
datatool v3: Performance, localisation, databases and more

Nicola Talbot

Abstract

A major new release of `datatool` (version 3.0) has improved performance with added localisation support. This article gives an overview of key new features.

1 Introduction

In December 2022, following on from a major update to `glossaries-extra` (and intertwined `mfirstuc`, `glossaries` and `bib2gls` changes), I finally started a long-overdue rewrite of `datatool` [7]. A large part of the new code uses \LaTeX 3, although there are still a few old commands that haven't been updated. Two years later, in March 2025, I finally released `datatool` version 3.0. Unfortunately, I slipped up on the packaging and accidentally missed out the new files. Version 3.0.1 followed on soon after as a bit of a damp squib. I'm sorry for any disruptions caused by this.

The `datatool` bundle consists of the following:

- the main `datatool` package;
- the `databar` package for plotting bar charts;
- the `datapie` package for plotting pie charts;
- the `dataplot` package for plotting 2D line charts or scatter plots;
- the `person` package for mail merging or document templates (added 2009);
- the `databib` package (added in August 2007, but it's better to use `biblatex` [2] now);
- the `datagidx` package (added in January 2013, but it's better to use the `\printnoidxglossary` command provided with `glossaries` [9] now).

The underlying package that's used by all the above is `datatool-base`. This deals with parsing (to determine data type and numeric value), sorting, and providing an interface to lower-level arithmetical functions.

Originally, the commands provided by the `fp` package [3] were used to perform floating point arithmetic. In version 2.10, the `math` package option was added to switch from `fp` to `pgfmath` [4], if preferred. As of version 3.0, there are new options: `math=13fp` (use `13fp` functions) or `math=lua` (use `\directlua`). The new default is `lua` (if \LaTeX) or `13fp` (otherwise).

2 Datum variables

In the context of `datatool`, a “plain number” is a value that consists of a sequence of digits 0–9, optionally prefixed by a single `+` or `-`, and (for decimals)

using a decimal point to separate the integer part from the fractional part. For example, 82145 (integer) or 48271.3 (decimal).

A “formatted number” is a number that is formatted using the designated number group separator and decimal character, where the number may optionally have a currency symbol before or after it. The default is a group separator of comma and a decimal point. For example, 82,145 (integer) or 48,271.3 (decimal) or `\$48,271.30` (currency).

Arithmetic commands that expect formatted numbers in the arguments, such as `\DTLadd`, need to parse the values, perform the appropriate calculation, and then format the result. If the result is used in another calculation, the formatted value would then have to be parsed.

With v3.0, the result is now stored as a “datum variable” which contains not only the formatted text but also the plain number value (if applicable). This reduces the amount of parsing required.

The new command `\DTLparse` may be used to parse some text, which may or may not be numerical. The result is a datum variable, and the component parts (formatted text, numeric value, currency symbol and data type) can all be extracted. This underlying function is used by all parsing commands.

For example, with the default settings:

```
\DTLparse\result{\$48,271.30}
Formatted: \result.
Currency symbol: \DTLdatumcurrency{\result}.
Value: \DTLdatumvalue{\result}.
Type:
\DTLgetDataTypeName{\DTLdatumtype{\result}}.
```

This produces:

```
Formatted: $48,271.30. Currency symbol: $.
Value: 48271.30. Type: currency.
```

The currency symbol needs to be declared. Common symbols are added by default. The following hides the currency symbol:

```
\newcommand{\mypound}{\textbf{\large\pounds}}
\DTLparse\result{\mypound 48,271.30}
Formatted: \result.
Currency symbol: \DTLdatumcurrency{\result}.
Value: \DTLdatumvalue{\result}.
Type:
\DTLgetDataTypeName{\DTLdatumtype{\result}}.
```

This produces:

```
Formatted: £48,271.30. Currency symbol: .
Value: . Type: string.
```

The value and currency symbol are both empty since the data is considered non-numeric, because the

parser doesn't recognise `\mypound` as a currency symbol. `\result` will only expand as far as the formatted value. So if `\edef\tmp{\result}` was added to the end of the last example, `\mypound` would not be expanded. For a full expansion, you need `\DTLusedatum{<datum-var>}`.

If the original value needs reformatting (for example, the source the data was obtained from was a little untidy) then `\DTLparse` can be instructed to do this:

```
\DTLsetup{numeric={auto-reformat}}
\DTLparse\result{-\$12,345.328}\result.
\DTLparse\result{\$-12345.3289}\result.
```

This produces:

```
-$12,345.33.  -$12,345.33.
```

This has adjusted the sign and rounded to two decimal places.

Scientific notation is now supported. If the auto-reformat setting is on and `siunitx` [11] has been loaded, then scientific notation will be reformatted with `\si`. The original value can again be extracted:

```
\DTLsetup{numeric={auto-reformat}}
\DTLparse\result{1.3e4}
Formatted: \result.
Value: \DTLdatumvalue{\result}.
```

This produces (assuming `siunitx` is also loaded):

```
Formatted: 1.3 × 104. Value: 1.3e4.
```

Although `datatool-base` is mostly concerned with parsing, there are also commands for converting from plain numbers to decimal or currency:

```
\DTLdecimaltocurrency{12345.6}{\price}
Price: \price.
\DTLdecimaltolocale{12345.6}{\decimalval}
Decimal: \decimalval.
```

This produces:

```
Price: $12,345.60.  Decimal: 12,345.6.
```

These commands are used by the auto-reformat feature (but not for scientific notation).

3 Localisation

There is now also localisation support. This is separated into support for regions (currency and number formatting, see Section 3.1) and languages (see Section 3.2). Searching for the appropriate localisation files is dealt with via `tracklang` (see, for example, [5]). Therefore, make sure you also have an up-to-date version of `tracklang` [10].

The localisation support is a little more complex than for `datetimer2` (for which `tracklang` was originally written). Suppose an English document that doesn't use either `babel` or `polyglossia` has:

```
\usepackage[locales={en-BE,en-GB,en-US}]
{datatool}
```

This requires support for English in three regions: BE (Belgium), GB (United Kingdom) and US (United States). The packages `datatool-regions` [8] and `datatool-english` [6] are needed, and this document will load the region files `datatool-BE.ldf`, `datatool-GB.ldf`, and `datatool-US.ldf`, and the language file `datatool-english.ldf`.

In this case, there are no `\captions...` hooks, but each LDF file provides its own hook to switch in this situation:

- `\DTLenLocaleHook` (switch to English orthography);
- `\DTLBELocaleHook` (switch to Belgian number formatting and currency);
- `\DTLGBLocaleHook` (switch to UK number formatting and currency);
- `\DTLUSLocaleHook` (switch to US number formatting and currency).

If a language package has been loaded and `tracklang` has a mapping from its own internal labelling system to the dialect label used by the language support, then these hooks will be added to the captions hook.

Some regions with multiple official languages may have different number formatting according to the language. For example, Canada may have a decimal point in English but a decimal comma in French. This is dealt with by the region file checking for the existence of a function that combines both the language tag and region tag in its name. The applicable language bundle needs to supply support for that specific language and region combination.

For example, `datatool-english` provides the file `datatool-en-CA.ldf`. If French localisation support were to be developed, it would not only need to provide `datatool-french.ldf` but also `datatool-fr-CA.ldf` (for Canadian French), `datatool-fr-BE.ldf` (for Belgian French), etc.

However, for the most part, the language and region files are independent of each other, which is why `en-BE` may be specified as a locale, even though there's no "en-BE" LDF file.

Suppose now that you have a document where `tracklang` only detects the root languages:

```
\documentclass[french,dutch,german,english]{book}
\usepackage{babel}
\usepackage[locales={BE}]{datatool-base}
```

In this case, `datatool-base` will add the region BE to each of `tracklang`'s dialects that don't have an associated region. Now `\DTLenLocaleHook` can be added to `\captionsenglish` but the region hook will also be added.

If all tracked dialects already had an associated region, then `und-BE` would be added to the list of tracked dialects instead. In that case, the `\DTLBELocaleHook` would need to be used explicitly in order to switch to the settings for that region, since there’s no applicable language hook.

In the following, the `babel` dialects already have an associated region so the `locales` option isn’t necessary:

```
\usepackage{babel}
\babelprovide{english-ca,french-ca}
\usepackage{datatool-base}
```

3.1 Regions

Region support is provided by the package `datatool-regions`, which needs to be installed separately. There are currently only a limited number of available regions. More may be added via pull request on GitHub.¹ There is an interactive Perl script provided to create a file based on your responses.

Each region file can provide options specific to that region. These can be set with:

```
\DTLsetLocaleOptions{<module>}{<key=value list>}
```

If you want to make use of these options, then switch on auto-reformatting so that the style commands governed by those options can be included in the formatting.

For example, the GB region has a numeric style called “education” that will allow a normal space or a thin space as the number separator when parsing, but will use `\`, (thin space) when formatting. (The default is the “official” style, which uses a comma number group separator.)

```
\DTLsetup{numeric={auto-reformat}}
\DTLsetLocaleOptions{GB}{number-style=education}
```

The following switches to each region in turn, to parse and reformat according to that region:

```
\DTLGBLocaleHook
\DTLparse\priceGB{£12 345.239}
\DTLUSLocaleHook
\DTLparse\priceUS{\$54,321.989}
\DTLBELocaleHook
\DTLparse\priceBE{24.678,42€}
```

All regional-sensitive commands can be reset:

```
\DTLresetRegion
\DTLdecimaltocurrency{12345.6}{\price}
```

Now each of the earlier datum variables are displayed in the text along with their embedded value:

```
No region price: \price.
(Value: \DTLdatumvalue{\price}.)
BE price: \priceBE.
(Value: \DTLdatumvalue{\priceBE}.)
```

¹ github.com/nlct/datatool-regions

```
US price: \priceUS.
(Value: \DTLdatumvalue{\priceUS}.)
```

```
GB price: \priceGB.
(Value: \DTLdatumvalue{\priceGB}.)
```

This produces:

```
No region price: ∅12,345.60. (Value: 12345.6.)
BE price: 24.678,42€. (Value: 24678.42.)
US price: $54,321.99. (Value: 54321.989.)
GB price: £12 345.24. (Value: 12345.239.)
```

Changing regional settings won’t have an effect unless or until the specified region is active:

```
\DTLUSLocaleHook
\DTLsetLocaleOptions{US}{
  currency-symbol-prefix,
  currency-symbol-sep=thin-space
}
```

```
Prefixed US price: \priceUS.
```

```
\DTLGBLocaleHook
\DTLsetLocaleOptions{GB}{number-style=official}
No change to GB price separator: \priceGB.
VAT: \DTLmul\vat{\priceGB}{0.2}\vat.
(Value: \DTLdatumvalue{\vat}.)
```

This produces:

```
Prefixed US price: US$ 54,321.99.
No change to GB price separator: £12 345.24.
VAT: £2,469.05. (Value: 2469.0478.)
```

While the options that affect the way the currency is displayed may alter content that has already been formatted, options that affect the number group and decimal characters won’t change already formatted content. In general, it’s best to set up all the localisation options in the preamble before any data parsing takes place.

3.2 Language

The language files not only provide fixed text translations (as is usual for language support) but also provide support for the language’s alphabet when sorting. Old versions of `datatool-base` provided `\dtsortlist` for sorting a comma-separated list according to a comparison handler function. Modern \LaTeX kernels provide `\clist_sort:Nn` which is far more efficient than the old method used by `datatool-base`. With version 3.0, `\dtsortlist` has been reimplemented to use `\seq_sort:Nn`. However, there is a new command that works better with localisation:

```
\DTLsortwordlist{<clist-var>}{<handler-cs>}
```

There is also a lower-level command for sequence variables (now used by `\printnoidxglossary`):

```
\datatool_sortwordseq:NN <seq-var> <handler-cs>
```

In this case, the handler function isn't a comparison macro but is used to preprocess the sort value. It works along a similar principle as Java's collation keys (which is used by `bib2gls`). In the case of Java, the sort value is converted into a series of bits. Since TEX doesn't provide an array data-type, a token list is used instead, where each token's character code represents a subset of bits (eight for single-byte tokens).

Bear in mind that we are only concerned with the character code of each token. The token list is not intended to be typeset and may include control codes (with category code "other"). A sequence variable is then set so that each item consists of both the pseudo-byte array token list and the original value. The sequence is then sorted using a comparison function that extracts the token list and performs a simple character code comparison. Afterwards, the original values can be extracted from the sorted sequence.

There are four predefined handler functions:

```
\DTLsortwordhandler{⟨original⟩}{⟨tl-var⟩}
\DTLsortwordcasehandler{⟨original⟩}{⟨tl-var⟩}
\DTLsortletterhandler{⟨original⟩}{⟨tl-var⟩}
\DTLsortlettercasehandler{⟨original⟩}{⟨tl-var⟩}
```

Each function converts $\langle original \rangle$ into a pseudo-byte array that's stored in the token list variable $\langle tl-var \rangle$. First a protected expansion is applied to the original content. The letter handlers then strip all hyphens and spaces. They then all use the default handler that applies the localisation support. Finally, the result is purified to remove any formatting commands and the case-insensitive handlers convert to lower-case.

Each language file needs to redefine (in its hook):

```
\DTLCurrentLocaleWordHandler{⟨tl-var⟩}
```

to convert the content of the supplied token list variable. The trick here is to ensure that any letters or punctuation that may be used in the language are converted to one or more characters that reflect the relative order in that language's alphabet. Support for foreign letters may be useful, but since this is done by regular expression, the more complex the expression, the longer the processing time.

English is quite straight-forward. The basic Latin characters can remain as they are. Common letters from the Latin-1 Supplemental block can be converted to their unaccented counterpart. For example, è, é, ê and ë can be replaced with e, and æ can be replaced with ae. Punctuation characters are prefixed with `0x22`. This ensures that they are all placed before any letters. Currency signs are all prefixed with `0x24`, which keeps them all together. If the conversions result in identical sort values, a

character code comparison of the original values will be used to differentiate them. For example:

```
\DTLenLocaleHook
\newcommand{\mylist}{r  sum  ,zebra,  therial,
resupply,resume}
\DTLsortwordlist{\mylist}{\DTLsortwordhandler}
\DTLformatlist{\mylist}.
```

```
  therial, resume, r  sum  , resupply and zebra.
```

The language support can be reset. This shows the result if there's no localisation:

```
\DTLresetLanguage
\newcommand{\mylist}{r  sum  ,zebra,  therial,
resupply,resume}
\DTLsortwordlist{\mylist}{\DTLsortwordhandler}
\DTLformatlist{\mylist}.
```

This produces:

```
resume, resupply, r  sum  , zebra &   therial.
```

Control codes can be useful if a diacritic or ligature is a distinct letter of the alphabet. For example, if "  " should come after "A" and before "B" then replace "  " with `0x41 0x7F`.

The `datatool-english` bundle includes support for Old English (Anglo-Saxon) for both `ang-Latn` (Latin script) and `ang-Runr` (Runic script). This is mainly included to provide an example of how to support extended Latin and non-Latin alphabets. Here, "script" refers to what is used in the source code. For example, if a package is used that provides a command called, say, `\runic` where `\runic{fuporc}` is rendered as runes in the PDF, then the script should be specified as Latin, not Runic, so the locale would need to be `ang-Latn`.

4 Databases

The `datatool` package deals with data arranged in rows and columns. This can either be defined within the document via commands or obtained from a CSV (or TSV) file. There are new commands for reading and writing:

```
\DTLread[⟨options⟩]{⟨filename⟩}
\DTLwrite[⟨options⟩]{⟨filename⟩}
```

The older commands have been redefined in terms of these new commands. The `format` option specifies the file type. For example, `csv` for CSV files or `tsv` (which will locally change the category code of the Tab character). The `csv-content` option may be: `tex` (data is valid $\text{L}\text{A}\text{T}\text{E}\text{X}$ syntax) or `literal` (literal content that will require converting special characters like `$` to commands) or (v3.2+) `no-parse` (data should not be parsed, the fastest setting for text-only content).

If the data contains numeric values that may need plotting or aggregating, switch on the “store-datum” option to store each item in the datum format for easy access to the numeric value without having to constantly re-parse. For example, to handle a CSV file that has one column with $\$$ prices and another with \pounds prices:

```
\usepackage[locales={en-GB,en-US}]{datatool}
\DTLsetup{store-datum}
\DTLread[format=csv,csv-content=tex,name=data]
{profits.csv}
```

If the numeric values are plain numbers (with no currency symbols) then use the `csv-content=no-parse` option. You will also need `data-types`. If omitted, the data will be considered non-numeric. For example, if all columns should be considered decimals:

```
\DTLsetup{store-datum}
\DTLread[format=csv,name=data,
csv-content=no-parse,data-types={decimal}]
{results.csv}
```

The old database sorting command `\dtsort` has been rewritten, but there is a newer command analogous to `\DTLsortwordlist`:

```
\DTLsortdata[{options}]{{db-name}}{{criteria}}
```

Some commands have quite lengthy names and need to have the database name supplied in an argument. This can be quite tiresome, so a new general purpose action command is provided:

```
\DTLaction[{options}]{{action-name}}
```

The default database name can be set:

```
\DTLsetup{default-name=products,
store-datum,numeric=auto-reformat}
```

The “products” database can now be created in the document:

```
\DTLaction{new}% same as \DTLgnewdb{products}
\DTLaction[assign={Product={Élite Book},
Price={\$,2,345}, Quantity=4}]{new row}
\DTLaction[assign={Product={Elephant Puzzles},
Price={\$,2.99}, Quantity=20}]{new row}
\DTLaction[assign={Product={Zebra Print},
Price={\$,2.99}, Quantity={1,500}}]{new row}
```

The “sort” action internally uses `\DTLsortdata` and the “display” action internally uses `\DTLdisplaydb`:

```
\begin{table}
\caption{Price then Quantity Descending}
\label{tab:productprice}
\centering
\DTLaction
[assign={Price=desc,Quantity=desc}]{sort}
\DTLaction{display}
\end{table}
```

The result is shown in Table 1. For long tables, use “display long” instead, which will load and use the `longtable` package instead of `tabular` [1].

Table 1: Price then Quantity Descending

| Product | Price | Quantity |
|------------------|------------|----------|
| Élite Book | \$2,345.00 | 4 |
| Zebra Print | \$2.99 | 1,500 |
| Elephant Puzzles | \$2.99 | 20 |

Table 2: Product Name Ascending

| Product | Price | Quantity |
|------------------|------------|----------|
| Elephant Puzzles | \$2.99 | 20 |
| Élite Book | \$2,345.00 | 4 |
| Zebra Print | \$2.99 | 1,500 |

In Table 1, the order is numerical as the “Price” column has been identified as currency by the parser. String columns will be sorted using the current localisation setting. For example,

```
\begin{table}
\caption{Product Name Ascending}
\label{tab:productname}
\centering
\DTLaction[assign={Product}]{sort}
\DTLaction{display}
\end{table}
```

This produces Table 2 (assuming English localisation has been set up).

References

- [1] D. Carlisle. The `longtable` package, 2024. ctan.org/pkg/longtable.
- [2] P. Kime, M. Wemheuer, P. Lehman. The `biblatex` package, 2024. ctan.org/pkg/biblatex.
- [3] M. Mehlich. The `fp` package, 1999. ctan.org/pkg/fp.
- [4] PGF/TikZ Team. The `pgf` package, 2023. ctan.org/pkg/pgf.
- [5] N. Talbot. Localisation of T \E X documents: `tracklang`. *TUGboat* 37(3):337–351, 2016. tug.org/TUGboat/tb37-3/tb117talbot.pdf
- [6] N. Talbot. The `datatool-english` bundle, 2025. ctan.org/pkg/datatool-english.
- [7] N. Talbot. The `datatool` package, 2025. ctan.org/pkg/datatool.
- [8] N. Talbot. The `datatool-regions` bundle, 2025. ctan.org/pkg/datatool-regions.
- [9] N. Talbot. The `glossaries` package, 2025. ctan.org/pkg/glossaries.
- [10] N. Talbot. The `tracklang` package, 2025. ctan.org/pkg/tracklang.
- [11] J. Wright. The `siunitx` package, 2025. ctan.org/pkg/siunitx.

◇ Nicola Talbot
dickimaw-books.com