.py3, 33  
csvkit.sniffer, 34  
csvkit.unicsv, 34



**C**

csvkit (module), 32  
 csvkit.py2 (module), 32  
 csvkit.py3 (module), 33  
 csvkit.sniffer (module), 34  
 csvkit.unicsv (module), 34  
 CSVKitDictReader (class in csvkit.py2), 32  
 CSVKitDictReader (class in csvkit.py3), 33  
 CSVKitDictWriter (class in csvkit.py2), 33  
 CSVKitDictWriter (class in csvkit.py3), 33  
 CSVKitReader (class in csvkit.py2), 32  
 CSVKitReader (class in csvkit.py3), 33  
 CSVKitWriter (class in csvkit.py2), 32  
 CSVKitWriter (class in csvkit.py3), 33

**D**

dialect (csvkit.py3.CSVKitReader attribute), 33

**F**

fieldnames (csvkit.py2.CSVKitDictReader attribute), 33  
 fieldnames (csvkit.py3.CSVKitDictReader attribute), 33  
 fieldnames (csvkit.unicsv.UnicodeCSVDictReader attribute), 34

**L**

line\_num (csvkit.py2.CSVKitReader attribute), 32  
 line\_num (csvkit.py3.CSVKitReader attribute), 33  
 line\_num (csvkit.unicsv.UnicodeCSVReader attribute), 34

**N**

next() (csvkit.py2.CSVKitDictReader method), 33  
 next() (csvkit.py2.CSVKitReader method), 32  
 next() (csvkit.py3.CSVKitDictReader method), 33  
 next() (csvkit.py3.CSVKitReader method), 33  
 next() (csvkit.unicsv.UnicodeCSVDictReader method), 34  
 next() (csvkit.unicsv.UnicodeCSVReader method), 34  
 next() (csvkit.unicsv.UTF8Recoder method), 34

**R**

reader() (in module csvkit.py2), 33  
 reader() (in module csvkit.py3), 33

**S**

sniff\_dialect() (in module csvkit.sniffer), 34

**U**

UnicodeCSVDictReader (class in csvkit.unicsv), 34  
 UnicodeCSVDictWriter (class in csvkit.unicsv), 34  
 UnicodeCSVReader (class in csvkit.unicsv), 34  
 UnicodeCSVWriter (class in csvkit.unicsv), 34  
 UTF8Recoder (class in csvkit.unicsv), 34

**W**

writeheader() (csvkit.py2.CSVKitDictWriter method), 33  
 writeheader() (csvkit.py3.CSVKitDictWriter method), 33  
 writeheader() (csvkit.unicsv.UnicodeCSVDictWriter method), 34  
 writer() (in module csvkit.py2), 33  
 writer() (in module csvkit.py3), 33  
 writerow() (csvkit.py2.CSVKitDictWriter method), 33  
 writerow() (csvkit.py2.CSVKitWriter method), 32  
 writerow() (csvkit.py3.CSVKitDictWriter method), 33  
 writerow() (csvkit.py3.CSVKitWriter method), 33  
 writerow() (csvkit.unicsv.UnicodeCSVDictWriter method), 34  
 writerow() (csvkit.unicsv.UnicodeCSVWriter method), 34  
 writerows() (csvkit.py2.CSVKitDictWriter method), 33  
 writerows() (csvkit.py2.CSVKitWriter method), 32  
 writerows() (csvkit.py3.CSVKitDictWriter method), 33  
 writerows() (csvkit.py3.CSVKitWriter method), 33  
 writerows() (csvkit.unicsv.UnicodeCSVDictWriter method), 34  
 writerows() (csvkit.unicsv.UnicodeCSVWriter method), 34