handlecsv

User Manual

Pablo Rodríguez

2017

http://www.handlecsv.tk

Dedicated to Hans Hagen for such powerful and awesome software pieces: ConTEXt and LuaTEX.

ἄλλος μὲν τεκεῖν δυνατὸς τὰ τέχνης,
ἄλλος δὲ κρῖναι τίν᾽ ἔχει μοῖραν βλάβης
τε καὶ ὠφελίας τοῖς μέλλουσι χρῆσθαι

one has the ability to beget arts, but the ability to judge of their utility or harm belongs to another

(Plato, *Phaedrus*, 274E)

Contents

Introduction 9

1 Before Your Start 11
2 Basic Usage 14
3 Conditional Processing 17
4 Loops inside Loops 21
5 List of Loop Commands 23

Conclusions 25

A Some LuaTEX Code Snippets 27
B Single–Document Processing 29

Introduction

The `handlecsv` module enables data merging for automatic document creation. The most common case for automatic document creation is mail merge.

Every user knows that ConTEXt is an awesome typesetting tool. With `handlecsv`—and the right scripts—, those beatifully typeset documents can be created automatically. This enables ordinary computer users to create high–quality typeset documents with no requirements at all about ConTEXt knowledge.

I have used `handlecsv` to develop a whole document creation system at my workplace. Although my employer may want me to keep the details not publicly available—and I will honor these indications—, my workmates use this document creation system each day at work.

No one at work knows that ConTEXt does the actual document generation. In fact, they only know that the have to enter data using *LibreOffice Calc* and select the right options in the script. Not even the guy from technical support knows exactly what ConTEXt is—which is my fault in explaining. And everyone seems to be happy with the new system.

The document creation system works under *Windows*—this is the operative system we have at work. But if we migrate to *Linux*, only the *Windows*–specific scripts will have to be translated.

The `handlecsv` module may be used to develop a full document merging system. This brief manual introduces to its usage.

Typographic Conventions

Italics are used to refer to intellectual works and registered marks. These include names of corporations, programs or programming languages. Italics may also emphasize words or expressions.

The monospaced typeface is used for everything that has to be typewritten by the user. This includes computer commands, program options and source code in many different formats.

Typewritten inline words will be hyphenated using an underscore instead of the standard hyphen. Underscore hyphenation aims to avoid confusion. Because `handlecsv`, `han-dlecsv` and `handle-csv` would be three different modules.[1]

ConTEXt and related programs have special logotypes. This is the reason why they aren't typeset in italics.

Code Snippets

Relevant code snippets—which contain more than four lines, or which are too complex to be typewritten without errors—are offered as comments in the PDF document. PDF comments enable direct copying and pasting for inmediate testing.[2] If code snippets are copied from text, also unwanted line and page breaks will be copied too.

Comments for code snippets only work in the electronic version from the PDF document. The printed version shouldn't even print the comment icon. The ePub format cannot include this goodie, but it doesn't really need it. Code snippets can be copied directly from text.

Comments

If you find errors in this document or you want to comment anything related to `handlecsv`, open an issue at `https://github.com/ousia/handlecsv/issues/new`.

Acknowledgments

I sincerely thank Jaroslav Hajtmar for the development of the `handlecsv` module. The composition of this text is a sign of gratitude for his excellent work. I also thank him for helping me in making the module available on the *ConTEXt Suite*. This important step will surely make `handlecsv` more known inside the ConTEXt community.

I want also to thank Hans Hagen for his extraordinary work in the development of both ConTEXt and LuaTEX. Without those excellent pieces of software, digital typesetting would be a much more difficult activity.

1 Before Your Start

A Disclaimer

The development of the `handlecsv` module is done by Jaroslav Hajtmar. Pablo Rodríguez has suggested improvements, but he didn't contribute any actual code to `handlecsv`. He only wrote the documentation. The errors this introductory manual may contain are only to be attributed to him.

B Requirements

To start using the `handlecsv` module, you need:

- The latest beta from the *ConTEXt Suite*.

  It includes the latest stable version from `handlecsv`.

- A spreadsheet program that can handle `.csv` files.[3]

C ConTEXt *Suite*

The first requirement is mandatory. Not only because it includes the module itself, but because it is the most developed and fixed version from ConTEXt.

The *ConTEXt Suite* is a portable distribution. So, it may be installed beside any other TEX distribution[4] without interferences. Installation instructions are provided at `http://wiki.contextgarden.net/ConTeXt_Standalone`.

D Spreadsheet Program

The spreadsheet program must be able to read and write `.csv` files with their character set enconded in UTF-8. Although they are many spreadsheet programs, I have tested the following two applications:

- *LibreOffice Calc* is the one I use at work. Main advantages are multi–platform availability and free licensing.

- *Microsoft Excel*, but I had only tested once briefly. I remember not knowing how to specify the character set for the `.csv` spreadsheet.

I must warn that I only use *Windows* at work and I don't have *Microsoft Excel* installed. I recommend that you check the compatibility yourself.

With *LibreOffice Calc*, you have to set reading and writing options. Import options should include UTF-8 as character set, semicolon as separator character and double quote mark as text delimiter. The same settings should be applied when exporting to `.csv` format.

E *The File Format*

The spreadsheet format is a pure text file with values separated by commas. In fact, `.csv` stands for "comma–separated values". But this file format has some drawbacks when compared to full spreadsheet format.

These limitations are related to the fact that `.csv` files are simple–text files. Some of these shortcommings are:

- `.csv` files contain only a single sheet.
- Special fields are lost when the file is saved. This happens even in the case that data were correctly saved in the pure–text spreadsheet.
- Formulas may be saved as formulas, but they won't work with ConTEXt without recreating them—either as LuaTEX code or as TEX math expression.

The use of formulas in merged documents isn't the common case. But they might come be extremely handy in invoice or budget generation. In that case, the field with formula should be recreated in ConTEXt using LuaTEX code.[5]

Special fields present a similar case. Standard special fields are numbers and date fiels. *LibreOffice Calc* automatically converts the values `04109` and `3/5/2015` into `4109` and `03/05/15`. The first sample is a postal code from Leipzig and the saved value would be a wrong postal code. The second value is a date, but you may require `3/5/15` or even `3.5.2015`—when dates are written in German.

The way to solve the invalid postal code is to create a command that adds a leading zero when the value includes only four digits. The pure TEX code for that would read:[6]

```
\def\PLZ#1{\ifnum#1<10000 0\fi#1}
```

In this case, the postal code should be invoked—assuming that postal code values are located in `\cC`:

```
\PLZ{\cC}
```

Converting dates to a standarized format is explained in *2. Date Format Normalization*.

2 Basic Usage

Although handlecsv merges data, its main usage is mail merging. Of course, nothing prevents you from using it in other scenarios you may imagine. But the common examples are from mail merging.

A The First Example

Imagine this basic spreadsheet, which has to be saved as a.csv:

```
"Name";"Surname";"Birthdate"
"John";"Smith";10/03/02
"Jane";"Amr";03/03/92
```

You may want to turn this data into the form:

```
Name Surname was born on Birthdate.
```

The required code for this output would be:

```
\usemodule[handlecsv]
\setheader
\opencsvfile{a.csv}

\starttext

\startbuffer[loop]
\cA\ \cB\ was born on \cC.\crlf
\stopbuffer

\doloopforall{\getbuffer[loop]}
\stoptext
```

B Module Loading

Some general comments on the code:

- \usemodule[handlecsv] is mandatory. Of course, you have to load the module to be able to use it.
- \setheader ignores the first row for data merging.

- `\setsep{;}` will change the character for cell separation.

  Semicolon is the default character. It doesn't need to be specified.

- `\opencsvfile{a.csv}` setups the file where data are taken from—`a.csv` in this sample.

  Avoid using blank spaces in file names. Characters outside the ASCII range should be avoided. This helps compatibility with *Windows* systems.

  Configuration commands—such as the previous ones—should be placed before the file loading command. This is required to parse the file after opening it.

C *Column Invocation*

There are two ways of invoking column contents in `handlecsv`:

1. You may use a command with the title from the first row. This is automatically generated by `handlecsv`. In the previous sample, commands would be: `\Name`, `\Surname` and `\Birthdate`.

   This option only works when the first row contains the column titles. `\setheader` has to be enabled. Otherwise, command names won't be taken from the title row, but from the first values.

2. You may use a command formed with the uppercase column letter prefixed with `\c`—from "column". This gives the three commands `\cA`, `\cB` and `\cC` used in the sample.

I strongly advise to use the second method, because of the following reasons:

- It is easy to invoke a column by a convention—such as `\cD`—than by its actual title—which may change from file to file.

  I'm not saying that column titles should be avoided. In fact, they are extremely useful when introducing data in the `.csv` file.

- If you use titles to invoke columns, you will have to avoid the following characters:

  – Blank spaces.

– Any character outside the ASCII range, such as accented characters or characters outside the Latin script.

In my experience, users feel more confortable with blank spaces and all required characters in column titles. So, it is easier also to write the title in full form and use column commands—such as `\cC`—in the ConTEXt source document.

D Loop Structure

The document automatically created with `handlecsv` may be as complex as you want. You should define the document as you would define any other ConTEXt source document.

Actual data merge is enabled by looping the document portion where those data have to change for each output. The output may be a different paragraph or a different page.

The loop structure is simple. You have to wrap the data to be merged in a buffer. And then, you have to add a command that actually loops that buffer. From the previous sample, those two parts read:

```
\startbuffer[loop]
\cA\ \cB\ was born on \cC.\crlf
\stopbuffer

\doloopforall{\getbuffer[loop]}
```

The following conventions have to be considered:

- You may use the name you want for the buffer with data.

  But the same name should be used in both `\startbuffer` and `\getbuffer`.

- This name should not be used for defining or invoking another buffer.

  The name match is required to get the right buffer and not another one.

- A cell is considered to be empty when it only contains spaces.

- A row is empty when none of its cells has any content other than spaces.

3 Conditional Processing

Automatic document creation may be easily improved with conditionals.

A Basic Conditionals

The most basic ConTEXt conditional for cell content is to check whether it has any text. These conditionals are two: `\doiftext` and `\doiftextelse`. They may be used as it follows:

```
\doiftext{\cA}{it has text}

\doiftextelse{\cA}{it has text}{it doesn't have any text}
```

There is no `\doifnottext`, but `\doiftextelse` with no positive consequence may be used. It could read:

```
\doiftextelse{\cA}{}{it doesn't have any text}
```

Another conditional is to check a cell equals a certain value. These conditionals are: `\doif`, `\doifnot` and `\doifelse`. Usage examples are:

```
\doif{\cD}{male}{Mr}

\doifnot{\cD}{male}{Ms}

\doifelse{\cD}{female}{Ms}{Mr}
```

There is an important reminder about string equality: upper and lowercase letters are different characters. `Male` and `male` are different strings. The same way, `F` and `f` aren't equal.[7]

B Conditional Looping

In some cases, you might want to deal with some part of your spreadsheet. This is when conditional looping comes extremely handy.

Inclusion

Imagine that your file contains a list of customers from Germany, but you only need to generate documents for the ones residing in Munich. Your final loop should read:

```
\doloopif{\cD}{==}{München}{\getbuffer[loop]}
```

The structure of the conditional loop is the following:

```
\doloopif{value1}{operator}{value2}{action}
```

1. As in all loops, it has an action to perform when condition is met.

2. The condition checks whether two selected values match the operator equal, different (non–equal), greater than, less than, equal or greater than, and equal or less than.

   Texts can be checked to be equal or different. Since they are compared as strings, equality is met only when the same character sequence is compared, as explained before.

   Besides a string match, to select or avoid a particular name—for persons or places—, the most interesting feature is the ability to check whether a cell is empty or not.

   Numerical comparision allows to check whether a value is greater or less than other, besides being equal or different.

Exclusion

Exclusion is the opposite operation from inclusion. Using it, you could also generate documents for all customers in the spreadsheet, excluding those ones from Berlin. The final loop reads:

```
\doloopif{\cD}{~=}{Berlin}{\getbuffer[loop]}
```

As you can see, the conditional loop is the similar to the previous one. The only difference is that the requirement that triggers the loop is that both values have to be different.

C *Nested Conditionals*

There may be some cases in which you have to extract the data from different fields. In this case, nesting conditional loops is extremely useful.

You might want to generate documents for customers in the Austrian federal state of Vorarlberg.[8] The spreadsheet may contain a column with the state infor-

mation. But if it doesn't, you may extract this information from the postal code column.

```
\doloopif{\cC}{>=}{6700}{%
    \doloopif{\cC}{<}{7000}{\getbuffer[loop]}}
```

The postal code numbers for the state of Vorarlberg have values in the range from 6700 to 6999. To express the range, we need two conditionals. To get the action triggered, we need to have both conditions met.

1. The first conditional loop checks whether the postal code number is equal or greater than 6700.

2. The action that this first conditional loop triggers is the second conditional loop.

3. The second conditional loop checks whether the postal code number is less than 7000.

4. The second conditional triggers the action, which gets the buffer. Since the second conditional is nested, the action can be only triggered if and only if both conditionals are met.

5. Order of conditions in this case is irrelevant. You might check the highest value first, and then the lowest one.

As in all three conditionals, `\getbuffer[loop]` should be enclosed as consequence for a condition. In more complex cases, nothing prevents you from nesting how many conditional loops you may need.

D *LuaTEX Deployment*

LuaTEX might help you to formulate more complex conditions.

```
\doloopif
    {\ctxlua{if string.len("\cC")==5
            and string.sub("\cC",1)=="8" then
            context("x") end}{~=}{}
    {\getbuffer[loop]}
```

The approach is different. The double condition for the looping is that the postal code contains five characters and the first digit is `8`.[9] This positive condition

has the consequence of getting data from the buffer. The condition is met when there is text in from the `\ctxlua` command. If the condition isn't met, the loop will continue to next row or the end of the file.

In the previous sample, there are two conditionals. The first or external one is from ConTEXt, and the second or internal one is in *Lua*. The second conditional is double: only if the field `\cC` is five characters long and the first one is `8`, the expression will output some text. The loop gets the buffer when the second conditional outputs text, when its result is other than empty.

4 Loops inside Loops

The most common case for document merging is one that has a unique loop. But there may be documents that need loops inside the main loop. Imagine that you have to add a list of requirements to each addressee of a letter, being that requirements list different for recipient. Or you might want to add different attachments—using the ConTEXt command `\attachment`—to each single document for each recipient.[10]

The sample I'm going to show is a list of subjects and grades for each student:

```
\setuppapersize[A6]
\setuppagenumbering[location=]
\usemodule[handlecsv]
\setheader

\startbuffer[notes]
    \item \cC: \cD.
\stopbuffer

\startbuffer[text]
This is to certify that \cA\ \cB\ got the following grades:
\startitemize
    \getbuffer[notes]\nextrow
    \doloopwhile{\cA}{}{\getbuffer[notes]}
\stopitemize
\page
\stopbuffer

\opencsvfile{grades.csv}

\starttext
\doloopif{\cA}{~=}{}{\getbuffer[text]}

\stoptext
```

There are three loops here, although two loops share the buffer they loop. Here are the conditions of the previous sample:

1. Colums are: name, surname, subject and grade.[11]

2. The main loop is triggered only when the first column (`\cA`) has text in it. The student must have a name to generate a certificate.

3. After the row with the student name and surname—with a subject and grade—, there may be other rows with only subject and grade, which belong to the student name contained in the previous row.

4. A loop is needed to get those lines without name, which are the lines that only contain the different subjects and grades from the student.

   `\doloopuntil{\cA}{}{\getbuffer[notes]}` would get subject and grade from the line where name and surname are contained. But if the next row includes a new student, that loop would include the subjects and grades from that next student. After all, the command loops until `\cA` is empty.

   With the data from this sample, it won't make a difference, since all students have more than a single subject. But as a general rule, it is safer not to invoke a loop when it could be triggered when unintended.

   This second loop is triggered while the name is empty—while `\cA` doesn't contain any text or data.

5 List of Loop Commands

`\doloopforall{action}` it loops the action—normally, a buffer—for all non–empty lines.

`\doloopfromto{from}{to}{action}` it loops the action from a row number to another row.

Loops may advance in decreasing order, when `{to}` is greater than `{from}`.

`\doloopif{value1}{operator}{value2}{action}` it loops the action if a condition is met comparing both values.

Basic operators for comparision are: `<`, `>`, `==`[12], `~=`, `>=` and `<=`.

`\doloopifnum{number1}{operator}{number2}{action}` it loops the action if a condition is met comparing both numbers.

It is exactly the same conditional loop as the previous one, but only numbers are compared. Comparision operators are the same.

`\doloopuntil{value1}{value2}{action}` it loops the action until the condition is met—that both values are equal.

The loop is finished when the condition is met.

`\doloopwhile{value1}{value2}{action}` it loops the action while the condition is met—that both values are equal.

Contrary to `\doloopuntil`, this loop is finished when the condition is not met.

`\doloopfornext{number}{action}` it loops the action for the next `{number}` of lines.

With a negative integer, loop goes backwards.

Conclusions

`handlecsv` is an extremely useful module. But utility is not a universal value, it is inherently related to each particular usage. Because of that, I want to describe some situations in which it may be used. Other people may find other interesting uses of `handlecsv` not considered here.

My basic usage has been generation of office documents. It should be noted that I wrote the ConTEXt templates. A small team over ten people use them for their everyday work. Again, standard usage only involves data saving using *LibreOffice Calc* and running extremely basic *Windows* scripts with simple options.

Documents that can be generated automatically from the same data:

- Office letters.
- Envelopes or labels for those letters.
- Return–receipts for all letters.
- Lists of recipients of those letters.

Although I have no experience with this, `handlecsv` could be used to generate budgets or invoices. In that case, I think some LuaTEX code may be required. Also automated `.csv` conversion from fully–featured spreadsheets would be advised.

A Some LuaTEX Code Snippets

1 Adding Missing Zero to Postal Code

This simple code snippet adds a leading zero to postal code. It works in any country with postal codes with five digits. It only checks whether the command—such as `\cC`—contains four characters and adds a leading zero to those digits.

```
\startluacode
function document.plz(str)
    if str:len()==4 then
        str = "0" .. str
    end
    context(str);
end
\stopluacode

\unexpanded\def\PLZ#1%
   {\ctxlua{document.plz("#1")}}
```

It should be invoked as a command for the column with the postal code:

```
\PLZ{\cC}
```

Of course, it could be adapted to add a leading zero in postal–code systems using four digits by replacing `str:len()==4` with `str:len()==3` in the snippet above. This may be useful for Austria, Norway or Slovakia.

2 Date Format Normalization

The basic date formalization should convert from a generic format—at least, in *LibreOffice Calc* to a proper date.

```
\startluacode
function document.standarddate(str,insep,outsep)
    if str:find(insep)==3 and str:find(insep, 4)==6 then
        local day = str:sub(1,2)
        local month = str:sub(4,5)
        local year = tonumber(str:sub(7,8))
        if str:len()<9 and year>=50 then year=year+1900 end
```

```
        if str:len()<9 and year<50 then year=year+2000 end
        context(day..outsep..month..outsep..year)
    end
end
\stopluacode

\unexpanded\def\formatfulldate#1#2#3%
   {\ctxlua{document.formatdate("#1","#2","#3")}}

\def\standarizedate#1{\formatfulldate{#1}{/}{/}}
```

The LuaTEX performs the following steps:

1. Date separator is to be met in the third and sixth position.
2. The first two characters before the first date separator are the day.
3. The first two characters after the first date separator are the month.
4. The first two characters after the second date separator are the year.
5. If the year has only two digits, it will be converted simply to a four–digit format.
6. Year values greater than 50 will belong to the twenty–first century, and year values equal or less than 50 will belong to the twentieth century.

In that case, date fields should be invoked:

```
\standarizeddate{\cF}
```

In some cases, it may be required to add a leading zero to the date[13]. But that is left as an exercise to the reader, since it was explained in the previous section.

B Single–Document Processing

When you have to print the output file, it is better to have all pages to be printed in a single PDF document. To the best of my knowledge, *Acrobat* requires multiple documents to be printed individually.

But there may be scenarios where each row in our spreadsheet requires an individual PDF document. Digitally–signed letters or certificates require single documents with no extra content, even if they may be also printed. More generally, when a document is going to be consumed in electronic format, it must be single.

Imagine attendance diplomas after participating in a conference. If they are to be only printed, it is better to have a single file with all diplomas. But if these diplomas are going to be sent to participants, each document must contain only one diploma.

In order to get the single processing of multiple files, we need two source files. The first one should be the template document with contents and layout for the final document. The second file has to trigger compilation of each individual document.[14]

1 *Certificate*

A sample minimal certificate may read:

```
\setuppapersize[A8, landscape]
\setupbodyfont[helvetica, 13.75pt]
\setuppagenumbering[location=]
\setupwhitespace[big]
\usemodule[handlecsv]
\opencsvfile{participants.csv}
\starttext
\startalign[middle]To Whom It Might Concern\stopalign

This is to certify that \cA\ \cB\ completed the course.
\stoptext
```

This is the standard source, exactly the same as you would made with any other document using `handlecsv`. Of course, it may be as complex as you want.

To get independent PDF documents for each participant[15], the sample should be saved as `certificate.tex`.

2 *Execution*

After the document to be combined is created, you need to create a document that performs a loop. The loop includes a command that runs ConTEXt to compile each row in a different file.

```
\usemodule[handlecsv]

\startbuffer[single-certificates]
\executesystemcommand{contextjit --purgeall
--result=certificate-\lineno.pdf
--arguments="MainLinePointer={\lineno}" certificate.tex}
\stopbuffer

\starttext
\opencsvfile{participants.csv}
\startitemize[n]
\doloopif{\cA}{~=}{}{\getbuffer[single-certificates]\item \cA}
\stopitemize
\stoptext
```

This source file uses `handlecsv` in standard way. It creates a buffer, which is looped given certain conditions. In this case, that `\cA`—the participant's name—does have actual content. But there are other considerations:

1. The looped buffer is a ConTEXt compilation command.

2. It is essential to rename the output file to avoid being overwritten by next output file. `--result=certificate-\cA.pdf`[16] is also possible. But there are two issues.

   a. If the name contains spaces, they have to be removed.[17]

   b. If two different persons have the same name, the second generated certificate would erase the first generated one.

3. What makes the data loop is `--arguments="MainLinePointer={\lineno}"`.

   Otherwise you may get as many single documents as you need, but all will have the data from the first row.

It is important that you compile this execution document only once, such as explained below. Otherwise, ConTEXt will generate individual documents as many times as the execution source is run. This is only a waste of time, since individual documents are compiled with so many runs as each requires. The simple file list doesn't require more than one run to include all generated files.

```
contextjit --purgeall --runs=1 execution-source.tex
```

3 Why `contextjit --purgeall`?

In both previous invocations of ConTEXt, you may have noticed that I suggested `contextjit --purgeall source.tex` instead of `context source.tex`. There are two main reasons for that.

1. `contextjit` is faster than `context`. At the time of writing, compiling this document with `contextjit` takes about 17 seconds, with `context` takes about 20 seconds.[18]

2. `--purgeall` removes extra files required from compilation. This migh not be a wise choice when generating a single file, but I think it is better to remove them when having multiple files from the same source.

My previous practices have reasons behind. You might want to check them, to see whether they fit to your task.

4 Better Output File Names

In some cases, you may want to have more descriptive—or simply useful—file names for the individual PDF documents generated with the method described in this appendix. You may use `Lua` to get this.

In my case, I use a small command to remove spaces form the contents of a field, since *Windows* didn't allow me to add spaces to file names.

Another useful goodie was to be able to number the files in order, but adding leading zeros where necessary. I mean, PDF documents named from `certificate-1` to `certificate-100` are fine for people. But for computers, files have to named `certificate-001` to `certificate-100`.

This command adds as many leading zeros as required—based on the total number of rows: `\zeroedlineno`.[19] To use it, replace `--result=certificate-\lineno.pdf` with `--result=certificate-\zeroedlineno.pdf`.

This is useful when you have to print all single documents. Digitally signed documents require that. To have all of them in paper, you may merge into a single PDF document and print this common document. Of course, this won't work with digital signatures, but it won't be problematic for having the paper version of these documents. Digital signed documents have to be preserved. This is only a simple way to print them.[20]

1 This is the reason why `handlecsv` is written always with its first letter lowercase. `Handlecsv` and `handlecsv` would be two different modules too.

2 Of course, you need a viewer that can handle PDF comments. Besides *Acrobat*, *Evince* and *Okular* can handle them. *SumatraPDF* handles PDF comments, but you have to copy its contents not opening the comment, but displaying the options right–clicking with your mouse in the comment icon and copying the contents. *MuPDF*—as far as I know, for *Windows* or *Linux*—cannot copy contents from PDF comments.

3 Of course, you may edit `.csv` files with a pure text editor, if you insist. But a proper spreadsheet program will simplify the task.

4 *TEX Live*, *MikTEX* , *MacTEX* or even another version from the *ConTEXt Suite* itself.

5 Final results may be saved instead of formulas. But in that case, formulas must be rewritten if the spreadsheet has new data.

6 Command kindly provided by Juan José Torrens from the Spanish mailing list on TEX . If you'd rather use LuaTEX code, see *1. Adding Missing Zero to Postal Code*.

7 Case–changing commands don't help here: `\WORD{HAHA}` isn't equal to `\WORD{haha}`.

But a bit of LuaTEX code helps the `\doifelse` command to ignore the case difference:

```
\def\GoUpper#1{\ctxlua{string.upper("#1")}}
```

The `\doifelse` command should be rewritten into:

```
\doifelse{\GoUpper{\cD}}{\GoUpper{female}}{Ms}{Mr}
```

Wolfgang Schuster corrected my first wrong definition.

8 It was extremely tricky to find an administrative division for a country in the European Union that has only one continuous range of postal code numbers. And there might be exceptions with some places outside Vorarlberg having a postal code number which belongs to that Austrian federal state.

9 A postal code number may contain four digits, due to automatic removal of leading zeros. When postal code aren't required to contain five digits, requesting their first character to be `8` may lead to include customers from an area in Sachsen in the selection. This is because their postal codes range between `08000` and `08999`.

10 Although this would require using *B. Single–Document Processing*, the step described in *2. Execution*. Otherwise, attachments will be indeed added to the document, but a single PDF document will contain all attachments and all letters.

11 Contents of `grades.csv` may be:

```
"Name";"Surname";"Subject";"Note"
```

```
"Name 1";"Surname 1";"Subject 1";"Note 1"
;;"Subject 2";"Note 2"
;;"Subject 3";"Note 3"
;;"Subject 4";"Note 4"
"Name 2";"Surname 2";"Subject 5";"Note 5"
;;"Subject 6";"Note 6"
;;"Subject 7";"Note 7"
;;"Subject 8";"Note 8"
;;"Subject 9";"Note 9"
"Name 3";"Surname 3";"Subject 10";"Note 10"
;;"Subject 11";"Note 11"
;;"Subject 12";"Note 12"
;;"Subject 13";"Note 13"
;;"Subject 14";"Note 14"
"Name 4";"Surname 4";"Subject 15";"Note 15"
;;"Subject 16";"Note 16"
;;"Subject 17";"Note 17"
;;"Subject 18";"Note 18"
;;"Subject 19";"Note 19"
;;"Subject 20";"Note 20"
;;"Subject 21";"Note 21"
;;"Subject 22";"Note 22"
;;"Subject 23";"Note 23"
;;"Subject 24";"Note 24"
;;"Subject 25";"Note 25"
;;"Subject 26";"Note 26"
"Name 5";"Surname 5";"Subject 27";"Note 27"
;;"Subject 28";"Note 28"
;;"Subject 29";"Note 29"
;;"Subject 30";"Note 30"
;;"Subject 31";"Note 31"
;;"Subject 32";"Note 32"
```

12 For those not familiar with programming, **=** is not the same as **==**.

Single equal sign is used to asign a value to a variable, such as in:

```
a = 5
name = "John"
```

Double equal sign is used when checking the equality of two values such as in:

```
if 5 + 3 == 4 + 4
```

The single equal sign is to be read "make equal to", such as `a = 5` is read "make *a* equal to *five*". The double equal sign is to be read "is equal to", such as `if 5 == 2 + 3` is read "if *five* is equal to *two plus three*".

Since we deal with conditions and we are comparing, the use of double equal sign is mandatory here.

13 *LibreOffice Calc* removes the leading zeroes in dates. At least, with the Spanish locale.

14 Actually, the loop goes for a single line and ConTEXt renames the resulting file.

15 The contents for `participants.csv` might be:

```
"John";"Smith"
"Jane";"Doe"
"Werner";"Müller"
"María";"Rodríguez"
```

16 File name for output file must always end with the `.pdf` extension.

17 We can replace blank spaces with underscores with the following command:

```
\def\NamesWithoutSpaces{%
  \ctxlua{string.gsub(thirddata.handlecsv.getcellcontent(1,
   thirddata.handlecsv.gCurrentLinePointer), " ", "_")}%
}
```

In that case, `--result=certificate-\cA.pdf` should be renamed to `--result=certificate-\NamesWithoutSpaces.pdf`.

18 Results are a rough average of more than five compilations using each command.

If you think compilation times are too slow, consider that my laptop is already a decade old. Your times would be much better.

19 To avoid problems with blank lines and wrong line numbers, use `\removeemptylines`, right after `\opencsvfile`.

20 The standard license from *Adobe Acrobat* doesn't allow the printing of multiple documents except individually. The software forces the user to print PDF documents that way. Merging multiple documents is the only way to have them printed at once.

This document was generated with `pandoc` (`http://pandoc.org/`)
and typeset with ConTEXt (`http://contextgarden.net/`).

TEX Gyre Pagella, *GFS Didot* and *Cousine* were
the typefaces deployed for the PDF document.