What every (LA)TEX newbie should know

Barbara Beeton

Abstract

LaTeX has a reputation for producing excellent results, but at the cost of a steep learning curve. That's true, but by understanding a few basic principles, and learning how to avoid some techniques that may seem obvious but often lead one into the weeds, it's possible to avoid some of that pain. Our goal here is to encourage good habits before bad habits have had a chance to develop.

Introduction

The examples presented here are drawn from two main sources.

- In the author's years as part of the TEXnical support team for a major math publisher, responsibilities included fielding questions from authors and writing user documentation.
- The online TEX forum at StackExchange[1] has provided a surfeit of questions both basic and advanced. A community effort has collected a list of "Often referenced questions", by topic, at `tex.meta.stackexchange.com/q/2419`.

Exhortation: Read the documentation. (This will be repeated.)

Vocabulary

There are several concepts that seem to be either missing from a new user's bag of tricks, or not clearly understood. Let's get them out of the way up front.

Template Many new (LA)TEX users think that the document class is the template for a particular style or publication. Not so, although the thought is going in the right direction. A template is a source (`.tex`) file that is an outline. It begins with `\documentclass` and contains a minimum of basic structural commands into which additional definitions and text can be inserted as appropriate. Ideally, the template itself can be compiled with no errors resulting, but without producing any useful output.

Command line Most new users these days enter (LA)TEX from an editor or other GUI, and launch a non-interactive job that will blithely keep on processing the file until it finishes (with or without errors) or hangs in a loop. Launching the compilation from the command line, on the other hand, allows one to interact with the session and, in certain cases, make corrections "on the fly", or if that's not possible, halt the job in case of an error before the collection of reported errors becomes unhelpful. One type of "fixable" error is a misspelled command:

```
! Undefined control sequence.
l.37 \scetion
                {Section}
?
```

Respond to this with the correct spelling;

```
i\section
```

hit "return", and continue; don't forget to correct the file when you come to a good stopping point.

A misspelled environment name can't be corrected this way; if that happens, cancel the job with an `x`, fix the file, and start over. Continuing a run after an unfixable error will just result in more error messages, most of which are meaningless and confusing, so it's best to avoid them.

Log file Every time a TEX job runs, it will create a log file. Learn where to find this file! In addition to errors and warnings, it will report all files that were read in, including version numbers for document class files and packages, pages processed, and, at the end, resources used. Only a few relevant items will be mentioned here, but in a paper based on an earlier talk [1], instructions are given for how to undertake serious debugging.

Conventions

In order to avoid overfull lines, error and warning messages shown here may be broken to fit the narrow columns of the *TUGboat* style. Many error messages output by LaTeX will consist of several lines, the first being the message, and the next showing the number of the line on which the error is identified along with the content of that line, up through the error text. A following line, indented so that it, with the numbered line, completes the line as it appears in the input.

Although we will deal here mostly with details, please remember that the basic concept of LaTeX is to separate content from structure.

Basic structure: Commands, modes, and scope

Here we deal with some fundamentals of LaTeX.

Commands Instructions are given to (LA)TEX by means of commands, or "control sequences", which by default begin with a backslash (\). There are two varieties: those which consist of the backslash followed by one non-letter character ("control symbol"), and those of one or more letters ("control words") in which only letters (upper- or lowercase A–Z) are permitted (no digits or special characters). A control word may

[1] `tex.stackexchange.com`

doi.org/10.47397/tb/44-2/tb137beeton-basic

have one or more arguments (`\title{...}`) or stand by itself (`\alpha`). A "standalone" control word will be terminated by a space or any other non-letter. But a space after a control symbol will appear as a space in the output. Several control symbols are predefined to produce their own character in the output: `\#`, `\%`, `\$`, `\&`. For example, `\$` produces '$'.

A user can define new commands, or assign new meanings to existing commands. LaTeX provides `\newcommand` to create a brand-new definition. `\newcommand` checks to make sure that the command name hasn't been used before, and complains if it has. (The basic TeX `\def` *does not.*) If it's necessary to redefine a command that already exists, the recommended way is to use `\renewcommand` — but be sure you know what you're doing. For example, redefining `\par` is chancy, as LaTeX uses this "under the covers" for many different formatting adjustments, and it's very easy to mess things up.

Single-letter commands are also bad candidates for (re)definition by users, as many of them are predefined as accents or forms of letters not usual in English text; `\i` might very well occur with (or without) an accent in a references list. For (a bad) example, consider the author Haïm Brezis:

```
\renewcommand{\i}{\ensuremath{\sqrt{-1}}}
Brezis, Ha\"{\i}m
```
$\Longrightarrow$ Brezis, Haïm

Single-digit commands (`\0`, `\1`, etc.) are not predefined in core LaTeX, so are available for ad hoc use.

Environments An *environment* is a block of material between

`\begin{`⟨*env-name*⟩`} ... \end{`⟨*env-name*⟩`}`.

The environment name must match at beginning and end; if it doesn't, this error is reported in the log file and on the terminal:

```
! LaTeX Error: \begin{xxx} on input line nn
               ended by \end{yyy}
```

Most environments can be nested, but the proper sequence must be maintained.

Other commands are available to provide new definitions — `\NewDocumentCommand`, `\NewEnvironment`, `\NewDocumentEnvironment` and similar ones for redefinitions. For details on these, consult a current reference.

Modes Generally speaking, the current mode identifies where you are on the (output) page, but here we will take a point of view based on the input/source file.

There are three modes: vertical, horizontal and math.

Starting after `\documentclass` or after a blank line or an explicit `\par`, LaTeX is in vertical mode. Certain operations are best launched in vertical mode; more about this later.

Starting to input ordinary text is one way to enter horizontal mode. Other transitions from vertical to horizontal mode are `\indent`, `\noindent` and `\leavevmode`. Within horizontal mode, multiple consecutive spaces are treated as a single space; *consecutive* is essential here. An end-of-line (EOL) is treated as a space, even though it's not explicitly visible in the source file; a GUI that wraps lines may or may not (usually not) insert an EOL, and different operating systems define an EOL differently, but such differences are taken care of by the TeX engine. Spaces at the beginning of a line are ignored. More about spaces later on.

The third mode, math, can be embedded in-line in text or set as display material in vertical mode. Inline math is wrapped in `$` signs or surrounded by `\(...\)`. An unnumbered one-line display can be indicated by `\[...\]`. Multi-line math displays are best entered using the environments provided by the amsmath and mathtools packages. (Refer to the user documentation. mathtools loads amsmath, so it's not necessary to load both.) A math display is usually a continuation of the preceding paragraph, so don't leave a blank line between a display and the preceding text; to do so can result in an unwanted page break.

Within math mode, blank lines are not allowed; this was a decision made by Knuth, to catch unintentional input lapses, since math never continues across a paragraph break.

Scope Along with modes, there is the concept of scope, making it possible to localize definitions and operations.

Math mode is one instance of scope; certain characters and operations are valid only within math, and others are invalid there. Within text, math usually begins and ends with `$`, and these must be matched. Display math breaks the flow of text; closing a display returns to text mode unless followed by a blank line or `\par`. More about math later.

Another way of delimiting scope is to wrap the material in braces: `{...}`. Within this scope, the meaning of a command may be changed for temporary effect; the definition in effect before the opening brace will be restored as soon as the closing brace is digested. Instead of a brace pair, the commands `\begingroup...\endgroup` have the same effect.

Another way to have a scoped environment is to pack the material in a "box". This may be a

`minipage`, `\mbox` or `\parbox`. Other boxes are defined in packages like tcolorbox.

Some environments (not all) are defined to be a scope. One such is the `theorem` environment, inside which text is italic; when the theorem ends, the text style automatically reverts to the document default.

Spacing in text

A goal of high-quality typesetting is even spacing in text. This is really possible only with ragged-right setting, where spaces are "natural width". But even margins are usually preferred, so TEX is designed to optimize spacing in that context.

In U.S. documents, spaces that end sentences are wider than interword spaces. This is not true for documents in other languages, and can be turned off with `\frenchspacing`. But in academic documents, frequent abbreviations can make it difficult to tell where sentences end. To avoid a too-wide space after an abbreviation, follow it by "\␣" (backslash-space):

```
abc vs. xyz  (abc vs. xyz) vs.
abc vs.\ xyz (abc vs. xyz)
```

If the line shouldn't break after the abbreviation, follow the period by `~`: `seen on p.~23.` (seen on p. 23.)

A similar, but reverse, situation can occur when an uppercase letter is followed by a period. This is assumed to be the initial of a name; it usually is, and an ordinary interword space is set. But sometimes the uppercase letter is at the end of an acronym, and that ends a sentence. In such a case, add `\@` before the period, and it will restore the wider end-of-sentence space.

All this boils down to a simple rule: Except at the end of a sentence (and to a lesser extent after other punctuation symbols or within math), all spaces within the same line should be the same width. If they're not, something is fishy.

Spurious spaces Multiple spaces can infiltrate a source file in several ways, but the overwhelming majority are the result of trying too hard to define commands in such a way that they are visually pleasing (and easily readable). For example:

```
\newcommand{\abc}{
   \emph{abc def}
}
```

With this, the input "`word \abc\ word`" results in "word *abc def* word" with extra spaces inserted by our `\abc` command. The offending spaces can be evicted by inserting `%` where it will "hide" an EOL:

```
\newcommand{\abc}{%
   \emph{abc def}%
}
```

to produce the desirable "word *abc def* word". The `%` character starts a comment, i.e., ignores the rest of the input line, including the EOL.

Another source of extra spaces in the output can be caused by the presence of multiple consecutive elements that aren't part of the main text, like footnotes or index entries:

```
An important topic\index{abc}
\index{def}
\index{xyz}
is indexed several ways.
```

An important topic is indexed several ways.

Here, the EOL effect has again occurred (after "topic"), and these spaces are no longer contiguous. Again the `%` comes to the rescue:

```
An important topic\index{abc}%
\index{def}%
\index{xyz}
is indexed several ways.
```

An important topic is indexed several ways.

Do remember to leave *one* space.

Sometimes using a % is a bad idea Remember that a space terminates a control word and it's then discarded; that's one place where it's not necessary to input a `%`. But there are places where adding a `%` can really cause trouble.

After defining any numeric value, TEX will keep looking for anything else that can be interpreted as numeric, so if a line ends with `\xyz=123`, no `%` should be added. Or, if setting a rubber length (glue), say `\parskip=2pc`, TEX will keep looking for `plus` or `minus`; a better "stopper" is an empty token, `{}`. (If "plus" or "minus" is there and happens to be actual text, a confusing error message will be produced, but that is rare, and beyond the scope of this discussion.)

Really unexpected extra spaces Other possibilities exist that aren't so predictable. Here's one that was the subject of an online question. A text with `\usepackage{colorbox}` (it can also happen with tcolorbox) had a colorized letter surrounded by spaces in the middle of a word. Oo p s! A small frame was applied around the colored element by the package:

```
\usepackage{colorbox}
\newcommand{\pink}[1]{%
  \colorbox{red!20}{#1}}
Oo\pink{p}s!
```

Oo p s!

Explicitly omitting the buffer inside the frame was the solution provided by the package documentation:

```
\renewcommand{\pink}[1]{{%
  \fboxsep=0pt
  \colorbox{red!20}{#1\strut}}}
Oo\pink{p}s!
```

Oops!

I added the `\strut` so that the color would be obvious above and below the highlighted element, rather than covering only the "p". While this isn't really a newbie problem, it's wise to be aware that such possibilities exist, and be ready in such cases to seek expert assistance.

Paragraph endings and vertical mode

The end of a paragraph is a transition from horizontal to vertical mode. A blank line or `\par` will accomplish this transition. It's important to be aware of what mode you're in, since some operations are best performed in vertical mode; the most important is the insertion of floats (figures, tables, algorithms).

Another important consideration is that some features of text are not "frozen" until a paragraph is ended. One important feature is the vertical spacing of baselines, which depends on the font size. Too many newbies try to end a paragraph with a double backslash, resulting in horrors like the following.

```
\Huge Texts with inconsistent descenders
can result in surprises when the font
size changes without a proper paragraph
ending.\\
```

Texts with inconsistent descenders can result in surprises when the font size changes without a proper paragraph ending.

Some environments (but not *all*) are defined with a paragraph break at the end. A problem such as the one shown here won't result in an error or warning message, so adding a proper paragraph break is the proper correction.

The vertical space between paragraphs is determined by the value of `\parskip`; this is set in the document class, but can be reset as needed. But often, it's convenient to add occasional extra space between paragraphs explicitly; this is done with `\vspace` or `\vskip` while in vertical mode (that is, after the blank line or `\par` that ends a paragraph).

The double backslash What a paragraph does *not* end with is the control symbol `\\`. `\\` does end a line. It is the designated command to end lines in tables, poetry, multi-line math environments, and some other situations. But it does not end a paragraph and can trigger a number of error or warning messages.

If `\\` is alone on a line in vertical mode, this error is reported:

```
! LaTeX Error:
    There's no line here to end.
```

Further, if the `\\` is preceded by a (typed) space, in addition to the warning, there may be an extra, unwanted, blank line in the output.

If a line ending with `\\` is very short:

```
Underfull \hbox (badness 10000)
    in paragraph at lines ...
```

This may be okay, but check.

If `\\` is followed by bracketed text, as in `[stuff to be typeset]`, the result will be the mysterious `! Missing number, treated as zero.` For `\\`, a following (optional) `[...]` is defined to indicate a vertical distance to be skipped; insert `\relax` before the opening bracket.

If extra vertical space *is* wanted after a line broken with `\\`, it can be added by inserting an optional rubber length (usually just a dimension), wrapped in brackets: `\\[`⟨*value*⟩`]`. If such a bracketed expression is really meant to be typeset, it must be preceded by `\relax`.

`\newline` is often a reasonable alternative to break a line.

Font changes

Font changes are a time-honored method of communicating shades of meaning or pointing out distinct or particularly important concepts. Many such instances are built into document classes and packages; for example, theorems are set in an *italic* font, section headings in **bold**, and some journals set figure captions in sans serif to distinguish them from the main text.

LaTeX provides two distinct methods for making font changes. Commands of one class take an argument and limit the persistence of the change to the content of that argument; these have the form of `\textbf{...}` for **bold**, `\textit{...}` for *italic*, etc. The other class sets the font style so that it will not change until another explicit change is made, or it is limited by the scope of an environment; some examples are `{\itshape...}`, `{\bfseries...}`, and `{\sffamily...}`. These commands are best looked up in a good user guide.

Several font-changing commands do different things depending on the context. `\emph{...}` will switch to italic if the current text is upright, or to upright if the current text is italic. Within math, `\text{...}` will set a text string in the same style as the surrounding text; thus, within a theorem, `\text{...}` will be set in italic. If this string should always be upright, like function words, `\textup{...}` should be used instead.

Basic TEX defined two-letter names for most font styles. All of these are of the persistent type. They should be avoided with LATEX, as some of the LATEX forms provide improvements, such as a smoother transition between italic and upright type.

Math

Math is always a scoped environment. If started, it must be ended explicitly and unambiguously. Within text, math begins and ends with `$`. LATEX also provides `\(...\)` for in-text math, but most users stick with the `$`. Many different display environments are defined by the packages amsmath and mathtools, and it is worthwhile to learn them by reading the user guides.

Within math, all input spaces are meaningless to (LA)TEX; they can be entered in the source file as useful to make it readable to a human. Blank lines, however, are considered errors. In both in-text math and displays, the error message will be

```
! Missing $ inserted.
```

This will also result if in-text math is not ended before the paragraph ends, or if a math-only symbol or command is found outside of math mode.

If a blank line occurs in a multi-line display environment from amsmath, the *first* error message will be

```
! Paragraph ended before ⟨env-name⟩
    was complete.
<to be read again>
```

This will be followed by *many* more error messages, all caused by the first. These will be confusing and misleading. Always fix the problem identified by the first error and ignore the rest; they will disappear once the first error is fixed; here, by removing the blank line.

If the appearance of a blank line is wanted for readability, instead use a line with just a `%`.

As with all environments, the `\end` name must exactly match the name specified at `\begin`. A "shorthand" for a single-line, unnumbered display is `\[...\]`. The environments designed for multi-line displays should not be used for a single-line display.

Although LATEX provided `eqnarray` as a display environment, don't use it. If the display is numbered and the equation is long, the equation can be overprinted by the equation number.

Tables, figures, and other floats

The allowed number of floats, their positions on a page, and the spacing around and between them is defined by the document class. So if something doesn't work as you expect (hope for?), any potential helper will insist on learning what document class is being used.

Input for a float must appear in the source file while there is still enough space on the output page to fit it in. In particular, on two-column pages, a `figure*` or `table*` must occur in the source *before* anything else is set on the page. LATEX's core float handling does not allow full-width floats to be placed anywhere but at the top of a page; some packages extend this capability, but those won't be discussed here.

Here are the defaults for the basic `article` class.

- Total number of floats allowed on a page with text: 3.
- Number of floats allowed at top of page: 2. Percentage of page allowed for top-of-page floats: 70%.
- Number of floats allowed at bottom of page: 1. Percentage of page allowed for bottom-of-page floats: 30%.
- Minimum height of page required for text: 20%.
- Minimum height of float requiring a page by itself: 50%.

The reference height is `\textheight`. That is, the height of page headers and footers is excluded.

If an insertion is small, must be placed precisely and fits in that location, don't use a float. `\includegraphics` or one of several available table structures should be used directly, often wrapped in `\begin{center} ... \end{center}` (Within a float, use `\centering` instead.)

The wrapfig package supports cut-in inserts at the sides of a page or column. Refer to the documentation for details.

By tradition, captions are applied at the top of tables and the bottom of figures. If an insertion is not a float, the usual `\caption` can't be used. Instead, `\usepackage{caption}` and the command `\captionof`.

The document class and preamble

When embarking on a new document, start by choosing the document class. If the goal is publication in a

particular journal, check the publisher's instructions to see what is required. Many popular journal classes are available from CTAN.[2]

If the project is a thesis or dissertation, find out the special requirements, and if your institution provides a tailored class, obtain a copy. Try to determine whether it is actively maintained, and if there is local support. Read the documentation.

It is the responsibility of the document class to define the essential structure of the intended document. If the document you are preparing differs in essential ways from what is supported by the document class, the time to get help is *now*.

There will be features not natively supported by the document class; for example, the choice of how to prepare a bibliography may be left to the author. This is why packages have been created.

Organizing your document Most packages are loaded in the preamble. There is one exception: `\RequirePackage`. This is usually specified before `\documentclass`, and is the place where certain special options should be loaded.

Some authors create a preamble that is suitable for one document, then use the same preamble for their next document, adding more packages as they go. And some unwitting newbies "adopt" such secondhand "templates" without understanding how they were created. *Don't do it!*

Start with a suitable document class and add features (packages, options, and definitions) as they become necessary. Organize the loading of packages into logical groups (all fonts together, for example), and be careful not to load a package more than once; if options are needed, any loaded with a non-first `\usepackage` will be ignored. Some packages automatically load other packages; for example, mathtools loads amsmath and amssymb loads amsfonts. And, very important, pay attention to the order of package loading: hyperref must be loaded (almost) last; the few packages that must come after hyperref are all well documented.

Read the documentation.

Processing the job

Once the file is created, it's time to produce output. There are several engines to choose from: pdfLaTeX, XeLaTeX, and LuaLaTeX. These can be run interactively from the command line, or initiated from an editor. Assuming there are no errors, how many times a document must be processed to produce the final output depends on what features it contains.

In particular, if any cross-references or `\cite`s are present, this information is written out to an `.aux` file; information for a table of contents is written to a `toc` file, and other tables are also possible. The bibliography must be processed by a separate program (and its log checked for errors) with the reformatted bib data written out to yet another file. Then LaTeX must be run (at least) twice more — once to read in the `.aux` and other secondary files and include the bibliography and resolved cross-references, and the second time to resolve the correct page numbers (which will change when the TOC and similar bits are added at the beginning).

All this assumes that there are no errors. Errors will be recorded in the log file. Learn where the log file is located, and make a habit of referring to it. Warnings, such as those for missing characters, will also be recorded there, but may not be shown online:

`Missing character: There is no` ⟨*char*⟩ `in font` ⟨*font*⟩`!`

In the log, some errors may appear with closely grouped line numbers. If so, and the first is one that interrupts the orderly processing of a scoped environment, following errors may be spurious. So fix the first error and try processing before trying to understand the others; often, they may just go away.

Good luck. With practice comes understanding.

Oh... Remember to read the documentation.

Acknowledgment

Thanks to samcarter, Mikael Sundquist, and (as always) Karl Berry for suggestions and for finding and exterminating my typos. I can find them in other people's writing, but often not in my own.

References

[1] Barbara Beeton. Debugging LaTeX files — Illegitimi non carborundum, *TUGboat* 38:2, 159–164 (2017). `tug.org/TUGboat/tb38-2/tb119beet.pdf`

◇ Barbara Beeton
TUGboat
Providence, RI, USA
`bnb (at) tug dot org`

[2] `ctan.org/search`