CSVS Specification Working Draft

Karsten Held, 16.12.2016 - 21.02.2019
karstenheld@gmail.com

Mission Statement

Primary goals

- Create modern version of CSV file format that supports structured data
- Be backwards compatible with 99% of CSV, DSV files out there
- Allow lossless roundtrip conversions from/to JSON, XML and others

Secondary goals

- Design for good data recovery on format/read failures
- Design for streamable data content
- Allow both, a very compact and a human readable form
- Native support for typed data, binary data, metadata and comments
- Native support for globalization (comma) and encoding (unicode, UTF8/16)
- Design parser for binary file equality if data is read and saved unchanged

What CSV can do

Raw data

	A	B	C
1	234.233		0.34e+07
2	0.441	Hello!	83.34
3			
4	DB04DA	08AB02	FF10FC

Simple Tables

	A	B	C
1	X	Y	Name
2	0.1	12.34	Frank
3	0.2	23.45	Cindy
4	0.3	34.43	George

Config Files

	A	B
1	server-ip	192.268.1.10
2	log-level	4
3	input-file	e:\data.txt
4	timeout	5000

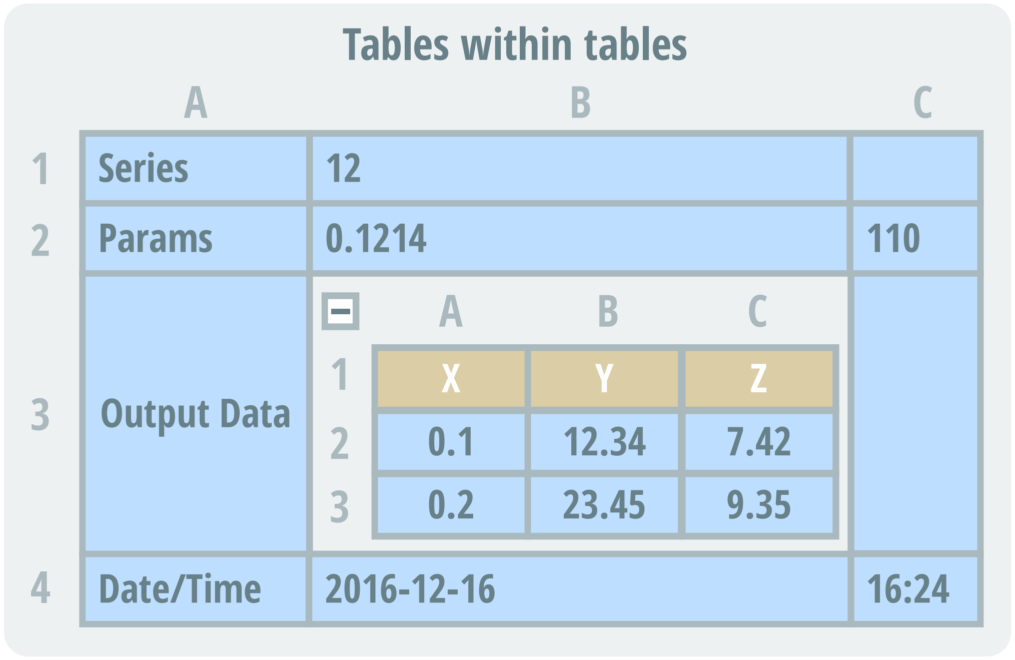
Spreadsheet Export

	A	B	C
1	Monthly Re		
2		Gross	Net
3	US/CA	320233	237332
4	EUROPE	120434	#DIV/0!

Time Series

	A	B	C
1	Date	AAPL	MSFT
2	2016-12-14	115.04	62.68
3	2016-12-15	115.38	62.58
4	2016-12-16	116.47	62.28

What CSV can't do


Tables within tables

	A	B	C
1	Series	12	
2	Params	0.1214	110
3	Output Data		
4	Date/Time	2016-12-16	16:24

	A	B	C
1	X	Y	Z
2	0.1	12.34	7.42
3	0.2	23.45	9.35

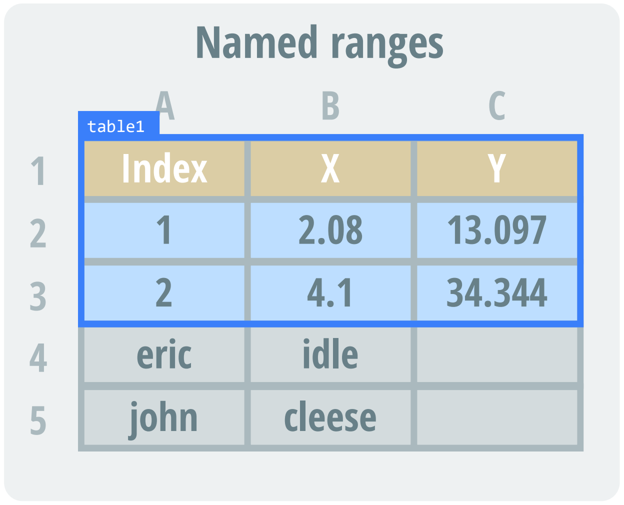

Named ranges

table1

	A	B	C
1	Index	X	Y
2	1	2.08	13.097
3	2	4.1	34.344
4	eric	idle	
5	john	cleese	

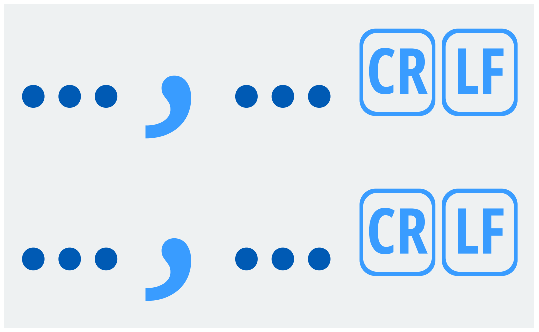
Structure!

Introducing

CSVS

Comma Separated Values with Structure

Cells, rows and columns



column separator: **Comma**

row separator: **Newline** (ASCII 13+10 or ASCII 10)

← Default row and column separators

These are 4 cells

Cells contain data or they can be empty. They are layed out in a grid of rows and columns. Columns are separated using **column separators**. Rows are separated using **row separators**.

Escaping of cell content

Cells containing reserved characters must be escaped using **doublequotes** (or **single quotes**). If they contain doublequotes (or single quotes), these have to be doubled. **Backticks** and **triple backticks** are also supported.

```
'"Al''s Pub"',"Saul ""Slash"" Hudson,CRLF
guitarist of ""Guns 'N Roses"" (🤘)",…
```

Flexible delimiters

You can specify the column/row separators using directives. These changes will be effective after the next row separator. Column/row separators also have fixed character sequences that can't be changed.

		default		fixed
column separator				
`!colsep[","]`	→	`,`	or	`||`
row separator				
`!rowsep["CRLF"]`	→	CRLF	or	`::`

CSVS is isomorphic. These files represent the same data:

```
Pi,π,3.1416::EulerNumber,e,2.7183::GoldenRatio
,ϕ,1.6180
```

```
!colsep[;]
"Pi"||"π"||3.1416
"EulerNumber";"e";2.7183
"GoldenRatio";"ϕ";1.6180
```

```
!colsep[" "]
Pi π 3.1416
EulerNumber e 2.7183
GoldenRatio ϕ 1.6180
```

Number formats and multivalue cells

```
!decsep["."]
!thsep[","," "]
!valsep["|"]
42,1.2,-1.9e-4
"120,000",250 234
1|2|3,amy|bob|zoe
```

Directives

The file format is specified via directives. Directives are executed after the next row separator. They can also be used to alter a previously specified format if the format changes within a file. Directives also support multiple characters or character sequences.

decimal separator	`!decsep["."]`	→	`.`	default values
thousand separator	`!thsep[","," "]`	→	`,` SP	
value separator	`!valsep["|"]`	→	`|`	

Multivalue Cells

Cells can hold multiple values. Those values are delimited using the value separator.

```
!valsep["#","_"]
r#g#b,red_blue_green
1#2#3,one_two_three
```

Fixed characters and keywords

For better readability and compatibility arcoss various CSVS variants some characters and keywords are considered as fixed and should not be re-used in unquoted contexts:

`!` `?` `"` `'` `` ` `` `[` `]` `{` `}` `(` `)`

`true` `false` `null` `//` `/*` `*/` `::` `..` `||` `!!` ```` ``` ````

Changing fixed characters and keywords is possible but not recommended.

Tables

This is a table

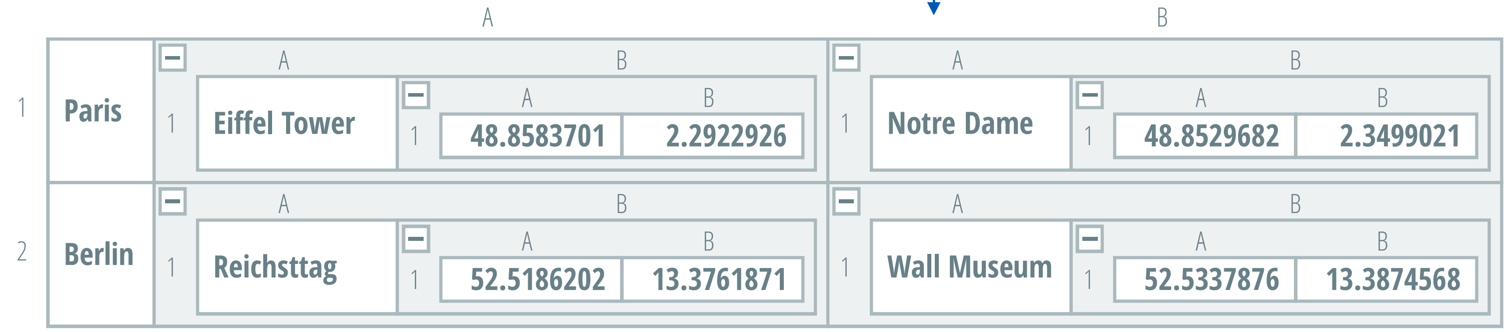
Tables structure data. They can be empty or contain one or many cells. Every table is contained in exactly one cell. A cell can be empty or contain a value or a table. Tables can also be nested:

`..` = line continuation

```
Paris, ['Eiffel Tower',[48.8583701, 2.2922926]],..
       ['Notre Dame',  [48.8529682, 2.3499021]]
Berlin,['Reichsttag',  [52.5186202,13.3761871]],..
       ['Wall Museum', [52.5337876,13.3874568]]
```

Whitespace around cell content will be ignored.

This is how this data looks visualized.



1	Paris	Eiffel Tower	48.8583701	2.2922926	Notre Dame	48.8529682	2.3499021
2	Berlin	Reichsttag	52.5186202	13.3761871	Wall Museum	52.5337876	13.3874568

Named tables

Tables are named by specifying a name directly before or after them (quoted or unquoted).

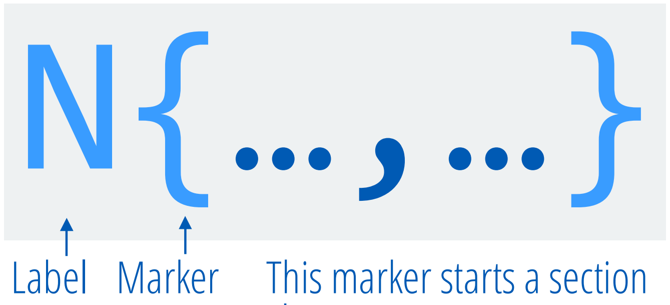
```
!rowsep[]
MyTable[
  1,"The first row"::
  2,"The second row"::
],
[1,1,0,0,1]"MyOtherTable"
```

`!rowsep[]` -> disable any row separator other than `::`.

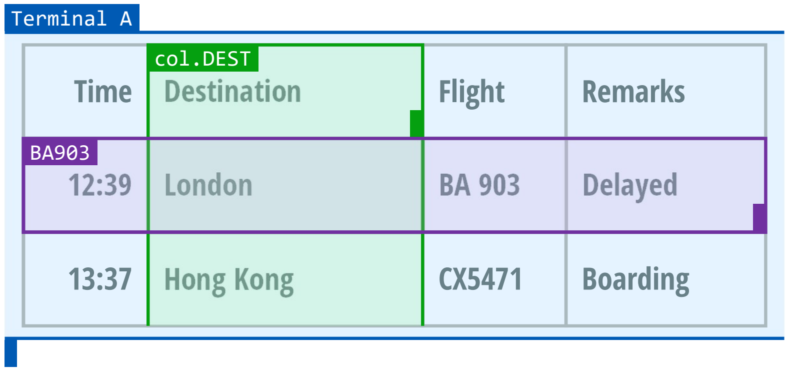
Names can be nested using the name qualifier `.`

```
Data.TableA[12.435]
"Data"."Table B"[1,2,3]
Data.[X]
["Row below X"]Data
```

Names



```
"Terminal A"{
Time,col.DEST{Destination},Flight,Remarks
BA903{12:39,"London","BA 903","Delayed"}
13:37,"Hong Kong","CX5471","Boarding"
}
```



Time	Destination	Flight	Remarks
12:39	London	BA 903	Delayed
13:37	Hong Kong	CX5471	Boarding

This is how this data looks visualized.

This is a named range

Named ranges are collections of cells that can be referenced by a name. Sections are ranges that cover a continuous group of lines.

Nested range names

Ranges can have nested names using the name qualifier (`.`): `N.M` `"1"."2"`
Subtracting is also possible: `-N.M` `-"1"."2"`

Row, column and table names

You can name infinite ranges by adding any of their cells to **keyword names:**

`row.N`	-> row
`col.N`	-> column
`table.N`	-> table

Naming options

These examples all do the same: they add a range to the name "N".

`N{…}` `{…}N` `N{…}N` `"N"{…}` `{…}'N'`
`'N'{…}"N"` `N.{…}` `{…}N.` `N.{…}'N'`
`{…}'N'.` `"N" . {…} 'N' .`

If a nested name is directly followed by a column separator, the last name becomes the cell's value:

```
1,"Name"."Value",2
1,"Name"{"Value"},2
```

the same

Formatting

Whitespace

Whitespace characters can be used in directives when escaped:

Tab	TAB	(09)
Newline	LF	(10) or CR LF (10+13)
Space	SP	(32) or NBSP (160)

```
!rowsep"SP"SP"TAB"SP"CRLF"
```

This example sets space as the column separator and tab or newline as the row separator. Any ASCII/Unicode character can be used by directives but whitespace characters must be escaped. If newline is disabled as rowsep, all whitespace is treated as insignificant (except when used by other directives) -> `!rowsep,::`

Escaping

Data of cells, names, directives and comments can be escaped using 4 alternative escaping methods:

Doublequotes	`"`	-> `"...""...'..."`
Single quotes	`'`	-> `'...''..."...'`
Backticks	`` ` ``	-> ``` `...``...` ```
Triple Backticks (ignores line break before first and after last line)	```` ``` ````	-> ```` ``` line 1 line 2 ``` ````

The escaping behavior can be changed using the **!escaping** directive.

`!escaping,csvs`	-> CSVS escaping (default)
`!escaping,csv`	-> Doublequote escaping
`!escaping,json`	-> JSON escaping

Comments

`//`	-> start of a comment that ends with rowsep
`/*` `*/`	-> start and end of a multiline comment

```
/* expected values:
- between 0 and +Inf = good
- 0, NaN, Empty = neutral
- between 0 and -Inf = bad */
Output[-1,-2,-Inf,-81 // all bad
NaN,0,,1] // NaN, +Inf, -Inf..
are supported in CSVS!
```

Line formatting

Lines can be joined and extended using these character sequences:

`::`	-> non-breaking row separator
`..`	-> line continues on next line

Types

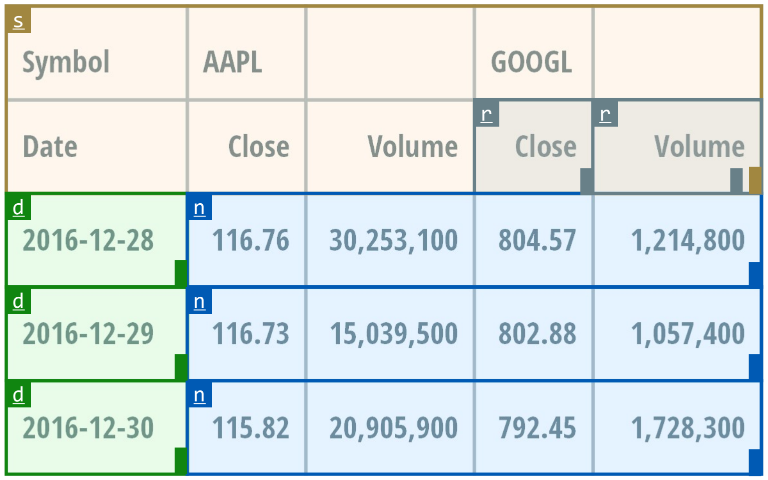
Why not make ? default and introduce ! for forced conversion?

T(…,…)

This is a typed range

Types work like names but they have meaning. They specify what type of data is contained within cells.

```
s(Symbol,AAPL,,GOOGL::Date,Close,Volume,r(B2),r(C2))
date(2016-12-28),n(116.76,30253100,804.57,1214800)
date(2016-12-29),n(116.73,15039500,802.88,1057400)
d(2016-12-30),num(115.82,20905900,792.45,1728300)
```



s Symbol	AAPL		GOOGL	
Date	Close	Volume	r Close	r Volume
d 2016-12-28	n 116.76	30,253,100	804.57	1,214,800
d 2016-12-29	n 116.73	15,039,500	802.88	1,057,400
d 2016-12-30	n 115.82	20,905,900	792.45	1,728,300

Rules

Crossing basic types (except with reference) results in the last assigned type winning. Optional types prevent #N/A errors after parsing.

`T?(…)` -> type is optional

Standard types

`s`	or	`string`	**String** (ASCII/Unicode)
`n`	or	`num`	**Number** (numeric types)
`b`	or	`bool`	**Boolean** ("true"or "false")
`u`	or	`url`	**URL** (string)
`d`	or	`date`	**ISO Date/Time** (see spec)
`t`	or	`time`	**ISO Time** (see spec)
`x`	or	`xldate`	**Excel Date/Time** (double)
`et`	or	`etime`	**Epoch Time** (Unix/Posix long)
`c`	or	`crypt`	**Base64 Encrypted Data** (base64)
`ba`	or	`base64`	**Base64 Encoded Data** (base64)
`e`	or	`email`	**Email address** (string)
`p`	or	`phone`	**Phone Number** (string/int)
`na`	or	`name`	**Person/Thing Name** (string)
`r`	or	`ref`	**Reference** (string)
`uu`	or	`uuid`	**Global Unique Identifier** (guid)
`ge`	or	`geo`	**Geo-Coordinates** (x|y = num|num)

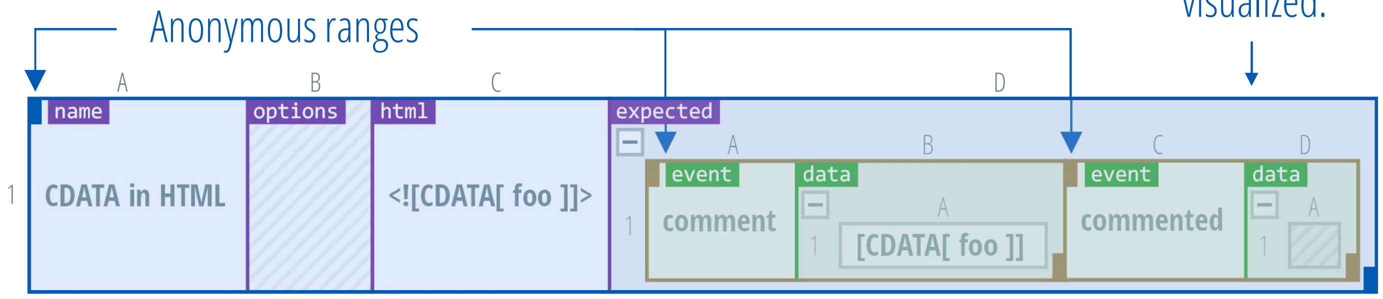
JSON is a special form of CSVS

```
!rowsep[],!colsep[","],!namequal[":"],!escaping["json"]
{
  "name": "CDATA in HTML",
  "options": {},
  "html": "<![CDATA[ foo ]]>",
  "expected": [
    { "event": "comment",      "data": [ "[CDATA[ foo ]]" ] },
    { "event": "commentend",  "data": [] }
  ]
}
```

Changes are effective after the line break

Objects and arrays can become anonymous ranges. Since JSON allows the {…} and […] forms, JSON data always becomes an anonymous range or table.

This is how this data looks visualized.



Rows or columns

Since JSON does not include topology information (columns and rows), JSON data can be loaded either as columns

```
!rowsep[],!escaping["json"]
!colsep[","],!namequal[":"]
```

or as rows

```
!rowsep[","],!escaping[json]
!colsep[],!namequal[":"]
```

disables column separator completely

JSON compatibility rule:

If a nested name is directly followed by a column separator the last nested label becomes a cell value:

```
…,"Name":"Value",…
…,"Name"{"Value"},…
```

the same

CSVS usage for config files and scripts

```
!namequal["="]::escaping[csvs] // insert this line at top of csvs file
servers = { login = "https://www.google.com/accounts/ClientLogin", test = "111.22.33.2" }
ping { host = ref(servers.test), n = 3 } // ref = reference another cell's value
curl {
  ref(servers.login) // first CURL argument = URL (Google Client login API)
  data-urlencode = { "Email=name@gmail.com", "Passwd=T0pS3cr3t!" }
  data = { "accountType=GOOGLE", "service=lh2" }, insecure = true
}
create_scheduled_task {
  name = "A task that runs every jan/mar/may/aug/oct/dec twice a month"
  enabled = true
  command = "C:\path\to\command.exe ${servers.test}" // string interpolation
  trigger {
    start_date = date(2017-01-01), start_time = time(06:30) // date and time are types
    period = monthly, months = [1,3,5,8,10,12], on = [15,last] // [] = table or array
  }
}
```