User Manual for datatool bundle version 2.18

Nicola L.C. Talbot
http://www.dickimaw-books.com/

2013-09-06

The datatool bundle comes with the following documentation:

datatool-user.pdf This document is the main user guide for the datatool bundle.

datatool-code.pdf Advanced users wishing to know more about the inner workings of all the packages provided in the datatool bundle should read "Documented Code for datatool v2.18"

INSTALL Installation instructions.

CHANGES Change log.

README Package summary.

There's an old adage, "use the right tool for the right job." A carpenter's fine chisel is the right tool for delicate carving, but if you try to use it to hack off a tree branch it will take a long time. That doesn't mean there's something wrong with the chisel. It just means you're using the wrong tool for the job.
The datatool bundle is provided to help perform repetitive commands, such as mail merging, but since TEX is designed as a typesetting language, don't expect this bundle to perform as efficiently as custom database systems or a dedicated mathematical or scripting language. **If the provided packages take a frustratingly long time to compile your document, use another language to perform your calculations or data manipulation and save the results in a file that can be input into your document.** For large amounts of data that need to be sorted or filtered or joined, consider storing your data in an SQL database and use `datatooltk`[a] to import the data, using SQL syntax to filter, sort and otherwise manipulate the values.

[a] http://www.dickimaw-books.com/apps/datatooltk/

This bundle consists of the following packages:

datatool Main package providing database support. Automatically loads datatool-base.

datatool-base Provides the main library code for numerical and string functions. Automatically loads datatool-fp or datatool-pgfmath depending on package options.

datagidx Package for generating indexes and glossaries. Automatically loads datatool.

databar Package for drawing bar charts. Automatically loads datatool.

datapie Package for drawing pie charts. Automatically loads datatool.

dataplot Package for drawing simple line graphs. Automatically loads datatool.

databib Package for loading a bibliography into a database. Automatically loads datatool.

person Package for referencing people by the appropriate gender pronouns. Automatically loads datatool.

In addition, there are two mutually exclusive packages datatool-fp and datatool-pgfmath that provide mathematical related commands that are just wrapper functions for fp or pgfmath commands. These can be loaded individually without loading datatool. For example, the following documents produce the same results, but the first uses the fp package and the second uses the pgfmath package:

1. Using fp macros:

```
\documentclass{article}
\usepackage{datatool-fp}
\begin{document}
1=2: \dtlifnumeq{1}{2}{true}{false}.
\end{document}
```

2. Using pgfmath macros:

```
\documentclass{article}
\usepackage{datatool-pgfmath}
\begin{document}
1=2: \dtlifnumeq{1}{2}{true}{false}.
\end{document}
```

Contents

List of Examples

List of Figures

List of Tables

1 Introduction

The datatool bundle consists of the following packages: datatool (which loads datatool-base and either datatool-fp or datatool-pgfmath), datagidx, datapie, dataplot, databar, databib and person.

- The datatool package can be used to:
  - Create or load databases.
  - Sort rows of a database (either numerically or alphabetically, ascending or descending).
  - Perform repetitive operations on each row of a database (e.g. mail merging). Conditions may be imposed to exclude rows.

  Package Options:

  verbose Boolean key. If true, prints informational messages in transcript.

  math May take one of two values: fp (load datatool-fp) or pgfmath (load datatool-pgfmath). Default is: fp.

  delimiter Delimiter used in CSV files. Default is a double quote (").

  separator Delimiter used in CSV files. Default is a comma (,).

- The datatool-base package can be used to:
  - Determine whether an argument is an integer, a real number, currency or a string. (Scientific notation is currently not supported.) Locale dependent number settings are supported (such as a comma as a decimal character and a full stop as a number group character).
  - Convert locale dependent numbers/currency to the decimal format required by the fp or pgfmath packages, enabling fixed point arithmetic to be performed on elements of the database.
  - Names can be converted to initials.
  - Determine if strings are all upper or lower case.
  - Perform string comparisons (both case sensitive and case insensitive).

  Package Options:

verbose Boolean key. If true, prints informational messages in transcript.

math May take one of two values: fp (load datatool-fp) or pgfmath (load datatool-pgfmath). Default is: fp.

- The datagidx package (see chapter 6) can be used to generate indexes or glossaries as an alternative to packages such as glossaries.
- The datapie package (see chapter 7) can be used to convert a database into a pie chart:
  - Segments can be separated from the rest of the chart to make them stand out.
  - Colour/grey scale options.
  - Predefined segment colours can be changed.
  - Hooks provided to add extra information to the chart
- The databar package (see chapter 9) can be used to convert a database into a bar chart:
  - Colour/grey scale options.
  - Predefined bar colours can be changed.
  - Hooks provided to add extra information to the chart

  (The datapie and databar packages do not support the creation of 3D charts, and I have no plans to implement them at any later date. The use of 3D charts should be discouraged. They may look pretty, but the purpose of a chart is to be informative. Three dimensional graphics cause distortion, which can result in misleading impressions. The pgf manual provides a more in-depth discussion on the matter.)
- The dataplot package (see chapter 8) can be used to convert a database into a two dimensional plot using markers and/or lines. Three dimensional plots are currently not supported.
- The databib package (see chapter 10) can be used to convert a BibTeX database into a datatool database.
- The person package (see chapter 11) can be used for gender-specific mail-merging and similar uses to avoid the cumbersome use of the impersonal "he/she".

2 Data Types

The datatool-base package recognises four data types: integers, real numbers, currency and strings.

Integers An integer is a sequence of digits, optionally groups of three digits may be separated by the number group character. The default number group character is a comma (,) but may be changed using `\DTLsetnumberchars` (see below).

Real Numbers A real number is an integer followed by the decimal character followed by one or more digits. The decimal character is a full stop (.) by default. The number group and decimal characters may be changed using

`\DTLsetnumberchars`

`\DTLsetnumberchars{`⟨*number group character*⟩`}{`⟨*decimal character*⟩`}`

Note that scientific notation is not supported, and the number group character may not be used after the decimal character.

Currency A currency symbol followed by an integer or real number is considered to be the currency data type. There are two predefined currency symbols, `\$` and `\pounds`. In addition, if any of the following commands are defined at the start of the document, they are also considered to be a currency symbol: `\texteuro`, `\textdollar`, `\textstirling`, `\textyen`, `\textwon`, `\textcurrency`, `\euro` and `\yen`. Additional currency symbols can be defined using

`\DTLnewcurrencysymbol`

`\DTLnewcurrencysymbol{`⟨*symbol*⟩`}`

Strings Anything that doesn't belong to the above three types is considered to be a string.

2.1 Conditionals

The following conditionals are provided by the datatool-base package:

\DTLifint

\DTLifint{⟨*text*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

If ⟨*text*⟩ is an integer then do ⟨*true part*⟩, otherwise do ⟨*false part*⟩. For example

```
\DTLifint{2536}{integer}{not an integer}
```

produces: integer.

The number group character may appear in the number, for example:

```
\DTLifint{2,536}{integer}{not an integer}
```

produces: integer. However, the number group character may only be followed by a group of three digits. For example:

```
\DTLifint{2,5,3,6}{integer}{not an integer}
```

produces: not an integer. The number group character may be changed. For example:

```
\DTLsetnumberchars{.}{,}%
\DTLifint{2,536}{integer}{not an integer}
```

this now produces: not an integer, since 2,536 is now a real number.

Note that nothing else can be appended or prepended to the number. For example:

```
\DTLsetnumberchars{,}{.}%
\DTLifint{2,536m}{integer}{not an integer}
```

produces: not an integer.

\DTLifreal

\DTLifreal{⟨*text*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

If ⟨*text*⟩ is a real number then do ⟨*true part*⟩, otherwise do ⟨*false part*⟩. For example

```
\DTLifreal{1000.0}{real}{not real}
```

produces: real.

Note that an integer is not considered a real number:

```
\DTLifreal{1,000}{real}{not real}
```

produces: not real.
Whereas

```
\DTLifreal{1,000.0}{real}{not real}
```

produces: real.
However

```
\DTLsetnumberchars{.}{,}%
\DTLifreal{1,000}{real}{not real}
```

produces: real since the comma is now the decimal character.
Currency is not considered to be real:

```
\DTLsetnumberchars{,}{.}%
\DTLifreal{\$1.00}{real}{not real}
```

produces: not real.

\DTLifcurrency

\DTLifcurrency{⟨*text*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

If ⟨*text*⟩ is currency, then do ⟨*true part*⟩, otherwise do false part. For example:

```
\DTLifcurrency{\$5.99}{currency}{not currency}
```

produces: currency. Similarly:

```
\DTLifcurrency{\pounds5.99}{currency}{not currency}
```

produces: currency. Note, however, that

```
\DTLifcurrency{US\$5.99}{currency}{not currency}
```

produces: not currency. If you want this to be considered currency, you will have to add the sequence `US\$` to the set of currency symbols:

```
\DTLnewcurrencysymbol{US\$}%
\DTLifcurrency{US\$5.99}{currency}{not currency}
```

this now produces: currency.
This document has used the textcomp package which defines `\texteuro`, so this is also considered to be currency. For example:

```
\DTLifcurrency{\texteuro5.99}{currency}{not currency}
```

produces: currency.
The preferred method is to display the euro symbol in a sans-serif font, but

```
\DTLifcurrency{\textsf{\texteuro}5.99}{currency}{not currency}
```

will produce: not currency.

It is better to define a new command, for example:

```
\DeclareRobustCommand*{\euro}{\textsf{\texteuro}}
```

and add that command to the list of currency symbols. In fact, in this case, if you define the command `\euro` in the preamble, it will automatically be added to the list of known currency symbols. If however you define `\euro` in the document, you will have to add it using `\DTLnewcurrencysymbol`. For example:

```
\newcommand*{\euro}{\textsf{\texteuro}}%
\DTLnewcurrencysymbol{\euro}%
\DTLifcurrency{\euro5.99}{currency}{not currency}
```

produces: currency.

`\DTLifcurrencyunit`

`\DTLifcurrencyunit{`⟨*text*⟩`}{`⟨*symbol*⟩`}{`⟨*true part*⟩`}{`⟨*false part*⟩`}`

If ⟨*text*⟩ is currency, and uses ⟨*symbol*⟩ as the unit of currency, then do ⟨*true part*⟩ otherwise do ⟨*false part*⟩. For example:

```
\DTLifcurrencyunit{\$6.99}{\$}{dollars}{not dollars}
```

produces: dollars. Another example:

```
\def\cost{\euro10.50}%
\DTLifcurrencyunit{\cost}{\euro}{euros}{not euros}
```

produces: euros.

`\DTLifnumerical`

`\DTLifnumerical{`⟨*text*⟩`}{`⟨*true part*⟩`}{`⟨*false part*⟩`}`

If ⟨*text*⟩ is numerical (either an integer, real number or currency) then do ⟨*true part*⟩ otherwise do ⟨*false part*⟩. For example:

```
\DTLifnumerical{1,000.0}{number}{string}.
```

produces: number. Whereas

```
\DTLsetnumberchars{.}{,}%
\DTLifnumerical{1,000.0}{number}{string}.
```

produces: string. Since the number group character is now a full stop, and the decimal character is now a comma. (The number group character may only appear before the decimal character, not after it.)

Currency is also considered to be numerical:

```
\DTLsetnumberchars{,}{.}%
\DTLifnumerical{\$1,000.0}{number}{string}.
```

produces: number.

\DTLifstring

\DTLifstring{⟨*text*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

This is the opposite of \DTLifnumerical. If ⟨*text*⟩ is not numerical, do ⟨*true part*⟩, otherwise do ⟨*false part*⟩.

\DTLifcasedatatype

\DTLifcasedatatype{⟨*text*⟩}{⟨*string case*⟩}{⟨*int case*⟩}{⟨*real case*⟩}{⟨*currency case*⟩}

If ⟨*text*⟩ is a string do ⟨*string case*⟩, if ⟨*text*⟩ is an integer do ⟨*int case*⟩, if ⟨*text*⟩ is a real number do ⟨*real case*⟩, if ⟨*text*⟩ is currency do ⟨*currency case*⟩. For example:

```
\DTLifcasedatatype{1,000}{string}{integer}{real}{currency}
```

produces: integer.

\dtlifnumeq

\dtlifnumeq{⟨*num1*⟩}{⟨*num2*⟩}{⟨*true part*⟩}{⟨*false paty*⟩}

If ⟨*num1*⟩ is equal to ⟨*num2*⟩, then do ⟨*true part*⟩, otherwise to ⟨*false part*⟩ where ⟨*num1*⟩ and ⟨*num2*⟩ are plain numbers using a full stop as the decimal point and no number group separator. For currency or locale dependent numbers use \DTLifnumeq.

\DTLifnumeq

\DTLifnumeq{⟨*num1*⟩}{⟨*num2*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

If ⟨*num1*⟩ is equal to ⟨*num2*⟩, then do ⟨*true part*⟩, otherwise do ⟨*false part*⟩. Note that both ⟨*num1*⟩ and ⟨*num2*⟩ must be numerical (either integers, real numbers or currency). The currency symbol is ignored when determining equality. For example

```
\DTLifnumeq{\pounds10.50}{10.5}{true}{false}
```

produces: true, since they are considered to be numerically equivalent. Likewise:

```
\DTLifnumeq{\pounds10.50}{\$10.50}{true}{false}
```

produces: true.

\DTLifstringeq

\DTLifstringeq{⟨*string1*⟩}{⟨*string2*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

\DTLifstringeq*

\DTLifstringeq*{⟨*string1*⟩}{⟨*string2*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

If ⟨*string1*⟩ and ⟨*string2*⟩ are the same, then do ⟨*true part*⟩, otherwise do ⟨*false part*⟩. The starred version ignores the case, the unstarred version is case sensitive. Both ⟨*string1*⟩ and ⟨*string2*⟩ are considered to be strings, so for example:

```
\DTLifstringeq{10.50}{10.5}{true}{false}
```

produces: false.

Note that

```
\DTLifstringeq{Text}{text}{true}{false}
```

produces: false, whereas

```
\DTLifstringeq*{Text}{text}{true}{false}
```

produces: true, however it should also be noted that many commands will be ignored, so:

```
\DTLifstringeq{\uppercase{t}ext}{text}{true}{false}
```

produces: false.

Spaces are considered to be equivalent to `\space` and `~`. For example:

```
\DTLifstringeq{an apple}{an~apple}{true}{false}
```

produces: true. Consecutive spaces are treated as the same, for example:

```
\DTLifstringeq{an  apple}{an apple}{true}{false}
```

produces: true.

\DTLifeq

\DTLifeq{⟨*arg1*⟩}{⟨*arg2*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

\DTLifeq*

\DTLifeq*{⟨*arg1*⟩}{⟨*arg2*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

If both ⟨*arg1*⟩ and ⟨*arg2*⟩ are numerical, then this is equivalent to `\DTLifnumeq`, otherwise it is equivalent to `\DTLifstringeq` (when using `\DTLifeq`) or `\DTLifstringeq*` (when using `\DTLifeq*`).

\dtlifnumlt

\dtlifnumlt{⟨*num1*⟩}{⟨*num2*⟩}{⟨*true part*⟩}{⟨*false paty*⟩}

If ⟨*num1*⟩ is less than ⟨*num2*⟩, then do ⟨*true part*⟩, otherwise to ⟨*false part*⟩ where ⟨*num1*⟩ and ⟨*num2*⟩ are plain numbers using a full stop as the decimal point and no number group separator. For currency or locale dependent numbers use `\DTLifnumlt`.

\DTLifnumlt

\DTLifnumlt{⟨*num1*⟩}{⟨*num2*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

If ⟨*num1*⟩ is less than ⟨*num2*⟩, then do ⟨*true part*⟩, otherwise do ⟨*false part*⟩. Note that both ⟨*num1*⟩ and ⟨*num2*⟩ must be numerical (either integers, real numbers or currency).

\DTLifstringlt

\DTLifstringlt{⟨*string1*⟩}{⟨*string2*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

\DTLifstringlt*

\DTLifstringlt*{⟨*string1*⟩}{⟨*string2*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

If ⟨*string1*⟩ is alphabetically less than ⟨*string2*⟩, then do ⟨*true part*⟩, otherwise do ⟨*false part*⟩. The starred version ignores the case, the unstarred version is case sensitive. For example:

```
\DTLifstringlt{aardvark}{zebra}{less}{not less}
```

produces: less.

Note that both ⟨*string1*⟩ and ⟨*string2*⟩ are considered to be strings, so for example:

```
\DTLifstringlt{2}{10}{less}{not less}
```

produces: not less, since the string `2` comes after the string `10` when arranged alphabetically.

The case sensitive (unstarred) version considers uppercase characters to be less than lowercase characters, so

```
\DTLifstringlt{B}{a}{less}{not less}
```

produces: less, whereas

```
\DTLifstringlt*{B}{a}{less}{not less}
```

produces: not less.

\DTLiflt

\DTLiflt{⟨*arg1*⟩}{⟨*arg2*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

\DTLiflt*

\DTLiflt*{⟨*arg1*⟩}{⟨*arg2*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

If ⟨*arg1*⟩ and ⟨*arg2*⟩ are both numerical, then this is equivalent to \DTLifnumlt, otherwise it is equivalent to \DTLstringlt (when using \DTLiflt) or \DTLstringlt* (when using \DTLiflt*).

\DTLifnumgt

\DTLifnumgt{⟨*num1*⟩}{⟨*num2*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

If ⟨*num1*⟩ is greater than ⟨*num2*⟩, then do ⟨*true part*⟩, otherwise do ⟨*false part*⟩. Note that both ⟨*num1*⟩ and ⟨*num2*⟩ must be numerical (either integers, real numbers or currency).

\DTLifstringgt

\DTLifstringgt{⟨*string1*⟩}{⟨*string2*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

\DTLifstringgt*

\DTLifstringgt*{⟨*string1*⟩}{⟨*string2*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

If ⟨*string1*⟩ is alphabetically greater than ⟨*string2*⟩, then do ⟨*true part*⟩, otherwise do ⟨*false part*⟩. The starred version ignores the case, the unstarred version is case sensitive. For example:

```
\DTLifstringgt{aardvark}{zebra}{greater}{not greater}
```

produces: not greater.

Note that both ⟨*string1*⟩ and ⟨*string2*⟩ are considered to be strings, so for example:

```
\DTLifstringgt{2}{10}{greater}{not greater}
```

produces: greater, since the string `2` comes after the string `10` when arranged alphabetically.

As with \DTLifstringlt, uppercase characters are considered to be less than lower case characters when performing a case sensitive comparison so:

```
\DTLifstringgt{B}{a}{greater}{not greater}
```

produces: not greater, whereas

```
\DTLifstringgt*{B}{a}{greater}{not greater}
```

produces: greater.

`\DTLifgt`

`\DTLifgt{`⟨*arg1*⟩`}{`⟨*arg2*⟩`}{`⟨*true part*⟩`}{`⟨*false part*⟩`}`

`\DTLifgt*`

`\DTLifgt*{`⟨*arg1*⟩`}{`⟨*arg2*⟩`}{`⟨*true part*⟩`}{`⟨*false part*⟩`}`

If ⟨*arg1*⟩ and ⟨*arg2*⟩ are both numerical, then this is equivalent to `\DTLifnumgt`, otherwise it is equivalent to `\DTLstringgt` (when using `\DTLifgt`) or `\DTLstringgt*` (when using `\DTLifgt*`).

`\DTLifnumclosedbetween`

`\DTLifnumclosedbetween{`⟨*num*⟩`}{`⟨*min*⟩`}{`⟨*max*⟩`}{`⟨*true part*⟩`}{`⟨*false part*⟩`}`

If ⟨*min*⟩ ≤ ⟨*num*⟩ ≤ ⟨*max*⟩ then do ⟨*true part*⟩, otherwise do ⟨*false part*⟩. Note that ⟨*num*⟩, ⟨*min*⟩ and ⟨*max*⟩ must be numerical (either integers, real numbers or currency). The currency symbol is ignored when determining equality. For example:

```
\DTLifnumclosedbetween{5.4}{5}{7}{inside}{outside}
```

produces: inside. Note that the closed range includes end points:

```
\DTLifnumclosedbetween{5}{5}{7}{inside}{outside}
```

produces: inside.

`\DTLifstringclosedbetween`

`\DTLifstringclosedbetween{`⟨*string*⟩`}{`⟨*min*⟩`}{`⟨*max*⟩`}{`⟨*true part*⟩`}{`⟨*false part*⟩`}`

`\DTLifstringclosedbetween*`

`\DTLifstringclosedbetween*{`⟨*string*⟩`}{`⟨*min*⟩`}{`⟨*max*⟩`}{`⟨*true part*⟩`}{`⟨*false part*⟩`}`

This determines if ⟨*string*⟩ is between ⟨*min*⟩ and ⟨*max*⟩ in the alphabetical sense, or is equal to either ⟨*min*⟩ or ⟨*max*⟩. The starred version ignores the case, the unstarred version is case sensitive.

\DTLifclosedbetween

\DTLifclosedbetween{⟨*arg*⟩}{⟨*min*⟩}{⟨*max*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

\DTLifclosedbetween*

\DTLifclosedbetween*{⟨*arg*⟩}{⟨*min*⟩}{⟨*max*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

If ⟨*arg*⟩, ⟨*min*⟩ and ⟨*max*⟩ are numerical, then this is equivalent to
\DTLifnumclosedbetween
otherwise it is equivalent to
\DTLifstringclosedbetween
(when using \DTLifclosedbetween) or
\DTLifstringclosedbetween*
(when using \DTLifclosedbetween*).

\DTLifnumopenbetween

\DTLifnumopenbetween{⟨*num*⟩}{⟨*min*⟩}{⟨*max*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

If ⟨*min*⟩ ⟨*num*⟩ ⟨*max*⟩ then do ⟨*true part*⟩, otherwise do ⟨*false part*⟩. Note that ⟨*num*⟩, ⟨*min*⟩ and ⟨*max*⟩ must be numerical (either integers, real numbers or currency). Again, the currency symbol is ignored when determining equality. For example:

```
\DTLifnumopenbetween{5.4}{5}{7}{inside}{outside}
```

produces: inside. Note that end points are not included. For example:

```
\DTLifnumopenbetween{5}{5}{7}{inside}{outside}
```

produces: outside.

\DTLifstringopenbetween

\DTLifstringopenbetween{⟨*string*⟩}{⟨*min*⟩}{⟨*max*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

\DTLifstringopenbetween*

\DTLifstringopenbetween*{⟨*string*⟩}{⟨*min*⟩}{⟨*max*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

This determines if ⟨*string*⟩ is between ⟨*min*⟩ and ⟨*max*⟩ in the alphabetical sense. The starred version ignores the case, the unstarred version is case sensitive.

\DTLifopenbetween

\DTLifopenbetween{⟨arg⟩}{⟨min⟩}{⟨max⟩}{⟨true part⟩}{⟨false part⟩}

\DTLifopenbetween*

\DTLifopenbetween*{⟨arg⟩}{⟨min⟩}{⟨max⟩}{⟨true part⟩}{⟨false part⟩}

If ⟨*arg*⟩, ⟨*min*⟩ and ⟨*max*⟩ are numerical, then this is equivalent to \DTLifnumopenbetween otherwise it is equivalent to \DTLifstringopenbetween (when using \DTLifopenbetween) or \DTLifstringopenbetween* (when using \DTLifopenbetween*).

\DTLifFPclosedbetween

\DTLifFPclosedbetween{⟨num⟩}{⟨min⟩}{⟨max⟩}{⟨true part⟩}{⟨false part⟩}

If ⟨*min*⟩ ≤ ⟨*num*⟩ ≤ ⟨*max*⟩ then do ⟨*true part*⟩, otherwise do ⟨*false part*⟩ where ⟨*num*⟩, ⟨*min*⟩ and ⟨*max*⟩ are all in standard fixed point notation (i.e. no number group separator, no currency symbols and a full stop as a decimal point).

\DTLifFPopenbetween

\DTLifFPopenbetween{⟨num⟩}{⟨min⟩}{⟨max⟩}{⟨true part⟩}{⟨false part⟩}

If ⟨*min*⟩ ⟨*num*⟩ ⟨*max*⟩ then do ⟨*true part*⟩, otherwise do ⟨*false part*⟩ where ⟨*num*⟩, ⟨*min*⟩ and ⟨*max*⟩ are all in standard fixed point notation (i.e. no number group separator, no currency symbols and a full stop as a decimal point).

\DTLifAllUpperCase

\DTLifAllUpperCase{⟨string⟩}{⟨true part⟩}{⟨false part⟩}

Tests if ⟨*string*⟩ is all upper case. For example:

```
\DTLifAllUpperCase{WORD}{all upper}{not all upper}
```

produces: all upper, whereas

```
\DTLifAllUpperCase{Word}{all upper}{not all upper}
```

produces: not all upper. Note also that:

```
\DTLifAllUpperCase{\MakeUppercase{word}}{all upper}{not all upper}
```

also produces: all upper. \MakeTextUppercase (defined in David Carlisle's textcase package) and \uppercase are also detected, otherwise, if a command is encountered, the case of the command is considered. For example:

```
\DTLifAllUpperCase{MAN{\OE}UVRE}{all upper}{not all upper}
```

produces: all upper.

\DTLifAllLowerCase

\DTLifAllLowerCase{⟨*string*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

Tests if ⟨*string*⟩ is all lower case. For example:

```
\DTLifAllLowerCase{word}{all lower}{not all lower}
```

produces: all lower, whereas

```
\DTLifAllLowerCase{Word}{all lower}{not all lower}
```

produces: not all lower. Note also that:

```
\DTLifAllLowerCase{\MakeLowercase{WORD}}{all lower}{not all lower}
```

also produces: all lower. \MakeTextLowercase (defined in David Carlisle's textcase package) and \lowercase are also detected, otherwise, if a command is encountered, the case of the command is considered. For example:

```
\DTLifAllLowerCase{man{\oe}uvre}{all lower}{not all lower}
```

produces: all lower.

\DTLifSubString

\DTLifSubString{⟨*string*⟩}{⟨*substring*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

This tests if ⟨*substring*⟩ is a sub-string of ⟨*string*⟩. This command performs a case sensitive match. For example:

```
\DTLifSubString{An apple}{app}{is substring}{isn't substring}
```

produces: is substring. Note that spaces are considered to be equivalent to \space or ~, so

```
\DTLifSubString{An apple}{n~a}{is substring}{isn't substring}
```

produces: is substring, but other commands are skipped, so

```
\DTLifSubString{An \uppercase{a}pple}{app}{is substring}{isn't
substring}
```

produces: is substring, since the \uppercase command is ignored. Note also that grouping is ignored, so:

```
\DTLifSubString{An {ap}ple}{app}{is substring}{isn't substring}
```

produces: is substring.

\DTLifSubString is case sensitive, so:

```
\DTLifSubString{An Apple}{app}{is substring}{isn't substring}
```

produces: isn't substring.

\DTLifStartsWith

\DTLifStartsWith{⟨*string*⟩}{⟨*substring*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

This is like \DTLifSubString, except that ⟨*substring*⟩ must occur at the start of ⟨*string*⟩. This command performs a case sensitive match. For example,

```
\DTLifStartsWith{An apple}{app}{prefix}{not a prefix}
```

produces: not a prefix. All the above remarks for \DTLifSubString also applies to \DTLifStartsWith. For example:

```
\DTLifStartsWith{\uppercase{a}n apple}{an~}{prefix}{not a prefix}
```

produces: not a prefix, since \uppercase is ignored, and ~ is considered to be the same as a space, whereas

```
\DTLifStartsWith{An apple}{an~}{prefix}{not a prefix}
```

produces: not a prefix.

2.2 ifthen conditionals

The commands described in the previous section can not be used as the conditional part of the \ifthenelse or \whiledo commands provided by the ifthen package. This section describes analogous commands which may only be used in the conditional argument of \ifthenelse and \whiledo. These may be used with the boolean operations \not, \and and \or provided by the ifthen package. See the ifthen documentation for further details.

\DTLisstring

\DTLisstring{⟨*text*⟩}

Tests if ⟨*text*⟩ is a string. For example:

```
\ifthenelse{\DTLisstring{some text}}{string}{not a string}
```

produces: string.

\DTLisnumerical

\DTLisnumerical{⟨*text*⟩}

Tests if ⟨*text*⟩ is numerical (i.e. not a string). For example:

```
\ifthenelse{\DTLisnumerical{\$10.95}}{numerical}{not numerical}
```

produces: numerical.

Note however that \DTLisnumerical requires more care than \DTLifnumerical when used with some of the other currency symbols. Consider:

```
\DTLifnumerical{\pounds10.95}{numerical}{not numerical}
```

This produces: numerical. However

```
\ifthenelse{\DTLisnumerical{\pounds10.95}}{numerical}{not numerical}
```

produces: not numerical. This is due to the expansion that occurs within \ifthenelse. This can be prevented using \noexpand, for example:

```
\ifthenelse{\DTLisnumerical{\noexpand\pounds10.95}}{numerical}{not numerical}
```

produces: numerical.

Likewise:

```
\def\cost{\pounds10.95}%
\ifthenelse{\DTLisnumerical{\noexpand\cost}}{numerical}{not numerical}
```

produces: numerical.

\DTLiscurrency

\DTLiscurrency{⟨*text*⟩}

Tests if ⟨*text*⟩ is currency. For example:

```
\ifthenelse{\DTLiscurrency{\$10.95}}{currency}{not currency}
```

produces: currency.

The same warning given above for \DTLisnumerical also applies here.

\DTLiscurrencyunit

\DTLiscurrencyunit{⟨*text*⟩}{⟨*symbol*⟩}

Tests if ⟨*text*⟩ is currency and that currency uses ⟨*symbol*⟩ as the unit of currency. For example:

```
\ifthenelse{\DTLiscurrencyunit{\$6.99}{\$}}{dollars}{not dollars}
```

produces: dollars. Another example:

```
\def\cost{\euro10.50}%
\ifthenelse{\DTLiscurrencyunit{\noexpand\cost}{\noexpand\euro}}%
{euros}{not euros}
```

produces: euros. Again note the use of `\noexpand`.

`\DTLisreal`

`\DTLisreal{`⟨*text*⟩`}`

Tests if ⟨*text*⟩ is a fixed point number (again, an integer is not considered to be a fixed point number). For example:

```
\ifthenelse{\DTLisreal{1.5}}{real}{not real}
```

produces: real.

`\DTLisint`

`\DTLisint{`⟨*text*⟩`}`

Tests if ⟨*text*⟩ is an integer. For example:

```
\ifthenelse{\DTLisint{153}}{integer}{not an integer}
```

produces: integer.

`\DTLislt`

`\DTLislt{`⟨*arg1*⟩`}{`⟨*arg2*⟩`}`

This checks if ⟨*arg1*⟩ is less than ⟨*arg2*⟩. As with `\DTLiflt`, if ⟨*arg1*⟩ and ⟨*arg2*⟩ are numerical, a numerical comparison is used, otherwise a case sensitive alphabetical comparison is used. (Note that there is no starred version of this command, but you can instead use `\DTLisilt` to ignore the case.)

`\DTLisilt`

`\DTLisilt{`⟨*arg1*⟩`}{`⟨*arg2*⟩`}`

This checks if ⟨*arg1*⟩ is less than ⟨*arg2*⟩. As with `\DTLiflt*`, if ⟨*arg1*⟩ and ⟨*arg2*⟩ are numerical, a numerical comparison is used, otherwise a case insensitive alphabetical comparison is used.

`\DTLisgt`

`\DTLisgt{`⟨*arg1*⟩`}{`⟨*arg2*⟩`}`

This checks if ⟨*arg1*⟩ is greater than ⟨*arg2*⟩. As with `\DTLifgt`, if ⟨*arg1*⟩ and ⟨*arg2*⟩ are numerical, a numerical comparison is used, otherwise a case sensitive alphabetical comparison is used. (Note that there is no starred version of this command, instead use `\DTLisigt` to ignore the case.)

`\DTLisigt`

`\DTLisigt{`⟨*arg1*⟩`}{`⟨*arg2*⟩`}`

This checks if ⟨*arg1*⟩ is greater than ⟨*arg2*⟩. As with `\DTLifgt*`, if ⟨*arg1*⟩ and ⟨*arg2*⟩ are numerical, a numerical comparison is used, otherwise a case insensitive alphabetical comparison is used.

`\DTLiseq`

`\DTLiseq{`⟨*arg1*⟩`}{`⟨*arg2*⟩`}`

This checks if ⟨*arg1*⟩ is equal to ⟨*arg2*⟩. As with `\DTLifeq`, if ⟨*arg1*⟩ and ⟨*arg2*⟩ are numerical, a numerical comparison is used, otherwise a case sensitive alphabetical comparison is used. (Note that there is no starred version of this command, instead use `\DTLisieq`.)

`\DTLisieq`

`\DTLisieq{`⟨*arg1*⟩`}{`⟨*arg2*⟩`}`

This checks if ⟨*arg1*⟩ is equal to ⟨*arg2*⟩. As with `\DTLifeq*`, if ⟨*arg1*⟩ and ⟨*arg2*⟩ are numerical, a numerical comparison is used, otherwise a case insensitive alphabetical comparison is used.

`\DTLisclosedbetween`

`\DTLisclosedbetween{`⟨*arg*⟩`}{`⟨*min*⟩`}{`⟨*max*⟩`}`

This checks if ⟨*arg*⟩ lies between ⟨*min*⟩ and ⟨*max*⟩ (end points included). As with `\DTLifclosedbetween`, if the arguments are numerical, a numerical comparison is used, otherwise a case sensitive alphabetical comparison is used. (Note that there is no starred version of this command, instead use `\DTLisiclosedbetween`.)

`\DTLisiclosedbetween`

`\DTLisiclosedbetween{`⟨*arg*⟩`}{`⟨*min*⟩`}{`⟨*max*⟩`}`

This checks if ⟨*arg*⟩ lies between ⟨*min*⟩ and ⟨*max*⟩ (end points included). As with `\DTLifclosedbetween*`, if the arguments are numerical, a

numerical comparison is used, otherwise a case insensitive alphabetical comparison is used.

\DTLisopenbetween

\DTLisopenbetween{⟨*arg*⟩}{⟨*min*⟩}{⟨*max*⟩}

This checks if ⟨*arg*⟩ lies between ⟨*min*⟩ and ⟨*max*⟩ (end points excluded). As with \DTLifopenbetween, if the arguments are numerical, a numerical comparison is used, otherwise a case sensitive alphabetical comparison is used. (Note that there is no starred version of this command, instead use \DTLisiopenbetween.)

\DTLisiopenbetween

\DTLisiopenbetween{⟨*arg*⟩}{⟨*min*⟩}{⟨*max*⟩}

This checks if ⟨*arg*⟩ lies between ⟨*min*⟩ and ⟨*max*⟩ (end points excluded). As with \DTLifopenbetween*, if the arguments are numerical, a numerical comparison is used, otherwise a case insensitive alphabetical comparison is used.

\DTLisFPlt

\DTLisFPlt{⟨*num1*⟩}{⟨*num2*⟩}

This checks if ⟨*num1*⟩ is less than ⟨*num2*⟩, where both numbers are in standard fixed point format (i.e. no number group separators, no currency and a full stop as a decimal point).

\DTLisFPlteq

\DTLisFPlteq{⟨*num1*⟩}{⟨*num2*⟩}

This checks if ⟨*num1*⟩ is less than or equal to ⟨*num2*⟩, where both numbers are in standard fixed point format (i.e. no number group separators, no currency and a full stop as a decimal point).

\DTLisFPgt

\DTLisFPgt{⟨*num1*⟩}{⟨*num2*⟩}

This checks if ⟨*num1*⟩ is greater than ⟨*num2*⟩, where both numbers are in standard fixed point format (i.e. no number group separators, no currency and a full stop as a decimal point).

\DTLisFPgteq

\DTLisFPgteq{⟨*num1*⟩}{⟨*num2*⟩}

This checks if ⟨*num1*⟩ is greater than or equal to ⟨*num2*⟩, where both numbers are in standard fixed point format (i.e. no number group separators, no currency and a full stop as a decimal point).

`\DTLisFPeq`

`\DTLisFPeq{`⟨*num1*⟩`}{`⟨*num2*⟩`}`

This checks if ⟨*num1*⟩ is equal to ⟨*num2*⟩, where both numbers are in standard fixed point format (i.e. no number group separators, no currency and a full stop as a decimal point).

`\DTLisFPclosedbetween`

`\DTLisFPclosedbetween{`⟨*num*⟩`}{`⟨*min*⟩`}{`⟨*max*⟩`}`

This checks if ⟨*num*⟩ lies between ⟨*min*⟩ and ⟨*max*⟩ (end points included). All arguments must be numbers in standard fixed point format (i.e. no number group separators, no currency and a full stop as a decimal point).

`\DTLisFPopenbetween`

`\DTLisFPopenbetween{`⟨*num*⟩`}{`⟨*min*⟩`}{`⟨*max*⟩`}`

This checks if ⟨*num*⟩ lies between ⟨*min*⟩ and ⟨*max*⟩ (end points excluded). All arguments must be numbers in standard fixed point format (i.e. no number group separators, no currency and a full stop as a decimal point).

`\DTLisSubString`

`\DTLisSubString{`⟨*string*⟩`}{`⟨*substring*⟩`}`

This checks if ⟨*substring*⟩ is contained in ⟨*string*⟩. The remarks about `\DTLifSubString` also apply to `\DTLisSubString`. This command performs a case sensitive match.

`\DTLisPrefix`

`\DTLisPrefix{`⟨*string*⟩`}{`⟨*prefix*⟩`}`

This checks if ⟨*string*⟩ starts with ⟨*prefix*⟩. The remarks about `\DTLifStartsWith` also apply to `\DTLisPrefix`. This command performs a case sensitive match.

3 Fixed Point Arithmetic

The datatool bundle doesn't support scientific notation.

The datatool-base package uses either the fp or the pgfmath package to perform fixed point arithmetic, however all numbers must be converted from the locale dependent format into the format required by the fp or pgfmath packages. A numerical value (i.e. an integer, a real or currency) can be converted into a plain decimal number using

\DTLconverttodecimal

\DTLconverttodecimal{⟨*num*⟩}{⟨*cmd*⟩}

The decimal number will be stored in ⟨*cmd*⟩ which must be a control sequence. For example:

```
\DTLconverttodecimal{1,563.54}{\mynum}
```

will define \mynum to be 1563.54. The command \mynum can then be used in any of the arithmetic macros provided by the fp or pgfmath packages.

The arguments to \DTLconverttodecimal don't get fully expanded so, for example,

```
\def\myval{1.23}
\DTLconverttodecimal{\myval}{\mynum}
```

will work, but the following ***won't*** work:

```
\def\myval{1.23}
\def\myotherval{\myval}
\DTLconverttodecimal{\myotherval}{\mynum}
```

Nor will the following work:

```
\def\myval{9}
\DTLconverttodecimal{\myval 9}{\mynum}
```

There are two commands provided to perform the reverse:

\DTLdecimaltolocale

\DTLdecimaltolocale{⟨*number*⟩}{⟨*cmd*⟩}

This converts a plain decimal number ⟨*number*⟩ (that uses a full stop as the decimal character and has no number group characters) into a locale dependent format. The resulting number is stored in ⟨*cmd*⟩, which must be a control sequence. For example:

```
\DTLdecimaltolocale{6795.3}{\mynum}
```

will define \mynum to be 6,795.3.

\DTLdecimaltocurrency

\DTLdecimaltocurrency{⟨*number*⟩}{⟨*cmd*⟩}

This will convert a plain decimal number ⟨*number*⟩ into a locale dependent currency format. For example:

```
\DTLdecimaltocurrency{267.5}{\price}\price
```

will produce: £267.50.

The currency symbol used by \DTLdecimaltocurrency is initially \$, but it will use the currency last encountered. So, for example

```
\DTLifcurrency{\texteuro45.00}{}{}%
\DTLdecimaltocurrency{267.5}{\price}\price
```

will produce: €267.50. This is because the last currency symbol to be encountered was \texteuro. You can reset the currency symbol using the command:

\DTLsetdefaultcurrency

\DTLsetdefaultcurrency{⟨*symbol*⟩}

For example:

```
\DTLsetdefaultcurrency{\textyen}%
\DTLdecimaltocurrency{267.5}{\price}\price
```

will produce: ¥267.50

The datatool-base package provides convenience commands which use \DTLconverttodecimal, and then use the basic macros provided by the fp/pgfmath package. The resulting value is then converted back into the locale format using \DTLdecimaltolocale or \DTLdecimaltocurrency. Note that since these commands use \DTLconverttodecimal the caveat above regarding expansion also applies to all the commands.

If you don't require currency or locale conversion, you can reduce the package overheads by using the commands defined in the datatool-fp or datatool-pgfmath packages which provide interface commands to fp or pgfmath, respectively. (See sections 2 and 3 of the documented code, `datatool-code.pdf`.) Alternatively, you can just use the fp or pgfmath commands explicitly. (See the fp or pgf manuals for further details.)

`\DTLadd`

`\DTLadd{`⟨*cmd*⟩`}{`⟨*num1*⟩`}{`⟨*num2*⟩`}`

`\DTLgadd`

`\DTLgadd{`⟨*cmd*⟩`}{`⟨*num1*⟩`}{`⟨*num2*⟩`}`

This sets the control sequence ⟨*cmd*⟩ to ⟨*num1*⟩+⟨*num2*⟩. `\DLTadd` sets ⟨*cmd*⟩ locally, while `\DTLgadd` sets ⟨*cmd*⟩ globally.

For example:

```
\DTLadd{\result}{3,562.65}{412.2}\result
```

will produce: 3,974.85. Since `\DTLconverttodecimal` can convert currency to a real number, you can also add prices. For example:

```
\DTLadd{\result}{\pounds3,562.65}{\pounds452.2}\result
```

produces: £4,014.85.

Note that datatool isn't aware of exchange rates! If you use different currency symbols, the last symbol will be used. For example

```
\DTLadd{\result}{\pounds3,562.65}{\euro452.2}\result
```

produces: €4,014.85.

Likewise, if one value is a number and the other is a currency, the type of the last value, ⟨*num2*⟩, will be used for the result. For example:

```
\DTLadd{\result}{3,562.65}{\$452.2}\result
```

produces: $4,014.85.

`\DTLaddall`

`\DTLaddall{`⟨*cmd*⟩`}{`⟨*number list*⟩`}`

`\DTLgaddall`

`\DTLgaddall{`⟨*cmd*⟩`}{`⟨*number list*⟩`}`

This sets the control sequence ⟨*cmd*⟩ to the sum of all the numbers in ⟨*number list*⟩. `\DLTaddall` sets ⟨*cmd*⟩ locally, while `\DTLgaddall` sets ⟨*cmd*⟩ globally. Example:

```
\DTLaddall{\total}{25.1,45.2,35.6}\total
```

produces: 105.9. Note that if any of the numbers in ⟨*number list*⟩ contain a comma, you must group the number. Example:

```
\DTLaddall{\total}{{1,525},{2,340},500}\total
```

produces: 4,365.

`\DTLsub`

`\DTLsub{`⟨*cmd*⟩`}{`⟨*num1*⟩`}{`⟨*num2*⟩`}`

`\DTLgsub`

`\DTLgsub{`⟨*cmd*⟩`}{`⟨*num1*⟩`}{`⟨*num2*⟩`}`

This sets the control sequence ⟨*cmd*⟩ to ⟨*num1*⟩−⟨*num2*⟩. `\DLTsub` sets ⟨*cmd*⟩ locally, while `\DTLgsub` sets ⟨*cmd*⟩ globally.

For example:

```
\DTLsub{\result}{3,562.65}{412.2}\result
```

will produce: 3,150.45. As with `\DTLadd`, ⟨*num1*⟩ and ⟨*num2*⟩ may be currency.

`\DTLmul`

`\DTLmul{`⟨*cmd*⟩`}{`⟨*num1*⟩`}{`⟨*num2*⟩`}`

`\DTLgmul`

`\DTLgmul{`⟨*cmd*⟩`}{`⟨*num1*⟩`}{`⟨*num2*⟩`}`

This sets the control sequence ⟨*cmd*⟩ to ⟨*num1*⟩×⟨*num2*⟩. `\DLTmul` sets ⟨*cmd*⟩ locally, while `\DTLgmul` sets ⟨*cmd*⟩ globally.

For example:

```
\DTLmul{\result}{568.95}{2}\result
```

will produce: 1,137.9. Again, ⟨*num1*⟩ or ⟨*num2*⟩ may be currency, but unlike `\DTLadd` and `\DTLsub`, currency overrides integer/real. For example:

```
\DTLmul{\result}{\pounds568.95}{2}\result
```

will produce: £1,137.90. Likewise,

```
\DTLmul{\result}{2}{\pounds568.95}\result
```

will produce: £1,137.90. Although it doesn't make sense to multiply two currencies, datatool will allow

```
\DTLmul{\result}{\$2}{\pounds568.95}\result
```

which will produce: £1,137.90.

\DTLdiv

\DTLdiv{⟨*cmd*⟩}{⟨*num1*⟩}{⟨*num2*⟩}

\DTLgdiv

\DTLgdiv{⟨*cmd*⟩}{⟨*num1*⟩}{⟨*num2*⟩}

This sets the control sequence ⟨*cmd*⟩ to ⟨*num1*⟩÷⟨*num2*⟩. \DLTdiv sets ⟨*cmd*⟩ locally, while \DTLgdiv sets ⟨*cmd*⟩ globally.

For example:

```
\DTLdiv{\result}{501}{2}\result
```

will produce: 250.5. Again, ⟨*num1*⟩ or ⟨*num2*⟩ may be currency, but the resulting type will be not be a currency if both ⟨*num1*⟩ and ⟨*num2*⟩ use the same currency symbol. For example:

```
\DTLdiv{\result}{\$501}{\$2}\result
```

will produce: 250.5. Whereas

```
\DTLdiv{\result}{\$501}{2}\result
```

will produce: $250.50.

\DTLabs

\DTLabs{⟨*cmd*⟩}{⟨*num*⟩}

\DTLgabs

\DTLgabs{⟨*cmd*⟩}{⟨*num*⟩}

This sets ⟨*cmd*⟩ to the absolute value of ⟨*num*⟩. \DLTabs sets ⟨*cmd*⟩ locally, while \DTLgabs sets ⟨*cmd*⟩ globally. Example:

```
\DTLabs{\result}{-\pounds2.50}\result
```

produces: £2.50.

\DTLneg

\DTLneg{⟨*cmd*⟩}{⟨*num*⟩}

\DTLgneg

\DTLgneg{⟨*cmd*⟩}{⟨*num*⟩}

This sets ⟨*cmd*⟩ to the negative of ⟨*num*⟩. \DLTneg sets ⟨*cmd*⟩ locally, while \DTLgneg sets ⟨*cmd*⟩ globally. Example:

```
\DTLneg{\result}{\pounds2.50}\result
```

produces: -£2.50.

\DTLsqrt

\DTLsqrt{⟨*cmd*⟩}{⟨*num*⟩}

\DTLgsqrt

\DTLgsqrt{⟨*cmd*⟩}{⟨*num*⟩}

This sets ⟨*cmd*⟩ to the sqrt root of ⟨*num*⟩. \DLTsqrt sets ⟨*cmd*⟩ locally, while \DTLgsqrt sets ⟨*cmd*⟩ globally. Example:

```
\DTLsqrt{\result}{2}\result
```

produces: 1.414213562373095042.

\DTLmin

\DTLmin{⟨*cmd*⟩}{⟨*num1*⟩}{⟨*num2*⟩}

\DTLgmin

\DTLgmin{⟨*cmd*⟩}{⟨*num1*⟩}{⟨*num2*⟩}

This sets the control sequence ⟨*cmd*⟩ to the minimum of ⟨*num1*⟩ and ⟨*num2*⟩. \DLTmin sets ⟨*cmd*⟩ locally, while \DTLgmin sets ⟨*cmd*⟩ globally. For example:

```
\DTLmin{\result}{256}{32}\result
```

produces: 32. Again, ⟨*num1*⟩ and ⟨*num2*⟩ may be currency. For example:

```
\DTLmin{\result}{256}{\pounds32}\result
```

produces: £32, whereas

```
\DTLmin{\result}{\pounds256}{32}\result
```

produces: 32. As mentioned above, datatool doesn't know about exchange rates, so be careful about mixing currencies. For example:

```
\DTLmin{\result}{\pounds5}{\$6}\result
```

produces: £5, which may not necessarily be true!

\DTLminall

\DTLminall{⟨*cmd*⟩}{⟨*number list*⟩}

\DTLgminall

\DTLgminall{⟨*cmd*⟩}{⟨*number list*⟩}

This sets the control sequence ⟨*cmd*⟩ to the minimum of all the numbers in ⟨*number list*⟩. \DLTminall sets ⟨*cmd*⟩ locally, while \DTLgminall sets ⟨*cmd*⟩ globally. Example:

```
\DTLminall{\theMin}{25.1,45.2,35.6}\theMin
```

produces: 25.1. Note that if any of the numbers in ⟨*number list*⟩ contain a comma, you must group the number. Example:

```
\DTLminall{\theMin}{{1,525},{2,340},500}\theMin
```

produces: 500.

\DTLmax

\DTLmax{⟨*cmd*⟩}{⟨*num1*⟩}{⟨*num2*⟩}

\DTLgmax

\DTLgmax{⟨*cmd*⟩}{⟨*num1*⟩}{⟨*num2*⟩}

This sets the control sequence ⟨*cmd*⟩ to the maximum of ⟨*num1*⟩ and ⟨*num2*⟩. \DLTmax sets ⟨*cmd*⟩ locally, while \DTLgmax sets ⟨*cmd*⟩ globally. For example:

```
\DTLmax{\result}{256}{32}\result
```

produces: 256. Again, ⟨*num1*⟩ and ⟨*num2*⟩ may be currency, but the same warnings for \DTLmin apply.

\DTLmaxall

\DTLmaxall{⟨*cmd*⟩}{⟨*number list*⟩}

\DTLgmaxall

\DTLgmaxall{⟨*cmd*⟩}{⟨*number list*⟩}

This sets the control sequence ⟨*cmd*⟩ to the maximum of all the numbers in ⟨*number list*⟩. `\DLTmaxall` sets ⟨*cmd*⟩ locally, while `\DTLgmaxall` sets ⟨*cmd*⟩ globally. Example:

```
\DTLmaxall{\theMax}{25.1,45.2,35.6}\theMax
```

produces: 45.2. Note that if any of the numbers in ⟨*number list*⟩ contain a comma, you must group the number. Example:

```
\DTLmaxall{\theMax}{{1,525},{2,340},500}\theMax
```

produces: 2,340.

\DTLmeanforall

\DTLmeanforall{⟨*cmd*⟩}{⟨*number list*⟩}

\DTLgmeanall

\DTLgmeanforall{⟨*cmd*⟩}{⟨*number list*⟩}

This sets the control sequence ⟨*cmd*⟩ to the arithmetic mean of all the numbers in ⟨*number list*⟩. `\DLTmeanforall` sets ⟨*cmd*⟩ locally, while `\DTLgmeanforall` sets ⟨*cmd*⟩ globally. Example:

```
\DTLmeanforall{\theMean}{25.1,45.2,35.6}\theMean
```

produces: 35.3. Note that if any of the numbers in ⟨*number list*⟩ contain a comma, you must group the number. Example:

```
\DTLmeanforall{\theMean}{{1,525},{2,340},500}\theMean
```

produces: 1,455.

\DTLvarianceforall

\DTLvarianceforall{⟨*cmd*⟩}{⟨*number list*⟩}

\DTLgvarianceforall

\DTLgvarianceforall{⟨*cmd*⟩}{⟨*number list*⟩}

This sets the control sequence ⟨*cmd*⟩ to the variance of all the numbers in ⟨*number list*⟩. `\DLTvarianceforall` sets ⟨*cmd*⟩ locally, while `\DTLgvarianceforall` sets ⟨*cmd*⟩ globally. Example:

```
\DTLvarianceforall{\theVar}{25.1,45.2,35.6}\theVar
```

produces: 67.38. Again note that if any of the numbers in ⟨*number list*⟩ contain a comma, you must group the number.

\DTLsdforall

\DTLsdforall{⟨*cmd*⟩}{⟨*number list*⟩}

\DTLgsdforall

\DTLgsdforall{⟨*cmd*⟩}{⟨*number list*⟩}

This sets the control sequence ⟨*cmd*⟩ to the standard deviation of all the numbers in ⟨*number list*⟩. \DTLsdforall sets ⟨*cmd*⟩ locally, while \DTLgsdforall sets ⟨*cmd*⟩ globally. Example:

```
\DTLsdforall{\theSD}{25.1,45.2,35.6}\theSD
```

produces: 8.208532146492453016. Note that if any of the numbers in ⟨*number list*⟩ contain a comma, you must group the number. Example:

```
\DTLsdforall{\theSD}{{1,525},{2,340},500}\theSD
```

produces: 752.805862534735216539.

\DTLround

\DTLround{⟨*cmd*⟩}{⟨*num*⟩}{⟨*num digits*⟩}

\DTLground

\DTLground{⟨*cmd*⟩}{⟨*num*⟩}{⟨*num digits*⟩}

This sets ⟨*cmd*⟩ to ⟨*num*⟩ rounded to ⟨*num digits*⟩ after the decimal character. \DTLround sets ⟨*cmd*⟩ locally, while \DTLground sets ⟨*cmd*⟩ globally. Example:

```
\DTLround{\result}{3.135276}{2}\result
```

produces: 3.14.

\DTLtrunc

\DTLtrunc{⟨*cmd*⟩}{⟨*num*⟩}{⟨*num digits*⟩}

\DTLgtrunc

\DTLgtrunc{⟨*cmd*⟩}{⟨*num*⟩}{⟨*num digits*⟩}

This sets ⟨*cmd*⟩ to ⟨*num*⟩ truncated to ⟨*num digits*⟩ after the decimal character. \DTLtrunc sets ⟨*cmd*⟩ locally, while \DTLgtrunc sets ⟨*cmd*⟩ globally. Example:

```
\DTLtrunc{\result}{3.135276}{2}\result
```

produces: 3.13.

\DTLclip

\DTLclip{⟨*cmd*⟩}{⟨*num*⟩}

\DTLgclip

\DTLgclip{⟨*cmd*⟩}{⟨*num*⟩}

This sets ⟨*cmd*⟩ to ⟨*num*⟩ with all unnecessary 0's removed. \DLTclip sets ⟨*cmd*⟩ locally, while \DTLgclip sets ⟨*cmd*⟩ globally.

4 Strings

Strings are considered to be anything non-numerical. The datatool package loads the substr package, so you can use the commands defined in that package to determine if one string is contained in another string. In addition, the datatool provides the following macros:

\DTLsubstitute

\DTLsubstitute{⟨*cmd*⟩}{⟨*original*⟩}{⟨*replacement*⟩}

This replaces the first occurrence of ⟨*original*⟩ in ⟨*cmd*⟩ with ⟨*replacement*⟩. Note that ⟨*cmd*⟩ must be the name of a command. For example:

```
\def\mystr{abcdce}\DTLsubstitute{\mystr}{c}{z}\mystr
```

produces: abzdce.

\DTLsubstituteall

\DTLsubstituteall{⟨*cmd*⟩}{⟨*original*⟩}{⟨*replacement*⟩}

This replaces all occurrences of ⟨*original*⟩ in ⟨*cmd*⟩ with ⟨*replacement*⟩, where again, ⟨*cmd*⟩ must be the name of a command. For example:

```
\def\mystr{abcdce}\DTLsubstituteall{\mystr}{c}{z}\mystr
```

produces: abzdze.

\DTLsplitstring

\DTLsplitstring{⟨*string*⟩}{⟨*split text*⟩}{⟨*before cmd*⟩}{⟨*after cmd*⟩}

This splits ⟨*string*⟩ at the first occurrence of ⟨*split text*⟩ and stores the before part in the command ⟨*before cmd*⟩ and the after part in the command ⟨*after cmd*⟩. For example:

```
\DTLsplitstring{abcdce}{c}{\beforepart}{\afterpart}%
Before part: ``\beforepart''. After part: ``\afterpart''
```

produces: Before part: “ab”. After part: “dce”. Note that for \DTLsplitstring, ⟨*string*⟩ is not expanded, so

```
\def\mystr{abcdce}%
\DTLsplitstring{\mystr}{c}{\beforepart}{\afterpart}%
Before part: ``\beforepart''. After part: ``\afterpart''
```

produces: Before part: “abcdce”. After part: “”. If you want the string expanded, you will need to use `\expandafter`:

```
\def\mystr{abcdce}%
\expandafter\DTLsplitstring\expandafter
{\mystr}{c}{\beforepart}{\afterpart}%
Before part: ``\beforepart''. After part: ``\afterpart''
```

which produces: Before part: “ab”. After part: “dce”.

`\DTLinitials`

`\DTLinitials{`⟨*string*⟩`}`

This converts ⟨*string*⟩ (typically a name) into initials. For example:

```
\DTLinitials{Mary Ann}
```

produces: M.A. (including the final full stop). Note that

```
\DTLinitials{Mary-Ann}
```

produces: M.-A. (including the final full stop). Be careful if the initial letter has an accent. The accented letter needs to be placed in a group, if you want the initial to also have an accent, otherwise the accent command will be ignored. For example:

```
\DTLinitials{{\'E}lise Adams}
```

produces: É.A., whereas

```
\DTLinitials{\'Elise Adams}
```

produces: E.A. In fact, any command which appears at the start of the name that is not enclosed in a group will be ignored. For example:

```
\DTLinitials{\MakeUppercase{m}ary ann}
```

produces: m.a., whereas

```
\DTLinitials{{\MakeUppercase{m}}ary ann}
```

produces: M.a., but note that

```
\DTLinitials{\MakeUppercase{mary ann}}
```

produces: mary ann.

`\DTLstoreinitials`

`\DTLstoreinitials{`⟨*string*⟩`}{`⟨*cmd*⟩`}`

This converts ⟨*string*⟩ into initials and stores the result in ⟨*cmd*⟩ which must be a command name. The remarks about `\DTLinitials` also relate to `\DTLstoreinitials`. For example

```
\DTLstoreinitials{Marie-{\'E}lise del~Rosario}{\theInitials}\theInitials
```

produces: M.-É.d.R.

Both the above commands rely on the following to format the initials:

`\DTLafterinitials`

```
\DTLafterinitials
```

This indicates what to do at the end of the initials. This simply does a full stop by default.

`\DTLbetweeninitials`

```
\DTLbetweeninitials
```

This indicates what to do between initials. This does a full stop by default.

`\DTLinitialhyphen`

```
\DTLinitialhyphen
```

This indicates what to do at a hyphen. This simply does a hyphen by default, but can be redefined to do nothing to prevent the hyphen appearing in the initials.

`\DTLafterinitialbeforehyphen`

```
\DTLafterinitialbeforehyphen
```

This indicates what to do between an initial and a hyphen. This simply does a full stop by default.

For example

```
\renewcommand*{\DTLafterinitialbeforehyphen}{}%
\DTLinitials{Marie-{\'E}lise del~Rosario}
```

produces: M-É.d.R. whereas

```
\renewcommand*{\DTLafterinitialbeforehyphen}{}%
\renewcommand*{\DTLafterinitials}{}%
\renewcommand*{\DTLbetweeninitials}{}%
\renewcommand*{\DTLinitialhyphen}{}%
\DTLinitials{Marie-{\'E}lise del~Rosario}
```

produces: MÉdR

5 Databases

The datatool package provides a means of creating and loading databases. Once a database has been created (or loaded), it is possible to iterate through each row of data, to make it easier to perform repetitive actions, such as mail merging.

Whilst TEX is an excellent typesetting language, it is not designed as a database management system, and attempting to use it as such is like trying to fasten a screw with a knife instead of a screwdriver: it can be done, but requires great care and is more time consuming. Version 2.0 of the datatool package uses a completely different method of storing the data to previous versions.[a] As a result, the code is much more efficient, however, large databases and complex operations will still slow the time taken to process your document. Therefore, if you can, it is better to do the complex operations using whatever system created the data in the first place.

[a] Many thanks to Morten Høgholm for providing the new code.

Some advanced commands for accessing database information are described in section 5.11, but using TEX is nowhere near as efficient as, say, using a SQL database, so don't expect too much from this package.

I've written a Java helper application to accompany datatool called `datatooltk`. The installer is available on CTAN at `http://mirrors.ctan.org/support/datatooltk/datatooltk-installer.jar`. The application will allow you to edit files saved using `\DTLsaverawdb` or `\DTLprotectedsaverawdb` in a graphical interface or import data from a SQL database, a CSV file or a probsoln dataset.

5.1 Creating a New Database

`\DTLnewdb`

`\DTLnewdb{`⟨*db name*⟩`}`

`\DTLgnewdb`

`\DTLgnewdb{`⟨*db name*⟩`}`

This command creates a new empty database called ⟨*db name*⟩. The second form is for global definitions. You can test if a database is empty using:

\DTLifdbempty

\DTLifdbempty{⟨*db name*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

If the database called ⟨*db name*⟩ is empty, do ⟨*true part*⟩, otherwise do ⟨*false part*⟩.

\DTLrowcount

\DTLrowcount{⟨*db name*⟩}

This command displays the number of rows in the database called ⟨*db name*⟩.

\DTLcolumncount

\DTLcolumncount{⟨*db name*⟩}

This command displays the number of columns (or keys) in the database called ⟨*db name*⟩.

\DTLnewrow

\DTLnewrow{⟨*db name*⟩}

This command starts a new row in the database called ⟨*db name*⟩. This new row becomes the current row when adding new entries.

For example, the following creates an empty database called `mydata`:

```
\DTLnewdb{mydata}
```

The following tests if the database is empty:

```
\DTLifdbempty{mydata}{empty}{not empty}!
```

This produces: empty!

The following adds an empty row to the database, this is the first row of the database:

```
\DTLnewrow{mydata}
```

Note that even though the only row in the database is currently empty, the database is no longer considered to be empty:

```
\DTLifdbempty{mydata}{empty}{not empty}!
```

This now produces: not empty! The row count is given by

```
\DTLrowcount{mydata}
```

which produces: 1. The column count is given by

```
\DTLcolumncount{mydata}
```

which produces: 0.

\DTLnewdbentry

\DTLnewdbentry{⟨*db name*⟩}{⟨*key*⟩}{⟨*value*⟩}

This creates a new entry with the identifier ⟨*key*⟩ whose value is ⟨*value*⟩ and adds it to the last row of the database called ⟨*db name*⟩. For example:

```
\DTLnewdbentry{mydata}{Surname}{Smith}
\DTLnewdbentry{mydata}{FirstName}{John}
```

Adds an entry with identifier `Surname` and value `Smith` to the last row of the database named `mydata`, and then adds an entry with identifier `FirstName` and value `John`. Note that the key should not contain any fragile commands. It is generally best to only use non-active characters in the key.

The value isn't expanded by default, but you can change this using the declaration:

\dtlexpandnewvalue

\dtlexpandnewvalue

This can be localised by placing it in a group, or you can switch back using:

\dtlnoexpandnewvalue

\dtlnoexpandnewvalue

Note that database entries can't contain paragraph breaks as many of the macros used by datatool are short commands. If you do need a paragraph break in an entry, you can instead use the command:

\DTLpar

\DTLpar

For example:

```
\DTLnewdbentry{mydata}{Description}{First paragraph.\DTLpar
Second paragraph.}
```

\DTLaddentryforrow

\DTLaddentryforrow{⟨db⟩}{⟨assign list⟩}{⟨condition⟩}{⟨key⟩}{⟨value⟩}

This adds the entry with the key given by ⟨*key*⟩ and value given by ⟨*value*⟩ to the first row in the database ⟨*db*⟩ which satisfies the condition given by ⟨*condition*⟩. The ⟨*assign list*⟩ argument is the same as for `\DTLforeach` (described in section 5.4) and may be used to set the values which are to be tested in ⟨*condition*⟩ (where, again, ⟨*condition*⟩ is the same as for `\DTLforeach`). For example:

```
\DTLaddentryforrow{mydata}{\firstname=FirstName,\surname=Surname}%
{\DTLiseq{\firstname}{John}\and\DTLiseq{\surname}{Smith}}%
{Score}{75}
```

Note that unlike `\DTLnewdbentry`, the value is always expanded when adding an entry using `\DTLaddentryforrow`.

`\DTLsetheader`

\DTLsetheader{⟨db⟩}{⟨key⟩}{⟨header⟩}

This assigns a header for a given key in the database named ⟨*db*⟩. This is used by `\DTLdisplaydb` and `\DTLdisplaylongdb` in the header row (see section 5.3). If you don't assign a header, the header will be given by the key. For example:

```
\DTLsetheader{mydata}{Price}{Price (\$)}
```

`\DTLaddcolumn`

\DTLaddcolumn{⟨db⟩}{⟨key⟩}

Adds a new column with the given key to the database ⟨*db*⟩. This doesn't add any data to the column, just identifies it as an available column. The starred version doesn't check if the database exists.

5.2 Loading a Database from an External ASCII File

`\DTLloaddb` and `\DTLloadrawdb`, described in this section, can't parse files that have newline characters within entries. The `datatooltk` application (see page 36) can parse them, so if you have multilined entries in a CSV file, you can convert it to datatool's internal database format using `datatooltk` and the input it using `\input`. See the `datatooltk` documentation for further details.

Instead of using the commands described in section 5.1 to create a new database, you can load a database from an external ASCII file using:

\DTLloaddb

```
\DTLloaddb[⟨options⟩]{⟨db name⟩}{⟨filename⟩}
```

Make sure your document uses the same encoding as ⟨*filename*⟩. For example, if ⟨*filename*⟩ is UTF-8, then include the following in your document:

```
\usepackage[utf8]{inputenc}
```

By default, \DTLloaddb creates a new database called ⟨*db name*⟩ before it loads the data given in the file ⟨*filename*⟩. If you want to append the data, use

```
\DTLnewdbonloadfalse
```

before you use \DTLloaddb. You can reverse this using

```
\DTLnewdbonloadtrue
```

The file (⟨*filename*⟩) may have a header row at the start of the file, which provides the ⟨*key*⟩ when creating a new database entry using \DTLnewdbentry. The optional argument ⟨*options*⟩ is a key=value list of options. Available options are:

noheader This is a boolean value and indicates if the file does not contain a header. If no value is supplied, true is assumed (i.e. the file doesn't contain a header row). If this option is omitted, it is assumed that the file contains a header row.

keys This is a comma-separated list of keys to use, where the keys are listed in the same order as the columns. If the file has a header, these keys will override the values given in the header row. If the file has no header row and no keys are supplied in ⟨*options*⟩, then the keys will be given by \dtldefaultkey⟨*n*⟩, where ⟨*n*⟩ is the column number and \dtldefaultkey defaults to "Column". Note that the list of keys must be delimited by braces since they contain commas. For example:

\dtldefaultkey

```
\DTLloaddb[noheader,keys={Temperature,Time,T2G}]{data}{data.csv}
```

headers This is a comma-separated list of headers. If not supplied, the header will be the same as that given in the header row, or the key if there is no header row. Note that the list of headers must be delimited by braces since they contain commas. For example:

```
\DTLloaddb[noheader,keys={Temperature,Time,T2G},%
headers={\shortstack{Incubation\\Temperature},%
\shortstack{Incubation\\Time},%
\shortstack{Time to\\Growth}}]{data}{data.csv}
```

omitlines This should be a non-negative integer that specifies how many rows to skip at the start of the file.

By default, the entries in the database must be separated by a comma, and optionally delimited by the double quote character (`"`). The separator can be changed to a tab separator using the command:

`\DTLsettabseparator`

```
\DTLsettabseparator
```

To set the separator to a character other than a tab, you need to use

`\DTLsetseparator`

```
\DTLsetseparator{⟨character⟩}
```

The delimiter can be changed using

`\DTLsetdelimiter`

```
\DTLsetdelimiter{⟨character⟩}
```

For example, suppose you have a file called `mydata.csv` which contains the following:

```
FirstName,Surname,Score
John,"Smith, Jr",68
Jane,Brown,75
Andy,Brown,42
Z\"oe,Adams,52
```

then

```
\DTLloaddb{mydata}{mydata.csv}
```

is equivalent to:

```
\DTLnewdb{mydata}
\DTLnewrow{mydata}%
\DTLnewdbentry{mydata}{FirstName}{John}%
```

```
\DTLnewdbentry{mydata}{Surname}{Smith, Jr}%
\DTLnewdbentry{mydata}{Score}{68}%
\DTLnewrow{mydata}%
\DTLnewdbentry{mydata}{FirstName}{Jane}%
\DTLnewdbentry{mydata}{Surname}{Brown}%
\DTLnewdbentry{mydata}{Score}{75}%
\DTLnewrow{mydata}%
\DTLnewdbentry{mydata}{FirstName}{Andy}%
\DTLnewdbentry{mydata}{Surname}{Brown}%
\DTLnewdbentry{mydata}{Score}{42}%
\DTLnewrow{mydata}%
\DTLnewdbentry{mydata}{FirstName}{Z\"oe}%
\DTLnewdbentry{mydata}{Score}{52}%
\DTLnewdbentry{mydata}{Surname}{Adams}%
```

Note that the entry `Smith, Jr` had to be delimited in `mydata.csv` using the double quote character since it contained a comma which is used as the separator. The percent symbol % can be used as a comment character within the file.

The file used in the above example contained a LaTeX command, namely `\"`. When using `\DTLloaddb` all the special characters that appear in the command retain their LaTeX meaning when the file is loaded. It is likely however that the data file may have been created by another application that is not TeX-aware, such as a spreadsheet application. For example, suppose you have a file called, say, `products.csv` which looks like:

```
Product,Cost
Fruit & Veg,$1.25
Stationary,$0.80
```

This file contains two of TeX's special characters, namely `&` and `$`. In this case, if you try to load the file using `\DTLloaddb`, you will encounter errors. Instead you can use:

`\DTLloadrawdb`

`\DTLloadrawdb[`⟨*options*⟩`]{`⟨*db name*⟩`}{`⟨*filename*⟩`}`

This is the same as `\DTLloaddb` except that it maps nine of the ten special characters onto commands which produce that symbol. The only character that retains its active state is the backslash character, so you will still need to check the file for backslash characters. The mappings used are listed in Table 5.1. So using the file `products.csv`, as described above,

```
\DTLloadrawdb{mydata}{products.csv}
```

is equivalent to:

```
\DTLnewdb{mydata}
\DTLnewrow{mydata}%
\DTLnewdbentry{mydata}{Product}{Fruit \& Veg}%
\DTLnewdbentry{mydata}{Cost}{\$1.25}%
\DTLnewrow{mydata}%
\DTLnewdbentry{mydata}{Product}{Stationary}%
\DTLnewdbentry{mydata}{Cost}{\$0.80}%
```

As with `\DTLloaddb`, you can govern whether or not a new database should be created with `\DTLnewdbonloadtrue` and `\DTLnewdbonloadfalse`.

Table 5.1: Special character mappings used by `\DTLloadrawdb` (note that the backslash retains its active state)

Character	Mapping
%	\%
$	\$
&	\&
#	\#
_	_
{	\{
}	\}
~	\textasciitilde
^	\textasciicircum

It may be that there are other characters that require mapping. For example, the file `products.csv` may instead look like:

```
Product,Cost
Fruit & Veg,£1.25
Stationary,£0.80
```

The pound character is not an internationally standard keyboard character, and does not generally achieve the desired effect when used in a LaTeX document. It may therefore be necessary to convert this symbol to an appropriate control sequence. This can be done using the command:

`\DTLrawmap`

`\DTLrawmap{`⟨*string*⟩`}{`⟨*replacement*⟩`}`

For example:

```
\DTLrawmap{£}{\pounds}
```

will replace all occurrences[1] of £ with \pounds. Naturally, the mappings must be set *prior* to loading the data with \DTLloadrawdb.

Note that the warning in the previous section about no paragraph breaks in an entry also applies to entries loaded from a database. If you do need a paragraph break, use \DTLpar instead of \par, but remember that each row of data in an external data file must not have a line break.

5.3 Displaying the Contents of a Database

Once you have created a database, either loading it from an external file, as described in section 5.2, or using the commands described in section 5.1, you can display the entire database in a tabular or longtable environment.

\DTLdisplaydb

\DTLdisplaydb[⟨*omit list*⟩]{⟨*db*⟩}

This displays the database given by ⟨*db*⟩ in a tabular environment. The first row displays the headers for the database in bold, the subsequent rows display the values for each key in each row of the database. The optional argument ⟨*omit list*⟩ is a comma-separated list of column keys to omit. (All columns displayed by default.)

\DTLdisplaylongdb

\DTLdisplaylongdb[⟨*options*⟩]{⟨*db*⟩}

This is like \DTLdisplaydb except that it uses the longtable environment instead of the tabular environment. Note that if you use this command, you must load the longtable package, as it is not loaded by datatool. The optional argument ⟨*options*⟩ is a comma-separated list of key=value pairs. The following keys are available:

caption The caption for the longtable.

contcaption The continuation caption.

shortcaption The caption to be used in the list of figures.

label The label for this table.

omit Comma-separated list of column keys to omit.

[1] when it is loaded into the LaTeX database, it does not modify the data file!

foot The longtable's foot.

lastfoot The foot for the last page of the longtable.

For example, suppose I have a database called `iris`, then I can display the contents in a longtable using:

```
\DTLdisplaylongdb[%
caption={Iris Data},%
label={tab:iris},%
contcaption={Iris Data (continued)},%
foot={\em Continued overleaf},%
lastfoot={}%
]{iris}
```

I can then reference the table using `\ref{tab:iris}`.

See the longtable documentation for details on how to change the longtable settings, such as how to change the table so that it is left aligned instead of centred on the page.

Note that if you want more control over the way the data is displayed, for example, you want to filter rows or columns, you will need to use `\DTLforeach`, described in section 5.4.

Example 1 (Displaying the Contents of a Database)

Suppose I have a file called `t2g.csv` that contains the following:

```
40,120,40
40,90,60
35,180,20
55,190,40
```

This represents time to growth data, where the first column is the incubation temperature, the second column is the incubation time and the third column is the time to growth. This file has no header row, so when it is loaded, the noheaders option is required. Note that `\DTLdisplaydb` only puts the data in a tabular environment, so `\DTLdisplaydb` needs to be put in a table environment with a caption to make it a float.

First load the data base, setting the keys and headers:

```
\DTLloaddb[noheader,%
keys={Temperature,Time,T2G},%
headers={\shortstack{Incubation\\Temperature},%
\shortstack{Incubation\\Time},\shortstack{Time to\\Growth}}%
]{t2g}{t2g.csv}
```

Now display the data in a table:

```
\begin{table}[htbp]
\caption{Time to Growth Data}
```

```
\centering
\DTLdisplaydb{t2g}
\end{table}
```

The result is shown in Table 5.2.

Table 5.2: Time to Growth Data

Incubation Temperature	Incubation Time	Time to Growth
40	120	40
40	90	60
35	180	20
55	190	40

Each column in the database has an associated data type which indicates what type of data is in that column. This may be one of: string, integer, real number or currency. If a column contains more than one type, the data type is determined as follows:

- If the column contains at least one string, then the column data type is string.
- If the column doesn't contain a string, but contains at least one currency, then the column data type is currency.
- If the column contains only real numbers and integers, the column data type is real number.
- The column data type is integer if the column only contains integers.

The column data type is updated whenever a new entry is added to the database. Note that the column data type is not adjusted when an entry is removed from the database.

The column alignments used by `\DTLdisplaydb` are given by:

`\dtlstringalign`

```
\dtlstringalign
```

The string alignment defaults to `l` (left aligned).

`\dtlintalign`

```
\dtlintalign
```

The integer alignment defaults to `r` (right aligned).

`\dtlrealalign`

```
\dtlrealalign
```

The alignment for real numbers defaults to `r` (right aligned).

`\dtlcurrencyalign`

```
\dtlcurrencyalign
```

The currency alignment defaults to `r` (right aligned).

You can redefine these to change the column alignments. For example, if you want columns containing strings to have the alignment `p{2in}`, then you can redefine `\dtlstringalign` as follows:

```
\renewcommand{\dtlstringalign}{p{2in}}
```

You can't use siunitx's `S` column alignment with either `\DTLdisplaydb` or `\DTLdisplaylongdb`. Instead, you will need to use `\DTLforeach`. The siunitx documentation provides an example.

`\dtlbeforecols` `\dtlbetweencols` `\dtlaftercols`

In addition to the `\dtl`⟨*type*⟩`align` commands above, you can also modify the tabular column styles by redefining `\dtlbeforecols`, `\dtlbetweencols` and `\dtlaftercols`. For example, to place a vertical line before the start of the first column and after the last column, do:

```
\renewcommand{\dtlbeforecols}{|}
\renewcommand{\dtlaftercols}{|}
```

If you additionally want vertical lines between columns, do:

```
\renewcommand{\dtlbetweencols}{|}
```

Limited modifications can be made to the way the data is displayed with `\DTLdisplaydb` and `\DTLdisplaylongdb`. The commands controlling the formatting are described below. If a more complicated layout is required, you will need to use `\DTLforeach` described in section 5.4.

`\dtlheaderformat`

```
\dtlheaderformat{⟨header⟩}
```

This indicates how to format a column header, where the header is given by ⟨*header*⟩. This defaults to `\null\hfil\textbf{`⟨*header*⟩`}\hfil\null`.

\dtlstringformat

\dtlstringformat{⟨*text*⟩}

This specifies how to format each entry in the columns that contain strings. This defaults to just displaying ⟨*text*⟩.

\dtlintformat

\dtlintformat{⟨*text*⟩}

This specifies how to format each entry in the columns that contain only integers. This defaults to just displaying ⟨*text*⟩.

\dtlrealformat

\dtlrealformat{⟨*text*⟩}

This specifies how to format each entry in the columns that contain only real numbers or a mixture of real numbers and integers. This defaults to just displaying ⟨*text*⟩.

\dtlcurrencyformat

\dtlcurrencyformat{⟨*text*⟩}

This specifies how to format each entry in the columns that contain only currency or currency mixed with real numbers and/or integers. This defaults to just displaying ⟨*text*⟩.

\dtldisplayvalign

\dtldisplayvalign

Specifies the vertical alignment of the tabular environment used by `\DTLdisplaydb`. Defaults to `c` (centred). May be redefined to `t` (top) or `b` (bottom).

\dtldisplaystarttab

\dtldisplaystarttab

This is a hook to add something at the beginning of the tabular environment. This defaults to nothing.

\dtldisplayendtab

\dtldisplayendtab

This is a hook to add something at the end of the tabular environment. This defaults to nothing.

\dtldisplayafterhead

```
\dtldisplayafterhead
```

This is a hook to add something after the header row, before the first row of data. This defaults to nothing.

\dtldisplaystartrow

```
\dtldisplaystartrow
```

This is a hook to add something at the start of each row, but not including the header row or the first row of data. This defaults to nothing.

If you want to use the booktabs package, you can redefine the above three commands to use \toprule, \midrule and \bottomrule:

```
\renewcommand{\dtldisplaystarttab}{\toprule}
\renewcommand{\dtldisplayafterhead}{\midrule}
\renewcommand{\dtldisplayendtab}{\\\bottomrule}
```

Example 2 (Balance Sheet)

Suppose you have a file called balance.csv that contains the following:

```
Description,In,Out,Balance
Travel expenses,,230,-230
Conference fees,,400,-630
Grant,700,,70
Train fare,,70,0
```

The data can be loaded using:

```
\DTLloaddb[headers={Description,In (\pounds),Out (\pounds),Balance
(\pounds)}]{balance}{balance.csv}
```

Suppose I want negative numbers to be displayed in red. I can do this by redefining \dtlrealformat to check if the entry is negative. For example:

```
\begin{table}[htbp]
\caption{Balance Sheet}
\renewcommand*{\dtlrealformat}[1]{\DTLiflt{#1}{0}{\color{red}}{}#1}
\centering
\DTLdisplaydb{balance}
\end{table}
```

This produces Table 5.3.

Table 5.3: Balance Sheet

Description	In (£)	Out (£)	Balance (£)
Travel expenses		230.00	-230.00
Conference fees		400.00	-630.00
Grant	700.00		70.00
Train Fare		70.00	0.00

5.4 Iterating Through a Database

Once you have created a database, either loading it from an external file, as described in section 5.2, or using the commands described in section 5.1, you can then iterate through each row of the database and access elements in that row.

\DTLforeach

\DTLforeach[⟨*condition*⟩]{⟨*db name*⟩}{⟨*assign list*⟩}{⟨*text*⟩}

\DTLforeach*

\DTLforeach*[⟨*condition*⟩]{⟨*db name*⟩}{⟨*assign list*⟩}{⟨*text*⟩}

This will iterate through each row of the database called ⟨*db name*⟩, applying ⟨*text*⟩ to each row of the database where ⟨*condition*⟩ is met. The argument ⟨*assign list*⟩ is a comma separated list of ⟨*cmd*⟩=⟨*key*⟩ pairs. At the start of each row, each command ⟨*cmd*⟩ in ⟨*assign list*⟩ will be set to the value of the entry given by ⟨*key*⟩. These commands may then be used in ⟨*text*⟩.

Note that this assignment is done globally to ensure that \DLTforeach works correctly in a tabular environment. Since you may want to use the same set of commands in a later \DTLforeach, the commands are not checked to determine if they already exist. It is therefore important that you check you are not using an existing command whose value should not be changed.

The optional argument ⟨*condition*⟩ is a condition in the form allowed by \ifthenelse. This includes the commands provided by the ifthen package (such as \not, \and, \or), as well as the commands described in section 2.2. The default value of ⟨*condition*⟩ is \boolean{true}.

The starred version \DTLforeach* is a read-only version. If you want to modify the database using any of the commands described in section 5.6, you must use the unstarred version. The starred version is

faster.

As is generally the case with command arguments, verbatim (for example, using `\verb` or the verbatim environment) can't be used in any of the arguments of `\DTLforeach`, specifically verbatim can't be used in ⟨*text*⟩.

There are also environment alternatives:

`DTLenvforeach`

`\begin{DTLenvforeach}[`⟨*condition*⟩`]{`⟨*db name*⟩`}{`⟨*assign list*⟩`}`

`DTLenvforeach*`

`\begin{DTLenvforeach*}[`⟨*condition*⟩`]{`⟨*db name*⟩`}{`⟨*assign list*⟩`}`

However, note that since these environments gather the contents of their body, they also suffer from the above limitation.

Verbatim can't be used in the body of DTLenvforeach or DTLenvforeach*.

Example 3 (Student scores)

Suppose you have a data file called `studentscores.csv` that contains the following:

```
FirstName,Surname,StudentNo,Score
John,"Smith, Jr",102689,68
Jane,Brown,102647,75
Andy,Brown,103569,42
Z\"oe,Adams,105987,52
Roger,Brady,106872,58
Clare,Verdon,104356,45
```

and you load the data into a database called `scores` using:

```
\DTLloaddb{scores}{studentscores.csv}
```

you can then display the database in a table as follows:

```
\begin{table}[htbp]
\caption{Student scores}
\centering
\begin{tabular}{llr}
\bfseries First Name &
\bfseries Surname &
\bfseries Score (\%)%
```

```
\DTLforeach{scores}{%
\firstname=FirstName,\surname=Surname,\score=Score}{%
\\
\firstname & \surname & \score}
\end{tabular}
\end{table}
```

This produces Table 5.4. (Note that since I didn't need the student registration number, I didn't bother to assign a command to the key `StudentNo`.)

Table 5.4: Student scores

First Name	Surname	Score (%)
John	Smith, Jr	68
Jane	Brown	75
Andy	Brown	42
Zöe	Adams	52
Roger	Brady	58
Clare	Verdon	45

The macro `\DTLforeach` may be nested up to three times. Each level uses the corresponding counters: DTLrowi, DTLrowiiand DTLrowiiiwhich keep track of the current row.

DTLrowi
DTLrowii
DTLrowiii

Note that these counters are only incremented when ⟨*condition*⟩ is satisfied, therefore they will not have the correct value in ⟨*condition*⟩. These counters are incremented using `\refstepcounter` before the start of ⟨*text*⟩, so they may be referenced using `\label`, however remember that `\label` references the last counter to be incremented using `\refstepcounter` *in the current scope*. The `\label` should therefore be the first command in ⟨*text*⟩ to ensure that it references the current row counter.

\DTLcurrentindex

```
\DTLcurrentindex
```

At the start of each iteration in `\DTLforeach`, `\DTLcurrentindex` is set to the arabic value of the current row counter. Note that this is only set after the condition is tested, so it should only be used in the body of `\DTLforeach` not in the condition. It is also only set locally, so if you use it in a tabular environment, it can only be used before the first instance of `\\` or `&` in the current iteration.

Within the body of \DTLforeach (i.e. within ⟨*text*⟩) the following conditionals may be used:

\DTLiffirstrow

\DTLiffirstrow{⟨*true part*⟩}{⟨*false part*⟩}

If the current row is the first row, then do ⟨*true part*⟩, otherwise do ⟨*false part*⟩.

\DTLiflastrow

\DTLiflastrow{⟨*true part*⟩}{⟨*false part*⟩}

If the current row is the last row, then do ⟨*true part*⟩, otherwise do ⟨*false part*⟩.

\DTLifoddrow

\DTLifoddrow{⟨*true part*⟩}{⟨*false part*⟩}

If the current row number is an odd number, then do ⟨*true part*⟩, otherwise do ⟨*false part*⟩.

\DTLsavelastrowcount

\DTLsavelastrowcount{⟨*cmd*⟩}

This command will store the value of the row counter used in the last occurrence of \DTLforeach in the control sequence ⟨*cmd*⟩.

\DTLforeachkeyinrow

\DTLforeachkeyinrow{⟨*cmd*⟩}{⟨*text*⟩}

This iterates through each key in the current row, (globally) assigns ⟨*cmd*⟩ to the value of that key, and does ⟨*text*⟩ (⟨*cmd*⟩ must be a control sequence and may be used in ⟨*text*⟩). This command may only be used in the body of \DTLforeach. At each iteration, \DTLforeachkeyinrow sets \dtlkey to the current key, \dtlcol to the current column index, \dtltype to the data type for the current column, and \dtlheader to the header for the current column. Note that \dtltype corresponds to the column type but if the entries in the column have mixed types, it may not correspond to the type of the current entry.

\dtlbreak

\dtlbreak

You can break out of most of the loops provided by datatool using `\dtlbreak`. Note, however, that it doesn't break the loop until the end of the current iteration. There is no provision for a `next` or `continue` style command.

Additional loop commands provided by datatool are described in the documented code (datatool-code.pdf).

Example 4 (Student Scores—Labelling)

In the previous example, the student scores, stored in the database `scores` were placed in a table. In this example the table will be modified slightly to number each student according to the row. Suppose I also want to identify which row Jane Brown is in, and reference it in the text. The easiest way to do this is to construct a label on each row which uniquely identifies that student. The label can't simply be constructed from the surname, as there are two students with the same surname. In order to create a unique label, I can either construct a label from both the surname and the first name, or I can use the student's registration number, or I can use the student's score, since all the scores are unique. The former method will cause a problem since one of the names (Zöe) contains an accent command. Although the registration numbers are all unique, they are not particularly memorable, so I shall instead use the scores.

```
\begin{table}[htbp]
\caption{Student scores}
\centering
\begin{tabular}{cllc}
\bfseries Row &
\bfseries First Name &
\bfseries Surname &
\bfseries Score (\%)%
\DTLforeach*{scores}{%
\firstname=FirstName,\surname=Surname,\score=Score}{%
\label{row:\score}\\\theDTLrowi &
\firstname & \surname & \score}%
\end{tabular}
\end{table}

Jane Brown scored the highest (75\%), her score can be seen on
row~\ref{row:75}.
```

This produces Table 5.5 and the following text: Jane Brown scored the highest (75%), her score can be seen on row 2.

Notes:

- the `\label` command is placed before `\\` to ensure that it is in the same scope as the command `\refstepcounter{DTLrowi}`.

- To avoid unwanted spaces the end of line characters are commented out with the percent (%) symbol.

Table 5.5: Student scores

Row	First Name	Surname	Score (%)
1	John	Smith, Jr	68
2	Jane	Brown	75
3	Andy	Brown	42
4	Zöe	Adams	52
5	Roger	Brady	58
6	Clare	Verdon	45

Example 5 (Filtering Rows)

As mentioned earlier, the optional argument ⟨*condition*⟩ of `\DTLforeach` provides a means to exclude certain rows. This example uses the database defined in example 3, but only displays the information for students whose marks are above 60. At the end of the table, `\DTLsavelastrowcount` is used to store the number of rows in the table. (Note that `\DTLsavelastrowcount` is outside of `\DTLforeach`.)

```
\begin{table}[htbp]
\caption{Top student scores}
\centering
\begin{tabular}{llr}
\bfseries First Name &
\bfseries Surname &
\bfseries Score (\%)%
\DTLforeach*[\DTLisgt{\score}{60}]{scores}{%
\firstname=FirstName,\surname=Surname,\score=Score}{%
\\
\firstname & \surname & \score}
\end{tabular}

\DTLsavelastrowcount{\n}%
\n\ students scored above 60\%.
\end{table}
```

This produces Table 5.6. Note that in this example, I could have specified the condition as `\score>60` since all the scores are integers, however, as it's possible that an entry may feasibly have a decimal score I have used `\DTLisgt` instead.

Table 5.6: Top student scores

First Name	Surname	Score (%)
John	Smith, Jr	68
Jane	Brown	75

2 students scored above 60%.

Suppose now, I only want to display the scores for students whose surname begins with 'B'. I can do this as follows:

```
\begin{table}[htbp]
\caption{Student scores (B)}
\centering
\begin{tabular}{llr}
\bfseries First Name &
\bfseries Surname &
\bfseries Score (\%)%
\DTLforeach*[\DTLisopenbetween{\surname}{B}{C}]{scores}{%
\firstname=FirstName,\surname=Surname,\score=Score}{%
\\
\firstname & \surname & \score}
\end{tabular}
\end{table}
```

This produces Table 5.7.

Table 5.7: Student scores (B)

First Name	Surname	Score (%)
Jane	Brown	75
Andy	Brown	42
Roger	Brady	58

Example 6 (Checking for the First Row (booktabs))

Suppose I want to use the booktabs package and I want to use `\midrule` after the header row. I can use `\DTLiffirstrow` to check if the loop is on the first row of the iteration. (Remember that you need to load the booktabs package in the preamble with `\usepackage`.) Using the same database as before:

```
\begin{table}[htbp]
\caption{Student scores (booktabs)}
\centering
```

```
\begin{tabular}{llc}
\toprule
\bfseries First Name &
\bfseries Surname &
\bfseries Score (\%)%
\DTLforeach*{scores}%
 {\firstname=FirstName,\surname=Surname,\score=Score}%
 {%
   \DTLiffirstrow{\\\midrule}{\\}%
   \firstname & \surname & \score
 }% end of loop
 \\\bottomrule
\end{tabular}%
\end{table}
```

(The commands `\toprule`, `\midrule` and `\bottomrule` are all provided by booktabs.) This produces Table 5.8.

Table 5.8: Student scores (booktabs)

First Name	Surname	Score (%)
John	Smith, Jr	68
Jane	Brown	75
Andy	Brown	42
Zöe	Adams	52
Roger	Brady	58
Clare	Verdon	45

Example 7 (Breaking Out of a Loop)

Suppose I only want to display the first three rows of a database. I could do:[2]

```
\DTLforeach*[\value{DTLrowi}<3]{scores}%
{\firstname=FirstName,\surname=Surname,\score=Score}{%
\\\firstname & \surname & \score
}
```

However, this isn't very efficient, as it still has to iterate through the entire database, checking if the condition is met. If the database has over 100 entries, this will slow the time taken to create the table. It would

[2]Recall that DTLrowi is incremented after the condition is tested, so it will be out by 1 when the condition is tested which is why `<3` is used instead of `<4`.

therefore be much more efficient to break out of the loop when row count exceeds 3:

```
\begin{table}[htbp]
\caption{First Three Rows}
\centering
\begin{tabular}{llr}
\bfseries First Name & \bfseries Surname & \bfseries Score (\%)%
\DTLforeach*{scores}%
{\firstname=FirstName,\surname=Surname,\score=Score}{%
\ifthenelse{\DTLcurrentindex=3}{\dtlbreak}{}%
\\\firstname & \surname & \score
}%
\end{tabular}
\end{table}
```

This produces Table 5.9. Note that the loop is not broken until the end of the current iteration, so even though `\dtlbreak` occurs at the start of the third row, the loop isn't finished until the third row is completed. (Recall that `\DTLcurrentindex` must be used before the first instance of `\\` or `&`.) Alternatively, you can use DTLrowi instead:

```
\DTLforeach{scores}%
{\firstname=FirstName,\surname=Surname,\score=Score}{%
\\\firstname & \surname & \score
\ifthenelse{\value{DTLrowi}=3}{\dtlbreak}{}%
}%
```

Table 5.9: First Three Rows

First Name	Surname	Score (%)
John	Smith, Jr	68
Jane	Brown	75
Andy	Brown	42

Example 8 (Stripy Tables)

This example uses the same database as in the previous examples. It requires the colortbl package, which provides the command `\rowcolor`. The command `\DTLifoddrow` is used to produce a striped table.

```
\begin{table}[htbp]
\caption{A stripy table}\label{tab:stripy}
\centering
```

```
\begin{tabular}{llc}
\bfseries First Name &
\bfseries Surname &
\bfseries Score (\%)%
\DTLforeach*{scores}{%
\firstname=FirstName,\surname=Surname,\score=Score}{%
\\\DTLifoddrow{\rowcolor{blue}}{\rowcolor{green}}%
\firstname & \surname & \score}%
\end{tabular}
\end{table}
```

This produces Table 5.10.

Table 5.10: A stripy table

First Name	Surname	Score (%)
John	Smith, Jr	68
Jane	Brown	75
Andy	Brown	42
Zöe	Adams	52
Roger	Brady	58
Clare	Verdon	45

Example 9 (Two Database Rows per Tabular Row)

In order to save space, you may want two database rows per tabular row, when displaying a database in a tabular environment. This can be accomplished using `\DTLifoddrow`. For example

```
\begin{table}[htbp]
\caption{Two database rows per tabular row}
\centering
\begin{tabular}{llcllc}
\bfseries First Name &
\bfseries Surname &
\bfseries Score (\%) &
\bfseries First Name &
\bfseries Surname &
\bfseries Score (\%)%
\DTLforeach*{scores}{\firstname=FirstName,\surname=Surname,\score=Score}{%
\DTLifoddrow{\\}{&}%
\firstname & \surname & \score}%
\end{tabular}
\end{table}
```

Table 5.11: Two database rows per tabular row

First Name	Surname	Score (%)	First Name	Surname	Score (%)
John	Smith, Jr	68	Jane	Brown	75
Andy	Brown	42	Zöe	Adams	52
Roger	Brady	58	Clare	Verdon	45

produces Table 5.11
(To order column-wise, instead of row-wise, see example 20.)

Example 10 (Iterating Through Keys in a Row)

Suppose you have lots of columns in your database, and you want to display them all without having to set a variable for each column. You can leave the assignment list in `\DTLforeach` blank, and iterate through the keys using `\DTLforeachkeyinrow`. For example:

```
\begin{table}[htbp]
\caption{Student Scores (Iterating Through Keys)}
\centering
\begin{tabular}{llll}
\bfseries First Name & \bfseries Surname &
\bfseries Registration No. &
\bfseries Score (\%)%
\DTLforeach*{scores}{}{%
\\\gdef\doamp{\gdef\doamp{&}}%
\DTLforeachkeyinrow{\thisValue}{\doamp\thisValue}}%
\end{tabular}
\end{table}
```

This produces Table 5.12.

Table 5.12: Student Scores (Iterating Through Keys)

First Name	Surname	Registration No.	Score (%)
John	Smith, Jr	102689	68
Jane	Brown	102647	75
Andy	Brown	103569	42
Zöe	Adams	105987	52
Roger	Brady	106872	58
Clare	Verdon	104356	45

Note that the `&` must be between columns, so I have defined a command called `\doamp` that on first use redefines itself to do `&`. So, for each row, at the start of the key iteration, `\doamp` does nothing, and on

subsequent iterations it does `&`. This ensures that the correct number of `&`s are used. Since each cell in the tabular environment is scoped, `\gdef` is needed instead of `\def`.

In the above, I needed to know how many columns are in the database, and the order that the headings should appear. If you are unsure, you can use `\dtlforeachkey` to determine the number of columns and to display the header row. For example:

```
\begin{table}[htbp]
\caption{Student Scores}
\centering
% Work out the column alignments.
\def\colalign{}%
\dtlforeachkey(\theKey,\theCol,\theType,\theHead)\in{scores}\do
{\edef\colalign{\colalign l}}%
% Begin the tabular environment.
\edef\dobegintabular{\noexpand\begin{tabular}{\colalign}}%
\dobegintabular
% Do the header row.
\gdef\doamp{\gdef\doamp{&}}%
\dtlforeachkey(\theKey,\theCol,\theType,\theHead)\in{scores}\do
{\doamp\bfseries \theHead}%
% Iterate through the data.
\DTLforeach*{scores}{}{%
\\\gdef\doamp{\gdef\doamp{&}}%
\DTLforeachkeyinrow{\thisValue}{\doamp\thisValue}}%
\end{tabular}
\end{table}
```

Table 5.13: Student Scores (Using `\dtlforeachkey` and `\DTLforeachkeyinrow`)

FirstName	**Surname**	**StudentNo**	**Score**
John	Smith, Jr	102689	68
Jane	Brown	102647	75
Andy	Brown	103569	42
Zöe	Adams	105987	52
Roger	Brady	106872	58
Clare	Verdon	104356	45

Notes:

- In order to determine the column alignment for the tabular environment, I first define `\colalign` to nothing, and then I iterate through the keys appending `l` to `\colalign`. Since `\colalign` only contains alphabetical characters, I can just use `\edef` for this. I could modify this to check the data type and, say,

use `l` (left alignment) for columns containing strings and `c` (centred) for the other columns:

```
\dtlforeachkey(\theKey,\theCol,\theType,\theHead)\in{scores}\do
{\ifnum\theType=0\relax
   \edef\colalign{\colalign l}% column contains strings
 \else
   \edef\colalign{\colalign c}% column contains numerical values
 \fi
}%
```

- To ensure `\colalign` gets correct expanded when passed to the tabular environment I temporarily define `\dobegintabular` to the code required to start the tabular environment:

  ```
  \edef\dobegintabular{\noexpand\begin{tabular}{\colalign}}%
  ```

  This sets `\dobegintabular` to `\begin{tabular}{llll}`. After defining `\dobegintabular`, I then need to use it.

- As before, I use `\doamp` to put the ampersands between columns.

- Recall that I can set the headers using `\DTLsetheader` or using the headers key when loading the data from an external file. For example:

  ```
  \DTLsetheaders{scores}{FirstName}{First Name}
  \DTLsetheaders{scores}{Score}{Score (\%)}
  ```

Recall that `\DTLforeachkeyinrow` sets `\dtlkey` to the current key. This can be used to filter out columns. Alternatively, if you know the column index, you can test `\dtlcol` instead. The following code modifies the above example so that it filters out the column whose key is `StudentNo`:

```
\begin{table}[htbp]
\caption{Student Scores (Filtering Out a Column)}
\centering
\def\colalign{}%
\dtlforeachkey(\theKey,\theCol,\theType,\theHead)\in{scores}\do
{\DTLifeq{\theKey}{StudentNo}{}{\edef\colalign{\colalign l}}}%
\edef\dobegintabular{\noexpand\begin{tabular}{\colalign}}%
\dobegintabular
\gdef\doamp{\gdef\doamp{&}}%
\dtlforeachkey(\theKey,\theCol,\theType,\theHead)\in{scores}\do
{\DTLifeq{\theKey}{StudentNo}{}{\doamp\bfseries \theHead}}%
\DTLforeach*{scores}{}{%
\\\gdef\doamp{\gdef\doamp{&}}%
```

```
\DTLforeachkeyinrow{\thisValue}{%
  \DTLifeq{\dtlkey}{StudentNo}{}{\doamp\thisValue}}}%
\end{tabular}
\end{table}
```

The result is shown in Table 5.14.

Table 5.14: Student Scores (Filtering Out a Column)

FirstName	Surname	Score
John	Smith, Jr	68
Jane	Brown	75
Andy	Brown	42
Zöe	Adams	52
Roger	Brady	58
Clare	Verdon	45

Example 11 (Nested `\DTLforeach`)

In this example I have a CSV file called `index.csv` which contains:

```
File,Temperature,NaCl,pH
exp25a.csv,25,4.7,0.5
exp25b.csv,25,4.8,1.5
exp30a.csv,30,5.12,4.5
```

The first column of this file contains the name of another CSV file which has the results of a time to growth experiment performed at the given incubation temperature, salt concentration and pH. The file `exp25a.csv` contains the following:

```
Time,Log Count
0,3.75
23,3.9
45,4.0
```

The file `exp25b.csv` contains the following:

```
Time,Log Count
0,3.6
60,3.8
120,4.0
```

The file `exp30a.csv` contains the following:

```
Time,Log Count
0,3.73
23,3.67
60,4.9
```

Suppose I now want to iterate through `index.csv`, load the given file, and create a table for that data. I can do this using nested `\DTLforeach` as follows:

```
 % load index data file
\DTLloaddb{index}{index.csv}

 % iterate through index database
\DTLforeach{index}{\theFile=File,\theTemp=Temperature,%
\theNaCl=NaCl,\thepH=pH}{%
 % load results file into database of the same name
\DTLloaddb{\theFile}{\theFile}%
 % Create a table
\begin{table}[htbp]
\caption{Temperature = \theTemp, NaCl = \theNaCl,
pH = \thepH}\label{tab:\theFile}
\centering
\begin{tabular}{rl}
\bfseries Time & \bfseries Log Count
\DTLforeach{\theFile}{\theTime=Time,\theLogCount=Log Count}{%
\\\theTime & \theLogCount}%
\end{tabular}
\end{table}
}
```

This creates Table 5.15 to Table 5.17. (Note that each table is given a label that is based on the database name, to ensure that it is unique.)

Table 5.15: Temperature = 25, NaCl = 4.7, pH = 0.5

Time	Log Count
0	3.75
23	3.9
45	4.0

Table 5.16: Temperature = 25, NaCl = 4.8, pH = 1.5

Time	Log Count
0	3.6
60	3.8
120	4.0

Example 12 (Dynamically Allocating Field Name)

(This example was suggested by Bill Hobbs.) Suppose you have a

Table 5.17: Temperature = 30, NaCl = 5.12, pH = 4.5

Time	Log Count
0	3.73
23	3.67
60	4.9

directory containing members of multiple clubs. The CSV file (say, `clubs.csv`) may look something like:

```
First Name,Surname,Rockin,Single
John,"Smith, Jr",member,
Jane,Brown,,friend
Andy,Brown,friend,member
Z\"oe,Adams,member,member
Roger,Brady,friend,friend
Clare,Verdon,member,
```

(Blank entries indicate that the person is not a member of that club.) The data can be loaded as follows:

```
\DTLloaddb{clubs}{clubs.csv}
```

Suppose at the beginning of your document you have specified which club you are interested in (`Rockin` or `Single`) and store it in `\DIdent`:

```
\newcommand{\DIdent}{Rockin}
```

You can now display the members for this particular club as follows:

```
\begin{table}[htbp]
\caption{Club Membership}
\centering
\begin{tabular}{lll}
\bfseries First Name & \bfseries Surname & \bfseries Status
\DTLforeach*[\not\DTLiseq{\status}{}]{clubs}
{\firstname=First Name,\surname=Surname,\status=\DIdent}{%
\\\firstname & \surname & \status
}%
\end{tabular}
\end{table}
```

The result is shown in Table 5.18.

Table 5.18: Club Membership

First Name	Surname	Status
John	Smith, Jr	member
Andy	Brown	friend
Zöe	Adams	member
Roger	Brady	friend
Clare	Verdon	member

5.5 Null Values

If a database is created using `\DTLnewdb`, `\DTLnewrow` and `\DTLnewdbentry` (rather than loading it from an ASCII file), it is possible for some of the entries to have null values when a value is not assigned to a given key for a given row. (Note that a null value is not the same as an empty value. Empty values can be tested using etoolbox's `\ifdefempty` or similar.)

When you iterate through the database using `\DTLforeach` (described in section 5.4), if an entry is missing for a given row, the associated command given in the ⟨*values*⟩ argument will be set to a null value. This value depends on the data type associated with the given key.

`\DTLstringnull`

`\DTLstringnull`

This is the null value for a string.

`\DTLnumbernull`

`\DTLnumbernull`

This is the null value for a number.

`\DTLifnull`

`\DTLifnull{`⟨*cmd*⟩`}{`⟨*true part*⟩`}{`⟨*false part*⟩`}`

This checks if ⟨*cmd*⟩ is null where ⟨*cmd*⟩ is a command name, if it is, then ⟨*true part*⟩ is done, otherwise ⟨*false part*⟩ is done. This macro is illustrated in example 13 below.

Example 13 (Null Values)

Consider the following (which creates a database called `emailDB`):

```
\DTLnewdb{emailDB}
\DTLnewrow{emailDB}
```

```
\DTLnewdbentry{emailDB}{Surname}{Jones}
\DTLnewdbentry{emailDB}{FirstName}{Mary}
\DTLnewdbentry{emailDB}{Email1}{mj@my.uni.ac.uk}
\DTLnewdbentry{emailDB}{Email2}{mj@somewhere.com}
\DTLnewrow{emailDB}
\DTLnewdbentry{emailDB}{Surname}{Smith}
\DTLnewdbentry{emailDB}{FirstName}{Adam}
\DTLnewdbentry{emailDB}{Email1}{as@my.uni.ac.uk}
\DTLnewdbentry{emailDB}{RegNum}{12345}
```

In the above example, the first row of the database contains an entry with the key `Email2`, but the second row doesn't. Whereas the second row contains an entry with the key `RegNum`, but the first row doesn't.

The following code puts the information in a tabular environment:

```
\begin{tabular}{lllll}
\bfseries First Name &
\bfseries Surname &
\bfseries Email 1 &
\bfseries Email 2 &
\bfseries Reg Num%
\DTLforeach{emailDB}{\firstname=FirstName,\surname=Surname,%
\emailI=Email1,\emailII=Email2,\regnum=RegNum}{%
\\\firstname & \surname & \emailI & \emailII & \regnum}%
\end{tabular}
```

This produces the following:

First Name	**Surname**	**Email 1**	**Email 2**	**Reg Num**
Mary	Jones	mj@my.uni.ac.uk	mj@somewhere.com	0
Adam	Smith	as@my.uni.ac.uk	NULL	12345

Note that on the first row of data, the registration number appears as 0, while on the next row, the second email address appears as NULL. The datatool package has identified the key `RegNum` for this database as a numerical key, since all elements in the database with that key are numerical, whereas it has identified the key `Email2` as a string, since there is at least one element in this database with that key that is a string. Null numerical values are set to `\DTLnumbernull` (0), and null strings are set to `\DTLstringnull` (NULL).

The following code checks each value to determine whether it is null using `\DTLifnull`. If it is, the text *Missing* is inserted, otherwise the value itself is used:

```
\begin{tabular}{lllll}
\bfseries First Name &
\bfseries Surname &
\bfseries Email 1 &
```

```
\bfseries Email 2 &
\bfseries Reg Num%
\DTLforeach{emailDB}{\firstname=FirstName,\surname=Surname,%
\emailI=Email1,\emailII=Email2,\regnum=RegNum}{%
\\\DTLifnull{\firstname}{\emph{Missing}}{\firstname} &
\DTLifnull{\surname}{\emph{Missing}}{\surname} &
\DTLifnull{\emailI}{\emph{Missing}}{\emailI} &
\DTLifnull{\emailII}{\emph{Missing}}{\emailII} &
\DTLifnull{\regnum}{\emph{Missing}}{\regnum}}%
\end{tabular}
```

This produces the following:

First Name	**Surname**	**Email 1**	**Email 2**	**Reg Num**
Mary	Jones	mj@my.uni.ac.uk	mj@somewhere.com	*Missing*
Adam	Smith	as@my.uni.ac.uk	*Missing*	12345

If you want to do this, you may find it easier to define a convenience command that will display some appropriate text if an entry is missing, for example:

```
\newcommand*{\checkmissing}[1]{\DTLifnull{#1}{---}{#1}}
```

Then instead of typing, say,

```
\DTLifnull{\regnum}{---}{\regnum}
```

you can instead type:

```
\checkmissing{\regnum}
```

Now suppose that instead of defining the database using `\DTLnewdb`, `\DTLnewrow` and `\DTLnewdbentry`, you have a file with the contents:

```
Surname,FirstName,RegNum,Email1,Email2
Jones,Mary,,mj@my.uni.ac.uk,mj@somewhere.com
Smith,Adam,12345,as@my.uni.ac.uk,
```

and you load the data from this file using `\DTLloaddb` (defined in section 5.2). Now the database has no null values, but has an empty value for the key `RegNum` on the first row of the database, and an empty value for the key `Email2` on the second row of the database. Now, the following code

```
\begin{tabular}{lllll}
\bfseries First Name &
\bfseries Surname &
\bfseries Email 1 &
\bfseries Email 2 &
```

```
\bfseries Reg Number%
\DTLforeach{emailDB}{\firstname=FirstName,\surname=Surname,%
\emailI=Email1,\emailII=Email2,\regnum=RegNum}{%
\\\DTLifnull{\firstname}{\emph{Missing}}{\firstname} &
\DTLifnull{\surname}{\emph{Missing}}{\surname} &
\DTLifnull{\emailI}{\emph{Missing}}{\emailI} &
\DTLifnull{\emailII}{\emph{Missing}}{\emailII} &
\DTLifnull{\regnum}{\emph{Missing}}{\regnum}}%
\end{tabular}
```

produces:

First Name	**Surname**	**Email 1**	**Email 2**	**Reg Number**
Mary	Jones	mj@my.uni.ac.uk	mj@somewhere.com	
Adam	Smith	as@my.uni.ac.uk		12345

5.6 Editing Database Rows

A row can be removed from a data base using:

`\DTLremoverow`

`\DTLremoverow{`⟨*db name*⟩`}{`⟨*row index*⟩`}`

where ⟨*row index*⟩ is the index of the unwanted row. For example:

```
\DTLremoverow{scores}{2}
```

will delete the second row in the database labelled "scores". There is also a starred version that doesn't check for the existence of the database.

The following commands may be used in the body of the `\DTLforeach` loop,[3] to edit the current row of the loop. (See also subsection 5.11.1.)

`\DTLappendtorow`

`\DTLappendtorow{`⟨*key*⟩`}{`⟨*value*⟩`}`

This appends a new entry with the given ⟨*key*⟩ and ⟨*value*⟩ to the current row. (⟨*value*⟩ is expanded.)

`\DTLreplaceentryforrow`

`\DTLreplaceentryforrow{`⟨*key*⟩`}{`⟨*value*⟩`}`

This replaces the entry for ⟨*key*⟩ with ⟨*value*⟩. (⟨*value*⟩ is expanded.)

[3] Only the unstarred version of `\DTLforeach`; the starred version is read-only.

\DTLremoveentryfromrow

```
\DTLremoveentryfromrow{⟨key⟩}
```

This removes the entry for ⟨*key*⟩ from the current row.

\DTLremovecurrentrow

```
\DTLremovecurrentrow
```

This removes the current row from the database.

Example 14 (Editing Database Rows)

In this example I have a CSV file called `marks.csv` that contains student marks for three assignments:

```
Surname,FirstName,StudentNo,Assignment 1,Assignment
2,Assignment 3
"Smith, Jr",John,102689,68,57,72
"Brown",Jane,102647,75,84,80
"Brown",Andy,103569,42,52,54
"Adams",Zöe,105987,52,48,57
"Brady",Roger,106872,58,60,62
"Verdon",Clare,104356,45,50,48
```

First load this into a database called `marks`:

```
\DTLloaddb{marks}{marks.csv}
```

Suppose now I want to compute the average mark for each student, and append this to the database. I can do this as follows:

```
\DTLforeach{marks}{%
\assignI=Assignment 1,%
\assignII=Assignment 2,%
\assignIII=Assignment 3}{%
\DTLmeanforall{\theMean}{\assignI,\assignII,\assignIII}%
\DTLappendtorow{Average}{\theMean}}
```

For each row in the `marks` database, I now have an extra key called `Average` that contains the average mark over all three assignments for a given student. I can now put this data into a table:

```
\begin{table}[htbp]
\caption{Student marks}
\centering
\begin{tabular}{llcccc}
\bfseries Surname & \bfseries First Name &
\bfseries Assign 1 &
\bfseries Assign 2 &
\bfseries Assign 3 &
```

```
\bfseries Average Mark%
\DTLforeach{marks}{\surname=Surname,\firstname=FirstName,\average
=Average,\assignI=Assignment 1,\assignII=Assignment 2,\assignIII
=Assignment 3}{\\\surname
& \firstname & \assignI & \assignII & \assignIII &
\DTLround{\average}{\average}{2}\DTLclip{\average}\average}\relax
\end{tabular}
\end{table}
```

This produces Table 5.19.

Note that if I only wanted the averages for the table and nothing else, I could simply have computed the average in each row of the table and displayed it without adding the information to the database, however I am going to reuse this information in example 37, so adding it to the database means that I don't need to recompute the mean.

Table 5.19: Student marks

Surname	**First Name**	**Assign 1**	**Assign 2**	**Assign 3**	**Average Mark**
Smith, Jr	John	68	57	72	
Brown	Jane	75	84	80	
Brown	Andy	42	52	54	
Adams	Zöe	52	48	57	
Brady	Roger	58	60	62	
Verdon	Clare	45	50	48	

5.7 Arithmetical Computations on Database Entries

The commands used in chapter 3 can be used on database entries. You can, of course, directly use the commands provided by the fp package if you know that the values are in the correct format (i.e. no currency symbols, no number group separators and a full stop as the decimal point) but if this is not the case, then you should use the commands described in chapter 3. If you want to use a command provided by the fp package, that does not have a wrapper function in datatool, then you will need to convert the value using `\DTLconverttodecimal`, and convert it back using either `\DTLdecimaltolocale` or `\DTLdecimaltocurrency`.

Example 15 (Arithmetical Computations)

In this example, I am going to produce a table similar to Table 5.4, except that I want to add an extra row at the end which contains the

average score.

```
\begin{table}[htbp]
\caption{Student scores}\label{tab:mean}
\centering
\def\total{0}%
\begin{tabular}{llr}
\bfseries First Name &
\bfseries Surname &
\bfseries Score (\%)%
\DTLforeach{scores}{%
\firstname=FirstName,\surname=Surname,\score=Score}{%
\\
\firstname & \surname &
\DTLgadd{\total}{\score}{\total}%
\score
}\\
\multicolumn{2}{l}{\bfseries Average Score} &
\DTLsavelastrowcount{\n}%
\DTLdiv{\average}{\total}{\n}%
\DTLround{\average}{\average}{2}%
\average
\end{tabular}
\end{table}
```

This produces Table 5.20. **Notes:**

- I had to use `\DTLgadd` rather than `\DTLadd` since it occurs within a tabular environment which puts each entry in a local scope.
- I used `\DTLsavelastrowcount` to store the number of rows produced by `\DTLforeach` in the control sequence `\n`.
- I used `\DTLround` to round the average score to 2 decimal places.

Table 5.20: Student scores

First Name	**Surname**	**Score (%)**
John	Smith, Jr	68
Jane	Brown	75
Andy	Brown	42
Zöe	Adams	52
Roger	Brady	58
Clare	Verdon	45
Average Score		56.67

\DTLsumforkeys

\DTLsumforkeys[⟨*condition*⟩][⟨*assign list*⟩]{⟨*db list*⟩}{⟨*key list*⟩}{⟨*cmd*⟩}

This command sums all the entries over all the databases listed in the comma separated list of database names ⟨*db list*⟩ for each key in ⟨*key list*⟩ where the condition given by ⟨*condition*⟩ is true. The second optional argument ⟨*assign list*⟩ is the same as the assignment list used by `\DTLforeach`, so that you can use the information in ⟨*condition*⟩. The result is stored in ⟨*cmd*⟩ which must be a control sequence. For example:

```
\DTLsumforkeys{scores}{Score}{\total}
```

sets `\total` to the sum of all the scores in the database called `scores`.

\DTLsumcolumn

\DTLsumcolumn{⟨*db*⟩}{⟨*key*⟩}{⟨*cmd*⟩}

This is a faster version of `\DTLsumforkeys` that only sums the entries in a single column (specified by ⟨*key*⟩) for a single database (specified by ⟨*db*⟩) and doesn't provide any filtering. The result is stored in ⟨*cmd*⟩ which must be a control sequence.

\DTLmeanforkeys

\DTLmeanforkeys[⟨*condition*⟩][⟨*assign list*⟩]{⟨*db list*⟩}{⟨*key list*⟩}{⟨*cmd*⟩}

This command computes the arithmetic mean of all the entries over all the databases listed in ⟨*db list*⟩ for all keys in ⟨*key list*⟩ where the condition given by ⟨*condition*⟩ is true. The second optional argument ⟨*assign list*⟩ is the same as the assignment list used by `\DTLforeach`, so that you can use the information in ⟨*condition*⟩. The result is stored in ⟨*cmd*⟩ which must be a control sequence. For example:

```
\DTLmeanforkeys{scores}{Score}{\average}
```

sets `\average` to the mean of all the scores in the database called `scores`.

\DTLmeanforcolumn

\DTLmeanforcolumn{⟨*db*⟩}{⟨*key*⟩}{⟨*cmd*⟩}

This is a faster version of `\DTLmeanforkeys` that only computes the mean for a single column (specified by ⟨*key*⟩) for a single database (specified by ⟨*db*⟩) and doesn't provide any filtering. The result is stored in ⟨*cmd*⟩ which must be a control sequence.

\DTLvarianceforkeys

\DTLvarianceforkeys[⟨*condition*⟩][⟨*assign list*⟩]{⟨*db list*⟩}{⟨*key list*⟩}{⟨*cmd*⟩}

This command computes the variance of all the entries over all the databases listed in ⟨*db list*⟩ for all keys in ⟨*key list*⟩ where the condition given by ⟨*condition*⟩ is true. The second optional argument ⟨*assign list*⟩ is the same as the assignment list used by \DTLforeach, so that you can use the information in ⟨*condition*⟩. The result is stored in ⟨*cmd*⟩ which must be a control sequence.

\DTLvarianceforcolumn

\DTLvarianceforcolumn{⟨*db*⟩}{⟨*key*⟩}{⟨*cmd*⟩}

This is a faster version of \DTLvarianceforkeys that only computes the variance for a single column (specified by ⟨*key*⟩) for a single database (specified by ⟨*db*⟩) and doesn't provide any filtering. The result is stored in ⟨*cmd*⟩ which must be a control sequence.

\DTLsdforkeys

\DTLsdforkeys[⟨*condition*⟩][⟨*assign list*⟩]{⟨*db list*⟩}{⟨*key list*⟩}{⟨*cmd*⟩}

This command computes the standard deviation of all the entries over all the databases listed in ⟨*db list*⟩ for all keys in ⟨*key list*⟩ where the condition given by ⟨*condition*⟩ is true. The second optional argument ⟨*assign list*⟩ is the same as the assignment list used by \DTLforeach, so that you can use the information in ⟨*condition*⟩. The result is stored in ⟨*cmd*⟩ which must be a control sequence.

\DTLsdforcolumn

\DTLsdforcolumn{⟨*db*⟩}{⟨*key*⟩}{⟨*cmd*⟩}

This is a faster version of \DTLsdforkeys that only computes the standard deviation for a single column (specified by ⟨*key*⟩) for a single database (specified by ⟨*db*⟩) and doesn't provide any filtering. The result is stored in ⟨*cmd*⟩ which must be a control sequence.

\DTLminforkeys

\DTLminforkeys[⟨*condition*⟩][⟨*assign list*⟩]{⟨*db list*⟩}{⟨*key list*⟩}{⟨*cmd*⟩}

This command determines the minimum value over all entries for all keys in ⟨*key list*⟩ over all the databases listed in ⟨*db list*⟩ where ⟨*condition*⟩ is true. The second optional argument ⟨*assign list*⟩ is the same as the assignment list used by `\DTLforeach`, so that you can use the information in ⟨*condition*⟩. The result is stored in ⟨*cmd*⟩, which must be a control sequence. For example

```
\DTLminforkeys{scores}{Score}{\theMin}
```

sets `\theMin` to the minimum score in the database.

`\DTLminforcolumn`

`\DTLminforcolumn{`⟨*db*⟩`}{`⟨*key*⟩`}{`⟨*cmd*⟩`}`

This is a faster version of `\DTLminforkeys` that only computes the minimum for a single column (specified by ⟨*key*⟩) for a single database (specified by ⟨*db*⟩) and doesn't provide any filtering. The result is stored in ⟨*cmd*⟩ which must be a control sequence.

`\DTLmaxforkeys`

`\DTLmaxforkeys[`⟨*condition*⟩`][`⟨*assign list*⟩`]{`⟨*db list*⟩`}{`⟨*key list*⟩`}{`⟨*cmd*⟩`}`

This command determines the maximum value over all entries for all keys in ⟨*key list*⟩ over all the databases listed in ⟨*db list*⟩ where ⟨*condition*⟩ is true. The second optional argument ⟨*assign list*⟩ is the same as the assignment list used by `\DTLforeach`, so that you can use the information in ⟨*condition*⟩. The result is stored in ⟨*cmd*⟩, which must be a control sequence. For example

```
\DTLminforkeys{scores}{Score}{\theMax}
```

sets `\theMax` to the minimum score in the database.

`\DTLmaxforcolumn`

`\DTLmaxforcolumn{`⟨*db*⟩`}{`⟨*key*⟩`}{`⟨*cmd*⟩`}`

This is a faster version of `\DTLmaxforkeys` that only computes the maximum for a single column (specified by ⟨*key*⟩) for a single database (specified by ⟨*db*⟩) and doesn't provide any filtering. The result is stored in ⟨*cmd*⟩ which must be a control sequence.

`\DTLcomputebounds`

`\DTLcomputebounds{`⟨*db list*⟩`}{`⟨*x key*⟩`}{`⟨*y key*⟩`}{`⟨*minX cmd*⟩`}{`⟨*minY cmd*⟩`}{`⟨*maxX cmd*⟩`}{`⟨*maxY cmd*⟩`}`

Computes the maximum and minimum x and y values over all the databases listed in ⟨*db list*⟩ where the x value is given by ⟨*x key*⟩ and the y value is given by ⟨*y key*⟩. The results are stored in ⟨*minX cmd*⟩, ⟨*minY cmd*⟩, ⟨*maxX cmd*⟩ and ⟨*maxY cmd*⟩.

Example 16 (Mail Merging)

This example uses the database given in example 3 and uses `\DTLmeanforkeys` to determine the average score. A letter is then created for each student to inform them of their score and the class average.

```
\documentclass{letter}

\usepackage{datatool}

\begin{document}
 % load database
\DTLloaddb{scores}{studentscores.csv}
 % compute arithmetic mean for key 'Score'
\DTLmeanforkeys{scores}{Score}{\average}
 % Round the average to 2 decimal places
\DTLround{\average}{\average}{2}
 % Save the highest score in \maxscore
\DTLmaxforkeys{scores}{Score}{\maxscore}

\DTLforeach{scores}{\firstname=FirstName,\surname=Surname,%
\score=Score}{%
\begin{letter}{}
\opening{Dear \firstname\ \surname}

\DTLifnumgt{\score}{60}{Congratulations you}{You} achieved a score
of \score\% which was \DTLifnumgt{\score}{\average}{above}{below}
the average of \average\%. \DTLifnumeq{\score}{\maxscore}{You
achieved the highest score}{The top score was \maxscore}.

\closing{Yours Sincerely}
\end{letter}
}
\end{document}
```

To determine a person's gender when mail merging, see chapter 11.

5.8 Sorting a Database

`\dtlsort`

```
\dtlsort[⟨replacement key list⟩]{⟨sort criteria⟩}{⟨db name⟩}{⟨handler⟩}
```

This will sort the database called ⟨*db name*⟩ according to the criteria given by ⟨*sort criteria*⟩, which must be a comma separated list of keys and optionally =⟨*order*⟩, where ⟨*order*⟩ is either `ascending` or `descending`. If the order is omitted, `ascending` is assumed. The database keeps track of the data type for a given key, and uses this to determine whether an alphabetical or numerical sort is required.

The final argument ⟨*handler*⟩ is the command used for the comparisons. These handlers are described in more detail on page 45 of the documented code (datatool-code.pdf). The following handlers are provided:

`\dtlcompare` A case-sensitive comparison.

`\dtlicompare` A case-insensitive comparison.

`\dtlwordindexcompare` English word-ordering comparison for indexes, as described by the Oxford Style Manual.

`\dtlletterindexcompare` English letter-ordering comparison for indexes.

The last two handlers, `\dtlwordindexcompare` and `\dtlletterindexcompare`, assume that inversion commas are indicated using one of the following commands:

- To indicate name inversion:

`\datatoolpersoncomma`

```
\datatoolpersoncomma
```

Example: `Knuth\datatoolpersoncomma Donald E.`

- To indicate place inversion:

`\datatoolplacecomma`

```
\datatoolplacecomma
```

Example: `New York\datatoolplacecomma USA`

- To indicate subject inversion:

`\datatoolsubjectcomma`

```
\datatoolsubjectcomma
```

Example: `New York\datatoolsubjectcomma population`

In addition, the start of parenthetical material should be indicated with

\datatoolparenstart

```
\datatoolparenstart
```

Example: `High Water\datatoolparenstart play`

Following the guidelines of the Oxford Style Manual, when sorting terms that have identical pre-inversion parts, the following ordering is applied: people, places, subjects, no inversions and parenthetical.

Example 17 (Sorting a Database—Dealing with Inversions)

This uses the example given in Chapter 16 of the Oxford Style Manual. Suppose I define my database as follows:

```
\DTLnewdb{inversiondata}
\DTLnewrow{inversiondata}
\DTLnewdbentry{inversiondata}{Term}{New York, New York}
\DTLnewrow{inversiondata}
\DTLnewdbentry{inversiondata}{Term}{New York\datatoolsubjectcomma
population}
\DTLnewrow{inversiondata}
\DTLnewdbentry{inversiondata}{Term}{New York\datatoolplacecomma
USA}
\DTLnewrow{inversiondata}
\DTLnewdbentry{inversiondata}{Term}{New York\datatoolpersoncomma
Earl of}
```

First of all, display the unsorted data:

```
\DTLdisplaydb{inversiondata}
```

This produces:

Term
New York, New York
New York, population
New York, USA
New York, Earl of

Now sort the data using the `\dtlwordindexcompare` handler:

```
\dtlsort{Term}{inversiondata}{\dtlwordindexcompare}
```

and display again:

```
\DTLdisplaydb{inversiondata}
```

which now produces:

Term
New York, Earl of
New York, USA
New York, population
New York, New York

here are three entries with pre-inversion text as simply `New York`. Since each of these three entries has the same pre-inversion text, they need to be sorted according to the type of inversion: person, place, subject. The fourth entry (New York, New York) doesn't have an inversion since the comma is part of the title of the named work. It's therefore sorted according to `New York, New York` rather than just `New York` and so comes after all the `New York` entries.

If you want to write your own comparison handler, see the documented code for details on the syntax of the handler. (You may want to consider uploading your handler as a separate package to CTAN if you think it will be of general use.)

There are two shortcut commands:

`\DTLsort`

`\DTLsort[`⟨*replacement key list*⟩`]{`⟨*sort criteria*⟩`}{`⟨*db name*⟩`}`

`\DTLsort*`

`\DTLsort*[`⟨*replacement key list*⟩`]{`⟨*sort criteria*⟩`}{`⟨*db name*⟩`}`

these use the `\dtlcompare` and `\dtlicompare` handlers, respectively.

The optional argument ⟨*replacement key list*⟩ is a list of keys to use if the current key given in ⟨*sort criteria*⟩ is null for a given entry. Null keys are unlikely to occur if you have loaded the database from an external ASCII file, but may occur if the database is created using `\DTLnewdb`, `\DTLnewrow` and `\DTLnewdbentry`. For example:

```
\DTLsort[Editor,Organization]{Author}{mydata}
```

will sort according to the `Author` key, but if that key is missing for a given row of the database, the `Editor` key will be used, and if the `Editor` key is missing, it will use the `Organization` key. Note that this is not the same as:

```
\DTLsort{Author,Editor,Organization}{mydata}
```

which will first compare the `Author` keys, but if the author names are the same, it will then compare the `Editor` keys, and if the editor names are also the same, it will then compare the `Organization` keys.

The unstarred version uses a case sensitive comparison for strings, whereas the starred version ignores the case when comparing strings. Note that the case sensitive comparison orders uppercase characters before lowercase characters, so the letter B is considered to be lower than the letter a.

Example 18 (Sorting a Database)

This example uses the database called `scores` defined in example 3. First, I am going to sort the database according to the student scores in descending order (highest to lowest) and display the database in a table

```
\begin{table}[htbp]
\caption{Student scores (sorted by score)}
\centering
\DTLsort{Score=descending}{scores}%
\begin{tabular}{llr}
\bfseries First Name &
\bfseries Surname &
\bfseries Score (\%)%
\DTLforeach{scores}{%
\firstname=FirstName,\surname=Surname,\score=Score}{%
\\
\firstname & \surname & \score}
\end{tabular}
\end{table}
```

This produces Table 5.21.

Table 5.21: Student scores (sorted by score)

First Name	Surname	Score (%)
Jane	Brown	75
John	Smith, Jr	68
Roger	Brady	58
Zöe	Adams	52
Clare	Verdon	45
Andy	Brown	42

Now I am going to sort the database according to surname and then first name, and display it in a table. Note that since I want to sort in ascending order, I can omit the `=ascending` part of the sort criteria. I have also decided to reverse the first and second columns, so that the surname is in the first column.

```
\begin{table}[htbp]
\caption{Student scores (sorted by name)}
\centering
\DTLsort{Surname,FirstName}{scores}%
\begin{tabular}{llr}
\bfseries Surname &
\bfseries First Name &
\bfseries Score (\%)%
\DTLforeach{scores}{%
\firstname=FirstName,\surname=Surname,\score=Score}{%
\\
\surname & \firstname & \score}
\end{tabular}
\end{table}
```

This produces Table 5.22.

Table 5.22: Student scores (sorted by name)

Surname	**First Name**	**Score (%)**
Adams	Zöe	52
Brady	Roger	58
Brown	Andy	42
Brown	Jane	75
Smith, Jr	John	68
Verdon	Clare	45

Now suppose I add two new students to the database:

```
\DTLnewrow{scores}%
\DTLnewdbentry{scores}{Surname}{van der Mere}%
\DTLnewdbentry{scores}{FirstName}{Henk}%
\DTLnewdbentry{scores}{Score}{71}%
\DTLnewrow{scores}%
\DTLnewdbentry{scores}{Surname}{de la Mere}%
\DTLnewdbentry{scores}{FirstName}{Jos}%
\DTLnewdbentry{scores}{Score}{58}%
```

and again I try sorting the database, and displaying the contents as a table:

```
\begin{table}[htbp]
\caption{Student scores (case sensitive sort)}
\centering
\DTLsort{Surname,FirstName}{scores}%
\begin{tabular}{llr}
\bfseries Surname &
\bfseries First Name &
\bfseries Score (\%)%
```

```
\DTLforeach{scores}{%
\firstname=FirstName,\surname=Surname,\score=Score}{%
\\
\surname & \firstname & \score}
\end{tabular}
\end{table}
```

This produces Table 5.23. Notice that the surnames aren't correctly ordered. This is because a case-sensitive sort was used. Changing `\DTLsort` to `\DTLsort*` in the above code produces Table 5.24.

Table 5.23: Student scores (case sensitive sort)

Surname	**First Name**	**Score (%)**
Adams	Zöe	52
Brady	Roger	58
Brown	Andy	42
Brown	Jane	75
de la Mere	Jos	58
Smith, Jr	John	68
van der Mere	Henk	71
Verdon	Clare	45

Table 5.24: Student scores (case ignored when sorting)

Surname	**First Name**	**Score (%)**
Adams	Zöe	52
Brady	Roger	58
Brown	Andy	42
Brown	Jane	75
de la Mere	Jos	58
Smith, Jr	John	68
van der Mere	Henk	71
Verdon	Clare	45

Example 19 (Influencing the sort order)

Consider the data displayed in Table 5.24, suppose that you want the names "van der Mere" and "de la Mere" sorted according to the actual surname "Mere" rather than by the "von part". There are two ways you can do this: firstly, you could store the von part in a separate field, and

then sort by surname, then von part, then first name, or you could define a command called, say, `\switchargs`, as follows:

```
\newcommand*{\switchargs}[2]{#2#1}
```

then store the data as:

```
FirstName,Surname,StudentNo,Score
John,"Smith, Jr",102689,68
Jane,Brown,102647,75
Andy,Brown,103569,42
Z\"oe,Adams,105987,52
Roger,Brady,106872,58
Clare,Verdon,104356,45
Henk,\switchargs{Mere}{van der },106789,71
Jos,\switchargs{Mere}{de la },104256,58
```

Now sort the data, and put it in table (this is the same code as in the previous example:

```
\begin{table}[htbp]
\caption{Student scores (influencing the sort order)}
\centering
\DTLsort*{Surname,FirstName}{scores}%
\begin{tabular}{llr}
\bfseries Surname &
\bfseries First Name &
\bfseries Score (\%)%
\DTLforeach{scores}{%
\firstname=FirstName,\surname=Surname,\score=Score}{%
\\
\surname & \firstname & \score}
\end{tabular}
\end{table}
```

This produces Table 5.25.

Table 5.25: Student scores (influencing the sort order)

Surname	**First Name**	**Score (%)**
de la Mere	Jos	58
van der Mere	Henk	71
Adams	Zöe	52
Brady	Roger	58
Brown	Andy	42
Brown	Jane	75
Smith, Jr	John	68
Verdon	Clare	45

5.9 Saving a Database to an External File

TEX's write mechanism automatically inserts linebreaks every 80 characters. This may cause problems if you have long entries in your database.

\DTLsavedb

\DTLsavedb{⟨*db name*⟩}{⟨*filename*⟩}

This writes the database called ⟨*db name*⟩ to a file called ⟨*filename*⟩. The separator and delimiter characters used are as given by \DTLsetseparator (or \DTLsettabseparator) and \DTLsetdelimiter. For example:

```
\DTLsettabdelimiter
\DTLsavedb{scores}{scores.txt}
```

will create a file called scores.txt and will save the data in a tab separated format. (The delimiters will only be used if a given entry contains the separator character.)

\DTLsavetexdb

\DTLsavetexdb{⟨*db name*⟩}{⟨*filename*⟩}

This writes the database called ⟨*db name*⟩ to a LATEX file called ⟨*filename*⟩, where the database is stored as a combination of \DTLnewdb, \DTLnewrow and \DTLnewdbentry commands. This means that the file is in a user-friendly format, but may be so to load, particularly if the database is large. If you are more concerned with speed rather than readability you can use:

\DTLsaverawdb

\DTLsaverawdb{⟨*db name*⟩}{⟨*filename*⟩}

This saves the database to ⟨*filename*⟩ in its internal representation, which makes it faster to load. Fragile commands cause a problem for \DTLsaverawdb so if your database contains any use:

\DTLprotectedsaverawdb

\DTLprotectedsaverawdb{⟨*db name*⟩}{⟨*filename*⟩}

instead.

If you find a problem caused by TEX's automatic insertion of a line break every 80 characters when writing to a file, try loading the morewrites package before datatool.

Databases saved using `\DTLsavetexdb`, `\DTLsaverawdb` and `\DTLprotectedsaverawdb` can be loaded using LATEX's standard `\input` command. As from version 2.15, the last line of the database file defines

`\dtllastloadeddb`

```
\dtllastloadeddb
```

to the name of the database, in case it's required.

Databases saved using `\DTLsaverawdb` and `\DTLprotectedsaverawdb` can also be loaded and edited by `datatooltk` (see page 36).

5.10 Deleting or Clearing a Database

A database can be cleared or deleted when its contents are no longer required.

`\DTLcleardb`

```
\DTLcleardb{⟨db name⟩}
```

`\DTLgcleardb`

```
\DTLgcleardb{⟨db name⟩}
```

Clears the database given by ⟨*db name*⟩. The database is emptied but remains defined. The second form is required if you want a global effect.

`\DTLdeletedb`

```
\DTLdeletedb{⟨db name⟩}
```

`\DTLgdeletedb`

```
\DTLgdeletedb{⟨db name⟩}
```

Deletes (undefines) the database given by ⟨*db name*⟩. The second form is required if you want a global effect.

Although \DTLdeletedb and \DTLgdeletedb undefine the macros associated with the database, they don't unassign the registers used. (TEX doesn't provide a command that performs the reverse of commands such as \newcount.) If you want to keep making temporary databases, it's better to just define a single database (called, say, temp) and then just clear it rather than delete it and define a new database. For example, if you are iterating through a loop and want to have a temporary database on each iteration. In that case, define the database before the start of the loop and clear it on each iteration. If you are loading data from an external file, remember to use \DTLnewdbonloadfalse before \DTLloaddb (or \DTLloadrawdb).

5.11 Advanced Database Commands

This section describes more advanced commands. Further details can be found in the documented code (datatool-code.pdf).

\DTLgetdatatype

\DTLgetdatatype{⟨*cs*⟩}{⟨*db*⟩}{⟨*key*⟩}

Gets the data type for the given key ⟨*key*⟩ for the database given by ⟨*db*⟩. The data type is stored in ⟨*cs*⟩ which must be a command name. The type will be one of:

\DTLunsettype
- \DTLunsettype (not set),

\DTLstringtype
- \DTLstringtype (string),

\DTLinttype
- \DTLinttype (integer),

\DTLrealtype
- \DTLrealtype (real number) or

\DTLcurrencytype
- \DTLcurrenttype (currency).

\dtlforeachkey

\dtlforeachkey(⟨*key cs*⟩,⟨*col cs*⟩,⟨*type cs*⟩,⟨*header cs*⟩)\in{⟨*db*⟩}\do{⟨*body*⟩}

This iterates through all the keys in the database given by ⟨*db*⟩. In each iteration, ⟨*key cs*⟩ is set to the key, ⟨*col cs*⟩ is set to the column index, ⟨*type cs*⟩ is set to the data type (as for \DTLgetdatatype), ⟨*header cs*⟩ is set to the header for that column, and then ⟨*body*⟩ is done. Note that ⟨*key cs*⟩, ⟨*col cs*⟩, ⟨*type cs*⟩ and ⟨*header cs*⟩ must all be control sequences. No check

is performed to determine if that control sequence already exists, and the control sequences are defined globally (since it's likely that `\dtlforeachkey` may be used within a tabular environment), so you need to make sure you don't override an existing command of the same name.

`\dtlforcolumn`

`\dtlforcolumn{⟨cs⟩}{⟨db⟩}{⟨key⟩}{⟨body⟩}`

This iterates through the column given by ⟨*key*⟩ in the database given by ⟨*db*⟩ and applies ⟨*body*⟩. In each iteration, ⟨*cs*⟩ (which must be a control sequence) is set to the current element in the column and may be used in ⟨*body*⟩. Alternatively, if you want to identify the column by its index rather than its key, use:

`\dtlforcolumnidx`

`\dtlforcolumnidx{⟨cs⟩}{⟨db⟩}{⟨col index⟩}{⟨body⟩}`

`\DTLifdbexists`

`\DTLifdbexists{⟨db name⟩}{⟨true part⟩}{⟨false part⟩}`

Determines if the database given by ⟨*db name*⟩ exists.

`\DTLifhaskey`

`\DTLifhaskey{⟨db name⟩}{⟨key⟩}{⟨true part⟩}{⟨false part⟩}`

This determines if the database given by ⟨*db name*⟩ has any entries with the key given by ⟨*key*⟩. If so, it does ⟨*true part*⟩ otherwise it does ⟨*false part*⟩.

Each key has an associated column index. This can be obtained using:

`\DTLgetcolumnindex`

`\DTLgetcolumnindex{⟨cs⟩}{⟨db⟩}{⟨key⟩}`

where ⟨*cs*⟩ is a command name, ⟨*db*⟩ is the database label and ⟨*key*⟩ is the key. The column index is stored in ⟨*cs*⟩.

You can also do the reverse and find the key associated with a given column index:

`\DTLgetkeyforcolumn`

`\DTLgetkeyforcolumn{⟨key cs⟩}{⟨db⟩}{⟨column index⟩}`

The key is stored in ⟨*key cs*⟩ (which must be a command name).

There is also a full expandable way of obtaining the column index, but note that no check is performed to determine if the database exists, or if it contains the given key:

\dtlcolumnindex

\dtlcolumnindex{⟨*db name*⟩}{⟨*key*⟩}

\DTLgetkeydata

\DTLgetkeydata{⟨*key*⟩}{⟨*db*⟩}{⟨*col cs*⟩}{⟨*type cs*⟩}{⟨*header cs*⟩}

Gets data for given key in database ⟨*db*⟩: the column index is stored in ⟨*col cs*⟩ (as \DTLgetcolumnindex), the type is stored in ⟨*type cs*⟩ (as \DTLgetdatatype) and the header is stored in ⟨*header cs*⟩.

\DTLgetvalue

\DTLgetvalue{⟨*cs*⟩}{⟨*db*⟩}{⟨*r*⟩}{⟨*c*⟩}

This gets the value for row given by index ⟨*r*⟩ and column given by ⟨*c*⟩ for the database ⟨*db*⟩ and stores it in ⟨*cs*⟩ which must be a command name. If you want to get the value by key rather than column index you can use \dtlcolumnindex. For example, the following gets the value for row 3 with key Surname from the database data and stores in \myval:

```
\DTLgetvalue{\myval}{data}{3}{\dtlcolumnindex{data}{Surname}}
```

\DTLgetlocation

\DTLgetlocation{⟨*row cs*⟩}{⟨*column cs*⟩}{⟨*database*⟩}{⟨*value*⟩}

Assigns ⟨*row cs*⟩ and ⟨*column cs*⟩ to the indices of the first entry in ⟨*database*⟩ that matches ⟨*value*⟩.

\DTLgetvalueforkey

\DTLgetvalueforkey{⟨*cmd*⟩}{⟨*key*⟩}{⟨*db name*⟩}{⟨*ref key*⟩}{⟨*ref value*⟩}

This (globally) sets ⟨*cmd*⟩ (a control sequence) to the value of the key specified by ⟨*key*⟩ in the first row of the database called ⟨*db name*⟩ which contains the key ⟨*ref key*⟩ which has the value ⟨*value*⟩.

\DTLfetch

\DTLfetch{⟨*db name*⟩}{⟨*column1 name*⟩}{⟨*column1 value*⟩}{⟨*column2 name*⟩}

This fetches and displays the value for ⟨*column2 name*⟩ in the first row where the value of ⟨*column1 name*⟩ is ⟨*column1 value*⟩. (Note that all arguments are expanded.) So, for example, if you have a column labelled "regnum" and a column labelled "tutor", then to fetch and display the value of the tutor in the row where "regnum" is "12345" from the database called "students" you can do:

```
\DTLfetch{students}{regnum}{12345}{tutor}
```

See example 21 on page 94.

`\DTLassign`

`\DTLassign{`⟨*db name*⟩`}{`⟨*row idx*⟩`}{`⟨*assign list*⟩`}`

This (globally) assigns the list of commands in ⟨*assign list*⟩ for row ⟨*row idx*⟩ in database ⟨*db name*⟩, where ⟨*assign list*⟩ has the same format as in `\DTLforeach`.

Two rows can be swapped using:

`\DTLswaprows`

`\DTLswaprows{`⟨*db name*⟩`}{`⟨*row1 index*⟩`}{`⟨*row2 index*⟩`}`

where ⟨*row1 index*⟩ and ⟨*row2 index*⟩ are the indices of the rows to be swapped. For example:

```
\DTLswaprows{scores}{3}{5}
```

will swap the third and fifth rows.

`\DTLifinlist`

`\DTLifinlist{`⟨*element*⟩`}{`⟨*list*⟩`}{`⟨*true part*⟩`}{`⟨*false part*⟩`}`

If ⟨*element*⟩ is contained in the comma-separated list given by ⟨*list*⟩, then do ⟨*true part*⟩ otherwise do false part. (Does a one level expansion on ⟨*list*⟩, but no expansion on ⟨*element*⟩.)

`\DTLnumitemsinlist`

`\DTLnumitemsinlist{`⟨*list*⟩`}{`⟨*cmd*⟩`}`

Counts the number of non-empty elements in ⟨*list*⟩ and stores result in ⟨*cmd*⟩, which must be a control sequence.

Example 20 (Two Database Rows Per Tabular Row (Column-Wise))

This example adapts example 9 so that the list is ordered vertically rather than horizontally.

```
\begin{table}[htbp]
 \caption{Two database rows per tabular row (column-wise)}
 \centering
 % store half number of rows
 \edef\maxrows{\DTLrowcount{scores}}%
 \DTLdiv{\halfrowidx}{\maxrows}{2}
 \begin{tabular}{llcllc}
 \bfseries First Name &
 \bfseries Surname &
 \bfseries Score (\%) &
 \bfseries First Name &
 \bfseries Surname &
 \bfseries Score (\%)%
 \DTLforeach*[\value{DTLrowi}<\halfrowidx]{scores}%
 {\firstname=FirstName,\surname=Surname,\score=Score}%
 {%
   \\%
   \firstname & \surname & \score
   &
   \edef\currentrowidx{\arabic{DTLrowi}}%
   \DTLadd{\rowidxII}{\halfrowidx}{\currentrowidx}%
   \DTLassign{scores}{\rowidxII}%
     {\firstnameII=FirstName,\surnameII=Surname,\scoreII=Score}%
   \firstnameII & \surnameII & \scoreII
 }%
 \end{tabular}
\end{table}
```

This produces Table 5.26.

Table 5.26: Two database rows per tabular row (column-wise)

First Name	Surname	Score (%)	First Name	Surname	Score (%)
Zöe	Adams	52	Jos	de la Mere	58
Roger	Brady	58	John	Smith, Jr	68
Andy	Brown	42	Henk	van der Mere	71
Jane	Brown	75	Clare	Verdon	45

5.11.1 Operating on Current Row

If you want to select from or edit a particular row in a database without having to iterate through the database using `\DTLforeach`, you can use the commands described in this section.

`\DTLgetrowindex`

\DTLgetrowindex{⟨*row cs*⟩}{⟨*db name*⟩}{⟨*col idx*⟩}{⟨*value*⟩}

Gets the row index of the first row in database ⟨*db name*⟩ where the value for column ⟨*col idx*⟩ matches ⟨*value*⟩ and stores the result in ⟨*row cs*⟩, which must be a control sequence. An error message is given if not found.

\dtlgetrowindex

\dtlgetrowindex{⟨*row cs*⟩}{⟨*db name*⟩}{⟨*col idx*⟩}{⟨*value*⟩}

Similar to \DTLgetrowindex but doesn't produce an error if no match is found. You can test the result by using \ifx⟨*row cs*⟩\dtlnovalue. For example:

```
\dtlgetrowindex{\myrowidx}{data}{\dtlcolumnindex{data}{Surname}}{Smith}
\ifx\myrowidx\dtlnovalue
  Not Found
\else
  Found in row \myrowidx.
\fi
```

\dtlgetrow

\dtlgetrow{⟨*db name*⟩}{⟨*row idx*⟩}

Gets the row with index ⟨*row idx*⟩ from the database ⟨*db name*⟩. The required row is stored in the token register

\dtlcurrentrow

\dtlcurrentrow

the preceding rows are stored in the token register

\dtlbeforerow

\dtlbeforerow

the following rows are stored in the token register

\dtlafterrow

\dtlafterrow

the row index, ⟨*row idx*⟩, is stored in the register

\dtlrownum

\dtlrownum

and the database name is stored in the control sequence

\dtldbname

```
\dtldbname
```

No check is made in \dtlgetrow to see if the database exists or if the row index is valid. You will probably get a "Missing { inserted" error if you misspell the database name and a "Runaway argument" error if you specify a row index that is out of range.

\dtlgetrowforvalue

```
\dtlgetrowforvalue{⟨db name⟩}{⟨column index⟩}{⟨value⟩}
```

Like \dtlgetrow, but this gets the row where the entry in column ⟨*column index*⟩ matches ⟨*value*⟩. This command produces an error if no match is found. **Note that no expansion is performed when matching ⟨*value*⟩.** If you want ⟨*value*⟩ expanded before comparison, use:

\edtlgetrowforvalue

```
\edtlgetrowforvalue{⟨db name⟩}{⟨column index⟩}{⟨value⟩}
```

You can use the commands below to access or edit \dtlcurrentrow, but they won't change the database. Instead, once you've finished editing \dtlcurrentrow, you need to reconstruct the database token by recombining \dtlbeforerow, \dtlcurrentrow and \dtlafterrow using:

\dtlrecombine

```
\dtlrecombine
```

Alternatively, to recombine omitting the current row:

\dtlrecombineomitcurrent

```
\dtlrecombineomitcurrent
```

(This removes the current row from the database, shifting the row indices in \dtlafterrow.) Note that these recombining commands assume that you haven't altered \dtlrownum, \dtldbname, \dtlbeforerow and \dtlafterrow.

\dtlcurrentrow stores the row information using datatool's internal row syntax, described in the documented code (datatool-code.pdf). Don't explicitly modify \dtlcurrentrow unless you have a good understanding of the syntax.

\dtlgetentryfromcurrentrow

\dtlgetentryfromcurrentrow{⟨cs⟩}{⟨col idx⟩}

Gets the value from \dtlcurrentrow for the column given by ⟨*col idx*⟩ (an integer) and stores in ⟨*cs*⟩, which must be a control sequence.

\dtlreplaceentryincurrentrow

\dtlreplaceentryincurrentrow{⟨*new value*⟩}{⟨*col idx*⟩}

Replaces the value in \dtlcurrentrow for the column given by ⟨*col idx*⟩ (an integer) with ⟨*new value*⟩.

The new value doesn't get expanded.

\dtlremoveentryincurrentrow

\dtlremoveentryincurrentrow{⟨*col idx*⟩}

Removes the value in \dtlcurrentrow for the column given by ⟨*col idx*⟩.

\dtlswapentriesincurrentrow

\dtlswapentriesincurrentrow{⟨*col1 idx*⟩}{⟨*col2 idx*⟩}

Swaps entries in columns ⟨*col1 idx*⟩ and ⟨*col2 idx*⟩ in \dtlcurrentrow (where ⟨*col1 idx*⟩ and ⟨*col2 idx*⟩ are the column indices).

\dtlappendentrytocurrentrow

\dtlappendentrytocurrentrow{⟨*key*⟩}{⟨*value*⟩}

Appends ⟨*value*⟩ to the current row for column given by ⟨*key*⟩. (Produces an error if there is already an entry for that column in the current row.)

\dtlupdateentryincurrentrow

\dtlupdateentryincurrentrow{⟨*key*⟩}{⟨*value*⟩}

Behaves like \dtlappendentrytocurrentrow if the current row doesn't contain an entry for the column given by ⟨*key*⟩, otherwise

behaves like `\dtlreplaceentryincurrentrow`.

Example 21 (Joining Two Databases in a Single Table)

Suppose a lecturer has a CSV file for a particular course that contains student registration numbers and marks for the Autumn and Spring semesters. The file is called, say, `cmp101.csv` and contains the following:

```
regnum,Autumn Marks,Spring Marks
12345,80,85
12346,70,90
12347,75,60
```

This only contains the student registration numbers, not their names, but suppose there's another CSV file that contains the registration numbers and names for all students at the department (or university). This file called, say, `students.csv` may look something like:

```
regnum,name
12344,Mary Brown
12345,Joe Bloggs
12346,Jane Doe
12347,John Smith
12348,Alice Jones
```

Now suppose the lecturer wants a table of all the students on course CMP101 listing each student's name and marks. Here's the code:

```
\DTLloaddb{cmp101}{cmp101.csv}% load course data
\DTLloaddb{students}{students.csv}% load student data

\begin{table}[htbp]
 \caption{Student Marks (Joining Databases)}
 \centering
 \begin{tabular}{lrr}
 \bfseries Name & \bfseries Autumn Marks & \bfseries Spring Marks%
 \DTLforeach*{cmp101}%
 {\RegNum=regnum,\Autumn=Autumn Marks,\Spring=Spring Marks}%
 {\\\DTLfetch{students}{regnum}{\RegNum}{name} & \Autumn & \Spring}%
 \end{tabular}
\end{table}
```

The result is shown in Table 5.27.

Let's suppose now that the `students.csv` file has the first name and surname in separate columns rather than single columns. So the CSV file looks like:

Table 5.27: Student Marks (Joining Databases)

Name	**Autumn Marks**	**Spring Marks**
Joe Bloggs	80	85
Jane Doe	70	90
John Smith	75	60

```
regnum,forename,surname
12344,Mary,Brown
12345,Joe,Bloggs
12346,Jane,Doe
12347,John,Smith
12348,Alice,Jones
```

You may be tempted to replace

```
\DTLfetch{students}{regnum}{\RegNum}{name}
```

with

```
\DTLfetch{students}{regnum}{\RegNum}{forename}\space
\DTLfetch{students}{regnum}{\RegNum}{surname}
```

in the above code, but this is inefficient as it requires two searches for the same row. Instead, you can do:

```
\DTLfetch{students}{regnum}{\RegNum}{forename}\space
\dtlgetentryfromcurrentrow{\Surname}{\dtlcolumnindex{students}{surname}}%
\Surname
```

This can be done because

```
\DTLfetch{students}{regnum}{\RegNum}{forename}
```

is equivalent to

```
\edtlgetrowforvalue{students}{\dtlcolumnindex{students}{regnum}}{\RegNum}%
\dtlgetentryfromcurrentrow
  {\dtlcurrentvalue}{\dtlcolumnindex{students}{forename}}%
\dtlcurrentvalue
```

This means that `\dtlcurrentrow` has already been set by `\DTLfetch` so we can just do another `\dtlgetentryfromcurrentrow` for the surname field. The new code for the table is now:

```
\DTLloaddb{cmp101}{cmp101.csv}% load course data
\DTLloaddb{students}{students.csv}% load student data

\begin{table}[htbp]
```

```
 \caption{Student Marks (Joining Databases)}
 \centering
 \begin{tabular}{lrr}
 \bfseries Name & \bfseries Autumn Marks & \bfseries Spring Marks%
 \DTLforeach*{cmp101}%
 {\RegNum=regnum,\Autumn=Autumn Marks,\Spring=Spring Marks}%
 {\\%
  \DTLfetch{students}{regnum}{\RegNum}{forename}\space
  \dtlgetentryfromcurrentrow{\Surname}{\dtlcolumnindex{students}{surname}}%
  \Surname
  & \Autumn & \Spring}%
 \end{tabular}
\end{table}
```

The result is shown in Table 5.28.

Table 5.28: Student Marks (Joining Databases)

Name	**Autumn Marks**	**Spring Marks**
Joe Bloggs	80	85
Jane Doe	70	90
John Smith	75	60

Caveat: be careful of scoping issues. Suppose you want the first name and surname in separate columns, you may consider doing:

```
  \DTLfetch{students}{regnum}{\RegNum}{forename}%
  &
  \dtlgetentryfromcurrentrow{\Surname}{\dtlcolumnindex{students}{surname}}%
  \Surname
```

(and adding an extra column to the tabular environment). However this will result in undefined values for `\Surname` as `\dtlcurrentrow` is only locally set. After the `&` special character `\dtlcurrentrow` has lost its value as it's no longer in the same scope. You can fix this problem in a number of ways. Firstly you can make `\dtlcurrentrow` global after `\DTLfetch` via

```
\global\dtlcurrentrow=\dtlcurrentrow
```

or you could move the column break to just before `\Surname` and make `\Surname` global:

```
  \DTLfetch{students}{regnum}{\RegNum}{forename}%
  \dtlgetentryfromcurrentrow{\Surname}{\dtlcolumnindex{students}{surname}}%
  \global\let\Surname\Surname
  &
  \Surname
```

There are other possibilities as well, but the first method is probably the best, especially if you have multiple columns you want to fetch.

Here's the updated code:

```
\begin{table}[htbp]
 \caption{Student Marks (Joining Databases)}
 \centering
 \begin{tabular}{llrr}
 \bfseries Forename & \bfseries Surname &
 \bfseries Autumn Marks & \bfseries Spring Marks%
 \DTLforeach*{cmp101}%
 {\RegNum=regnum,\Autumn=Autumn Marks,\Spring=Spring Marks}%
 {\\%
  \DTLfetch{students}{regnum}{\RegNum}{forename}%
  \global\dtlcurrentrow=\dtlcurrentrow
  &
  \dtlgetentryfromcurrentrow{\Surname}{\dtlcolumnindex{students}{surname}}%
  \Surname
  & \Autumn & \Spring}%
 \end{tabular}
\end{table}
```

The result is shown in Table 5.29.

Table 5.29: Student Marks (Joining Databases)

Forename	**Surname**	**Autumn Marks**	**Spring Marks**
Joe	Bloggs	80	85
Jane	Doe	70	90
John	Smith	75	60

6 Creating an index, glossary or list of acronyms (datagidx package)

The datagidx package is provided as an alternative to the glossaries package. Rather than relying on an external indexing application, such as `xindy` or `makeindex`, it uses the database mechanism of the datatool package. *datagidx and glossaries are not compatible.*

First a repeat of the caveat at the start of this manual:

> Use the right tool for the right job.

Don't expect datagidx to perform as efficiently as an application that is designed specifically to sort and collate entries.

If, however, you are happy to exchange efficiency for the convenience of not having to invoke an external application in between LaTeX runs, read on.

Sections 6.1 and 6.3 describe how to create and populate a database that's used to store terms or acronyms. By default the database is sorted when it's displayed using `\printterms` (see section 6.8). This is where the main inefficiency lies in this package. A faster alternative is to use `datatooltk` (see page 36) and its `datagidx` plugin, which will allow you to enter terms in a graphical environment and sort the terms. This way, you only need to sort the database after you enter a new term and the sorting is done by a more efficient language than TeX. Note that this means returning to using an external helper application, but it only needs to be used when you add a new term rather than between each pair of LaTeX runs.

Once you've edited and sorted the database in `datatooltk`, you can then just load it using:

`\loadgidx`

`\loadgidx[`⟨*options*⟩`]{`⟨*filename*⟩`}{`⟨*title*⟩`}`

where ⟨*filename*⟩ is the name of the file saved in `datatooltk`. The remaining arguments ⟨*options*⟩ and ⟨*title*⟩ are the same as for `\newgidx`, described in section 6.1. This command automatically sets the default database to the loaded database. You can change the default database using `\DTLgidxSetDefaultDB`, described in section 6.3.

Since \loadgidx is intended for use with presorted databases, the sort key defaults to nothing.

If you've opted to use datagidx over glossaries because you don't want to install Perl, then don't bother with datatooltk because, although it's a Java application, it requires Perl for the plugins.

6.1 Defining Index/Glossary Databases

The databases and their associated entries described here can only be defined in the preamble. This is because the database must be set up before the auxiliary file is read. If you don't want to lose your place by constantly returning to the preamble to add a new term while you edit your document, consider putting all your definitions in a separate file which can be \input in the preamble. You can then switch between files without losing your place (provided you are using a decent text editor). Alternatively, use datatooltk's datagidx plugin as described above.

First you need to define a customised database that will be used to store the entries in your index, glossary or list of acronyms:

\newgidx

\newgidx[⟨*options*⟩]{⟨*label*⟩}{⟨*title*⟩}

This defines a new database with a unique label and a title. For example:

```
\newgidx{index}{Index}
```

I can now identify this database using the label index. The title "Index" is the default heading when the database is displayed using \printterms (see section 6.8).

The optional argument ⟨*options*⟩ should be a key=value list. Available options:

showgroups Boolean option that indicates whether or not to insert group headings (and a group separator) between index groups, if headings are supported by the given style. If no value is supplied, true is assumed.

style The style to use. The value should be the name of the style. Available styles are listed in subsection 6.8.1.

sort How to sort the database. See subsection 6.8.2 for further details.

balance This is a boolean option that is only applied if columns is greater than 1. If true, the columns are balanced. If false, the columns aren't balanced. If no value is specified, true is assumed.

heading The heading at the start of the index/glossary.

postheading What to put immediately after the heading.

6.2 Locations

Each term in an index or glossary database has an associated location list. This is initially null. When you display the database using `\printterms` (see section 6.8) only those entries with a non-null location list or with a "see" cross-reference are displayed. The location by default is the page number on which the entry has been used. This may be changed to another counter by redefining

`\DTLgidxCounter`

```
\DTLgidxCounter
```

to the name of the required counter. For example:

```
\renewcommand*{\DTLgidxCounter}{section}
```

The datagidx package knows about the following counter styles: `arabic`, `roman`, `Roman`, `alph` and `Alph`. If your location counter uses a different style, you will need to add a new location type. This will only work if the counter uses a command that expands to another command that takes a number as its argument. For example, suppose I want to use small caps Roman numeral page numbering. I need to define a command (say `\myscroman`) that takes a counter name as its argument but expands to another command that takes a number as its argument, like this:

```
\newcommand*{\myscroman}[1]{\myscrromannum{\value{#1}}}
\newcommand*{\myscromannum}[1]{\textsc{\romannumeral#1}}
```

Note that the font changing command `\textsc` is in the definition of `\myscromannum` not in the definition of `\myscroman`. The page counter can now be changed so that it uses `\myscroman`:

```
\renewcommand*{\thepage}{\myscroman{page}}
```

I now have to indicate that `\myscromannum` is a valid location type using:

`\DTLgidxAddLocationType`

```
\DTLgidxAddLocationType{⟨cs name⟩}
```

where ⟨*cs name*⟩ is the name of the control sequence without the initial backslash. Like this:

```
\DTLgidxAddLocationType{myscromannum}
```

Note that this is the command that takes a number as its argument (`\myscromannum`) not the command that takes a counter name as its argument (`\myscroman`).

As with `makeindex` and `xindy`, locations may have a compositor. The default compositor is a full stop but may be changed by redefining

`\DTLgidxSetCompositor`

```
\DTLgidxSetCompositor
```

Alternatively, you can use the package option compositor.

6.3 Defining Terms

Once you have defined the database, you can now define terms associated with that database using

`\newterm`

```
\newterm[⟨options⟩]{⟨name⟩}
```

where ⟨*name*⟩ is the term and ⟨*options*⟩ is a comma-separated list of ⟨*key*⟩=⟨*value*⟩ options. The following keys are available:

database Identifies the database in which to store this term. For example:

```
\newterm[database=index]{reptile}
```

It can be somewhat cumbersome having to keep typing the database for each new term. Instead you can identify the default database using

`\DTLgidxSetDefaultDB`

```
\DTLgidxSetDefaultDB{⟨label⟩}
```

Note: the argument ⟨*label*⟩ is not expanded.

Example:

```
% define two indexes:
\newgidx{index}{Index}
\newgidx{people}{People}
% Set "index" as the default database:
\DTLgidxSetDefaultDB{index}
% This batch of terms will be added to database "index":
\newterm{reptile}
\newterm{mammal}
\newterm{insect}
% Set "people" as the default database:
\DTLgidxSetDefaultDB{people}
% This batch of terms will be added to database "people":
\newterm{Bob}
\newterm{Mary}
\newterm{Jane}
```

label A unique identifying label. This should not contain any active characters. If omitted, the label is extracted from ⟨*name*⟩ (see below).

sort The sort key. If omitted, this is extracted from ⟨*name*⟩ (see below).

parent The parent entry, if this is a sub-term. An entry may only have one parent. If you want the same term to appear under two different parents, you'll have to define two separate terms with the same name but different parents (and different labels). This is the only way to avoid ambiguity with the hyperlinks (if enabled).

text How the entry should appear in the document text. This is ⟨*name*⟩ by default. If this option is used, ⟨*name*⟩ indicates how the entry should appear in the index, glossary or list of acronyms.

description An optional description. This is usually not required for an index but needed for a glossary.

plural The plural form of the term. If omitted this is formed by appending "s" to ⟨*name*⟩ (or the value of the text key if supplied).

symbol An associated symbol if required.

short An associated short form if required. (Default ⟨*name*⟩.)

long An associated long form if required. (Default ⟨*name*⟩.)

shortplural An associated short plural if required. (Default formed by appending "s" to the value of the short key.)

longplural An associated long plural if required. (Default formed by appending "s" to the value of the long key.)

see A cross-reference to a synonym. The value should be the label of another entry. This entry will not have a location list, just the reference to the other term.

seealso A cross-reference to a closely related term. This entry should have both a location list and a reference to the other term.

If the or key are omitted, datagidx tries to form sensible defaults. At the moment, this involves stripping certain commands (`\MakeUppercase`, `\MakeLowercase`, `\MakeTextUppercase`, `\MakeTextLowercase`, `\acronymfont`, `\textsc`, `\textbf`, `\textmd`, `\textit`, `\textsl`, `\textrm`, `\texttt`, `\textsf`, `\emph`, `\ensuremath` and `\textsuperscript`), stripping accents and replacing certain control characters or control sequences (~ is replace with a space and `\&` is replaced with `\andname` (if defined) or "and" (if `\andname` isn't defined)). The Greek letter commands (`\alpha` etc) are converted to their name.

Examples:

1. `\ensuremath` is stripped and `\alpha` is converted to "alpha" so the following:

   ```
   \newterm{\ensuremath{\alpha}}
   ```

   sets both the label and sort to `alpha` but the name and text fields are set to `\ensuremath{\alpha}`.

2. Accent commands are stripped so the following:

   ```
   \newterm{mac\'edoine}
   ```

   sets both the label and sort fields to `macedoine` but the name and text fields are set to `mac\'edoine`.

   The first letter must be grouped if it's an accent or ligature or a character outside the range `a...z` or `A...Z`.

3. This example must have the sort and label fields set manually because the first letter has an accent:

   ```
   \newterm[label=elite,sort=elite]{{\'e}lite}
   ```

4. The same applies if you are using the inputenc package:

   ```
   \newterm[label=elite,sort=elite]{{é}lite}
   ```

5. The same applies to plural terms set explicitly:

```
\newterm
 [%
   plural={{œ}sophagi},%
   label={oesophagus},%
   sort={oesophagus}%
 ]
 {{œ}sophagus}
```

6. Commands such as `\oe` aren't dealt with, so you must manually set the label and sort key:

```
\newterm[label=manoeuvre,sort=manoeuvre]{man\oe uvre}
```

7. The same applies if you are using the inputenc package:

```
\newterm[label=manoeuvre,sort=manoeuvre]{manœuvre}
```

> Take care if any of the values to fields contain a comma or equal sign. The value must be grouped.

8. This term contains a comma in some of the fields:

```
\newterm
 [%
   label={comma},%
   sort={,},%
   text={comma (,)}%
   plural={commas (,)}%
 ]
 {, (comma)}
```

   In the text, the entry is `comma (,)` but in the index the entry is sorted according to the comma symbol and is displayed as `, (comma)`.

6.3.1 Commands to Assist Sorting

There are some situations where you will have to specify the sort key, for example:

```
\newterm
[
  sort={Ten Downing Street}
]
{10 Downing Street}
```

However, there are some commands provided to help set the default sort for entries that are sorted differently from the way they are typeset in the index/glossary, which can help reduce the number of times you need to explicitly set the sort field.

\DTLgidxParen

\DTLgidxParen{⟨*text*⟩}

This command is provided for parenthetical material that should be typeset in the index, but should not contribute to the sort unless there is an identical entry without parenthetical material.

For example:

```
\newterm{0\DTLgidxParen{zero}}
```

This term is typeset as `0 (zero)`, but has the sort and label fields set to `0`.

The default sort used is word-order sorting. This has a special number group for entries where the sort field consists solely of digits and they are sorted numerically rather than by string comparison. Using `\DTLgidxParen` in this manner, the following terms will appear in numerical order in the index:

```
\newterm{0\DTLgidxParen{zero}}
\newterm{1\DTLgidxParen{one}}
\newterm{2\DTLgidxParen{two}}
\newterm{3\DTLgidxParen{three}}
\newterm{10\DTLgidxParen{ten}}
\newterm{100\DTLgidxParen{one hundred}}
\newterm{20\DTLgidxParen{twenty}}
```

If `\DTLgidxParen` was not used and the parentheses were explicitly included, e.g. `0 (zero)`, then the entries would be placed in the symbol group instead and be sorted according to string (so `10 (ten)` would come before `2 (two)`).

\DTLgidxPlace

\DTLgidxPlace{⟨*country/county*⟩}{⟨*city/town*⟩}

Use this command to indicate a place. For example:

```
\newterm{\DTLgidxPlace{USA}{New York}}
```

This sets the label and name to `New York, USA`, the text field is set to just `New York` and the sort field is set to `New York\datatoolplacecomma USA` (see section 5.8).

\DTLgidxSubject

\DTLgidxSubject{⟨*subject*⟩}{⟨*text*⟩}

Use this to indicate a subject, concept or object. Example:

```
\newterm{\DTLgidxSubject{population}{New York}}
```

Both the label and name fields default to `New York, population`, the text field defaults to `population` and the sort field is set to `New York\datatoolsubjectcomma population` (see section 5.8).

\DTLgidxName

\DTLgidxName{⟨*forename(s)*⟩}{⟨*surname*⟩}

Use this command to index a person. The entry will be sorted according to the surname then the forenames. The entry will be displayed as ⟨*surname*⟩, ⟨*forename(s)*⟩ in the index but will be displayed as ⟨*forename(s)*⟩ ⟨*surname*⟩ when referenced in the document. The label, on the other hand, is set to just the surname. Example:

```
\newterm{\DTLgidxName{Donald E.}{Knuth}}
```

This sets the name field to `Knuth, Donald E.`, the text field to `Donald E. Knuth`, the label to `Knuth` and the sort field to `Knuth\datatoolpersoncomma Donald E.` (see section 5.8).

A person's title (such as "Dr") should typically not affect the sort, unless there is another person with the same surname and forenames (or initials) without a title. To assist this, you can identify a person's title using:

\DTLgidxRank

\DTLgidxRank{⟨*title*⟩}{⟨*forename(s)/initial(s)*⟩}

Using examples from the Oxford Style Manual:

```
\newterm[label=AliceMeynell]{\DTLgidxName{Meynell}{Alice}}
\newterm[label=DrMeynell]{\DTLgidxName{Meynell}{\DTLgidxRank{Dr}{A.}}}
\newterm[label=AMeynell]{\DTLgidxName{Meynell}{A.}}
```

Here the labels must be set as the surnames are identical for each entry, but the entries will be sorted in the order: "Meynell, A.", "Meynell, Dr A." and "Meynell, Alice".

You can use

\DTLgidxNameNum

\DTLgidxNameNum{⟨*number*⟩}

to indicate a number associated with a name. The number is typeset as an uppercase Roman numeral in the text, but is sorted numerically.

For example:

```
\newterm{James~\DTLgidxNameNum{1}}
```

This is typeset as `James~I`, but gets the label `James I` (note no tilde) and the sort field is set to `James 01`. This means that if I want to index all the Kings whose name is James, they will appear in the correct order in the index.

If a term contains a variant of "Mac" you can also use:

`\DTLgidxMac`

`\DTLgidxMac{`⟨*text*⟩`}`

The entry will be typeset with ⟨*text*⟩ but the sort key will have ⟨*text*⟩ replaced with `Mac`. Examples:

```
\newterm{\DTLgidxName{Joe}{\DTLgidxMac{Mc}Cullers}}
\newterm{\DTLgidxName{Bob}{\DTLgidxMac{M'}Fingal}}
\newterm{\DTLgidxMac{Mc}Carthyism}
\newterm{\DTLgidxMac{Mc}Guffin}
```

Similarly saints can be identified using:

`\DTLgidxSaint`

`\DTLgidxSaint{`⟨*text*⟩`}`

Examples:

```
\newterm{\DTLgidxSaint{St} Julian}
\newterm{\DTLgidxName{Q.}{\DTLgidxSaint{St}~John-Smythe}}
\newterm{\DTLgidxPlace{\DTLgidxSaint{St}~Andrews}{Fife}}
```

These will be sorted according to `Saint Julian`, `Saint John-Smythe\datatoolpersoncomma Q.` and `Saint Andrews\datatoolplacecomma Fife`.

Particles, such as "de", "von" or "of" are usually ignored when sorting. These can be identified using:

`\DTLgidxParticle`

`\DTLgidxParticle{`⟨*text*⟩`}`

Examples:

```
\newterm{\DTLgidxName{Fred}{\DTLgidxParticle{de}{Winter}}}
\newterm{\DTLgidxName{Gustav}{\DTLgidxParticle{von}{Aschenbach}}}
```

Here the names are sorted according to `Winter\datatoolpersoncomma Fred` and `Aschenbach\datatoolpersoncomma Gustav` but the labels are set to `deWinter` and `vonAschenbach`.

A person can also be indicated by their office, for example "Henry, scribe of Bury St Edmunds". For this, you can use:

`\DTLgidxOffice`

`\DTLgidxOffice{`⟨*office*⟩`}{`⟨*name*⟩`}`

Here the label defaults to just ⟨*name*⟩, so you may need to set the label manually to ensure uniqueness. Examples:

```
\newterm
[
  label={HenrySonJohn}
]
{\DTLgidxOffice{son \DTLgidxParticle{of}{John}}{Henry}}

\newterm
[
  label={HenryBeaumont}
]
{\DTLgidxOffice{bishop \DTLgidxParticle{of}{Bayeux}}{Henry
\DTLgidxParticle{de}{Beaumont}}}

\newterm
[
  label={HenryScribe}
]
{\DTLgidxOffice{scribe \DTLgidxParticle{of}{Bury}
   \DTLgidxSaint{St}~Edmunds}{Henry}}
```

You can hook into the mechanism that sets the default sort key by adding to the definition of

`\newtermlabelhook`

`\newtermlabelhook`

You can use etoolbox's `\appto` command to append to this hook. For example, suppose you want to index the terms `\TeX`, `e\TeX` and `pdf\TeX`, but you want the terms to have the label and sort fields to be just `TeX`, `eTeX` and `pdfTeX`, then you can add to the hook so that it automatically converts `\TeX` to just `TeX`:

```
\appto\newtermlabelhook{\def\TeX{TeX}}
```

(Note that it's important to use the local `\def` rather than the global `\gdef` to ensure the redefinition is localised.)

Now the terms can simply be defined using:

```
\newterm{\TeX}
\newterm{e\TeX}
\newterm{pdf\TeX}
```

To assist in using this mechanism, the following commands are available (these commands may also be used in the mandatory argument of `\newterm`):

`\DTLgidxNoFormat`

`\DTLgidxNoFormat{`⟨*text*⟩`}`

This commands simply does its argument, so any commands that should be stripped from the label or sort field without the loss of their argument can be `\let` to `\DTLgidxNoFormat`. For example, suppose you want to define a command called, say, `\appname` that you want to use to identify application names, like this:

```
\newcommand*{\app}[1]{\texttt{#1}}
```

This command needs to be stripped from the label and sort, so it can be added to the hook like this:

```
\appto\newtermlabelhook{\let\app\DTLgidxNoFormat}
```

Now you can define terms like this:

```
\newterm{\app{makeindex}}
\newterm{\app{xindy}}
```

The label and sort keys are then set to `makeindex` (for the first term) and `xindy` (for the second term).

`\DTLgidxGobble`

`\DTLgidxGobble{`⟨*text*⟩`}`

This command discards its argument, so it can be used if you not only want to strip a command but also its argument from the label and sort fields.

For example, suppose you want some terms to have a footnote (both in the index/glossary and in the document text) but the footnote shouldn't form part of the sort or label fields. You can add to the hook like this:

```
\appto\newtermlabelhook{\let\footnote\DTLgidxGobble}
```

Now you can define some terms with footnotes:

```
\newterm{foo\footnote{a note about foo}}
\newterm{bar\footnote{a note about bar}}
```

The label and sort keys are then set to `foo` (for the first term) and `bar` (for the second term).

`\DTLgidxIgnore`

`\DTLgidxIgnore`

This is similar to `\DTLgidxGobble` but only affects the sort key not the label. Example:

```
\newterm{de\DTLgidxIgnore{-}escalate}
```

This is displayed as `de-escalate` and gets the label `de-escalate` but is sorted according to `deescalate`.

`\DTLgidxStripBackslash`

`\DTLgidxStripBackslash{`⟨*control sequence*⟩`}`

This can be used to "stringify" a control sequence and remove the leading backslash. For example, suppose you want to index the ampersand symbol (&) but you want to sort it according to the actual symbol `&`, you can do:

```
\newterm
 [%
   label={amp},
   sort={\DTLgidxStripBackslash{\&}},
   text={ampersand (\&)},
   plural={ampersands (\&)},
 ]
 {\& (ampersand)}
```

6.4 Referencing Terms

You can reference terms using

`\useentry`

`\useentry{`⟨*label*⟩`}{`⟨*field*⟩`}`

This fetches the given field for the term identified by ⟨*label*⟩, displays it and marks the term as having been used. Example, suppose I have previous (in the preamble) defined the term "reptile" using:

```
\newterm{reptile}
```

I can now reference this term in the document:

```
\useentry{reptile}{Text}
```

or if I want the plural, I can use:

```
\useentry{reptile}{Plural}
```

There are also uppercase versions:

`\Useentry`

`\Useentry{⟨label⟩}{⟨field⟩}`

This makes the first letter uppercase (using the mfirstuc package) or to make the whole text uppercase use:

`\USEentry`

`\USEentry{⟨label⟩}{⟨field⟩}`

If you use the hyperref package, the above commands will automatically create hyperlinks to the relevant entry in the index/glossary. You can suppress this action by using one of the following analogous commands instead:

`\useentrynl`

`\useentrynl{⟨label⟩}{⟨field⟩}`

`\Useentrynl`

`\Useentrynl{⟨label⟩}{⟨field⟩}`

`\USEentrynl`

`\USEentrynl{⟨label⟩}{⟨field⟩}`

In all the above commands, the ⟨*label*⟩ argument may optionally start with `[`⟨*format*⟩`]`, where format is the name of a control name *without* the preceding backslash. This command will be applied to this location in the entry's location list when it's displayed in the index/glossary.

For example:

```
\useentry{[textbf]reptile}{Text}
```

Note that the command (`\textbf` in the above example) should take one argument (the location). If you attempt to use, say, a declaration (such as `\bfseries`) the effect won't be localised.

You can display the value of a field without indexing it using:

\glsdispentry

\glsdispentry{⟨*label*⟩}{⟨*field*⟩}

To make the first letter uppercase, use:

\Glsdispentry

\Glsdispentry{⟨*label*⟩}{⟨*field*⟩}

The above commands aren't expandable. If you want to fetch a value without displaying or using it, you can use:

\DTLgidxFetchEntry

\DTLgidxFetchEntry{⟨*cs*⟩}{⟨*label*⟩}{⟨*field*⟩}

where ⟨*cs*⟩ is a control sequence, ⟨*label*⟩ is the label that uniquely identifies the entry and ⟨*field*⟩ is the required field. The value of that field is stored in ⟨*cs*⟩.

The predefined database fields are:

Name How the term appears in the index/glossary (as specified by the mandatory argument of \newterm).

Text The value of the text field.

Plural The value of the plural field.

Description The value of the description field.

Symbol The value of the symbol field.

Long The value of the long field.

Short The value of the short field.

LongPlural The value of the longplural field.

ShortPlural The value of the shortplural field.

See The value of the see field.

SeeAlso The value of the seealso field.

Sort The value of the sort field.

Parent The value of the parent field.

Label The entry's unique identifying label.

Used Has the value 1 (entry has been used) or either 0 or undefined (entry hasn't been used).

Location The entry's location list (picked up from the last LaTeX run).

In addition, there are some fields designed for internal use: `Child`, `FirstId` and `CurrentLocation`.

You can add an entry to the index/glossary without displaying any text using:

`\glsadd`

`\glsadd{`⟨*label*⟩`}`

As with `\useentry`, ⟨*label*⟩ maybe in the form `[`⟨*format*⟩`]{`⟨*label*⟩`}` where ⟨*format*⟩ is the name of a control sequence *without* the leading backslash.

You can also add all entries from a particular database using

`\glsaddall`

`\glsaddall{`⟨*db name*⟩`}`

where ⟨*db name*⟩ is the name of the database.

Unlike the commands of the same name provided by the glossaries package, here there is a difference between `\glsaddall` and using `\glsadd` on all entries in the database. In the case of `\glsadd` a location is added to the location list for that entry. However in the case of `\glsaddall` no location is added to each entry's location list, but the location list is set to non-null so the entry will appear in the index/glossary.

6.4.1 Shortcut Commands

There are some shortcuts to common fields (if you are used to the glossaries package, note that these commands have different formats to the commands provided by glossaries with the same name):

`\gls`

`\gls{`⟨*label*⟩`}`

This is equivalent to `\useentry{`⟨*label*⟩`}{Text}`.

`\glspl`

`\glspl{`⟨*label*⟩`}`

This is equivalent to `\useentry{`⟨*label*⟩`}{Plural}`.

\glsnl

\glsnl{⟨label⟩}

This is equivalent to \useentrynl{⟨label⟩}{Text}.

\glsplnl

\glsplnl{⟨label⟩}

This is equivalent to \useentrynl{⟨label⟩}{Plural}.

\Gls

\Gls{⟨label⟩}

This is equivalent to \Useentry{⟨label⟩}{Text}.

\Glspl

\Glspl{⟨label⟩}

This is equivalent to \Useentry{⟨label⟩}{Plural}.

\Glsnl

\Glsnl{⟨label⟩}

This is equivalent to \Useentrynl{⟨label⟩}{Text}.

\Glsplnl

\Glsplnl{⟨label⟩}

This is equivalent to \Useentrynl{⟨label⟩}{Plural}.

\glssym

\glssym{⟨label⟩}

This is equivalent to \useentry{⟨label⟩}{Symbol}.

\Glssym

\Glssym{⟨label⟩}

This is equivalent to \Useentry{⟨label⟩}{Symbol}.

6.5 Adding Extra Fields

You can add new fields to the index/glossary database using:

\newtermaddfield

\newtermaddfield[⟨*db list*⟩]{⟨*field name*⟩}{⟨*key name*⟩}{⟨*default value*⟩}

The optional argument ⟨*db list*⟩ is a comma-separated list of databases that should have this new field. If omitted, the field will be added to all the defined databases. The argument ⟨*field name*⟩ is the label to give this new column in the database(s). The argument ⟨*key name*⟩ is the name of the new key to use in the optional argument of \newterm. The final argument ⟨*default value*⟩ is the default value if the key isn't used. Within ⟨*default value*⟩, you may use

\field

\field{⟨*key*⟩}

to indicate the value of another key.

For example, suppose I want to be able to specify an alternative plural. I can add a new field like this:

```
\newtermaddfield{AltPlural}{altplural}{}
```

This adds a new column with the label AltPlural to each defined index/glossary database and adds a new key called altplural that I can now use in \newterm. The default is set to empty. Now I can define terms with an alternative plural:

```
\newterm[altplural=kine]{cow}
```

In the document, I can use \gls{cow} to display "cow", \glspl{cow} to display "cows" and \useentry{cow}{AltPlural} to display "kine". To make life a little easier, I can define a new command to save typing:

```
\newcommand*{\glsaltpl}[1]{\useentry{#1}{AltPlural}}
```

Now I can just do \glsaltpl{cow} to display "kine".

Here's another example. Suppose I want to add a field that produces the past tense of a verb. In this case, the default should be formed by appending "ed" to the text field. The new field can be defined as follows:

```
\newtermaddfield{Ed}{ed}{\field{text}ed}
```

This adds a new column labelled "Ed" and defines a new key called "ed" that can be used with \newterm. Now I can defined some verbs:

```
\newterm{jump}
\newterm[ed=went]{go}
```

Let's define a convenience command to access this field:

```
\newcommand*{\glsed}[1]{\useentry{#1}{Ed}}
```

This new field can now be referenced in the document:

```
He \glsed{jump} over the gate.
She \glsed{go} to the shop.
```

The above will be displayed as: He jumped over the gate. She went to the shop.

6.6 Acronyms

You may have noticed that you can specify short and long fields when you define a new term. There is a convenient shortcut command which uses `\newterm` to define an acronym. The syntax is:

`\newacro`

`\newacro[`⟨*options*⟩`]{`⟨*short*⟩`}{`⟨*long*⟩`}`

This is a shortcut for

```
\newterm
 [%
   description={\capitalisewords{⟨long⟩}},%
   short={\acronymfont{⟨short⟩}},%
   long={⟨long⟩},%
   text={\DTLgidxAcrStyle{⟨long⟩}{\acronymfont{⟨short⟩}}},%
   plural={\DTLgidxAcrStyle{⟨long⟩s}{\acronymfont{⟨short⟩s}}},%
   sort={⟨short⟩},%
   ⟨options⟩%
 ]%
 \MakeTextUppercase{⟨short⟩}
```

where `\capitalisewords` is defined in mfirstuc (automatically loaded by datagidx) and `\MakeTextUppercase` is defined in textcase (automatically loaded by datagidx). The other commands used are defined by datagidx:

`\acronymfont`

`\acronymfont`

By default this just typesets its argument but can be redefined if the acronyms need to be typeset in a certain style (such as small caps).

`\DTLgidxAcrStyle`

\DTLgidxAcrStyle{⟨*long*⟩}{⟨*short*⟩}

This governs how the acronym is typeset in the text field. This defaults to: ⟨*long*⟩ (⟨*short*⟩).

6.6.1 Using Acronyms

You can use terms that represent acronyms via commands such as `\useentry`. For example, if you define the following in the preamble:

```
\newacro{css}{cascading style sheet}
```

then later in the text you can use:

```
\useentry{css}{Short}
```

to access the short form and

```
\useentry{css}{Long}
```

to access the long form. You can also use

```
\useentry{css}{Text}
```

(or `\gls{css}`) to access the full version. However with acronyms you generally only want the full form on first use and just the short form on subsequent use. The following commands are provided to do that. The singular form is obtained using:

\acr

\acr{⟨*label*⟩}

The plural form is obtained using:

\acrpl

\acrpl{⟨*label*⟩}

Note that, unlike the glossaries package, `\acr` isn't the same as `\gls`. With datagidx, `\gls` always references the text field. There is no "first" field.

As a general rule, you're not supposed to capitalise the first letter of an acronym (especially if it is displayed in small caps) but if you need to you can use:

\Acr

\Acr{⟨*label*⟩}

and

\Acrpl

```
\Acrpl{⟨label⟩}
```

6.6.2 Unsetting and Resetting Acronyms

You can reset a term so it's marked as not used with:

\glsreset

```
\glsreset{⟨label⟩}
```

or you can unset a term so it's marked as used with:

\glsunset

```
\glsunset{⟨label⟩}
```

You can reset all the terms defined in a given database using:

\glsresetall

```
\glsresetall{⟨db name⟩}
```

or unset all the terms defined in a given database using:

\glsunsetall

```
\glsunsetall{⟨db name⟩}
```

where ⟨*db name*⟩ is the name of the database as supplied when the database was defined using \newgidx.

6.7 Conditionals

You can test if a term exists using

\iftermexists

```
\iftermexists{⟨label⟩}{⟨true part⟩}{⟨false part⟩}
```

You can test if a term has been used using:

\ifentryused

```
\ifentryused{⟨label⟩}{⟨true part⟩}{⟨false part⟩}
```

6.8 Displaying the Index or Glossary

The index or glossary can be displayed using

\printterms

```
\printterms[⟨options⟩]
```

You will need to run LaTeX at least twice to ensure your index/glossary is up-to-date. The first run will only display any entries that have a "See" field defined.

The optional argument ⟨*options*⟩ is a comma-separated list of ⟨*key*⟩=⟨*value*⟩ options. Available keys:

database The name of the database (as given in `\newgidx`).

postdesc This may have the value dot (put a full stop after the description) or none (don't put a full stop after the description).

prelocation This indicates what to put before the location list. Available values:

- **none** Nothing.
- **enspace** An en-space.
- **space** An ordinary space.
- **dotfill** A dotted line (`\dotfill`).
- **hfill** Expandable space (`\hfill`).

location This indicates how to display the location list. Available values:

- **hide** Don't display the location list.
- **list** Display the location list.
- **first** Only display the first location in the list.

symboldesc How to format the symbol in relation to the description. Available values:

- **symbol** Display the symbol but not the description.
- **desc** Display the description but not the symbol field.
- **(symbol) desc** Display the symbol (if defined) in parentheses followed by the description.
- **desc (symbol)** Display the description followed by the symbol (if defined) in parentheses.
- **symbol desc** Display the symbol (if defined) followed by the description.

desc symbol Display the description followed by the symbol (if defined).

columns This should be a positive number that indicates the page column layout. If the value is greater than 1, the multicols environment is used (defined in the multicol package, which is automatically loaded).

namecase Indicates whether any case change should be applied to the entry's name field. Available values:

nochange Don't apply a case change.

uc Convert the name to uppercase.

lc Convert the name to lowercase.

firstuc Convert the first letter to uppercase (using `\makefirstuc` defined in mfirstuc).

capitalise Capitalise initial letters of each word in the name (using `\capitalisewords` defined in mfirstuc).

namefont The font changing command to apply to the name. (Include the initial backslash.) Declarations may be used.

postname What to put after the name.

see Indicates how the cross-reference (given in the "See" field) should be displayed. Available values:

comma Insert a comma followed by a space in front of the cross-reference.

brackets Insert a space before the cross-reference and put the cross-reference in parentheses.

dot Insert a full stop followed by a space in front of the cross-reference.

space Insert a space before the cross-reference.

nosep Don't insert anything before the cross-reference.

semicolon Insert a semi-colon followed by a space in front of the cross-reference.

location Display the cross-reference in the same way as a location.

child Indicates whether child entries should have their name displayed. Available values: named (display the child's name) and noname (don't display the child's name).

showgroups Boolean option that indicates whether or not to insert group headings (and a group separator) between index groups, if headings are supported by the given style. If no value is supplied, true is assumed.

style The style to use. The value should be the name of the style. Available styles are listed in subsection 6.8.1.

symbolwidth Some of the styles allow you to specify a width for the symbol field. This width can be specified with this option. The value will be ignored by some of the styles.

locationwidth Some of the styles allow you to specify a width for the location field. This width can be specified with this option. The value will be ignored by some of the styles.

childsort A boolean option that indicates whether or not the child entries should be sorted. If true, the child entries are listed using the same sort order as the sort applied to the database. If false, the child entries are listed in the order they were defined. If the value is missing, true is assumed.

heading The heading at the start of the index/glossary.

postheading What to put immediately after the heading.

sort How to sort the database. See subsection 6.8.2 for further details.

balance This is a boolean option that is only applied if columns is greater than 1. If true, the columns are balanced. If false, the columns aren't balanced. If no value is specified, true is assumed.

condition This specifies a boolean condition (as used by `\DTLforeach`) so you can display only those entries where the condition is met. For example, to only display entries starting with "H" (not including any entry that is just the letter "H") you can do:

```
\printterms[condition={\DTLisiopenbetween{\Name}{H}{I}}]
```

6.8.1 Index or Glossary Styles

The index or glossary style is given by the style key in the optional argument of `\newgidx` or `\printterms`. The following styles are available:

index The "index" style is a basic style for an index. This style accepts the locationwidth and symbolwidth keys in `\printterms`. This is the default style.

indexalign The "indexalign" style is similar to the "index" style but aligns the descriptions.

align The "align" style aligns the fields. This style accepts the locationwidth and symbolwidth keys in `\printterms`.

gloss The "gloss" style is a basic glossary style. This style uses

`\DTLgidxChildSep`

```
\DTLgidxChildSep
```

as the separator between child entries (defaults to a space) and

`\DTLgidxPostChild`

```
\DTLgidxPostChild
```

to indicate what to put after the list of child entries (defaults to nothing).

dict The "dict" style is designed for dictionary-like glossaries. This assumes a hierarchical structure where the top level entries have a name. The next level is used to indicate a category (such as "adjective" or "noun"). If there is only one meaning for the term, this level also has a description. If there is more than one meaning, each meaning should be a child of the category entry. Only third level entries are numbered. No further levels are expected. The symbol field is ignored.

If showgroups is set, the group headers will be placed in a `\chapter` (if defined) or in a `\section` (if `\chapter` isn't defined).

This style uses:

`\DTLgidxCategoryNameFont`

```
\DTLgidxCategoryNameFont{⟨text⟩}
```

The font used to display the name of the category (first child level).

`\DTLgidxCategorySep`

```
\DTLgidxCategorySep
```

The category separator. (Defaults to a space).

\DTLgidxSubCategorySep

```
\DTLgidxSubCategorySep
```

The category separator. (Defaults to a space).

\DTLgidxDictPostItem

```
\DTLgidxDictPostItem
```

Indicates what to do at the end of each top-level item. (Defaults to \par).

The indentation is given by the length register

\datagidxdictindent

```
\datagidxdictindent
```

This value defaults to 1em.

For additional commands that affect the style of the indexes or glossaries, see the documented code datatool-code.pdf.

6.8.2 Sorting the Index or Glossary Database

By default the index/glossary databases are sorted according to the Sort field using the \dtlwordindexcompare handler (see section 5.8). Note that the *entire* database is sorted, which is less efficient that using external indexing applications, such as makeindex or xindy, which only sort the terms that have been used in the document. In addition, the sorting algorithm used by datatool is less efficient than that used by a custom-built sorting and collation application.

The database is sorted at the start of \printterms according to the value of the sort key supplied by \printterms. To completely suppress the sorting, set this key to empty. Example:

```
\printterms[database=index,sort={},showgroups=false]
```

Note that in the above, I also switched off the group headers as they don't make sense with an unsorted index or glossary.

If you want to use a different comparison handler, you can set the sort key to the required sort command, where you can use

\DTLgidxCurrentdb

```
\DTLgidxCurrentdb
```

to indicate the current database.

For example, to sort using letter rather than word comparison:

```
\printterms[database=index,
  sort={\dtlsort{Sort}{\DTLgidxCurrentdb}{\dtlletterindexcompare}}]
```

You may recall from earlier that the index/glossary databases have a column labelled "FirstId". This can be used if you want to sort the database according to the order of usage. Example:

```
\printterms[database=index,
 sort={\dtlsort{FirstId}{\DTLgidxCurrentdb}{\dtlcompare}}]
```

Note that here I've used the `\dtlcompare` handler (which is the fastest handler) as I'm only concerned with a numerical rather than a string comparison.

The default value of the sort key is actually:

```
\dtlsort{Sort,FirstId}{\DTLgidxCurrentdb}{\dtlwordindexcompare}}
```

This means that entries with duplicate "Sort" fields are then sorted according to use.

Optimization

If you have used `xindy` or `makeindex`, you'll be familiar with the document creation process. The document is first compiled, then the indexing application is run to sort and collate the entries, then the document is compiled again (and possible once more). This involves two (or three) LaTeX runs and one sort and collate run. With the datagidx package, the sorting and collation is done every LaTeX run. For a large index, this can be quite slow. If you're not editing the index or glossary, you might prefer not to have to keep sorting the database whenever you update the document. To assist this, datagidx provides the optimize package option. This may take the following values:

off Don't use the optimize facility. (The index/glossary databases will be sorted every run, unless the sort is switched off by setting the sort key to empty.)

low Use the "low" optimize setting. This only sorts the index/glossary databases every other run. (Assuming that the sorting is done via the `\printterms`sort key rather than explicitly using `\dtlsort` or `\DTLsort` somewhere else in the document.) Don't use this option if sorting the databases makes the document out-of-date. (For example, the group headers use sectioning commands.)

high Use the "high" optimize setting. This sorts the index/glossary databases on the first run, then writes the sorted databases to external files, which are read in on subsequent runs. Again this assumes that the sorting is done via the `\printtermssort` key. Don't use this option if you want to edit the index/glossary database.

6.9 Package Options

The following package options are available for datagidx:

optimize Sets the optimization. (See section 6.8.2.)

columns Sets the default number of columns to use for the indexes or glossaries. (See section 6.8.)

child Sets whether or not to show the name in child entries, where the style supports this option. (See section 6.8.)

namecase Sets the case change for the entry's name. (See section 6.8.)

namefont Sets the font for the entry's name. (See section 6.8.)

postname Indicates what to put after the entry's name. (See section 6.8.)

postdesc Indicates what to put after the entry's description. (See section 6.8.)

prelocation Indicates what to put before the entry's location. (See section 6.8.)

location Indicates how to display the entry's location. (See section 6.8.)

see Indicates how to display the entry's cross-reference list. (See section 6.8.)

symboldesc Indicates how to display the entry's symbol in relation to the description. (See section 6.8.)

compositor Sets the location compositor. (See section 6.2.)

draft Displays additional information, such as target names.

final Hides the draft information.

verbose Use datatool's verbose mode.

nowarn A boolean option that suppresses datagidx's rerun warnings.

Example 22 (Creating an Index)

In this document, I have used the datagidx package and the hyperref package. In the preamble, I have the following:

```
\usepackage{datagidx}
\usepackage[colorlinks]{hyperref}

\newgidx{index}{Index}% define a database for the index

\DTLgidxSetDefaultDB{index}% set this as the default

\newterm{mac\'edoine}
\newterm{macram\'e}
\newterm[label=elite]{{\'e}lite}
\newterm{reptile}
\newterm[seealso={reptile}]{crocodylian}

\newterm
 [%
   parent=crocodylian
 ]
 {crocodile}

\newterm
 [%
   parent=crocodylian
 ]
 {alligator}

\newterm
 [%
   parent=crocodylian,
   description={(also cayman)}
 ]
 {caiman}

\newterm[see={caiman}]{cayman}
```

Now here's some code to go in the document:

```
Here are some words containing accents: \gls{macedoine},
\gls{macrame} and \gls{elite}. \Gls{elite} requires extra care as it
starts with an accented letter. A \gls{crocodylian} is a family of
\glspl{reptile} consisting of \glspl{crocodile}, \glspl{alligator} and
\glspl{caiman}.
```

This produces the following:

Here are some words containing accents: macédoine, macramé and élite. Élite requires extra care as it starts with an accented letter. A

crocodylian is a family of reptiles consisting of crocodiles, alligators and caimans.

The index can then be displayed using:

```
\printterms[heading={\section*},database=index]
```

This requires two runs to ensure the index is up-to-date. The resulting index is as follows:

Index

Here's the code if you want to add the letter groups (I've also added a dotted line before the location):

```
\printterms
 [
   heading={\section*},
   database=index,
   prelocation=dotfill,
   showgroups
 ]
```

which produces:

Index

7 Pie Charts (datapie package)

The datapie package is not loaded by the datatool package, so you need to explicitly load datapie if you want to use any of the commands defined in this section. You will also need to have the pgf/tikz packages installed. The datapie package may be given the following options:

color=datapie Colour option (default).

gray=datapie Grey scale option.

rotateinner=datapie Rotate inner labels so that they are aligned with the pie chart radial axis.

norotateinner=datapie Don't rotate inner labels (default).

rotateouter=datapie Rotate outer labels so that they are aligned with the pie chart radial axis.

norotateouter=datapie Don't rotate outer labels (default).

Numerical information contained in a database created by the datatool package can be converted into a pie chart using

\DTLpiechart

\DTLpiechart[⟨*condition*⟩]{⟨*settings list*⟩}{⟨*db name*⟩}{⟨*values*⟩}

where ⟨*db name*⟩ is the name of the database, and ⟨*condition*⟩ has the same form as the optional argument to \DTLforeach described in section 5.4. If ⟨*condition*⟩ is false, that information is omitted from the construction of the pie chart. The argument ⟨*values*⟩ is a comma separated list of ⟨*cmd*⟩=⟨*key*⟩ pairs, the same as that required by the penultimate argument of \DTLforeach. The ⟨*settings list*⟩ is a comma separated list of ⟨*setting*⟩=⟨*value*⟩ pairs, where ⟨*setting*⟩ can be any of the following:

variable This specifies the control sequence to use that contains the value used to construct the pie chart. The control sequence must be one of the control sequences to appear in the assignment list ⟨*values*⟩. This setting is required.

start This is the starting angle of the first segment. The value is 0 by default.

radius This is the radius of the pie chart. The default value is 2cm.

innerratio The distance from the centre of the pie chart to the point where the inner labels are placed is given by this value multiplied by the ratio. The default value is 0.5.

outerratio The distance from the centre of the pie chart to the point where the outer labels are placed is given by this value multiplied by the ratio. The default value is 1.25.

cutawayratio The distance from the centre of the pie chart to the point of cutaway segments is given by this value multiplied by the ratio. The default value is 0.2.

inneroffset This is the absolute distance from the centre of the pie chart to the point where the inner labels are placed. You should use only one or other of innerratio and inneroffset, not both. If you also want to specify the radius, you must use ratio before inneroffset. If omitted, the inner offset is obtained from the ratio multiplied by the innerratio value.

outeroffset This is the absolute distance from the centre of the pie chart to the point where the outer labels are placed. You should use only one or other of outerratio and outeroffset, not both. If you also want to specify the radius, you must use ratio before outeroffset. If omitted, the outer offset is obtained from the ratio multiplied by the outerratio value.

cutawayoffset This is the absolute distance from the centre of the pie chart to the point of the cutaway segments. You should use only one or other of cutawayratio and cutawayoffset, not both. If you also want to specify the radius, you must use ratio before cutawayoffset. If omitted, the cutaway offset is obtained from the ratio multiplied by the cutawayratio value.

cutaway This is a list of cutaway segments. This should be a comma separated list of individual numbers, or number ranges (separated by a dash). For example `cutaway={1,3}` will separate the first and third segments from the rest of the pie chart, offset by the value of the cutawayoffset setting, whereas `cutaway={1-3}` will separate the first three segments from the rest of the pie chart. If omitted, the pie chart will be whole.

innerlabel The value of this is positioned in the middle of each segment at a distance of inneroffset from the centre of the pie chart. The default is the same as the value of variable.

outerlabel The value of this is positioned at a distance of outeroffset from the centre of the pie chart. The default is empty.

rotateinner This is a boolean setting, so it can only take the values `true` and `false`. If the value is omitted `true` is assumed. If true, the inner labels are rotated along the spokes of the pie chart, otherwise the inner labels are not rotated. There are analogous package options rotateinner=datapie and norotateinner=datapie.

rotateouter This is a boolean setting, so it can only take the values `true` and `false`. If the value is omitted `true` is assumed. If true, the outer labels are rotated along the spokes of the pie chart, otherwise the outer labels are not rotated. There are analogous package options rotateouter=datapie and norotateouter=datapie.

Example 23 (A Pie Chart)

This example loads data from a file called `fruit.csv` which contains the following:

```
Name,Quantity
"Apples",30
"Pears",25
"Lemons,Limes",40.5
"Peaches",34.5
"Cherries",20
```

First load the data:

```
\DTLloaddb{fruit}{fruit.csv}
```

Now create a pie chart in a figure:

```
\begin{figure}[htbp]
\centering
\DTLpiechart{variable=\quantity}{fruit}{\name=Name,\quantity=Quantity}
\caption{A pie chart}
\end{figure}
```

This creates Figure 7.1. The colours used are the defaults. See example 27 for an example that changes the default colours.

There are no outer labels by default, but they can be set using the outerlabel setting. The following sets the outer label to the value of the `Name` key:

```
\begin{figure}[htbp]
\centering
\DTLpiechart{variable=\quantity,outerlabel=\name}{fruit}{%
\name=Name,\quantity=Quantity}
\caption{A pie chart (outer labels set)}
\end{figure}
```

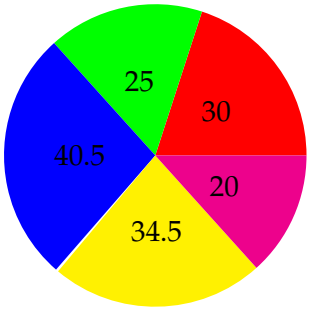

Figure 7.1: A pie chart

This creates Figure 7.2.

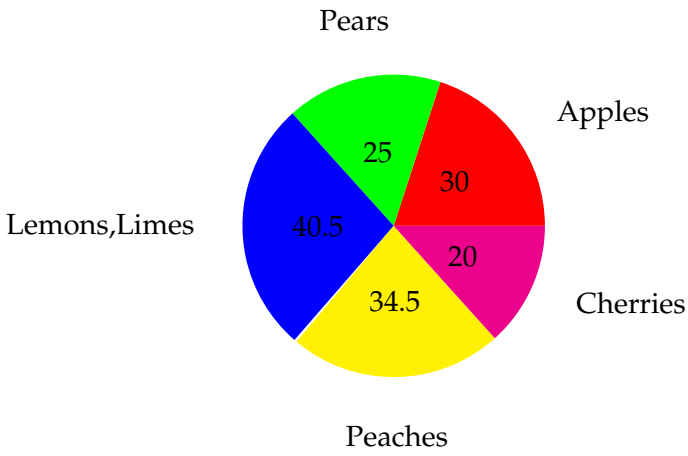

Figure 7.2: A pie chart (outer labels set)

You may prefer the labels to be rotated. The following switches on the rotation for the inner and outer labels:

```
\begin{figure}[htbp]
\centering
\DTLpiechart{variable=\quantity,outerlabel=\name,%
rotateinner,rotateouter}{fruit}{%
\name=Name,\quantity=Quantity}
\caption{A pie chart (rotation enabled)}
\end{figure}
```

This creates Figure 7.3.

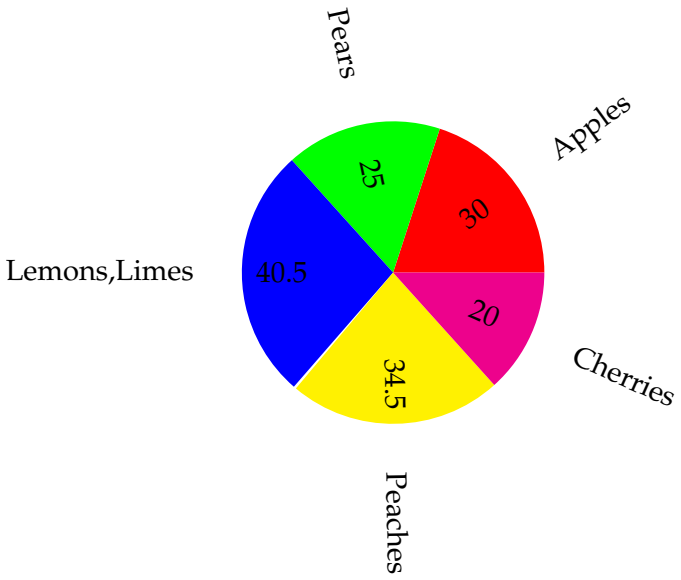

Figure 7.3: A pie chart (rotation enabled)

Example 24 (Separating Segments from the Pie Chart)

You may want to separate one or more segments from the pie chart, perhaps to emphasize them. You can do this using the cutaway setting. The following separates the first and third segments from the pie chart:

```
\begin{figure}[htbp]
\centering
\DTLpiechart{variable=\quantity,outerlabel=\name,%
cutaway={1,3}}{fruit}{%
\name=Name,\quantity=Quantity}
\caption{A pie chart with cutaway segments}
\end{figure}
```

This produces Figure 7.4.

Alternatively I can specify a range of segments. The following separates the first two segments:

```
\begin{figure}[htbp]
\centering
\DTLpiechart{variable=\quantity,outerlabel=\name,%
cutaway={1-2}}{fruit}{%
\name=Name,\quantity=Quantity}
\caption{A pie chart with cutaway segments (\texttt{cutaway=\{1-2\}})}
\end{figure}
```

This produces Figure 7.5.

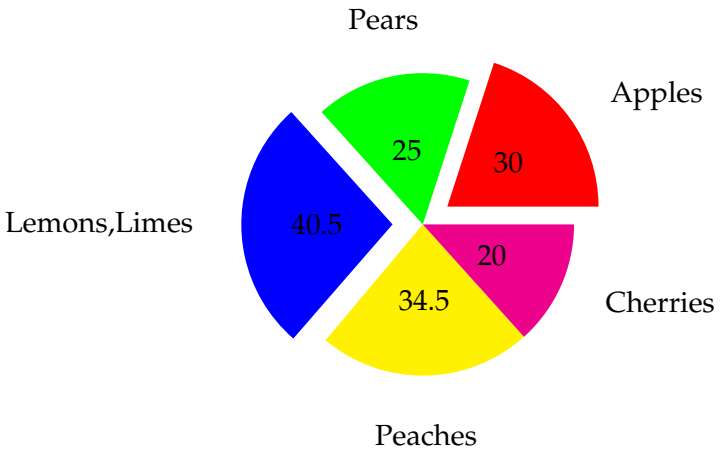

Figure 7.4: A pie chart with cutaway segments

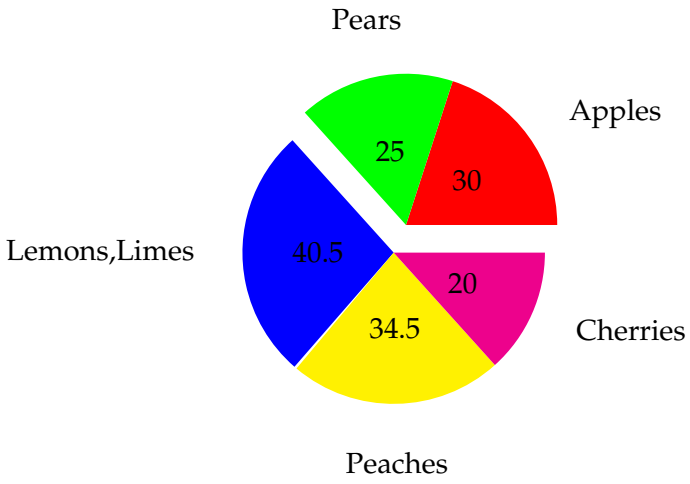

Figure 7.5: A pie chart with cutaway segments (`cutaway={1-2}`)

Notice the difference between Figure 7.5 and Figure 7.6 which was produced using:

```
\begin{figure}[htbp]
\centering
\DTLpiechart{variable=\quantity,outerlabel=\name,%
cutaway={1,2}}{fruit}{%
\name=Name,\quantity=Quantity}
\caption{A pie chart with cutaway segments (\texttt{cutaway=\{1,2\}})}
\end{figure}
```

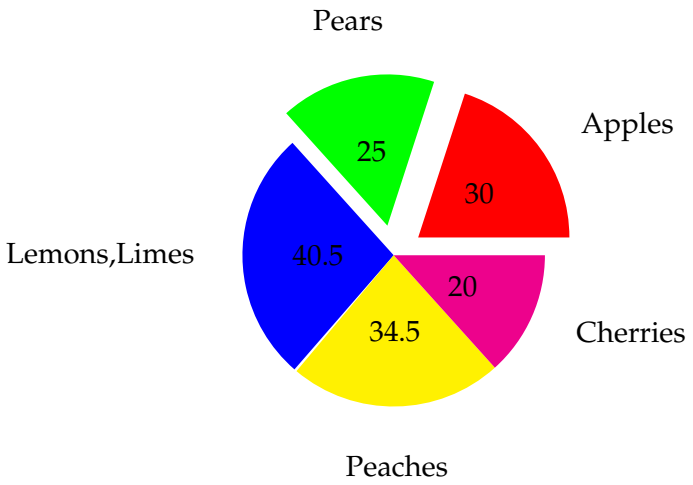

Figure 7.6: A pie chart with cutaway segments (`cutaway={1,2}`)

7.1 Pie Chart Variables

`\DTLpievariable`

```
\DTLpievariable
```

This command is set to the variable given by the variable setting in the ⟨*settings list*⟩ argument of `\DTLpiechart`. The innerlabel is set to `\DTLpievariable` by default.

`\DTLpiepercent`

```
\DTLpiepercent
```

This command is set to the percentage value of `\DTLpievariable`. The percentage value is rounded to ⟨*n*⟩ digits, where ⟨*n*⟩ is the value of the LaTeX counter DTLpieroundvar.

`DTLpieroundvar`

Example 25 (Changing the Inner and Outer Labels)

This example uses the database defined in example 23. The inner label is now set to the percentage value, rather than the actual value, and the outer label is set to the name with the actual value in parentheses.

```
\begin{figure}[htbp]
\centering
\DTLpiechart{variable=\quantity,%
innerlabel={\DTLpiepercent\%},%
outerlabel={\name\ (\DTLpievariable)}}{fruit}{%
\name=Name,\quantity=Quantity}
\caption{A pie chart (changing the labels)}
\end{figure}
```

This produces Figure 7.7.

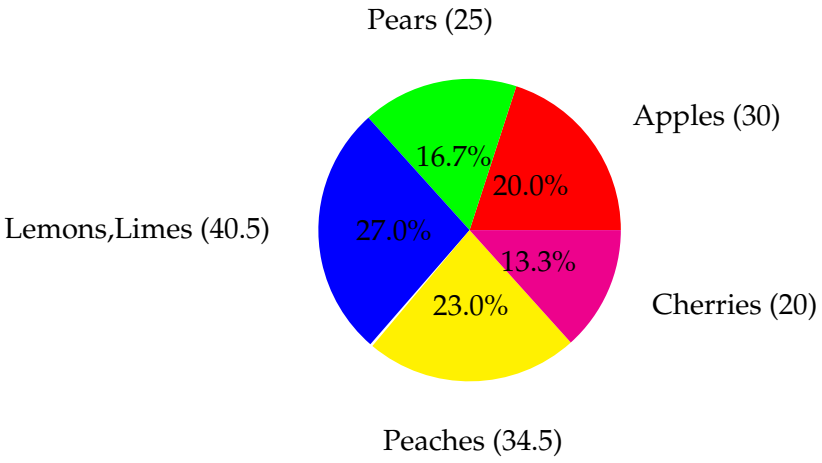

Figure 7.7: A pie chart (changing the labels)

7.2 Pie Chart Label Formatting

`\DTLdisplayinnerlabel`

```
\DTLdisplayinnerlabel{⟨text⟩}
```

This governs how the inner label is formatted, where ⟨*text*⟩ is the text of the inner label. The default is to just do ⟨*text*⟩.

`\DTLdisplayouterlabel`

```
\DTLdisplayouterlabel{⟨text⟩}
```

This governs how the outer label is formatted, where ⟨*text*⟩ is the text of the outer label. The default is to just do ⟨*text*⟩.

Example 26 (Changing the Inner and Outer Label Format)

This example extends example 25. The inner and outer labels are now both typeset in a sans-serif font:

```
\begin{figure}[htbp]
\centering
\renewcommand*{\DTLdisplayinnerlabel}[1]{\textsf{#1}}
\renewcommand*{\DTLdisplayouterlabel}[1]{\textsf{#1}}
\DTLpiechart{variable=\quantity,%
innerlabel={\DTLpiepercent\%},%
outerlabel={\name\ (\DTLpievariable)}}{fruit}{%
\name=Name,\quantity=Quantity}
\caption{A pie chart (changing the label format)}
\end{figure}
```

This produces Figure 7.8.

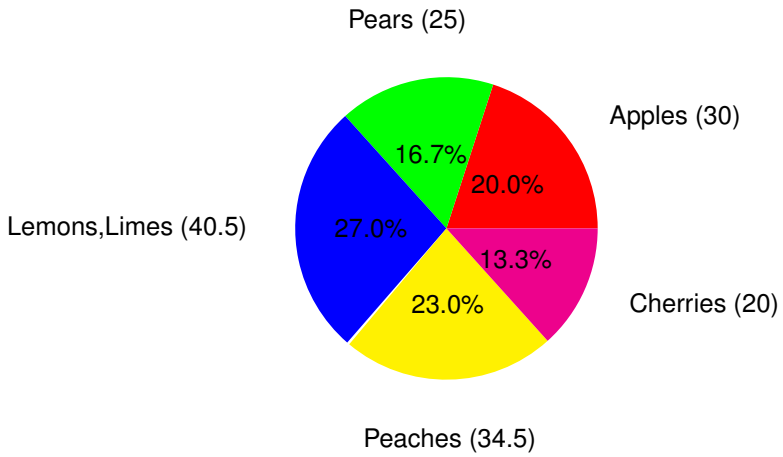

Figure 7.8: A pie chart (changing the label format)

7.3 Pie Chart Colours

The datapie package predefines colours for the first eight segments of the pie chart. If you require more than eight segments or if you want to change the default colours, you will need to use

\DTLsetpiesegmentcolor

```
\DTLsetpiesegmentcolor{⟨n⟩}{⟨color⟩}
```

The first argument ⟨*n*⟩ is the segment index (starting from 1), and the second argument ⟨*color*⟩ is a colour specifier as used in commands such as \color.

It is a good idea to set the colours so that each segment colour is somehow relevant to whatever the segment represents. For example, in the previous examples of pie charts depicting fruit, some of default colours were inappropriate. Whilst red is appropriate for apples and green is appropriate for pears, blue doesn't really correspond to lemons or limes.

\DTLdopiesegmentcolor

```
\DTLdopiesegmentcolor⟨n⟩
```

This sets the current text colour to that of the ⟨*n*⟩th segment.

\DTLdocurrentpiesegmentcolor

```
\DTLdocurrentpiesegmentcolor
```

This sets the current text colour to that of the current pie segment. This command may only be used within a pie chart, or within the body of \DTLforeach.

\DTLpieoutlinecolor

```
\DTLpieoutlinecolor
```

This sets the outline colour for the pie chart. The default is black.

\DTLpieoutlinewidth

```
\DTLpieoutlinewidth
```

This is a length that governs the line width of the outline. The default value is 0pt, but can be changed using \setlength. The outline is only drawn if \DTLpieoutlinewidth is greater than 0pt.

Example 27 (Pie Segment Colours)

This example extends example 26. It sets the outline thickness to 2pt, and the outer label is now set in the same colour as the fill colour of the segment to which it belongs. The third segment (lemons and limes) is set to yellow and the fourth segment (peaches) is set to pink. In addition, a legend is created using \DTLforeach.

```
\begin{figure}[htbp]
\centering
\setlength{\DTLpieoutlinewidth}{2pt}
\DTLsetpiesegmentcolor{3}{yellow}
\DTLsetpiesegmentcolor{4}{pink}
\renewcommand*{\DTLdisplayinnerlabel}[1]{\textsf{#1}}
\renewcommand*{\DTLdisplayouterlabel}[1]{%
\DTLdocurrentpiesegmentcolor
\textsf{\shortstack{#1}}}
\DTLpiechart{variable=\quantity,%
innerlabel={\DTLpiepercent\%},%
outerlabel={\name\\(\DTLpievariable)}}{fruit}{%
\name=Name,\quantity=Quantity}
\begin{tabular}[b]{ll}
\DTLforeach{fruit}{\name=Name}{\DTLiffirstrow{}{\\}%
\DTLdocurrentpiesegmentcolor\rule{10pt}{10pt} &
\name
}
\end{tabular}
\caption{A pie chart (using segment colours and outline)}
\end{figure}
```

This produces Figure 7.9. (The format of the outer label has been changed to use `\shortstack` to prevent the outer labels from taking up so much horizontal space. The outerlabel setting has also been modified to use `\\` after the name to move the percentage value onto the next row.)

7.4 Adding Extra Commands Before and After the Pie Chart

The pie charts created using `\DTLpiechart` are placed inside a tikzpicture environment (defined by the tikz package).

`\DTLpieatbegintikz`

```
\DTLpieatbegintikz
```

The macro `\DTLpieatbegintikz` is called at the start of the tikzpicture environment, allowing you to change the tikzpicture settings. By default `\DTLpieatbegintikz` does nothing, but you can redefine it to, say, scale the pie chart (but be careful not to distort the chart).

`\DTLpieatendtikz`

```
\DTLpieatendtikz
```

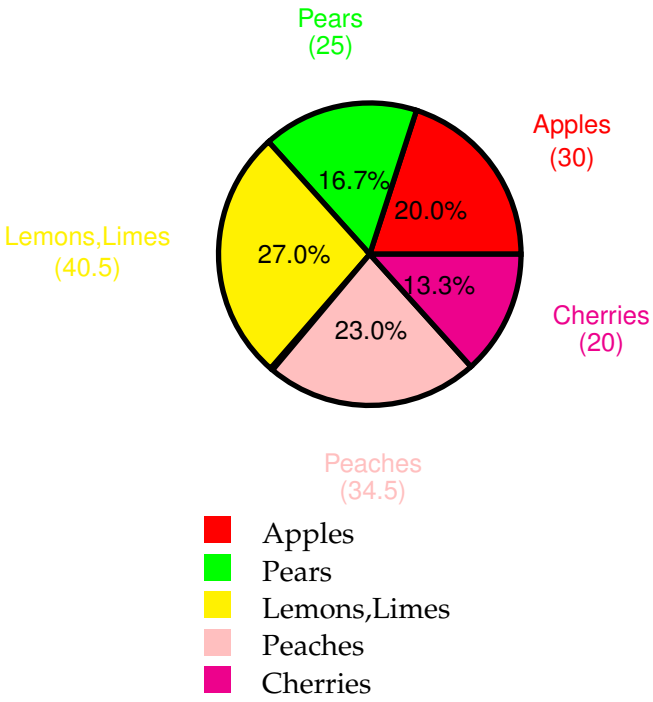

Figure 7.9: A pie chart (using segment colours and outline)

The macro `\DTLpieatendtikz` is called at the end of the tikzpicture environment, allowing you add additional graphics to the pie chart. This does nothing by default.

Example 28 (Adding Information to the Pie Chart)

This example modifies example 23. It redefines `\DTLpieatendtikz` to add an annotated arrow.

```
\begin{figure}[htbp]
\centering
\renewcommand*{\DTLpieatendtikz}{%
\draw[<-] (45:1.5cm) -- (40:2.5cm)node[right]{Apples};}
\DTLpiechart{variable=\quantity}{fruit}{%
\name=Name,\quantity=Quantity}
\caption{An annotated pie chart}
\end{figure}
```

This produces Figure 7.10. (Note that the centre of the pie chart is the origin of the TikZ picture.)

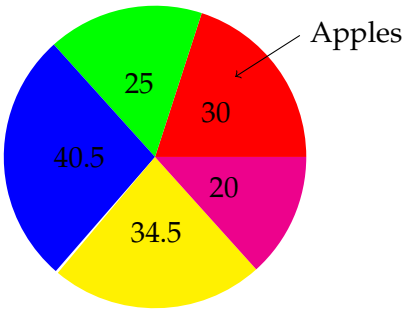

Figure 7.10: An annotated pie chart

8 Scatter and Line Plots (dataplot package)

The dataplot package provides commands for creating scatter or line plots from databases. It uses the pgf/TikZ plot handler library to create the plots. See the pgf manual for more detail on pgf streams and plot handles. The dataplot package is not loaded by datatool so if you want to use it you need to load it explicitly using `\usepackage{dataplot}`.

`\DTLplot`

`\DTLplot[`⟨*condition*⟩`]{`⟨*db list*⟩`}{`⟨*settings*⟩`}`

This command creates a plot (inside a tikzpicture environment) of all the data given in the databases listed in ⟨*db list*⟩, which should be a comma separated list of database names. The optional argument ⟨*condition*⟩ is the same as that for `\DTLforeach`. The ⟨*settings*⟩ argument is a comma separated list of ⟨*setting*⟩=⟨*value*⟩ pairs. There are two settings that must be specified x and y. The other settings are optional. Note that any value that contains a comma, must be enclosed in braces. For example `colors={red,cyan,blue}`. Note where any setting requires a number, or list of numbers (such as bounds) the number must be supplied in standard decimal notation (i.e. no currency, no number groups, and a full stop as the decimal point). Available settings are as follows:

x The database key that specifies the *x* co-ordinates. This setting is required.

y The database key that specifies the *y* co-ordinates. This setting is required.

markcolors A comma separated list of colour names for the markers. An empty value will use the current colour.

linecolors A comma separated list of colour names for the plot lines. An empty value will use the current colour.

colors A comma separated list of colour names for the lines and markers.

marks A comma separated list of code to generate plot marks. (This should typically be a list of `\pgfuseplotmark` commands, see the

pgf manual for further details.) You may use `\relax` as an element of the list to suppress markers for the corresponding plot. For example: `marks={\pgfuseplotmark{o},\relax}` will use an open circle marker for the first database, and no markers for the second database listed in ⟨*db list*⟩.

lines A comma separated list of line style settings. (This should typically be a list of `\pgfsetdash` commands, see the pgf manual for further details on how to set the line style.) An empty value will use the current line style. You may use `\relax` as an element of the list to suppress line for the corresponding plot. For example: `lines={\relax,\pgfsetdash{}{0pt}}` will have no lines for the first database, and a solid line for the second database listed in ⟨*db list*⟩.

width The width of the plot. This must be a length. The plot width does not include outer tick marks or labels.

height The height of the plot. This must be a length. The plot height does not include outer tick marks or labels.

style This setting governs whether to use lines or markers in the plot, and may take one of the following values: `both` (lines and markers), `lines` (only lines) or `markers` (only markers). The default is `markers`.

axes This setting governs whether to display the axes, and may take one of the following values: `both`, `x`, `y` or `none`. If no value is specified, `both` is assumed.

box This setting governs whether or not to surround the plot in a box. It is a boolean setting, taking only the values `true` and `false`. If no value is specified, `true` is assumed.

xtics This setting governs whether or not to display the *x* tick marks. It is a boolean setting, taking only the values `true` and `false`. If no value is specified `true` is assumed. If the axes setting is set to `both` or `x`, this value will automatically be set to `true`, otherwise it will be set to `false`.

ytics This setting governs whether or not to display the *y* ticks. It is a boolean setting, taking only the values `true` and `false`. If no value is specified `true` is assumed. If the axes setting is set to `both` or `y`, this value will automatically be set to `true`, otherwise it will be set to `false`.

xminortics This setting governs whether or not to display the x minor tick marks. It is a boolean setting, taking only the values `true` and `false`. If no value is specified `true` is assumed. This setting also sets the x major tick marks on if the value is `true`.

yminortics This setting governs whether or not to display the y minor tick marks. It is a boolean setting, taking only the values `true` and `false`. If no value is specified `true` is assumed. This setting also sets the y major tick marks on if the value is `true`.

xticdir This sets the x tick direction, and may only take the values `in` or `out`.

yticdir This sets the y tick direction, and may only take the values `in` or `out`.

ticdir This sets the x and y tick direction, and may only take the values `in` or `out`.

bounds The value must be in the form $\langle min\ x\rangle$, $\langle min\ y\rangle$, $\langle max\ x\rangle$, $\langle max\ y\rangle$. This sets the graph bounds to the given values. If omitted the bounds are computed from the maximum and minimum values of the data. For example

```
\DTLplot{data1,data2}{x=Height,y=Weight,bounds={0,0,10,20}}
```

Note that the bounds setting overrides the minx, maxx, miny and maxy settings.

minx The value is the minimum value of the x axis.

miny The value is the minimum value of the y axis.

maxx The value is the maximum value of the x axis.

maxy The value is the maximum value of the y axis.

xticpoints The value must be a comma separated list of decimal numbers indicating where to put the x tick marks. If omitted, the x tick marks are placed at equal intervals along the x axis such that each interval is not less than the length given by `\DTLmintickgap`. This setting overrides xticgap.

xticgap This value specifies the gap between the x tick marks.

yticpoints The value must be a comma separated list of decimal numbers indicating where to put the y tick marks. If omitted, the y tick marks are placed at equal intervals along the y axis such that each interval is not less than the length given by `\DTLmintickgap`. This setting overrides yticgap.

yticgap This value specifies the gap between the y tick marks.

grid This is a boolean value that specifies whether or not to display the grid. If no value is given, `true` is assumed. The minor grid lines are only displayed if the minor tick marks are set.

xticlabels The value must be a comma separated list of labels for each x tick mark. If omitted, the labels are the value of the x tick position, rounded $\langle n \rangle$ digits after the decimal point, where $\langle n \rangle$ is given by the value of the counter DTLplotroundXvar.

yticlabels The value must be a comma separated list of labels for each y tick mark. If omitted, the labels are the value of the y tick position, rounded $\langle n \rangle$ digits after the decimal point, where $\langle n \rangle$ is given by the value of the counter DTLplotroundYvar.

xlabel The value is the label for the x axis. If omitted, the axis has no label.

ylabel The value is the label for the y axis. If omitted, the axis has no label.

legend This setting governs whether or not to display the legend, and where it should be displayed. It may take one of the following values `none` (don't display the legend), `north`, `northeast`, `east`, `southeast`, `south`, `southwest`, `west` or `northwest`. If the value is omitted, `northeast` is assumed.

legendlabels The value must be a comma separated list of labels for the legend. If omitted, the database names are used.

Example 29 (A Basic Graph)

Suppose you have a file called `groupa.csv` that contains the following:

```
Height,Weight
1.54,48.0
1.55,45.4
1.56,58.0
1.56,50.2
1.57,46.0
1.58,48.3
1.59,56.5
1.59,58.1
1.60,60.9
1.62,56.3
```

First load this into a database called `groupa`:

```
\DTLloaddb{groupa}{groupa.csv}
```

The data can now be converted into a scatter plot as follows:

```
\begin{figure}[htbp]
\centering
\DTLplot{groupa}{x=Height,y=Weight}
\caption{A scatter plot}
\end{figure}
```

This produces Figure 8.1.

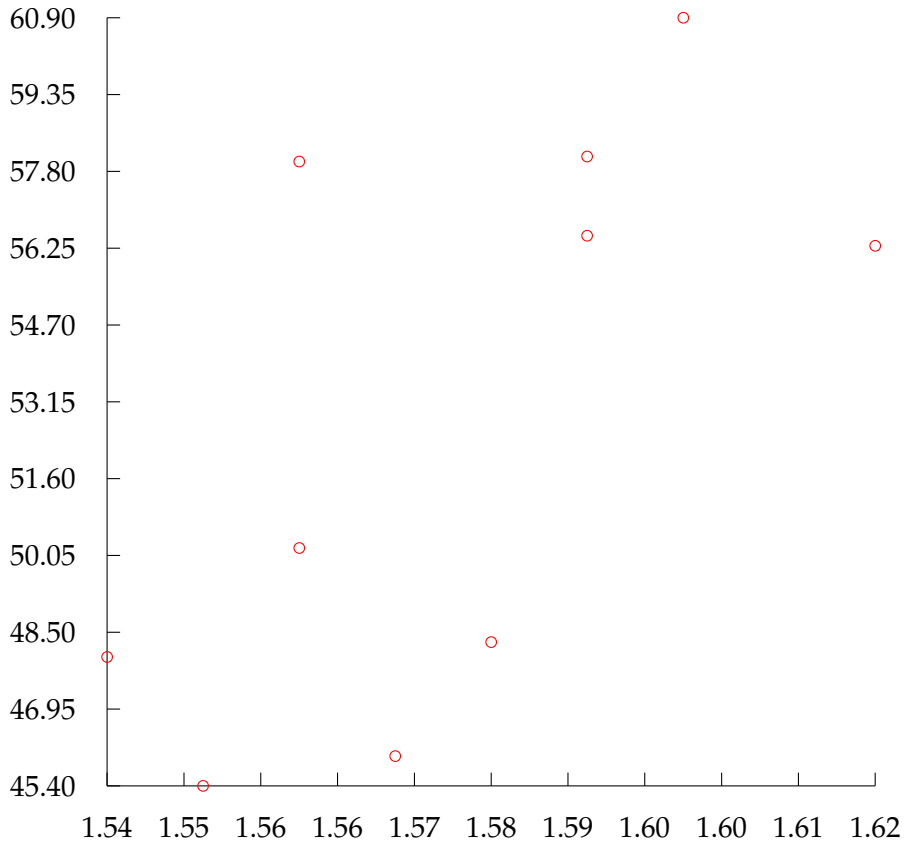

Figure 8.1: A scatter plot

Alternatively, you can use the style setting to change it into a line plot:

```
\begin{figure}[htbp]
\centering
\DTLplot{groupa}{x=Height,y=Weight,style=lines}
\caption{A line plot}
\end{figure}
```

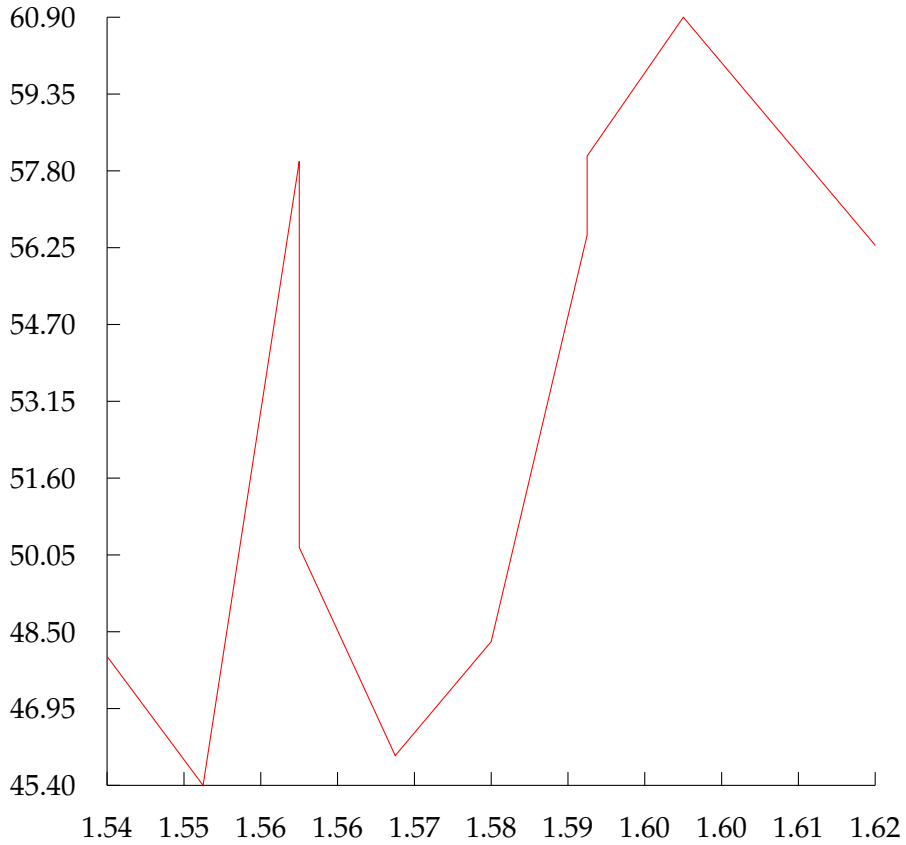

Figure 8.2: A line plot

This produces Figure 8.2.

Example 30 (Plotting Multiple Data Sets)

In this example, I shall use the database called `groupa` defined in example 29, and another database called `groupb` which is loaded from the file `groupb.csv` which contains the following:

```
Height,Weight
1.54,48.4
1.54,42.0
1.55,64.0
1.56,58.2
1.56,49.0
1.57,40.3
1.58,51.5
1.58,63.1
1.59,74.9
1.59,59.3
```

First load this into a database called `groupb`:

```
\DTLloaddb{groupb}{groupb.csv}
```

I can now plot both groups in the same graph, but I want a smaller graph than Figure 8.1 and Figure 8.2, so I am going to set the plot width and height to 3in:

```
\begin{figure}[htbp]
\centering
\DTLplot{groupa,groupb}{x=Height,y=Weight,width=3in,height=3in}
\caption{A scatter plot}
\end{figure}
```

This produces Figure 8.3.

Now let's add a legend using the legend setting, with the legend labels `Group A` and `Group B`, and set the x tick intervals using xticpoints setting. I am also going to set the x axis label to `Height (m)` and the y axis label to `Weight (kg)`, and place a box around the plot.

```
\begin{figure}[htbp]
\centering
\DTLplot{groupa,groupb}{x=Height,y=Weight,
width=3in,height=3in,legend,legendlabels={Group A,Group B},
xlabel={Height (m)},ylabel={Weight (kg)},box,
xticpoints={1.54,1.55,1.56,1.57,1.58,1.59,1.60,1.61,1.62}}
\caption{A scatter plot}
\end{figure}
```

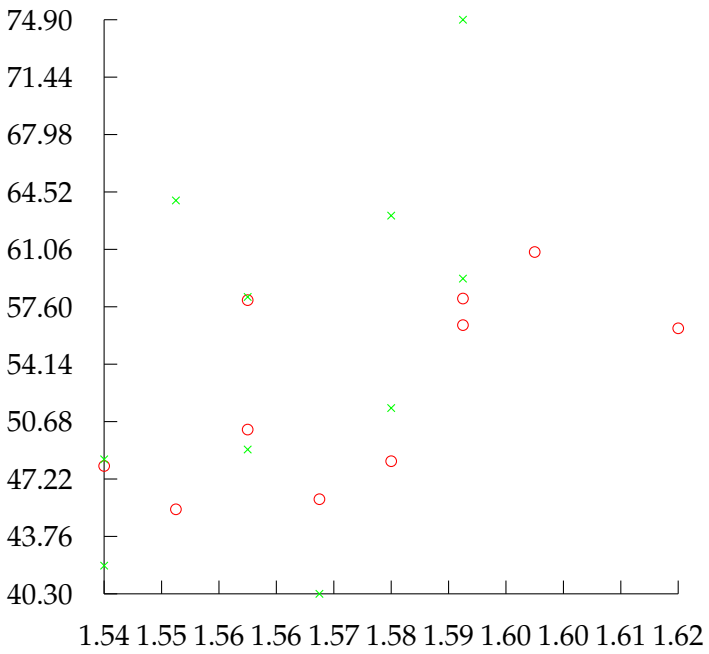

Figure 8.3: A scatter plot

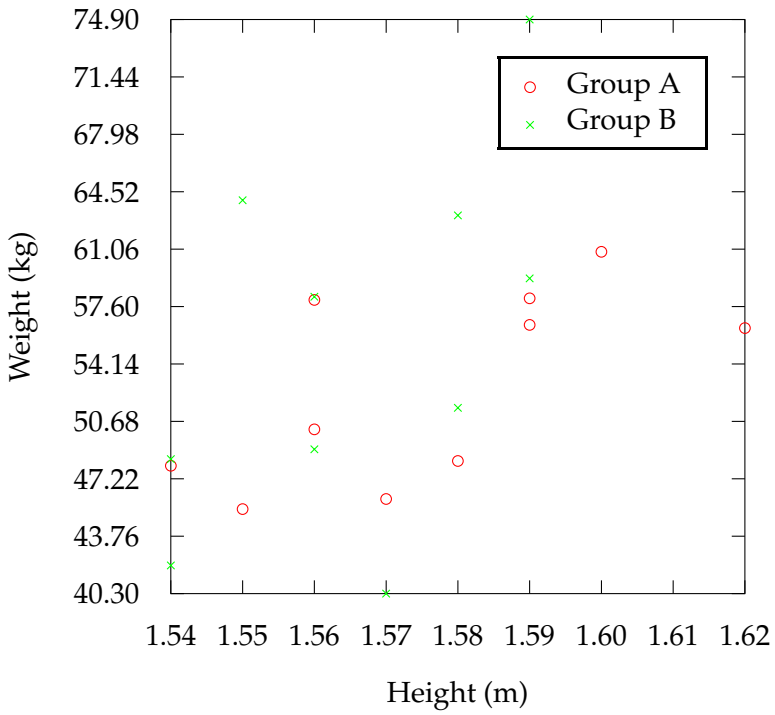

Figure 8.4: A scatter plot

This produces Figure 8.4.

8.1 Adding Information to the Plot

The datatool package provides two hooks used at the beginning and end of the tikzpicture environment:

`\DTLplotatbegintikz`

```
\DTLplotatbegintikz
```

and

`\DTLplotatendtikz`

```
\DTLplotatendtikz
```

They are both defined to do nothing by default, but can be redefined to add commands to the image. The unit vectors are set prior to using these hooks, so you can use the same co-ordinates as those in the data sets. However, to reduce the problem of exceeding TEX's maximum dimension, `\DTLplot` scales the plot which may distort plot marks. To get around this use

`\dtlplothandlermark`

```
\dtlplothandlermark{⟨pgf code⟩}
```

instead of `\pgfplothandlermark{`⟨*pgf code*⟩`}`. (See example 33.) Note that `\dtlplothandlermark` is only intended for use within the definition of `\DTLplotatbegintikz` or `\DTLplotatendtikz`. If used elsewhere it will produce a warning and act as though you'd just used `\pgfplothandlermark`.

`\DTLaddtoplotlegend`

```
\DTLaddtoplotlegend{⟨marker⟩}{⟨line style⟩}{⟨text⟩}
```

This adds a new row to the plot legend where ⟨*marker*⟩ is code to produce the marker, ⟨*line style*⟩ is code to set the line style and ⟨*text*⟩ is a textual label. You can use `\relax` to suppress the marker or line. For example:

```
\DTLaddtoplotlegend{\pgfuseplotmark{x}}{\relax}{Some Data}
```

Note that the legend is plotted before `\DTLplotatendtikz`, so if you want to add information to the legend you will need to do the in `\DTLplotatstarttikz`.

Example 31 (Adding Information to a Plot)

Returning to the plots created in example 30, suppose I now want to annotate the plot, say I want to draw your notice to a particular point, say the point (1.58,48.3), then I can redefine `\DTLplotatendtikz` to draw an annotated arrow to that point:

```
\renewcommand*{\DTLplotatendtikz}{%
\draw[<-,line width=1pt] (1.58,48.3) -- (1.6,43)
 node[below]{interesting point};
}
```

So Figure 8.4 now looks like Figure 8.5. (Obviously, `\DTLplotatendtikz` needs to be redefined before using `\DTLplot`.)

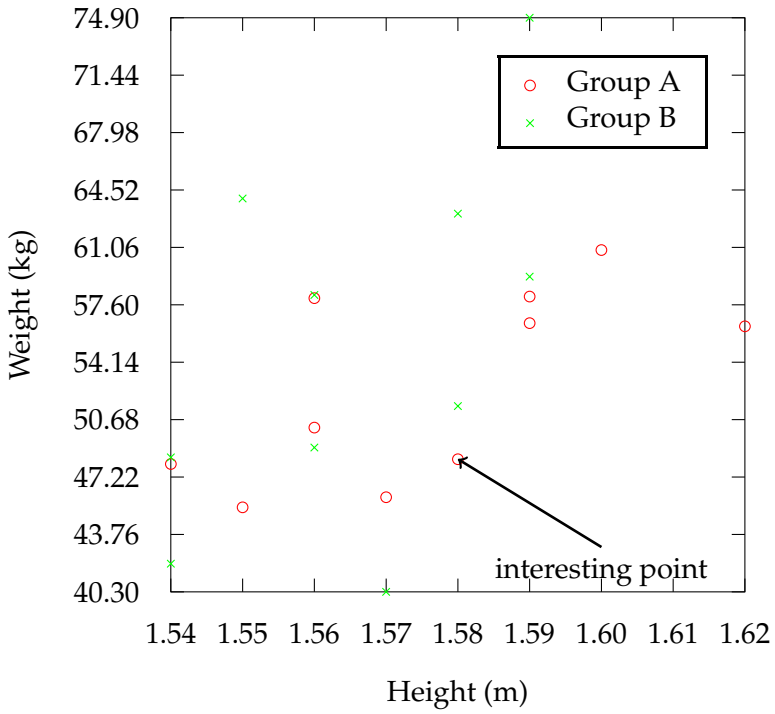

Figure 8.5: A scatter plot

8.2 Global Plot Settings

8.2.1 Lengths

This section describes the lengths that govern the appearance of the plot created using `\DTLplot`. These lengths can be changed using

\setlength.

\DTLplotwidth

\DTLplotwidth

This length governs the length of the x axis. Note that the plot width does not include any outer tick marks or labels. The default value is 4in.

\DTLplotheight

\DTLplotheight

This length governs the length of the y axis. Note that the plot height does not include any outer tick marks or labels. The default value is 4in

\DTLticklength

\DTLticklength

This governs the length of the tick marks. The default value is 5pt.

\DTLminorticklength

\DTLminorticklength

This governs the length of the minor tick marks. The default value is 2pt.

\DTLticklabeloffset

\DTLticklabeloffset

This governs the distance from the axis to the tick labels. The default value is 8pt.

\DTLmintickgap

\DTLmintickgap

This is the minimum distance allowed between tick marks. If the plot width or height is less than this distance there will only be tick marks at either end of the axis. The default value is 20pt.

\DTLlegendxoffset

\DTLlegendxoffset

This is the horizontal distance from the border of the plot to the outer border of the legend. The default value is 10pt.

\DTLlegendyoffset

\DTLlegendyoffset

This is the vertical distance from the border of the plot to the outer border of the legend. The default value is 10pt.

8.2.2 Counters

These counters govern the appearance of plots created using `\DTLplot`. The value of the counters can be changed using `\setcounter`.

`DTLplotroundXvar`

DTLplotroundXvar

Unless you specify your own tick labels, the x tick labels will be given by the tick points rounded to $\langle n \rangle$ digits after the decimal point, where $\langle n \rangle$ is the value of the counter DTLplotroundXvar.

`DTLplotroundYvar`

DTLplotroundYvar

Unless you specify your own tick labels, the y tick labels will be given by the tick points rounded to $\langle n \rangle$ digits after the decimal point, where $\langle n \rangle$ is the value of the counter DTLplotroundYvar.

8.2.3 Macros

These macros govern the appearance of plots created using `\DTLplot`. They can be changed using `\renewcommand`.

`\DTLplotmarks`

`\DTLplotmarks`

This must be a comma separated list of pgf code to create the plot marks. `\DTLplot` cycles through this list for each database listed. The pgf package provides convenient commands for generating plots using `\pgfuseplotmark`. See the pgf manual for more details.

`\DTLplotmarkcolors`

`\DTLplotmarkcolors`

This must be a comma separated list of defined colours to apply to the plot marks. `\DTLplot` cycles through this list for each database listed. If this macro is set to empty, the current colour will be used instead.

`\DTLplotlines`

`\DTLplotlines`

This must be a comma separated list of pgf code to set the style of the plot lines. \DTLplot cycles through this list for each database listed. Dash patterns can be set using \pgfsetdash, see the pgf manual for more details. If \DTLplotlines is set to empty the current line style will be used instead.

\DTLplotlinecolors

\DTLplotlinecolors

This must be a comma separated list of defined colours to apply to the plot lines. \DTLplot cycles through this list for each database listed. If this macro is set to empty, the current colour will be used instead. The default is the same as \DTLplotmarkcolors.

\DTLXAxisStyle

\DTLXAxisStyle

This governs the style of the *x* axis. It is passed as the optional argument to the TikZ \draw command. By default it is just - which is a solid line style with no start or end arrows. The *x* axis line starts from the bottom left corner of the plot and extends to the bottom right corner of the plot. So if you want the *x* axis to have an arrow head at the right end, you can do:

```
\renewcommand*{\DTLXAxisStyle}{->}
```

\DTLYAxisStyle

\DTLYAxisStyle

This governs the style of the *y* axis. It is analogous to \DTLXAxisStyle described above.

\DTLmajorgridstyle

\DTLmajorgridstyle

This specifies the format of the major grid lines. It may be set to any TikZ setting that you can pass to the optional argument of \draw. The default value is color=gray, - which indicates a grey solid line.

\DTLminorgridstyle

\DTLminorgridstyle

This specifies the format of the minor grid lines. It may be set to any TikZ setting that you can pass to the optional argument of \draw. The default

value is `color=gray,loosely dotted` which indicates a grey dotted line.

`\DTLformatlegend`

`\DTLformatlegend{`⟨*legend*⟩`}`

This formats the entire legend, which is passed as the argument. The default is to set the legend with a white background, a black frame.

8.3 Adding to a Plot Stream

`\DTLplotstream`

`\DTLplotstream[`⟨*condition*⟩`]{`⟨*db name*⟩`}{`⟨*x key*⟩`}{`⟨*y key*⟩`}`

This adds points to a stream from the database called ⟨*db name*⟩ where the *x* co-ordinates are given by the key ⟨*x key*⟩ and the *y* co-ordinates are given by the key ⟨*y key*⟩. (`\DTLconverttodecimal` is used to convert locale dependent values to a standard decimal that is recognised by the pgf package.) The optional argument ⟨*condition*⟩ is the same as that for `\DTLforeach`.

Example 32 (Adding to a Plot Stream)

Suppose you have a CSV file called `data.csv` containing the following:

```
x,y
0,0
1,1
2,0.5
1.5,0.3
```

First load the file into a database called `data`:

```
\DTLloaddb{data}{data.csv}
```

Now create a figure containing this data:

```
\begin{figure}[tbhp]
\centering
\begin{tikzpicture}
\pgfplothandlermark{\pgfuseplotmark{o}}
\pgfplotstreamstart
\DTLplotstream{data}{x}{y}%
\pgfplotstreamend
\pgfusepath{stroke}
```

```
\end{tikzpicture}
\caption{Adding to a plot stream}
\end{figure}
```

This produces Figure 8.6.

Figure 8.6: Adding to a plot stream

Example 33 (Plotting Multiple Keys in the Same Database)

Suppose I have conducted two time to growth experiments. For each experiment, I have recorded the log count at set times, and I have recorded this information in the same data file called, say, `growth.csv` which contains the following:

```
Time,Experiment 1,Experiment 2
0,3.73,3.6
23,3.67,3.7
60,4.9,3.8
```

I can load the data into a database using:

```
\DTLloaddb{growth}{growth.csv}
```

However, I'd like to plot both results on the same graph. Since they are contained in the same database, I can't use the method I used in example 30. Instead I can use a combination of `\DTLplot` and `\DTLplotstream`:

```
\begin{figure}[tbhp]
\centering
 % compute bounds
\DTLminforkeys{growth}{Time}{\minX}
\DTLminforkeys{growth}{Experiment 1,Experiment 2}{\minY}
\DTLmaxforkeys{growth}{Time}{\maxX}
\DTLmaxforkeys{growth}{Experiment 1,Experiment 2}{\maxY}
 % round x tick labels to 1 d.p.
\setcounter{DTLplotroundXvar}{1}
 % redefine \DTLplotatbegintikz to plot the data for Experiment 1
\renewcommand*{\DTLplotatbegintikz}{%
  % set plot mark
  \dtlplothandlermark{\color{green}\pgfuseplotmark{x}}
```

```
  % start plot stream
  \pgfplotstreamstart
  % add data from Experiment 1 to plot stream
  \DTLplotstream{growth}{Time}{Experiment 1}%
  % end plot stream
  \pgfplotstreamend
  % stroke path
  \pgfusepath{stroke}
  % add information to legend (no line is require so use \relax)
  \DTLaddtoplotlegend{\color{green}%
  \pgfuseplotmark{x}}{\relax}{Experiment 1}
}
 % now plot the data for Experiment 2
\DTLplot{growth}{x=Time,y=Experiment 2,legend,
width=3in,height=3in,bounds={\minX,\minY,\maxX,\maxY},
xlabel={Time},ylabel={Log Count},
legendlabels={Experiment 2}}
\caption{Time to growth data}
\end{figure}
```

This produces Figure 8.7. Notes:

- I redefined `\DTLplotatbegintikz` in order to add the new plot to the legend, since `\DTLplotatendtikz` is used after the legend is plotted. The x and y unit vectors are set before `\DTLplotatbegintikz` so I don't need to worry about the co-ordinates, however I've had to use `\dtlplothandlermark` instead of `\pgfplothandlermark` to prevent the plot marks from being distorted.
- I have used `\DTLminforkeys` and `\DTLmaxforkeys` to determine the bounds since `\DTLplot` won't take the data for Experiment 1 into account when computing the bounds.

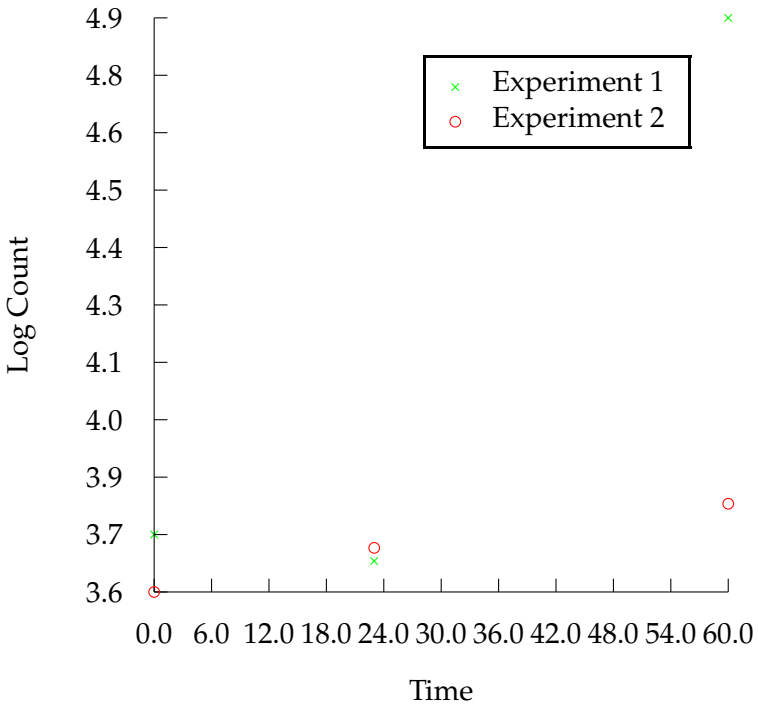

Figure 8.7: Time to growth data

9 Bar Charts (databar package)

The databar package provides commands for creating bar charts. It is not loaded by the datatool package, so if you want to use it you will need to load it explicitly using `\usepackage{databar}`. You must also have the pgf package installed.

Bar charts can either be vertical or horizontal, the default is vertical. In this section the x axis refers to the horizontal axis when plotting a vertical bar chart and to the vertical axis when plotting a horizontal bar chart. The x axis units are in increments of one bar. The y axis refers to the vertical axis when plotting a vertical bar chart and to the horizontal axis when plotting a horizontal bar chart. The y axis uses the same co-ordinates as the data. The bars may have an upper and lower label. In a vertical bar chart, the lower label is placed below the x axis and the upper label is placed above the top of the bar. In a horizontal bar chart, the lower label is placed to the left of the x axis and the upper label is placed to the right of the end of the bar. (This is actually a misnomer as it is possible for the "upper" label to be below the "lower" label if a bar has a negative value, however the bars are considered to be anchored on the x axis, and the other end of the bar is considered to be the "upper" end, regardless of its direction.)

The databar package options are as follows:

color=databar Created coloured bar charts (default).

gray=databar Created grey scale bar charts.

vertical=databar Created vertical bar charts (default).

horizontal=databar Created horizontal bar charts.

`\DTLbarchart`

`\DTLbarchart[`⟨*condition*⟩`]{`⟨*settings*⟩`}{`⟨*db name*⟩`}{`⟨*values*⟩`}`

`\DTLmultibarchart`

`\DTLmultibarchart[`⟨*condition*⟩`]{`⟨*settings*⟩`}{`⟨*db name*⟩`}{`⟨*values*⟩`}`

These commands both create a bar chart from the information in the database ⟨*db name*⟩, where ⟨*condition*⟩ is the same as the optional

argument for \DTLforeach described in section 5.4, and ⟨*values*⟩ is the same as the penultimate argument of \DTLforeach. The ⟨*settings*⟩ argument is a ⟨*setting*⟩=⟨*value*⟩ list of settings. The first command, \DLTbarchart, will draw a bar chart for a given column of data in the database, whereas the second command, \DTLmultibarchart, will draw a bar chart that is divided into groups of bars where each bar within a group represents data from several columns of a given row in the database.

The variable setting is required for \DTLbarchart and the variables, the other settings are optional (though some may only be used for one of \DTLbarchart and \DLTmultibarchart), and are as follows:

variable This specifies the control sequence to use that contains the value used to construct the bar chart. The control sequence must be one of the control sequences to appear in the assignment list ⟨*values*⟩. This setting is required for \DTLbarchart, and is unavailable for \DTLmultibarchart.

variables This specifies a list of control sequences to use which contain the values used to construct the bar chart. Each control sequence must be one of the control sequences to appear in the assignment list ⟨*values*⟩. This setting is required for \DTLmultibarchart, and is unavailable for \DTLbarchart.

max This specifies the maximum value on the *y* axis. (This should be a standard decimal value.)

length This specifies the overall length of the *y* axis, and must be a dimension.

maxdepth This must be a zero or negative number. It specifies the maximum depth of the *y* axis. (This should be a standard decimal value.)

axes This setting specifies which axes to display. This may take one of the following values: both, x, y or none.

barlabel This setting specifies the lower bar label. When used with \DTLmultibarchart it indicates the group label.

multibarlabels This setting should contain a comma separated list of labels for each bar within a group for \DTLmultibarchart. This setting is not available for \DTLbarchart.

upperbarlabel This setting specifies the upper bar label. This setting is not available for \DTLmultibarchart.

uppermultibarlabels This setting must be a comma separated list of upper bar labels for each bar within a group. This setting is not available for `\DTLbarchart`.

yticpoints This must be a comma separated list of tick locations for the y axis. (These should be standard decimal values.) This setting overrides yticgap.

yticgap This specifies the gap between the y tick marks. (This should be a standard decimal value.)

yticlabels This must be a comma separated list of tick labels for the y axis.

ylabel This specifies the label for the y axis.

groupgap This specifies the gap between groups when using `\DTLmultibarchart`. This value is given as a multiple of the bar width. The default value is 1, which indicates a gap of one bar width. This setting is not available for `\DTLbarchart`.

verticalbars This is a boolean setting, so it can only take the values `true` (do a vertical bar chart) or `false` (do a horizontal bar chart). If the value is omitted, `true` is assumed.

Example 34 (A Basic Bar Chart)

Recall example 23 defined a database called `fruit`. This example will be using that database to plot a bar chart. The following plots a basic vertical bar chart:

```
\begin{figure}[htbp]
\centering
\DTLbarchart{variable=\theQuantity}{fruit}{\theQuantity=Quantity}
\caption{A basic bar chart}
\end{figure}
```

This produces Figure 9.1.

9.1 Changing the Appearance of a Bar Chart

`\DTLbarchartlength`

`\DTLbarchartlength`

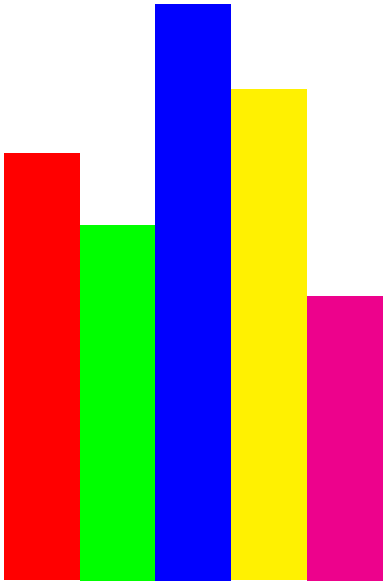

Figure 9.1: A basic bar chart

This specifies the total length of the y axis. You must use `\setlength` to change this value. The default value is 3in.

`\DTLbarwidth`

`\DTLbarwidth`

This specifies the width of each bar. You must use `\setlength` to change this value. The default value is 1cm.

`\DTLbarlabeloffset`

`\DTLbarlabeloffset`

This specifies the distance from the x axis to the lower bar label. You must use `\setlength` to change this value. The default value is 10pt.

`DTLbarroundvar`

DTLbarroundvar

The y tick labels are rounded to $\langle n \rangle$ digits after the decimal point, where $\langle n \rangle$ is given by the value of the counter DTLbarroundvar. You must use `\setcounter` to change this value.

`\DTLsetbarcolor`

```
\DTLsetbarcolor{⟨n⟩}{⟨color⟩}
```

This sets the ⟨*n*⟩th bar colour to ⟨*color*⟩. Only the first eight bars have a colour defined by default. If you need more than eight bars, you will need to define more bar colours. It is recommended that you set the colour of each bar to correspond with whatever the bar represents.

`\DTLdobarcolor`

```
\DTLdobarcolor{⟨n⟩}
```

This sets the current colour to the colour of the ⟨*n*⟩th bar.

`\DTLbaroutlinecolor`

```
\DTLbaroutlinecolor
```

This macro contains the colour of the bar outlines. This defaults to `black`.

`\DTLbaroutlinewidth`

```
\DTLbaroutlinewidth
```

This length specifies the line width for the bar outlines. If it is 0pt, the outline is not drawn. The default value is 0pt.

`\DTLbaratbegintikz`

```
\DTLbaratbegintikz
```

This specifies any additional commands to add to the start of the plot. It defaults to nothing, and is called after the unit vectors are set.

`\DTLbaratendtikz`

```
\DTLbaratendtikz
```

This specifies any additional commands to add to the end of the plot. It defaults to nothing.

`\DTLeverybarhook`

```
\DTLeverybarhook
```

`\DTLstartpt` `\DTLmidpt` `\DTLendpt`

The specifies code to apply at every bar. Within the definition of `\DTLeverybarhook` you can use the commands `\DTLstartpt` (the start of the bar), `\DTLmidpt` (the mid point of the bar) and `\DTLendpt` (the end of the bar). For example (using the earlier `fruit` database):

```
\renewcommand*{\DTLeverybarhook}{%
```

```
\pgftext[at=\DTLmidpt]{\insertName\space(\insertValue)}%
}
\DTLbarchart{variable=\insertValue,axes=both,
ylabel=Quantity,max=50,verticalbars=false
}%
{fruit}{\insertValue=Value,\insertName=Name}
```

This puts the name followed by the quantity in brackets in the middle of the bar.

`\ifDTLverticalbars`

```
\ifDTLverticalbars
```

This conditional governs whether the chart uses vertical or horizontal bars.

`\DTLbarXlabelalign`

```
\DTLbarXlabelalign
```

This specifies the text alignment of the lower bar labels. This defaults to `left,rotate=-90` if you use the vertical=databar package option or the verticalbars setting, and defaults to `right` if you use the horizontal=databar package option or the verticalbars=`false` setting.

`\DTLbarYticklabelalign`

```
\DTLbarYlabelalign
```

This specifies the text alignment of the *y* axis labels. This defaults to `right` for vertical bar charts and `center` for horizontal bar charts.

`\DTLbardisplayYticklabel`

```
\DTLbardisplayYticklabel{⟨text⟩}
```

This specifies how to display the *y* tick label. The argument is the tick label.

`\DTLdisplaylowerbarlabel`

```
\DTLdisplaylowerbarlabel{⟨text⟩}
```

This specifies how to display the lower bar label for `\DTLbarchart` and the lower bar group label for `\DTLmultibarchart`. The argument is the label.

`\DTLdisplaylowermultibarlabel`

```
\DTLdisplaylowermultibarlabel{⟨text⟩}
```

This specifies how to display the lower bar label for `\DTLmultibarchart`. The argument is the label. This command is ignored by `\DTLbarchart`.

`\DTLdisplayupperbarlabel`

`\DTLdisplayupperbarlabel{`⟨*text*⟩`}`

This specifies how to display the upper bar label for `\DTLbarchart` and the upper bar group label for `\DTLmultibarchart`. The argument is the label.

`\DTLdisplayuppermultibarlabel`

`\DTLdisplayuppermultibarlabel{`⟨*text*⟩`}`

This specifies how to display the upper bar label for `\DTLmultibarchart`. The argument is the label. This command is ignored by `\DTLbarchart`.

Example 35 (A Labelled Bar Chart)

This example extends example 34 so that the chart is a bit more informative (which is after all the whole point of a chart). This chart now has a label below each bar, as well as a label above the bar. The lower label uses the value of the `Name` key, and the upper label uses the quantity. I have also set the outline width so each bar has a border.

```
\begin{figure}[htbp]
\setlength{\DTLbaroutlinewidth}{1pt}
\centering
\DTLbarchart{variable=\theQuantity,barlabel=\theName,%
upperbarlabel=\theQuantity}{fruit}{%
\theQuantity=Quantity,\theName=Name}
\caption{A bar chart}
\end{figure}
```

This produces Figure 9.2.

Example 36 (Profit/Loss Bar Chart)

Suppose I have a file called `profits.csv` that looks like:

```
Year,Profit
2000,\pounds2,535
2001,\pounds3,752
2002,-\pounds1,520
2003,\pounds1,270
```

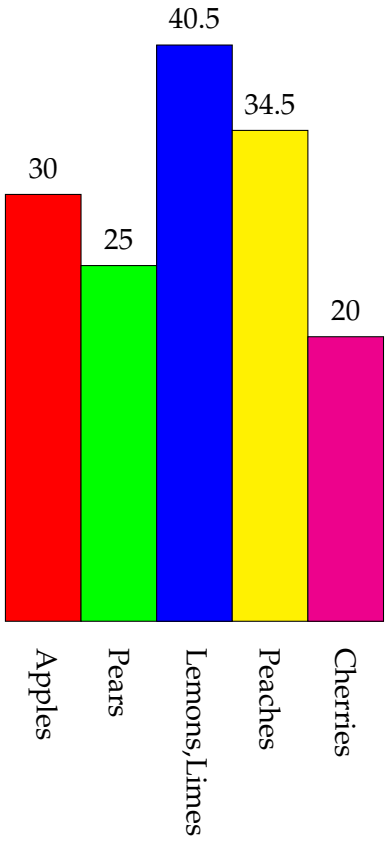

Figure 9.2: A bar chart

First I can load this file into a database called `profits`:

```
\DTLloaddb{profits}{profits.csv}
```

Now I can plot the data as a bar chart:

```
\begin{figure}[htbp]
\centering
 % Set the width of each bar to 10pt
\setlength{\DTLbarwidth}{10pt}
 % Set the outline width to 1pt
\setlength{\DTLbaroutlinewidth}{1pt}
 % Round the $y$ tick labels to integers
\setcounter{DTLbarroundvar}{0}
 % Adjust the tick label offset
\setlength{\DTLticklabeloffset}{20pt}
 % Change the y tick label alignment
\renewcommand*{\DTLbarYticklabelalign}{left}
 % Rotate the y tick labels
\renewcommand*{\DTLbardisplayYticklabel}[1]{\rotatebox{-45}{#1}}
 % Set the bar colours depending on the value of \theProfit
\DTLforeach{profits}{\theProfit=Profit}{%
\ifthenelse{\DTLislt{\theProfit}{0}}
{\DTLsetbarcolor{\DTLcurrentindex}{red}}
{\DTLsetbarcolor{\DTLcurrentindex}{blue}}}
 % Do the bar chart
\DTLbarchart{variable=\theProfit,upperbarlabel=\theYear,
ylabel={Profit/Loss (\pounds)},verticalbars=false,
maxdepth=-2000,max=4000}{profits}
{\theProfit=Profit,\theYear=Year}
\caption{Profits for 2000--2003}
\end{figure}
```

This produces Figure 9.3. Notes:

1. This example uses `\rotatebox`, so the graphics or graphicx package is required.
2. The *y* tick labels are too wide to fit horizontally so they have been rotated to avoid overlapping with their neighbour.
3. Rotating the *y* tick labels puts them too close to the *y* axis, so `\DTLticklabeloffset` is made larger to compensate.
4. Remember not to use `\year` as an assignment command as this command already exists!
5. Before the bar chart is created I have iterated through the database, setting the bar colour to red or blue depending on the value of `\theProfit`.

Both \DTLbarchart and \DTLmultibarchart set the following macros, which may be used in \DTLbaratbegintikz and \DTLbaratendtikz:

\DTLbarchartwidth

\DTLbarchartwidth

This is the overall width of the bar chart. In the case of \DTLbarchart this is just the number of bars. In the case of \DTLmultibarchart it is computed as:

$$m \times n + (m-1) \times g$$

where m is the number of bar groups (i.e. the number of rows of data), n is the number of bars within a group (i.e. the number of commands listed in the variables) setting and g is the group gap (as specified by the groupgap setting).

\DTLnegextent

\DTLnegextent

This is set to the negative extent of the bar chart. (This value may either be zero or negative, and corresponds to the maxdepth setting.)

\DTLbarmax

\DTLbarmax

This is set to the maximum extent of the bar chart. (This value corresponds to the max setting.)

Figure 9.3: Profits for 2000–2003

Example 37 (A Multi-Bar Chart)

This example uses the marks database described in example 14. Recall that this database stores student marks for three assignments. The keys for the assignment marks are Assignment 1, Assignment 2 and Assignment 3, respectively. I can convert this data into a bar chart using the following:

```
\begin{figure}[htbp]
\centering
\DTLmultibarchart{variables={\assignI,\assignII,\assignIII},
barwidth=10pt,uppermultibarlabels={\assignI,\assignII,\assignIII},
barlabel={\firstname\ \surname}}{marks}{%
\surname=Surname,\firstname=FirstName,\assignI=Assignment 1,%
\assignII=Assignment 2,\assignIII=Assignment 3}
\caption{Student marks}
\end{figure}
```

This produces Figure 9.4. Notes:

1. I used `variables={\assignI,\assignII,\assignIII}` to set the variable to use for each bar within a group. This means that there will be three bars in each group.
2. I have set the bar width to 10pt, otherwise the chart will be too wide.
3. I used `uppermultibarlabels={\assignI,\assignII,\assignIII}` to set the upper labels for each bar within a group. This will print the assignment mark above the relevant bar.
4. I used `barlabel={\firstname\ \surname}` to place the student's name below the group corresponding to that student.

Recall that example 14 computed the average score over for each student, and saved it with the key `Average`. This information can be added to the bar chart. It might also be useful to compute the average over all students and add this information to the chart. This is done as follows:

```
\begin{figure}[htbp]
\centering
% compute the overall mean
\DTLmeanforkeys{marks}{Average}{\overallmean}
% round it to 2 decimal places
\DTLround{\overallmean}{\overallmean}{2}
% draw a grey dotted line indicating the overall mean
% covering the entire width of the bar chart
\renewcommand*{\DTLbaratendtikz}{%
  \draw[lightgray,loosely dotted] (0,\overallmean) --
    (\DTLbarchartwidth,\overallmean)
    node[right,black]{Average (\overallmean)};}
% Set the lower bar labels to draw a brace across the current
% group, along with the student's name and average score
\renewcommand*{\DTLdisplaylowerbarlabel}[1]{%
\tikz[baseline=(current bounding box.center)]{
```

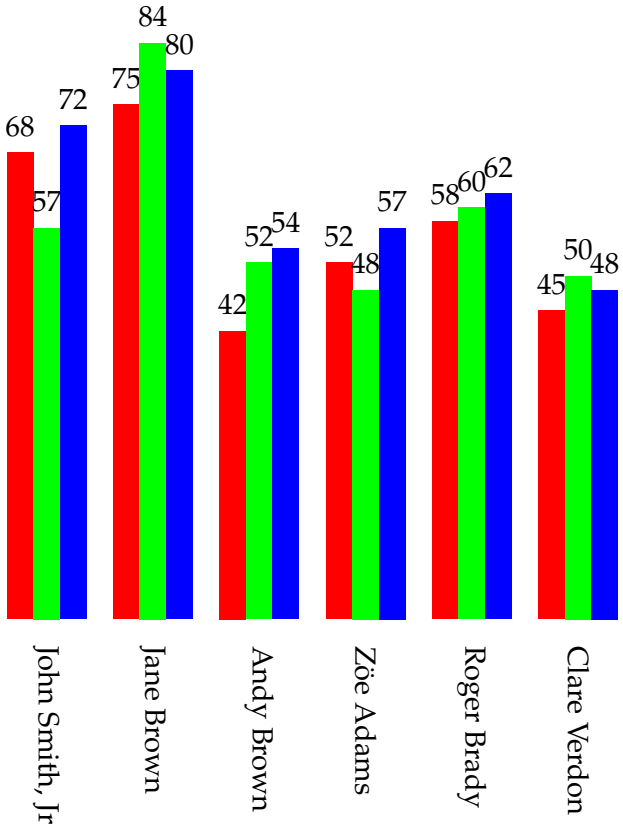

Figure 9.4: Student marks

```
\draw[snake=brace,rotate=-90](0,0) -- (\DTLbargroupwidth,0);}
\DTLround{\theMean}{\theMean}{2}%
\shortstack{#1\\(Average: \theMean)}}
% draw the bar chart
\DTLmultibarchart{variables={\assignI,\assignII,\assignIII},
barwidth=10pt,uppermultibarlabels={\assignI,\assignII,\assignIII},
barlabel={\firstname\ \surname}}{marks}
{\surname=Surname,\firstname=FirstName,\assignI=Assignment 1,%
\assignII=Assignment 2,\assignIII=Assignment 3,\theMean=Average}
\caption{Student marks}
\end{figure}
```

which produces Figure 9.5. Notes:

1. I've used the TikZ snake library to create a brace, so I need to put

   ```
   \usetikzlibrary{snakes}
   ```

   in the preamble. See the pgf manual for more details on how to use this library.

2. I used `\DTLbargroupwidth` to indicate the width of each bar group.

3. I used `\DTLbarchartwidth` to indicate the width of the entire bar chart

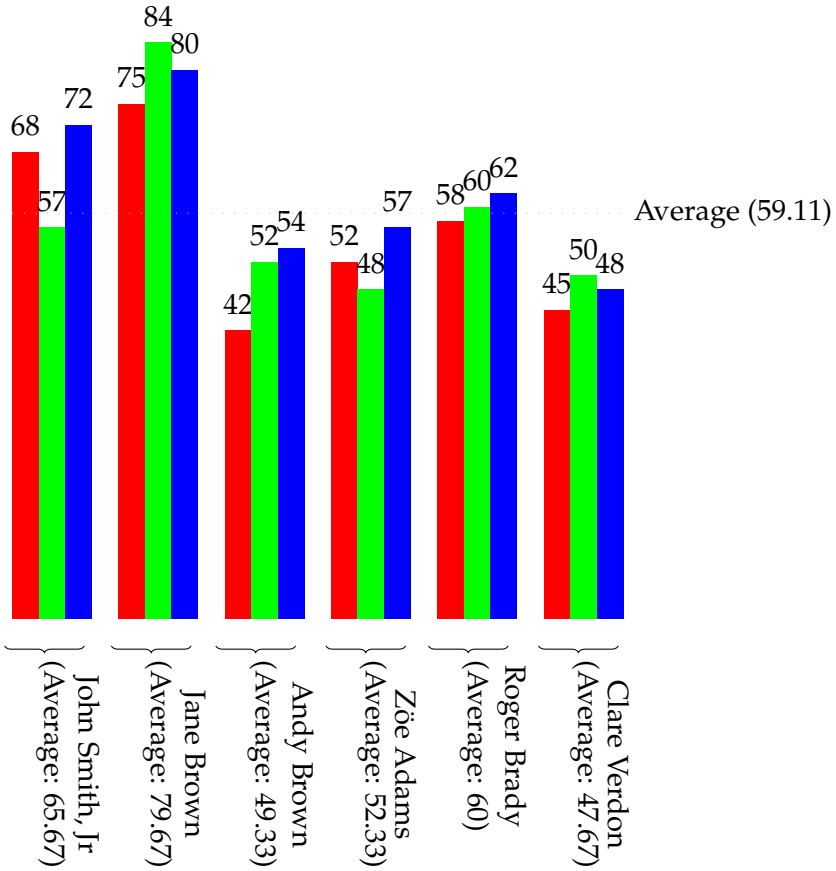

Figure 9.5: Student marks

10 Converting a BibTeX database into a datatool database (databib package)

The databib package provides the means of converting a BibTeX database into a datatool database. The database can then be sorted using `\DTLsort`, described in section 5.8. For example, you may want to sort the bibliography in reverse chronological order. Once you have sorted the bibliography, you can display it using `\DTLbibliography`, described in section 10.3, or you can iterate through the database using `\DTLforeachbib`, described in section 10.5.

Note that the databib package is not automatically loaded by datatool, so if you want to use it, you must load it using `\usepackage{databib}`.

The purpose of this package is to provide a means for authors to format their own bibliography style where there is no bibliography style file available that produces the desired results. The `\DTLsort` macro uses a much less efficient sorting algorithm than BibTeX, and loading the bibliography as a datatool database is much slower than loading a standard `bbl` file. If you have a large database, and you are worried that LaTeX may have become stuck, try using the verbose option to datatool or use the command `\dtlverbosetrue`. This will print informative messages to the console and transcript file, to let you know what's going on.

10.1 BibTeX: An Overview

This document assumes that you have at least some passing familiarity with BibTeX, but here follows a brief refresher.

BibTeX is an external application used in conjunction with LaTeX. When you run BibTeX, you need to specify the name of the document's auxiliary file (without the `aux` extension). BibTeX then reads this file and looks for the commands `\bibstyle` (which indicates which bibliography style (`bst`) file to load), `\bibdata` (which indicates which bibliography database (`bib`) files to load) and `\citation` (produced by `\cite` and `\nocite`, which indicates which entries should be included in the bibliography). BibTeX then creates a file with the extension `bbl` which

contains the bibliography, formatted according to the layout defined in the bibliography style file.

In general, given a document called, say, `mydoc.tex`, you will have to perform the following steps to ensure that the bibliography and all citations are up-to-date:

1. `latex mydoc`

   This writes the citation information to the auxiliary file. The bibliography currently doesn't exists, so it isn't displayed. Citations will appear in the document as ?? since the internal cross-references don't exist yet.

2. `bibtex mydoc`

   This reads the auxiliary file, and creates a file with the extension `bbl` which typically contains the typeset bibliography.

3. `latex mydoc`

   Now that the `bbl` file exists, the bibliography can be input into the document. The internal cross-referencing information for the bibliography can now be written to the auxiliary file.

4. `latex mydoc`

   The cross-referencing information can be read from the auxiliary file.

10.1.1 BibTeX database

The bibliographic data required by BibTeX must be stored in a file with the extension `bib`, where each entry is stored in the form:

@⟨*entry_type*⟩{⟨*cite_key*⟩,
  ⟨*field_name*⟩ = "⟨*value*⟩",
  ⋮
  ⟨*field_name*⟩ = "⟨*value*⟩"
}

Note that curly braces `{` and `}` may be used instead of `"` and `"`.

The entry type, given by ⟨*entry_type*⟩ above, indicates the type of document. This may be one of: `article`, `book`, `booklet`, `inbook`, `incollection`, `inproceedings`[1], `manual`, `mastersthesis`, `misc`, `phdthesis`, `proceedings`, `techreport` or `unpublished`.

[1] Note that `conference` is a synonym for `inproceedings`.

The ⟨*cite_key*⟩ above is a unique label identifying this entry, and is the label used in the argument of `\cite` or `\nocite`. The available fields depends on the entry type, for example, the field `journal` is required for the `article` entry type, but is ignored for the `inproceedings` entry type. The standard fields are: `address`, `author`, `booktitle`, `chapter`, `edition`, `editor`, `howpublished`, `institution`, `journal`, `key`, `month`, `note`, `number`, `organization`, `pages`, `publisher`, `school`, `series`, `title`, `type`, `volume` and `year`.

Author and editor names must be entered in one of the following ways:

1. ⟨*First names*⟩ ⟨*von part*⟩ ⟨*Surname*⟩, ⟨*Jr part*⟩

   The ⟨*von part*⟩ is optional and is identified by the name(s) starting with lowercase letters. The final comma followed by ⟨*Jr part*⟩ is also optional. Examples:

   ```
   author = "Henry James de Vere"
   ```

   In the above, the first names are Henry James, the "von part" is de and the surname is Vere. There is no "junior part".

   ```
   author = "Mary-Jane Brown, Jr"
   ```

   In the above, the first name is Mary-Jane, there is no von part, the surname is Brown and the junior part is Jr.

   ```
   author = "Peter {Murphy Allen}"
   ```

   In the above, the first name is Peter, and the surname is Murphy Allen. Note that in this case, the surname must be grouped, otherwise Murphy would be considered part of the forename.

   ```
   author = "Maria Eliza {\uppercase{d}e La} Cruz"
   ```

   In the above, the first name is Maria Eliza, the von part is De La, and the surname is Cruz. In this case, the von part starts with an uppercase letter, but specifying

   ```
   author = "Maria Eliza De La Cruz"
   ```

   would make BibTeX incorrectly classify "Maria Eliza De La" as the first names, and the von part would be empty. Since BibTeX doesn't understand LaTeX commands, using `{\uppercase{d}e La}` will trick BibTeX into thinking that it starts with a lower case letter.

2. ⟨*von part*⟩ ⟨*Surname*⟩, ⟨*Forenames*⟩

   Again the ⟨*von part*⟩ is optional, and is determined by the case of the first letter. For example:

   ```
   author = "de Vere, Henry James"
   ```

Multiple authors or editors should be separated by the key word `and`, for example:

```
author = "Michel Goossens and Frank Mittlebach and Alexander Samarin"
```

Below is an example of a book entry:

```
@book{latexcomp,
  title     = "The \LaTeX\ Companion",
  author    = "Michel Goossens and Frank Mittlebach and
               Alexander Samarin",
  publisher = "Addison-Wesley",
  year      = 1994
}
```

Note that numbers may be entered without delimiters, as in `year = 1994`. There are also some predefined strings, including those for the month names. You should always use these strings instead of the actual month name, as the way the month name is displayed depends on the bibliography style. For example:

```
@article{Cawley2007b,
author = "Gavin C. Cawley and Nicola L. C. Talbot",
title  = "Preventing over-fitting in model selection via {B}ayesian
          regularisation of the hyper-parameters",
journal = "Journal of Machine Learning Research",
volume  = 8,
pages   = "841--861",
month   = APR,
year    = 2007
}
```

You can concatenate strings using the # character, for example:

```
month  = JUL # "~31~---~" # AUG # "~4",
```

Depending on the bibliography style, this may be displayed as:
July 31 – August 4, or it may be displayed as: Jul 31 – Aug 4. For further information, see [1].

10.2 Loading a databib database

The databib package always requires the `databib.bst` bibliography style file (which is supplied with this bundle). You need to use `\cite` or `\nocite` as usual. If you want to add all entries in the `bib` file to the datatool database, you can use `\nocite{*}`.

`\DTLloadbbl`

`\DTLloadbbl[`⟨*bbl name*⟩`]{`⟨*db name*⟩`}{`⟨*bib list*⟩`}`

This command performs several functions:

1. it writes the following line in the auxiliary file:

   `\bibstyle{databib}`

   which tells BibTeX to use the `databib.bst` BibTeX style file,

2. it writes `\bibdata{`⟨*bib list*⟩`}` to the auxiliary file, which tells BibTeX which `bib` files to use,

3. it creates a datatool database called ⟨*db name*⟩,

4. it loads the file ⟨*bbl name*⟩ if it exists. (The value defaults to `\jobname.bbl`, which is the usual name for a `bbl` file.) If the `bbl` file doesn't exist, the database ⟨*db name*⟩ will remain empty.

You then need to run your document through LaTeX (or PDFLaTeX) and then run BibTeX on the auxiliary file, as described in section 10.1. This will create a `bbl` file which contains all the commands required to add the bibliography information to the datatool database called ⟨*db name*⟩. The next time you LaTeX your document, this file will be read, and the information will be added to ⟨*db name*⟩.

Note that `\DTLloadbbl` doesn't generate any text. Once you have loaded the data, you can display the bibliography uses `\DTLbibliography` (described below) or you can iterate through it using `\DTLforeachbibentry` described in section 10.5.

Note that the `databib.bst` BibTeX style file provides the following additional fields: `isbn`, `doi`, `pubmed`, `url` and `abstract`. However these fields are ignored by the three predefined databib styles (`plain`, `abbrv` and `alpha`). If you want these fields to be displayed in the bibliography you will need to modify the bibliography style (see subsection 10.4.1).

10.3 Displaying a databib database

A databib database which has been loaded using `\DTLloadbbl` (described in section 10.2) can be displayed using:

`\DTLbibliography`

`\DTLbibliography[⟨conditions⟩]{⟨db name⟩}`

where ⟨*db name*⟩ is the name of the database.

Within the optional argument ⟨*condition*⟩, you may use any of the commands that may be used within the optional argument of `\DTLforeach` *In addition*, you may use the following commands:

`\DTLbibfieldexists`

`\DTLbibfieldexists{⟨field label⟩}`

This tests whether the field with the given label exists for the current entry. The field label may be one of: `Address`, `Author`, `BookTitle`, `Chapter`, `Edition`, `Editor`, `HowPublished`, `Institution`, `Journal`, `Key`, `Month`, `Note`, `Number`, `Organization`, `Pages`, `Publisher`, `School`, `Series`, `Title`, `Type`, `Volume`, `Year`, `ISBN`, `DOI`, `PubMed`, `Abstract`, `Url` or `Eprints`.

For example, suppose you have loaded a databib database called `mybib` using `\DTLloadbbl` (described in section 10.2) then the following bibliography will only include those entries which have a `Year` field:

```
\DTLbibliography[\DTLbibfieldexists{Year}]{mybib}
```

`\DTLbibfieldiseq`

`\DTLbibfieldiseq{⟨field label⟩}{⟨value⟩}`

This tests whether the value of the field given by ⟨*field label*⟩ equals ⟨*value*⟩. If the field doesn't exist for the current entry, this evaluates to false. For example, the following will produce a bibliography which only contains entries which have the `Year` field set to 2004:

```
\DTLbibliography[\DTLbibfieldiseq{Year}{2004}]{mybib}
```

`\DTLbibfieldcontains`

`\DTLbibfieldcontains{⟨field label⟩}{⟨sub string⟩}`

This tests whether the value of the field given by ⟨*field label*⟩ contains ⟨*sub string*⟩. For example, the following will produce a bibliography which only contains entries where the author field contains the name `Knuth`:

```
\DTLbibliography[\DTLbibfieldcontains{Author}{Knuth}]{mybib}
```

\DTLbibfieldislt

\DTLbibfieldislt{⟨*field label*⟩}{⟨*value*⟩}

This tests whether the value of the field given by ⟨*field label*⟩ is less than ⟨*value*⟩. If the field doesn't exist for the current entry, this evaluates to false. For example, the following will produce a bibliography which only contains entries whose `Year` field is less than 1983:

```
\DTLbibliography[\DTLbibfieldislt{Year}{1983}]{mybib}
```

\DTLbibfieldisle

\DTLbibfieldisle{⟨*field label*⟩}{⟨*value*⟩}

This tests whether the value of the field given by ⟨*field label*⟩ is less than or equal to ⟨*value*⟩. If the field doesn't exist for the current entry, this evaluates to false. For example, the following will produce a bibliography which only contains entries whose `Year` field is less than or equal to 1983:

```
\DTLbibliography[\DTLbibfieldisle{Year}{1983}]{mybib}
```

\DTLbibfieldisgt

\DTLbibfieldisgt{⟨*field label*⟩}{⟨*value*⟩}

This tests whether the value of the field given by ⟨*field label*⟩ is greater than ⟨*value*⟩. If the field doesn't exist for the current entry, this evaluates to false. For example, the following will produce a bibliography which only contains entries whose `Year` field is greater than 1983:

```
\DTLbibliography[\DTLbibfieldisgt{Year}{1983}]{mybib}
```

\DTLbibfieldisge

\DTLbibfieldisge{⟨*field label*⟩}{⟨*value*⟩}

This tests whether the value of the field given by ⟨*field label*⟩ is greater than or equal to ⟨*value*⟩. If the field doesn't exist for the current entry, this evaluates to false. For example, the following will produce a bibliography which only contains entries whose `Year` field is greater than or equal to 1983:

```
\DTLbibliography[\DTLbibfieldisge{Year}{1983}]{mybib}
```

Note that `\DTLbibliography` uses `\DTLforeachbibentry` (described in section 10.5) so you may also use test the value of the counter DTLbibrow within ⟨*conditions*⟩. You may also use the boolean commands defined by the ifthen package, such as `\not`.

Example 38 (Creating a list of publications since a given year)

Suppose my boss has asked me to produce a list of my publications in reverse chronological order, but doesn't want any publications published prior to the year 2000. I have a file called `nlct.bib` which contains all my publications which I keep in the directory `$HOME/texmf/bibtex/bib/`. I could look through this file, work out the labels for all the publications whose year field is greater or equal to 2000, and create a file with a `\nocite` command containing all those labels in a comma separated list in reverse chronological order, but I really can't be bothered to do that. Instead, I can create the following document:

```
\documentclass{article}
\usepackage{databib}
\begin{document}
\nocite{*}
\DTLloadbbl{mybib}{nlct}
\DTLsort{Year=descending,Month=descending}{mybib}
\DTLbibliography[\DTLbibfieldisge{Year}{2000}]{mybib}
\end{document}
```

Suppose I save this file as `mypubs.tex`, then I need to do:

```
latex mypubs
bibtex mypubs
latex mypubs
```

Notes:

1. `\nocite{*}` is used to add all the citations in the bibliography file (`nlct.bib` in this case) to the databib database.
2. `\DTLloadbbl{mybib}{nlct}` does the following:
   a) writes the line

      ```
      \bibstyle{databib}
      ```

      to the auxiliary file. This tells BibTeX to use `databib.bst` (which is supplied with this package). You therefore shouldn't use `\bibliographystyle`.
   b) writes the line

      ```
      \bibdata{nlct}
      ```

to the auxiliary file. This tells BibTeX that the bibliography data is stored in the file `nlct.bib`. Since I have placed this file in TeX's search path, BibTeX will be able to find it.

c) creates a datatool database called `mybib`.

d) if the `bbl` file (`mypubs.bbl` in this example) exists, it loads this file (which adds the bibliography data to the database), otherwise it does nothing further.

3. In my BibTeX database (`nlct.bib` in this example), I have remembered to use the BibTeX month macros: `jan`, `feb` etc. This means that the months are stored in the database in the form `\DTLmonthname{`⟨*nn*⟩`}`, where ⟨*nn*⟩ is a two digit number from 01 to 12. `\DTLsort` ignores command names when it compares strings, which means I can not only sort by year, but also by month[2].

4. Once I have loaded and sorted my database, I can then display it using `\DTLbibliography`. This uses the style given by the databib style=databib package option, or the `\DTLbibliographystyle` command, both of which are described in section 10.4.

5. I have filtered the bibliography using the optional argument `[\DTLbibfieldisge{Year}{2000}]`, which checks if the year field of the current entry is greater than or equal to 2000. (Note that if an entry has no year field, the condition evaluates to false, and the entry will be omitted from the bibliography.)

6. If the bibliography database is large, sorting and creating the bibliography may take a while. Using databib is much slower than using a standard BibTeX style file.

Example 39 (Creating a list of my 10 most recent publications)

Suppose now my boss has asked me to produce a list of my ten most recent publications (in reverse chronological order). As in the previous example, I have a file called `nlct.bib` which contains all my publications. I can create the required document as follows:

```
\documentclass{article}
\usepackage{databib}
\begin{document}
```

[2] as long as I haven't put anything before the month name in the bibliography file, e.g. `month = 2  # apr` will sort by 2 03, instead of 03

```
\nocite{*}
\DTLloadbbl{mybib}{nlct}
\DTLsort{Year=descending,Month=descending}{mybib}
\DTLbibliography[\value{DTLbibrow}<10]{mybib}
\end{document}
```

10.4 Changing the bibliography style

The style of the bibliography produced using `\DTLbibliography` depends on the style=databib package option, or can be set using

`\DTLbibliographystyle`

```
\DTLbibliographystyle{⟨style⟩}
```

Note that this is *not* the same as `\bibliographystyle`, as the databib package uses its custom `databib.bst` bibliography style file.

Example:

```
\usepackage[style=plain]{databib}
```

This sets the plain bibliography style. This is, in fact, the default style, so it need not be specified.

Available styles are: `plain`, `abbrv` and `alpha`. These are similar to the standard BibTeX styles of the same name, but are by no means identical. The most notable difference is that these styles do not sort the bibliography. It is up to you to sort the bibliography using `\DTLsort` (described in section 5.8).

10.4.1 Modifying an existing style

This section describes some of the commands which are used to format the bibliography. You can choose whichever predefined style best fits your required style, and then modify the commands described in this section. A description of the remaining commands not listed in this section can be found in section 6.4, section 6.5 and section 6.6.

`\DTLformatauthor`

```
\DTLformatauthor{⟨von part⟩}{⟨surname⟩}{⟨jr part⟩}{⟨forenames⟩}
```

`\DTLformateditor`

```
\DTLformateditor{⟨von part⟩}{⟨surname⟩}{⟨jr part⟩}{⟨forenames⟩}
```

These commands are used to format an author/editor's name, respectively. The list of authors and editors are stored in the databib database as a comma separated list of {⟨*von part*⟩}{⟨*surname*⟩}{⟨*jr part*⟩}{⟨*forenames*⟩} data. This ensures that when you sort on the Author or Editor field, the names will be sorted by the first author or editor's surname.

Within `\DTLformatauthor` and `\DTLformateditor`, you may use the following commands:

`\DTLformatforenames`

`\DTLformatforenames{⟨forenames⟩}`

This is used by the `plain` style to display the author's forenames[3].

`\DTLformatabbrvforenames`

`\DTLformatabbrvforenames{⟨forenames⟩}`

This is used by the `abbrv` style to display the author's initials (which are determined from ⟨*forenames*⟩). Note that if any of the authors has a name starting with an accent, the accented letter must be grouped in order for this command to work. For example:

```
author = "{\'E}lise {\"E}awyn Edwards",
```

The initials are formed using `\DTLstoreinitials` described in chapter 4, so if you want to change the way the initials are displayed (e.g. put a space between them) you will need to redefine the commands used by `\DTLstoreinitials` (such as `\DTLbetweeninitials`).

`\DTLformatsurname`

`\DTLformatsurname{⟨surname⟩}`

This displays its argument by default[4].

`\DTLformatvon`

`\DTLformatvon{⟨von part⟩}`

If the ⟨*von part*⟩ is empty, this command does nothing, otherwise it displays its argument followed by a non-breakable space.

`\DTLformatjr`

[3] It also checks whether ⟨*forenames*⟩ ends with a full stop using `\DTLcheckendsperiod` to prevent a sentence ending full stop from following an abbreviation full stop

[4] It also checks whether the surname ends with a full stop using `\DTLcheckendsperiod`

\DTLformatjr{⟨*jr part*⟩}

If the ⟨*jr part*⟩ is empty, this command displays nothing, otherwise it displays a comma followed by its argument[5].

For example, suppose you want the author's surname to appear first in small capitals, followed by a comma and the forenames. This can be achieved by redefining \DTLformatauthor as follows:

```
\renewcommand*{\DTLformatauthor}[4]{%
\textsc{\DTLformatvon{#1}%
\DTLformatsurname{#2}\DTLformatjr{#3}},
\DTLformatforenames{#4}%
}
```

DTLmaxauthors

DTLmaxauthors

The counter DTLmaxauthors is used to determine the maximum number of authors to display for a given entry. If the entry's author list contains more than that number of authors, \etalname is used, the definition of which is given in section 6.4. The default value of DTLmaxauthors is 10.

DTLmaxeditors

DTLmaxeditors

The DTLmaxeditors counter is analogous to the DTLmaxauthors counter. It is used to determine the maximum number of editor names to display. The default value of DTLmaxeditors is 10.

\DTLandlast
\DTLandnotlast
\DTLtwoand

Within a list of author or editor names, \DTLandlast is used between the last two names, otherwise \DTLandnotlast is used between names. However, if there are only two author or editor names, \DTLtwoand is used instead of \DTLandlast.

\DTLendbibitem

The command \DTLendbibitem is a hook provided to add additional information at the end of each bibliography item. This does nothing by default, but if you want to display the additional fields provided by the databib.bst style file, you can redefine \DTLendbibitem so that it displays a particular field, if it is defined. Within this command, you may use the commands \DTLbibfield, \DTLifbibfieldexist and \DTLifanybibfieldexist, which are described in section 10.5. For example, if you have used the abstract field in any of your entries, you can display the abstract as follows:

```
\renewcommand{\DTLendbibitem}{%
```

[5] again, it also checks ⟨*jr part*⟩ to determine if it ends with a full stop

```
\DTLifbibfieldexists{Abstract}{\DTLpar\textbf{Abstract}
\begin{quote}\DTLbibfield{Abstract}\end{quote}}{}}
```

(Note that `\DTLpar` needs to be used instead of `\par`.)

Example 40 (Compact bibliography)

Suppose I don't have much space in my document, and I need to produce a compact bibliography. Firstly, I can use the bibliography style `abbrv`, either through the package option:

```
\usepackage[style=abbrv]{databib}
```

or using:

```
\DTLbibliographystyle{abbrv}
```

Once I have set the style, I can further modify it thus:

```
\renewcommand*{\andname}{\&}
\renewcommand*{\editorname}{ed.}
\renewcommand*{\editorsname}{eds.}
\renewcommand*{\pagesname}{pp.}
\renewcommand*{\pagename}{p.}
\renewcommand*{\volumename}{vol.}
\renewcommand*{\numbername}{no.}
\renewcommand*{\editionname}{ed.}
\renewcommand*{\techreportname}{T.R.}
\renewcommand*{\mscthesisname}{MSc thesis}
```

Now I can load[6] and display the bibliography:

```
% create a database called mybib from the information given
% in mybib1.bib and mybib2.bib
\DTLloadbbl{mybib}{mybib1,mybib2}
% display the bibliography
\DTLbibliography{mybib}
```

Example 41 (Highlighting a given author)

Suppose my boss wants me to produce a list of all my publications (which I have stored in the file `nlct.bib`, as in example 38). Most of my publications have multiple co-authors, but suppose my boss would like me to highlight my name so that when he skims through the document, he can easily see my name in the list of co-authors. I can do this by

[6] I can load the bibliography earlier, but obviously the bibliography should only be displayed after the bibliography styles have been set, otherwise they will have no effect

redefining \DTLformatauthor so that it checks if the given surname matches mine. (This assumes that none of the other co-author's share my surname.)

```
\renewcommand*{\DTLformatauthor}[4]{%
{\DTLifstringeq{#2}{Talbot}{\bfseries }{}%
\DTLformatforenames{#4}
\DTLformatvon{#1}%
\DTLformatsurname{#2}%
\DTLformatjr{#3}}}
```

Notes:

1. I have used \DTLifstringeq (described in section 2.1) to perform the string comparison.

2. If one or more of my co-authors shared the same surname as me, I would also have had to check the first name, however there is regrettably a lack of consistency in my bib file when it comes to my forenames. Sometimes my name is given as Nicola L. C. Talbot, sometimes the middle initials are omitted, Nicola Talbot, or sometimes, just initials are used, N. L. C. Talbot. This can cause problems when checking the forenames, but as long as the other authors who share the same surname as me, don't also share the same first initial, I can use \DTLifStartsWith or \DTLisPrefix, which are described in section 2.1 and section 2.2, respectively. Using the first approach I can do:

   ```
   \renewcommand*{\DTLformatauthor}[4]{%
   {\DTLifstringeq{#2}{Talbot}{\DTLifStartsWith{#4}{N}{\bfseries }{}}{}}%
   \DTLformatforenames{#4}
   \DTLformatvon{#1}%
   \DTLformatsurname{#2}%
   \DTLformatjr{#3}}}
   ```

   Using the second approach I can do:

   ```
   \renewcommand*{\DTLformatauthor}[4]{%
   {\ifthenelse{\DTLiseq{#2}{Talbot}\and
   \DTLisPrefix{#4}{N}}{\bfseries }{}%
   \DTLformatforenames{#4}
   \DTLformatvon{#1}%
   \DTLformatsurname{#2}%
   \DTLformatjr{#3}}}
   ```

3. I have used a group to localise the effect of \bfseries.

10.5 Iterating through a databib database

\DTLbibliography (described in section 10.3) may still not meet your needs. For example, you may be required to list journal papers and conference proceedings in separate sections. In which case, you may find it easier to iterate through the bibliography using:

\DTLforeachbib

\DTLforeachbib[⟨*condition*⟩]{⟨*db name*⟩}{⟨*text*⟩}

\DTLforeachbib*

\DTLforeachbib*[⟨*condition*⟩]{⟨*db name*⟩}{⟨*text*⟩}

This iterates through the databib database called ⟨*db name*⟩ and does ⟨*text*⟩ if ⟨*condition*⟩ is met. As with \DTLforeach, the starred version is read-only.

For each row of the database, the following commands are set:

\DBIBcitekey

- \DBIBcitekey This is the unique label which identifies the current entry (as used in the argument of \cite and \nocite).

\DBIBentrytype

- \DBIBentrytype This is the current entry type, and will be one of: article, book, booklet, inbook, incollection, inproceedings, manual, mastersthesis, misc, phdthesis, proceedings, techreport or unpublished. (Note that even if you used the entry type conference in your bib file, its entry type will be set to inproceedings).

The remaining fields may be accessed using:

\DTLbibfield

\DTLbibfield{⟨*field label*⟩}

where ⟨*field label*⟩ may be one of: Address, Author, BookTitle, Chapter, Edition, Editor, HowPublished, Institution, Journal, Key, Month, Note, Number, Organization, Pages, Publisher, School, Series, Title, Type, Volume, Year, ISBN, DOI, PubMed, Abstract or Url.

You can determine if a field exists for a given entry using

\DTLifbibfieldexists

\DTLifbibfieldexists{⟨*field label*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

If the field given by ⟨*field label*⟩ exists for the current bibliography entry, it does ⟨*true part*⟩, otherwise it does ⟨*false part*⟩.

\DTLifbibanyfieldexists

\DTLifanybibfieldexists{⟨*field label list*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

This is similar to \DTLifbibfieldexists except that the first argument is a list of field names. If one or more of the fields given in ⟨*field label list*⟩ exists for the current bibliography item, this does ⟨*true part*⟩, otherwise it does ⟨*false part*⟩.

\DTLformatbibentry

\DTLformatbibentry

This formats the bibliography entry for the current row. It checks for the existence of the command \DTLformat⟨*entry type*⟩, where ⟨*entry type*⟩ is given by \DBIBentrytype. These commands are defined by the bibliography style.

\DTLcomputewidestbibentry

\DTLcomputewidestbibentry{⟨*conditions*⟩}{⟨*db name*⟩}{⟨*bib label*⟩}{⟨*cmd*⟩}

This computes the widest bibliography entry over all entries satisfying ⟨*conditions*⟩ in the database ⟨*db name*⟩, where the label is given by ⟨*bib label*⟩, and the result is stored in ⟨*cmd*⟩, which may then be used in the argument of the thebibliography environment.

DTLbibrow

The counter DTLbibrowkeeps track of the current bibliography entry. This is reset at the start of each \DTLforeachbib and is incremented if ⟨*conditions*⟩ is met.

Within the optional argument ⟨*condition*⟩, you may use any of the commands that may be used within the optional argument of \DTLbibliography, described in section 10.3.

Example 42 (Separate List of Journals and Conference Papers)

Suppose now my boss has decided that I need to produce a list of all my publications, but they need to be separated so that all the journal papers appear in one section, and all the conference papers appear in another section. The journal papers need to be labelled [J1], [J2] and so on, while the conference papers need to be labelled [C1], [C2] and so on. (My boss isn't interested in any of my other publications!) Again, all my publications are stored in the BibTeX database nlct.bib. The following creates the required document:

```
\documentclass{article}
\usepackage{databib}
```

```
\begin{document}
\nocite{*}
\DTLloadbbl{mybib}{nlct}

\renewcommand*{\refname}{Journal Papers}
\DTLcomputewidestbibentry{\equal{\DBIBentrytype}{article}}
{mybib}{J\theDTLbibrow}{\widest}

\begin{thebibliography}{\widest}
\DTLforeachbibentry[\equal{\DBIBentrytype}{article}]{mybib}{%
\bibitem[J\theDTLbibrow]{\DBIBcitekey} \DTLformatbibentry}
\end{thebibliography}

\renewcommand*{\refname}{Conference Papers}
\DTLcomputewidestbibentry{\equal{\DBIBentrytype}{inproceedings}}
{mybib}{C\theDTLbibrow}{\widest}

\begin{thebibliography}{\widest}
\DTLforeachbibentry[\equal{\DBIBentrytype}{inproceedings}]{mybib}{%
\bibitem[C\theDTLbibrow]{\DBIBcitekey} \DTLformatbibentry}
\end{thebibliography}

\end{document}
```

10.6 Multiple Bibliographies

It is possible to have more than one bibliography in a document, but it then becomes necessary to have a separate auxiliary file for each bibliography, and each auxiliary file must then be passed to BibTeX. In order to do this, you need to use

\DTLmultibibs

```
\DTLmultibibs{⟨name list⟩}
```

where ⟨*name list*⟩ is a comma separated list of names, ⟨*name*⟩. For each ⟨*name*⟩, this command creates an auxiliary file called ⟨*name*⟩`.aux` (note that this command may only be used in the preamble).

When you want to cite an entry for a given bibliography named in `\DTLmultibibs`, you must use:

\DTLcite

```
\DTLcite[⟨text⟩]{⟨mbib⟩}{⟨cite key list⟩}
```

This is analogous to `\cite[`⟨*text*⟩`]{`⟨*cite key list*⟩`}`, but writes the

\citation command to ⟨*mbib*⟩.aux instead of to the document's main auxiliary file. It also ensures that the cross-referencing labels are based on ⟨*mbib*⟩, to allow you to have the same reference in more than one bibliography without incurring a "multiply defined" warning message. Note that you can still use \cite to add citation information to the main auxiliary file.

If you want to add an entry to the bibliography without producing any text, you can use

\DTLnocite

\DTLnocite{⟨*mbib*⟩}{⟨*cite key list*⟩}

which is analogous to \nocite{⟨*cite key list*⟩}, where again the citation information is written to ⟨*mbib*⟩.aux instead of the document's main auxiliary file.

Note that for both \DTLcite and \DTLnocite the ⟨*mbib*⟩ part must be one of the names listed in \DTLmultibibs.

\DTLloadmbbl

\DTLloadmbbl{⟨*mbib*⟩}{⟨*db name*⟩}{⟨*bib list*⟩}

This is analogous to \DTLloadbbl{⟨*db name*⟩}{⟨*bib list*⟩} described in section 10.2. (Again ⟨*mbib*⟩ must be one of the names listed in \DTLmultibibs.) This creates a new datatool database called ⟨*db name*⟩ and loads the bibliography information from ⟨*mbib*⟩.bbl (if it exists).

\DTLmbibliography

\DTLmbibliography[⟨*condition*⟩]{⟨*mbib*⟩}{⟨*db name*⟩}

This is analogous to \DTLbibliography[⟨*condition*⟩]{⟨*db name*⟩}, but is required when displaying a bibliography in which elements have been cited using \DTLcite and \DTLnocite.

Example 43 (Multiple Bibliographies)

Suppose I need to create a document which contains a section listing all my publications, but I also need to have separate sections covering each of my research topics, with a mini-bibliography at the end of each section. As in the earlier examples, all my publications are stored in the file nlct.bib which is somewhere on TEX's path. Note that there will be some duplication as the references in the mini-bibliographies will also appear in the main bibliography at the end of the document, but using \DTLcite and \DTLmbibliography ensures that all the

cross-referencing labels (and hyperlinks if they are enabled) are unique.

```
\documentclass{article}
\usepackage{databib}
\DTLmultibibs{kernel,food}
\begin{document}
\section{Kernel methods}
In this section I'm going to describe some research work into
kernel methods, and in the process I'm going to cite some related
papers \DTLcite{kernel}{Cawley2007a,Cawley2006a}.

\DTLloadmbbl{kernel}{kernelDB}{nlct}
\DTLmbibliography{kernel}{kernelDB}

\section{Food research}

In this section I'm going to describe some research work
in the area of food safety, and in the process, I'm going
to cite some related papers \DTLcite{food}{Peck1999,Barker1999a}

\DTLloadmbbl{food}{foodDB}{nlct}
\DTLmbibliography{food}{foodDB}

\cite{*}
\renewcommand{\refname}{Complete List of Publications}
\DTLloadbbl{fullDB}{nlct}
\DTLbibliography{fullDB}
\end{document}
```

Notes:

1. This will create the files `kernel.aux` and `food.aux`. These will have to be passed to BibTeX, in addition to the documents main auxiliary file. So, if my document is called `researchwork.tex`, then I need to do:

   ```
   latex researchwork
   bibtex researchwork
   bibtex kernel
   bibtex food
   latex researchwork
   latex researchwork
   ```

2. `\cite{*}` is used to add all the entries in the bib file to the main bibliography database. As before, `\DTLloadbbl` and `\DTLbibliography` are used to load and display the main bibliography.

Don't try to directly input the `.bbl` file using `\input` (or `\include`) instead of using `\DTLloadbbl` or `\DTLloadmbbl` as these commands store the name of the required database and initialise the database before loading the `.bbl` file. Similarly, don't just copy the contents of the `.bbl` file into your document without first defining the database using `\DTLnewdb` and setting `\DTLBIBdbname` to the name of the database.

11 Referencing People (person package)

Sometimes when mail-merging, it may be necessary to reference a person by their pronoun which can lead to the cumbersome and impersonal "he/she" construct. The person package allows you to define a person by their full name, familiar name and gender. You can then use the commands described in section 11.2 to produce the appropriate pronoun.

This can also be useful for other types of documents, such as an order of service for a baptism or funeral. Since the document is much the same from one person to the next, documents of this nature are frequently simply copied and a search and replace edit is used to change the relevant text. However this can lead to errors (especially if the previous person's name was Mary!) With the person package, you need only change the definition of the person by modifying the arguments of `\newperson`.

11.1 Defining and Undefining People

A person is defined (globally) using the command:

`\newperson`

`\newperson[⟨label⟩]{⟨full name⟩}{⟨familiar name⟩}{⟨gender⟩}`

The optional argument is a unique label identifying this person, in the event that there is more than one person. If ⟨*label*⟩ is omitted `anon` is used. (This is also the case for subsequent commands that take an optional label.) The gender may be any of those given by

`\malelabels`

`\malelabels`

or

`\femalelabels`

`\femalelabels`

The default definition of `\malelabels` is `male,Male,MALE,M,m` and the default definition of `\femalelabels` is `female,Female,FEMALE,F,f`. You can add extra identifiers using

\addmalelabel

```
\addmalelabel{⟨identifier⟩}
```

or

\addfemalelabel

```
\addfemalelabel{⟨identifier⟩}
```

For example:

```
\addmalelabel{boy}
\addfemalelabel{girl}
```

The total number of defined people is given by:

\thepeople

```
\thepeople
```

A person can be undefined using:

\removeperson

```
\removeperson[⟨label⟩]
```

where the person is given by ⟨*label*⟩.

If more than one person has been defined, they can all be removed using:

\removeallpeople

```
\removeallpeople
```

or you can remove a subset using:

\removepeople

```
\removepeople{⟨list⟩}
```

where ⟨*list*⟩ is a comma-separated list of labels.

11.2 Displaying Information

Once a person has been defined, you can display their name using:

\personfullname

```
\personfullname[⟨label⟩]
```

where ⟨*label*⟩ is the unique label used in the optional argument to `\newperson`. The person's familiar name is displayed using:

`\personname`

```
\personname[⟨label⟩]
```

The person's pronoun ("he" or "she") is displayed using:

`\personpronoun`

```
\personpronoun[⟨label⟩]
```

The objective pronoun ("him" or "her") is displayed using:

`\personobjpronoun`

```
\personobjpronoun[⟨label⟩]
```

The possessive adjective ("his" or "her") is displayed using:

`\personpossadj`

```
\personpossadj[⟨label⟩]
```

The possessive pronoun "his" or "hers" is displayed using:

`\personposspronoun`

```
\personposspronoun[⟨label⟩]
```

The person's relationship to their parent ("son" or "daughter") is displayed using:

`\personchild`

```
\personchild[⟨label⟩]
```

The person's relationship to their child ("mother" or "father") is displayed using:

`\personparent`

```
\personparent[⟨label⟩]
```

The person's relationship to their sibling ("brother" or "sister") is displayed using:

`\personsibling`

```
\personsibling[⟨label⟩]
```

If the word occurs at the start of a sentence, you will need one of the following commands, which are as the above, except the first letter is converted to upper case:

\Personpronoun

```
\Personpronoun[⟨label⟩]
```

\Personobjpronoun

```
\Personobjpronoun[⟨label⟩]
```

\Personpossadj

```
\Personpossadj[⟨label⟩]
```

\Personposspronoun

```
\Personposspronoun[⟨label⟩]
```

\Personchild

```
\Personchild[⟨label⟩]
```

\Personparent

```
\Personparent[⟨label⟩]
```

\Personsibling

```
\Personsibling[⟨label⟩]
```

Example 44 (Order of Service (Memorial))

This example is for a memorial order of service.

```
\documentclass{article}

\usepackage{person}

\newperson{Jane Doe}{Jane}{female}

\begin{document}
\begin{center}
\Large
In Memory of \personfullname
\end{center}
```

```
We are gathered here to remember our \personsibling\ \personname.
\Personpronoun\ will be much missed, and \personpossadj\
family are in our prayers.
\end{document}
```

In Memory of Jane Doe

We are gathered here to remember our sister Jane. She will be much missed, and her family are in our prayers.

If there is more than one person, you will need to use the optional argument ⟨*label*⟩ to `\newperson` to uniquely identify each person. You can then list all of the people's full or familiar names using:

`\peoplefullname`

```
\peoplefullname
```

`\peoplename`

```
\peoplename
```

Note that if there is only one person defined, these commands behave the same as `\personfullname[`⟨*label*⟩`]` and `\personname`[⟨*label*⟩].

Similarly for the pronouns:

`\peoplepronoun`

```
\peoplepronoun
```

`\Peoplepronoun`

```
\Peoplepronoun
```

`\peopleobjpronoun`

```
\peopleobjpronoun
```

`\Peopleobjpronoun`

```
\Peopleobjpronoun
```

`\peoplepossadj`

```
\peoplepossadj
```

\Peoplepossadj

```
\Peoplepossadj
```

\peoplepossproun

```
\peoplepossproun
```

\Peoplepossproun

```
\Peoplepossproun
```

where, again, if only one person has been defined, each of these commands is equivalent to `\person...[`⟨*label*⟩`]` or `\Person...[`⟨*label*⟩`]`. If more than one person has been defined, these commands will display they/them/their/theirs or They/Them/Their/Theirs, as appropriate.

Likewise for relationship commands:

\peoplechild

```
\peoplechild
```

\Peoplechild

```
\Peoplechild
```

\peopleparent

```
\peopleparent
```

\Peopleparent

```
\Peopleparent
```

\peoplesibling

```
\peoplesibling
```

\Peoplesibling

```
\Peoplesibling
```

Example 45 (Order of Service (Baptism))

In this example two people are defined.

```
\documentclass{article}
```

```
\usepackage{person}

\newperson[john]{John Joseph}{John}{male}
\newperson[jane]{Jane Mary}{Jane}{female}

\begin{document}
\begin{center}
\Large
Baptism of \peoplefullname.
\end{center}

Today we welcome \peoplename\ into God's family, may He guide
and protect \peopleobjpronoun.
\end{document}
```

This is produces the following text:

Baptism of John Joseph and Jane Mary.

Today we welcome John and Jane into God's family, may He guide and protect them.

Example 46 (Mail Merging Using Appropriate Gender)

In this example I have a CSV file called `students.csv` containing the following:

```
FirstName,Surname,Gender,Parent,Address
John,"Smith, Jr",M,Mr and Mrs Smith,1 The Street\\Newtown
Jane,Brown,F,Ms Brown,2 The Avenue\\Oldtown
Andy,Brown,male,Mr Brown and Miss Sepia,3 The Road\\Newtown
Z\"oe,Adams,f,Mr and Mrs Adams,5 The Street\\Newtown
Roger,Brady,m,Mrs Brady,6 The Avenue\\Oldtowm
Clare,Vernon,female,Mr Vernon,7 The Close\\Anytown
```

Suppose I have to write to each student's parents regarding their child. I can load the information using `\DTLloaddb` (described in section 5.2). I can then iterate through the database and define the student as a person and use the commands defined in the person package to display the correct gender related text. I could give each person a unique label based on the row count (`\DTLcurrentindex`), but since I don't need to reuse the information, I can use the default `anon` label and undefine the person when no longer required.

Note that in the CSV file, the gender label isn't consistent. For some students the gender is identified by a single letter ("m" or "f") and for others the gender is identified by a complete word ("male" or "female").

There's also no regard for case. This doesn't matter to \newperson as all the identifiers used are listed in \malelabels and \femalelabels.

The following is an example letter sent to all parents:

```
\documentclass{letter}
\usepackage{person}

 load student information from file "students.csv"
\DTLloaddb{students}{students.csv}
\begin{document}
 Iterate through the student database:
\DTLforeach{students}{\FirstName=FirstName,\Surname=Surname,%
\Gender=Gender,\Parent=Parent,\Address=Address}{%
 Define "anon":
  \newperson{\FirstName\space\Surname}{\FirstName}{\Gender}%
 Do the letter:
  \begin{letter}{\Parent\\\Address}
  \opening{Dear \Parent}
    Your \personchild\ \personname\ has been awarded a
    place. We look forward to seeing \personobjpronoun\
    on \personpossadj\ arrival.
  \closing{Yours Sincerely}
  \end{letter}
 Undefine "anon":
  \removeperson
}
\end{document}
```

The body of the first letter appears as follows:

Your son John has been awarded a place. We look forward to seeing him on his arrival.

Whereas the body of the second letter appears as follows:

Your daughter Jane has been awarded a place. We look forward to seeing her on her arrival.

11.3 Advanced Commands

This section describes additional commands provided by the person package. More detail can be found in the documented code (datatool-code.pdf).

11.3.1 Conditionals

\ifpersonexists

\ifpersonexists{⟨*label*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

Tests if the person identified by ⟨*label*⟩ has been defined. If true, do ⟨*true part*⟩ otherwise do ⟨*false part*⟩.

\ifmale

\ifmale{⟨*label*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

Test if the person identified by ⟨*label*⟩ is male. If true, do ⟨*true part*⟩ otherwise do ⟨*false part*⟩.

\iffemale

\iffemale{⟨*label*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

Test if the person identified by ⟨*label*⟩ is female. If true, do ⟨*true part*⟩ otherwise do ⟨*false part*⟩.

\ifallmale

\ifallmale[⟨*label list*⟩]{⟨*true part*⟩}{⟨*false part*⟩}

Tests if all the people listed in ⟨*label list*⟩ are male. If true, do ⟨*true part*⟩ otherwise do ⟨*false part*⟩. If ⟨*label list*⟩ is omitted, applied to all defined people.

\ifallfemale

\ifallfemale[⟨*label list*⟩]{⟨*true part*⟩}{⟨*false part*⟩}

Likewise to test if all the people tested are female.

To determine if a string is an allowed male label:

\ifmalelabel

\ifmalelabel{⟨*identifier*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

where ⟨*identifier*⟩ is the string to be tested. If true, do ⟨*true part*⟩ otherwise do ⟨*false part*⟩. For example:

```
\def\gender{M}
\ifmalelabel{\gender}{male}{not male}
```

Similarly to for an allowed female label:

\iffemalelabel

\iffemalelabel{⟨*identifier*⟩}{⟨*true part*⟩}{⟨*false part*⟩}

For example:

```
\ifmalelabel{\gender}{Male}{%
  \iffemalelabel{\gender}{Female}%
   {Undefined Gender}%
}
```

11.3.2 Iterating Through Defined People

You can iterate through all defined people using:

\foreachperson

\foreachperson(⟨*name cs*⟩,⟨*full name cs*⟩,⟨*gender cs*⟩,⟨*label cs*⟩)\do{⟨*body*⟩}

At each iteration, ⟨*name cs*⟩, ⟨*full name cs*⟩, ⟨*gender cs*⟩ and ⟨*label cs*⟩ are set to the current person's name, full name, gender and label, respectively. (These arguments must all be command names.) Note that the gender is set to the definition of \malename or \femalename, as appropriate.[1] Once these commands are set, ⟨*body*⟩ is applied.

\malename
\femalename

If you only want to iterate through a subset of defined people, you can use:

\foreachperson(⟨*name cs*⟩,⟨*full name cs*⟩,⟨*gender cs*⟩,⟨*label cs*⟩)\in{⟨*list*⟩}\do{⟨*body*⟩}

where ⟨*list*⟩ is a comma-separated list of labels.

11.3.3 Accessing Individual Information

\getpersongender

\getpersongender{⟨*cs*⟩}{⟨*label*⟩}

Gets the gender of the person identified by ⟨*label*⟩ and stores in ⟨*cs*⟩ (which must be a command name). This sets ⟨*cs*⟩ to the definition of \malename or \femalename as appropriate.

\getpersonname

[1] Predefined names provided by the person package are described in the documented code (datatool-code.pdf).

\getpersonname{⟨*cs*⟩}{⟨*label*⟩}

Gets the name of the person identified by ⟨*label*⟩ and stores in ⟨*cs*⟩ (which must be a command name).

\getpersonfullname

\getpersonfullname{⟨*cs*⟩}{⟨*label*⟩}

Gets the full name of the person identified by ⟨*label*⟩ and stores in ⟨*cs*⟩ (which must be a command name).

Bibliography

[1] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, 1994.

Acknowledgements

Many thanks to Morten Høgholm for suggesting a much more efficient way of storing the information in databases which has significantly improved the time it takes to LaTeX documents containing large databases.

Index

E

F

G

H

I

L

M

N