CSVSee Documentation

Release 0.2

Automation Excellence

September 20, 2012

CONTENTS

You are reading the documentation for CSVSee, a tool for manipulating and visualizing data in comma-separated (CSV) files.

This software is open source, under the terms of the simplified BSD license.

CHAPTER

ONE

MOTIVATION

This tool was originally developed to help with analyzing test results coming from Grinder and Performance Monitor. It was partly inspired by Grinder Analyzer, which serves a similar purpose in a more specific domain.

CHAPTER

TWO

FEATURES

- Generate graphs of just about any numerical data, particularly those having timestamps
- Match one or more CSV column names with regular expressions
- Plot the top `N` data sets, by average or peak value within each column
- Automatic color-coding when graphing multiple columns
- Display graphs in an interactive viewer with zooming/panning capability
- Export graphs to a `.png`, `.svg` or `.pdf` file
- Customizable line styles, title / axis labels, and timestamp formats
- Automatic guessing of date/time format

CHAPTER

THREE

INSTALLATION

You'll need Python before doing anything else. Most Linux distributions already have this installed, but you can use this to check:

```
$ which python
/usr/bin/python
```

To use the graphing features of CSVSee, you'll need matplotlib. On Ubuntu, this should work:

```
$ sudo apt-get install python-matplotlib
```

If you want to install an official release, first download one from the downloads page, and extract it somewhere.

Then, open that directory in a terminal and run:

```
$ sudo python setup.py install
```

Or use pip:

```
$ sudo apt-get install python-pip
$ sudo pip install .
```

One advantage of using `pip` is that you can uninstall later like so:

```
$ sudo pip uninstall CSVSee
```

If you'd rather use a copy of the latest development version, clone it using Git:

```
$ git clone git://github.com/a-e/csvsee.git
```

then install as before using `setup.py` or `pip`.

3.1 Using virtualenv

There are some hassles when installing CSVSee's dependencies in a virtualenv. Specifically, NumPy and matplotlib must be compiled from source, requiring extra development headers and other dependencies that are not easily installable using pip. For this reason, it's strongly recommended that you just install NumPy and matplotlib through your regular package manager (like `apt-get`).

If you really want to install them in a virtualenv, you could try this:

```
$ sudo apt-get install python-dev libpng-dev
```

In order to display an interactive graphing window, you'll also need a GUI backend that matplotlib can use. Qt4, Gtk, and Tkinter should all work. I use Qt4:

```
$ sudo apt-get install python-qt4
```

Then you may be able to do:

```
$ pip install numpy
$ pip install matplotlib
```

But I make no promises. In fact, I couldn't get it to work, so if you manage to do so, please open an issue describing how you did it, so I can include it in this documentation.

CHAPTER

FOUR

USAGE

`csvs` is the primary frontend for CSVSee. You can run this without arguments to see what it expects. At minimum, you'll need to provide the name of a command. Currently, two commands are implemented:

- `csvs graph`: Generate graphs from `.csv` files
- `csvs grep`: Search in text files and generate a `.csv` file
- `csvs grinder`: Create `.csv` reports based on Grinder output file

4.1 csvs graph

The `graph` command is designed to generate graphs of comma-separated (.csv) data files. It was originally designed for graphing data from the Windows Performance Monitor tool, but it can also be used more generally to graph any CSV data that includes timestamps.

The only thing you must provide is a `filename.csv` containing your data. By default, the first column of data is used as the X-coordinate; if it's a timestamp, its format will be guessed.

You can optionally specify one or more regular expressions to match the column names you want to graph. If you don't provide these, all columns will be graphed. All data must be integer or floating-point numeric values; anything that isn't a date or number will be plotted as a 0.

Column names can be specified as regular expressions that may match one or more column headings in the .csv file. For example, if you have a file called `perfmon.csv` with columns named like this:

```
"Eastern Time","CPU (user)","CPU (system)","CPU (idle)","Memory"
```

You can generate a graph of user, system, and idle CPU values over time like this:

```
csvs graph perfmon.csv "CPU.*"
```

Run `csvs graph` without arguments to see full usage notes.

4.2 csvs grep

The `grep` command generates a `.csv` file by matching strings in one or more timestamped log files. It would typically be used to generate a report of how frequently certain messages or errors appear through time.

For example, if you have `parrot.log` containing:

```
2010/08/30 13:57:14 Pushing up the daisies
2010/08/30 13:58:08 Stunned
2010/08/30 13:58:11 Stunned
2010/08/30 14:04:22 Pining for the fjords
2010/08/30 14:05:37 Pushing up the daisies
2010/08/30 14:09:48 Pining for the fjords
```

And you wanted to see how often each of these phrases occur, do:

```
csvs grep parrot.log \
    -match "Stunned" "Pushing up the daisies" "Pining for the fjords" \
    -out parrot.csv
```

By default, the `grep` command counts the number of occurrences each minute, so this would give you a `.csv` file looking something like this (whitespace added for readability):

```
"Timestamp",        "Stunned", "Pushing up the daisies", "Pining for the fjords"
2010/08/30 13:57,   0,         1,                        0
2010/08/30 13:58,   2,         0,                        0
2010/08/30 14:04,   0,         0,                        1
2010/08/30 14:05,   0,         1,                        0
2010/08/30 14:09,   0,         0,                        1
```

You can change the resolution using the `-seconds` option. For example, to count the occurrences each hour, use `-seconds 3600`.

Run `csvs grep` without arguments to see full usage notes.

4.3 csvs grinder

New in version 0.2. The `grinder` command generates `.csv` files from Grinder logs. You must provide the name of a `out*` file, and one or more `data*` files generated from the same test run:

```
csvs grinder out-0.log data-*.log foo
```

This will write four `.csv` files in the current directory:

- `foo_Errors.csv`
- `foo_HTTP_response_errors.csv`
- `foo_HTTP_response_length.csv`
- `foo_Test_time.csv`

By default, statistics are summarized with a 60-second resolution; that is, all statistics within each 60-second interval are summed (in the case of errors) or averaged (in the case of response length and test time). To change the interval resolution, pass the `-seconds` option. For instance, to summarize statistics in 10-minute intervals:

```
csvs grinder -seconds 600 out-0.log data-*.log foo
```

Run `csvs grinder` without arguments to see full usage notes.

CHAPTER

FIVE

DEVELOPMENT

If you'd like to hack on CSVSee, you'll probably want to install the development dependencies first:

```
$ pip install -r dev-req.txt
```

5.1 Testing

New in version 0.2. CSVSee's core modules include several doctests, along with a suite of unit tests in the `tests` directory that can be run with py.test:

```
$ py.test
```

To generate a coverage report, you can just get a plain-text report:

```
$ py.test --cov csvsee --cov-report=term-missing
```

Or a nice HTML report:

```
$ py.test --cov csvsee --cov-report=html
```

CHAPTER

SIX

API

All of the supporting libraries are in a module called `csvsee`. You can read the autogenerated documentation here:

6.1 csvsee

The `csvsee` module provides most of the functionality of CSVSee. It consists of the following submodules:

6.1.1 csvsee.dates

Date/time parsing and manipulation functions

exception `csvsee.dates.`**`CannotParse`**

Failure to parse a date or time.

`csvsee.dates.`**`date_chop`**`(`*line*, *dateformat='%m/%d/%y %I:%M:%S %p'*, *resolution=60*`)`

Given a `line` of text, get a date/time formatted as `dateformat`, and return a `datetime` object rounded to the nearest `resolution` seconds. If `line` fails to match `dateformat`, a `CannotParse` exception is raised.

Examples:

```
>>> date_chop('1976/05/19 12:05:17', '%Y/%m/%d %H:%M:%S', 60)
datetime.datetime(1976, 5, 19, 12, 5)

>>> date_chop('1976/05/19 12:05:17', '%Y/%m/%d %H:%M:%S', 3600)
datetime.datetime(1976, 5, 19, 12, 0)
```

`csvsee.dates.`**`format_regexp`**`(`*simple_format*`)`

Given a simplified date or time format string, return `(format, regexp)`, where `format` is a `strptime`-compatible format string, and `regexp` is a regular expression that matches dates or times in that format.

The `simple_format` string supports a subset of `strptime` formatting directives, with the leading `%` characters removed.

Examples:

```
>>> format_regexp('Y/m/d')
('%Y/%m/%d', '(?P<Y>\\d\\d\\d\\d)/(?P<m>\\d\\d?)/(?P<d>\\d\\d?)')

>>> format_regexp('H:M:S')
('%H:%M:%S', '(?P<H>[01]?[0-9]|2[0-3]):(?P<M>[0-5]\\d):(?P<S>[0-5]\\d)')
```

csvsee.dates.**guess_file_date_format** (*filename*)

Open the given file and use `guess_format` to look for a date/time at the beginning of each line. Return the format string for the first one that's found. Raise `CannotParse` if none is found.

csvsee.dates.**guess_format** (*string*)

Try to guess the date/time format of `string`, or raise a `CannotParse` exception.

Examples:

```
>>> guess_format('2010/01/28 13:25:49')
'%Y/%m/%d %H:%M:%S'

>>> guess_format('01/28/10 1:25:49 PM')
'%m/%d/%y %I:%M:%S %p'

>>> guess_format('01/28/2010 13:25:49.123')
'%m/%d/%Y %H:%M:%S.%f'

>>> guess_format('Aug 15 2009 15:24')
'%b %d %Y %H:%M'

>>> guess_format('3-14-15 9:26:53.589')
'%m-%d-%y %H:%M:%S.%f'
```

Leading and trailing text may be present:

```
>>> guess_format('FOO April 1, 2007 3:45 PM BAR')
'%B %d, %Y %I:%M %p'

>>> guess_format('[[2010-09-25 14:19:24]]')
'%Y-%m-%d %H:%M:%S'
```

csvsee.dates.**parse** (*string*, *format*)

Attempt to parse the given string as a date in the given format. This is similar to `datetime.strptime`, but this can handle date strings with trailing characters. If it still fails to parse, raise a `CannotParse` exception.

Examples:

```
>>> parse('2010/08/28', '%Y/%m/%d')
datetime.datetime(2010, 8, 28, 0, 0)

>>> parse('2010/08/28 extra stuff', '%Y/%m/%d')
datetime.datetime(2010, 8, 28, 0, 0)

>>> parse('2010/08/28', '%m/%d/%y')
Traceback (most recent call last):
CannotParse: time data '2010/08/28' does not match format '%m/%d/%y'
```

6.1.2 csvsee.utils

Shared utility functions for the csvsee library.

exception csvsee.utils.**NoMatch**

Exception raised when no column name matches a given expression.

class csvsee.utils.**ProgressBar** (*end*, *prefix=''*, *fill='='*, *units='secs'*, *width=40*)

An ASCII command-line progress bar with percentage.

Adapted from Corey Goldberg's version: http://code.google.com/p/corey-projects/source/browse/trunk/python2/progress_bar.py

update (*current*)
Set the current progress.

`csvsee.utils.`**`boring_columns`** (*csvfile*)
Return a list of column names in `csvfile` that are "boring"–that is, the data in them is always the same.

`csvsee.utils.`**`column_names`** (*csv_file*)
Return a list of column names in the given `.csv` file.

`csvsee.utils.`**`filter_csv`** (*csv_infile, csv_outfile, columns, match='regexp', action='include'*)
Filter `csv_infile` and write output to `csv_outfile`.

columns A list of regular expressions or exact column names

match `regexp` to treat each value in `columns` as a regular expression, `exact` to match exact literal column names

action `include` to keep the specified `columns`, or `exclude` to keep all columns *except* the specified `columns`

`csvsee.utils.`**`float_or_0`** (*value*)
Try to convert `value` to a floating-point number. If conversion fails, return `0`.

Examples:

```
>>> float_or_0(5)
5.0

>>> float_or_0('5')
5.0

>>> float_or_0('five')
0
```

`csvsee.utils.`**`grep_files`** (*filenames, matches, dateformat='guess', resolution=60, show_progress=True*)
Search all the given files for matching text, and return a list of `(timestamp, counts)` for each match, where `timestamp` is a `datetime`, and `counts` is a dictionary of `{match: count}`, counting the number of times each match was found during intervals of `resolution` seconds.

`csvsee.utils.`**`line_count`** (*filename*)
Return the total number of lines in the given file.

`csvsee.utils.`**`matching_fields`** (*expr, fields*)
Return all `fields` that match a regular expression `expr`, or raise a `NoMatch` exception if no matches are found.

Examples:

```
>>> matching_fields('a.*', ['apple', 'banana', 'avocado'])
['apple', 'avocado']

>>> matching_fields('a.*', ['peach', 'grape', 'kiwi'])
Traceback (most recent call last):
NoMatch: No matching column found for 'a.*'
```

`csvsee.utils.`**`matching_xy_fields`** (*x_expr, y_exprs, fieldnames, verbose=False*)
Match `x_expr` and `y_exprs` to all available column names in `fieldnames`, and return the matched `x_column` and `y_columns`.

Example:

```
>>> matching_xy_fields('x.*', ['y[12]', 'y[ab]'],
...     ['xxx', 'y1', 'y2', 'y3', 'ya', 'yb', 'yc'])
('xxx', ['y1', 'y2', 'ya', 'yb'])
```

If `x_expr` is empty, the first column name is used:

```
>>> matching_xy_fields('', ['y[12]', 'y[ab]'],
...     ['xxx', 'y1', 'y2', 'y3', 'ya', 'yb', 'yc'])
('xxx', ['y1', 'y2', 'ya', 'yb'])
```

If no match is found for any expression in `y_exprs`, a `NoMatch` exception is raised:

```
>>> matching_xy_fields('', ['y[12]', 'y[jk]'],
...     ['xxx', 'y1', 'y2', 'y3', 'ya', 'yb', 'yc'])
Traceback (most recent call last):
NoMatch: No matching column found for 'y[jk]'
```

`csvsee.utils.`**`read_xy_values`**(*reader*, *x_column*, *y_columns*, *date_format=''*, *gmt_offset=0*, *zero_time=False*)

Read values from a `csv.DictReader`, and return `(x_values, y_values)`. where `x_values` is a list of values found in `x_column`, and `y_values` is a dictionary of `{y_column: [values]}` for each column in `y_columns`.

Arguments:

x_column Name of the column you want to use as the X axis.

y_columns Names of columns you want to plot on the Y axis.

date_format If given, treat values in `x_column` as timestamps with the given format string.

gmt_offset Add this many hours to every timestamp. Only useful with `date_format`.

zero_time If `True`, adjust timestamps so the earliest one starts at `00:00` (midnight). Only useful with `date_format`.

`csvsee.utils.`**`strip_prefix`**(*strings*)

Strip a common prefix from a sequence of strings. Return `(prefix, [stripped])` where `prefix` is the string that is common (with leading and trailing whitespace removed), and `[stripped]` is all strings with the prefix removed.

Examples:

```
>>> strip_prefix(['first', 'fourth', 'fifth'])
('f', ['irst', 'ourth', 'ifth'])

>>> strip_prefix(['spam and eggs', 'spam and potatoes', 'spam and spam'])
('spam and', ['eggs', 'potatoes', 'spam'])
```

`csvsee.utils.`**`top_by`**(*func*, *count*, *y_columns*, *y_values*, *drop=0*)

Apply `func` to each column, and return the top `count` column names. Arguments:

func A function that takes a list of values and returns a single value. `max`, `min`, and average are good examples.

count How many of the "top" values to keep

y_columns A list of candidate column names. All of these must exist as keys in `y_values`

y_values Dictionary of `{column: values}` for each y-column. Must have data for each column in `y_columns` (any extra column data will be ignored).

drop How many top values to skip before returning the next `count` top columns

csvsee.utils.**top_by_average** (*count*, *y_columns*, *y_values*, *drop=0*)
    Determine the top count columns based on the average of values in y_values, and return the filtered y_columns names.

csvsee.utils.**top_by_peak** (*count*, *y_columns*, *y_values*, *drop=0*)
    Determine the top count columns based on the peak value in y_values, and return the filtered y_columns names.

6.1.3 csvsee.graph

Provides a Graph class for creating graphs from .csv data files.

class csvsee.graph.**Graph** (*csv_file*, ***kwargs*)
    A graph of data from a CSV file.

add_date_labels (*min_date*, *max_date*)
    Add date labels to the graph.

generate ()
    Generate the graph.

guess_date_format (*date_column*)
    Try to guess the date format used in the current .csv file, by reading from the first row of the date_column column.

save (*filename*)
    Save the graph to filename. The format is determined by the extension of filename; if it's not png, svg, or pdf, then a ValueError is raised.

show ()
    Display the graph in a GUI window.

6.1.4 csvsee.grinder

New in version 0.2. Tools for working with Grinder log output and generating CSV data.

This module defines three important classes:

- Bin: Statistics collected over a certain time interval
- Test: All statistics for a single Test in a Grinder test run
- Report: Collection of Test statistics and functions to write CSV reports

To generate reports, simply instantiate a Report instance, providing the report granularity in seconds, the name of the out_* file, and at least one data_* file generated by Grinder:

```
from csvsee import grinder
report = grinder.Report(60, 'out-0.log', 'data-0.log')
```

Or, if you have multiple data_* files:

```
report = grinder.Report(60, 'out-0.log', 'data-0.log', 'data-1.log', 'data-2.log')
```

The granularity determines how fine-grained the timestamps in your report will be; with 60-second granularity, all statistics during each 60-second interval are accumulated and reported together. For more detail, you could use 1-second granularity:

```
report = grinder.Report(1, 'out-0.log', 'data-0.log')
```

but this will increase the size of your CSV data considerably and result in much noisier-looking data. If you're doing a long-term load test spanning hours, you might use a larger value, say 10 minutes:

```
report = grinder.Report(600, 'out-0.log', 'data-0.log')
```

This will give smaller CSV files with smoother data, at the expense of some detail; you won't be able to see data spikes as easily.

Now, you can generate a bunch of predetermined CSV files like this:

```
report.write_all_csvs('my_results')
```

The given string will be prefixed all the CSV filenames.

class `csvsee.grinder.`**`Bin`**(*stat_names*)
: Accumulated statistics for an interval of time.

  `add`(*row*)
  : Accumulate a row of statistics in this bin. All statistics are accumulated as integers.

  `average`(*stat*)
  : Return the integer average (mean) of the given statistic.

exception `csvsee.grinder.`**`NoTestNames`**
: Failure to find any test names in a Grinder `out*` file.

class `csvsee.grinder.`**`Report`**(*granularity*, *grinder_outfile*, **grinder_datafiles*)
: A report of statistics for a Grinder test run.

  `add`(*row*)
  : Add a row from a `data*` file to the stats.

  `populate_stats`()
  : Add statistics for all tests in all Grinder data files.

  `timestamp_range`()
  : Return the `(start, end)` timestamps for this report, based on the timestamps of all tests within it.

  `write_all_csvs`(*csv_prefix*)
  : Write all CSV files for this report to files with the given prefix.

  `write_csv`(*stat*, *filename*)
  : Write the given statistic for all tests to `filename`.

class `csvsee.grinder.`**`Test`**(*number*, *name*, *granularity=1*)
: Statistics for a single Test in a Grinder test run.

  `add`(*row*)
  : Add a row of statistics for this test.

  `stat_at_time`(*stat*, *timestamp*)
  : Return a statistic at the given timestamp (either sum or average). Return `0` if there is no data at the given time.

  `timestamp_range`()
  : Return the `(start, end)` timestamps for this test.

`csvsee.grinder.`**`get_test_names`**(*outfile*)
: Return a dict of `{number: name}` for each test from the summary portion of the given Grinder `out*` file. If the summary portion is not found, look for test numbers and names as logged by grinder-webtest.

`csvsee.grinder.`**`grinder_files`**(*include_dir*)
: Return a list of full pathnames to all `out*` and `data*` files found in descendants of `include_dir`.

PYTHON MODULE INDEX