\NewDocumentCommand *versus* \newcommand *versus* ...

Joseph Wright

Creating new document commands in LaTeX has traditionally been the job of \newcommand. This lets you create a command with mandatory arguments, and also support a first optional argument. However, it can't create more complex commands: LaTeX uses stars, multiple optional arguments, and plenty more. To define commands using such syntaxes, the kernel itself uses lower-level TeX programming. But this is opaque to many users, and a variety of packages have been created to ease the burden.

Over the last decade, the LaTeX team have developed xparse, a generic document command parser, as a way to unify many ideas and provide a single consistent way to create document commands. The bulk of that code has now been moved to the LaTeX kernel, and in a parallel article (starts on the preceding page) I've provided some ideas about how best to exploit that.

In this article, I want to look at a related issue: why to use this 'xparse' approach, and how it compares to existing solutions, both in the LaTeX kernel and the wider package sphere. Here, I'm going to avoid talking about 'simple' shortcuts (things such as \newcommand\myname{Joseph}): these are best left to \newcommand. Instead, I want to deal with commands which take arguments and have some element of 'programming' to them.

What I'll seek to highlight here is that using \NewDocumentCommand, we get a single consistent and reliable way to create a variety of commands. There's no need to worry about clashes between approaches, and it all 'just works'.

1 Preliminaries: Protected commands and optional arguments

Before we start, a couple of things are worth mentioning. First, there is the idea of 'protected' commands. In some places, we need commands not to 'expand' (turn into their definition). With a modern TeX system, that can be arranged by the engine itself (pdfTeX or similar), using ε-TeX's \protected primitive (built-in). The LaTeX kernel doesn't use that mechanism in \newcommand, but lots of other tools do. I'm going to assume that we want to make protected commands unless I mention otherwise. Almost always, unless you are creating a 'shortcut' for some text, you want your commands to be protected.

The second thing to note is that TeX itself has no concept of optional arguments, so they are always arranged using some clever look-ahead code. In xparse, nested optional arguments are handled automatically, but again, \newcommand and similar do not do that.

2 The kernel: *versus* \newcommand

The kernel's \newcommand can, as I've said, create commands with multiple mandatory arguments but with only one optional one. A simple example:

```
\newcommand\foo[3][default]{%
    Code perhaps using #1 and
    definitely using #2 and #3%
}
```

We can of course create an equivalent command using \NewDocumentCommand:

```
\NewDocumentCommand\foo{+O{default} +m +m}{%
    Code perhaps using #1 and
    definitely using #2 and #3%
}
```


doi.org/10.47397/tb/42-1/tb130wright-newdoccmd

You may notice that I've used `+m` for both of the mandatory arguments, as that matches `\newcommand`: the arguments can accept paragraphs (is `\long`, in TEX terms). With `\newcommand`, all arguments either accept `\par` or do not: with `\NewDocumentCommand` we can select on a per-argument level what happens.

The optional argument with a default works using `O{default}`, and the result will be the same functionality as `\newcommand`. We gain the idea that nested optional arguments are parsed properly, some better error messages if we use `\foo` incorrectly, and an engine-robust definition of `\foo`.

We can't do a lot more with `\newcommand`, so rather than try to show other `\NewDocumentCommand` features here, we'll first consider how we might make more complex syntaxes using just the classical LATEX kernel.

3 ... *versus* `\def`: The primitive

Using the TEX primitive `\def`, plus the kernel internal commands `\@ifstar` and `\@ifnextchar`, we can construct more complex syntaxes. For example, let's create the syntax for `\section`: a star, an optional argument and a mandatory one. I'll assume we have `@` defined as a letter here. I'm also going to pass the presence of a star as the text `true` or `false`, as it makes things clearer.

```
\newcommand\section{%
    \@ifstar
      {\section@auxi{true}}
      {\section@auxi{false}}%
}
\def\section@starred#1{%
    \@ifnextchar[%]
      {\section@auxii{#1}}
      {\section@auxii{#1}[]}%
}
\long\def\section@auxii#1[#2]#3{%
    % Here:
    % #1 is "true"/"false" for a star
    % #2 is the optional argument
    % #3 is the mandatory argument
}
```

As you can see, this is a bit tricky already, and it doesn't cover the case where we want to have the optional argument default to the mandatory one, when it's not given. It also doesn't allow for nested optional arguments, and it's not engine-robust. We might of course use more complex paths for the star: we could have independent routes.

Using `\NewDocumentCommand`, things are much easier:

```
\NewDocumentCommand\section{s +O{#3} +m}{%
  % Here:
  % #1 is "true"/"false" for a star
  % #2 is the optional argument
  % #3 is the mandatory argument
}
```

The minor difference now is that `#1` is a special token that we can test for truth using `\IFBooleanTF`. I've also allowed for the optional argument picking up the mandatory one (`#3`), when it's not given.

We could make more complex examples, but the bottom line is: using `\NewDocumentCommand`, we are going to have simple one-line interface descriptions, and the behind-the-scenes TEX argument parsing is hidden away.

4 ... *versus* `\newrobustcmd`: etoolbox

The etoolboxpackage offers `\newrobustcmd` as a complement to `\newcommand`. It provides *exactly the same* interface as `\newcommand`, except it uses ε-TEX to make engine-protected commands. Here's an interface point of view, there's nothing new here.

5 ... *versus* `\newcommandtwoopt`: twoopt

The twoopt package supports a syntax similar to `\newcommand` but for creating two optional arguments. We'll take an example from its documentation:

```
\newcommandtwoopt\bsp[3][AA][BB]{%
    \typeout{\string\bsp: #1,#2,#3}%
}
```

This is reasonably clear: we have an optional argument `#1`, and optional argument `#2` and a mandatory argument `#3`. The two optional arguments each here have a default.

How does this look with `\NewDocumentCommand`?

```
\NewDocumentCommand\bsp{+O{AA} +O{BB} +m}{%
    \typeout{\string\bsp: #1,#2,#3}%
}
```

You'll see that we stay consistent here: the same syntax is used to create one, two or even more optional arguments. I wouldn't recommend using multiple optional arguments in most cases, but when we do, it's a lot easier using `\NewDocumentCommand`.

One thing that `\NewDocumentCommand` can do, but twoopt *cannot*, is create optional arguments that are not in the first or second positions. With twoopt, that would require either the TEX coding we've already seen, or using a different tool again.

6 ... *versus* \withsuffix: suffix

The suffix package allows one to extend an existing command to look for an optional token ('suffix') immediately after the command name. Taking a simple example from StackExchange (https://tex.stackexchange.com/a/4388), we start with

```
\newcommand\foo{blah}
\WithSuffix\newcommand\foo*{blahblah}
```

which translates to

```
\NewDocumentCommand\foo{s}{%
   \IFBooleanTF{#1}
     {blah}
     {blahblah}
}
```

This means we only need one line for the interface set up, and don't need, for example, to split up grabbing optional arguments into two different places (as in the previous example with \section).

7 ... *versus* \newcommandx: xargs

The xargs package is perhaps the most complete approach to extending \newcommand as far as optional arguments are concerned. It provides \newcommandx, which has the same syntax as \newcommand but where the second optional argument is a key–value list, which then describes which arguments are optional, and what their defaults are. Taking an example from the documentation:

```
\newcommandx*\coord[3][2=1,3=n]{%
  (#2_{#1},\ldots,#2_{#3})}
```

would create a command with two optional arguments, #2 and #3 (each with defaults), leaving #1 mandatory. Translating into \NewDocumentCommand syntax might make that clearer!

```
\NewDocumentCommand\coord{m O{1} O{n}}{%
    (#2_{#1},\ldots,#2_{#3})%
}
```

The xargs package has the idea of usedefault, which allows [] to be the same as [default]. That's not something xparse does, as it is pretty confusing: what happens when you want an empty optional argument? This links to something I've said before: avoid consecutive optional arguments *unless* the second is dependent on the first.

8 ... *versus* newcommand.py: newcommand

Stepping outside of TEX itself, Scott Pakin's Python script newcommand.py provides a description language somewhat like xparse, and converts this into a 'template' of TEX code, allowing a 'fill in the blanks' approach to creating commands. It can cover several of the ideas that xparse can, including a few that will *not* be migrated to the LATEX kernel. It can also set up a command taking more than 9 arguments, but that's always going to be tricky as a user.

What is important is that using a script means we have to work in two steps, and it's hard to see what's happening from the TEX source. It also doesn't offer anything that the kernel doesn't already do: no protected commands, no nested optional arguments, no improved error messages. So in many ways this is using techniques we've already seen, just made a little more accessible, at least if you have Python installed.

9 ... *versus* \NewEnviron: environ

As well as document commands, the xparse syntax can be used to create document *environments*: the same relationship we have between \newcommand and \newenvironment. What people sometimes want to do is grab an entire document environment body and use it like a command argument. Classically, one does that using the environ package. Again, taking an example from the documentation:

```
\NewEnviron{test}{%
  \fbox{\parbox{1.5cm}{\BODY}}\color{red}
  \fbox{\parbox{1.5cm}{\BODY}}%
}
```

would grab all of the body of the environment test and typeset it twice, the first time in red. That is, the environment body is saved as \BODY.

Using \NewDocumentEnvironment, we have a syntax similar to \newenvironment

```
\NewDocumentEnvironment{test}{+b}{%
  \fbox{\parbox{1.5cm}{#1}}\color{red}
  \fbox{\parbox{1.5cm}{#1}}%
}{}
```

with the argument grabbed in the normal way as (here) #1. We can therefore have 'real' arguments first, then grab the body.

10 Summary

Using the tools set up in \NewDocumentCommand, we can have a consistent way of creating a wide range of document commands. Rather than use a mixture of tools, from the kernel, the TEX engine, and the package sphere, it is far preferable to use the single interface of \NewDocumentCommand for defining new commands today.

◇ Joseph Wright
Northampton, United Kingdom
joseph dot wright (at)
morningstar2.co.uk