The newcommand.py utility[*]

Scott Pakin
scott+nc@pakin.org

2010/06/01

Abstract

LaTeX's `\newcommand` is fairly limited in the way it processes optional arguments, but the TeX alternative, a batch of `\def`s and `\futurelet`s, can be overwhelming to the casual LaTeX user. `newcommand.py` is a Python program that automatically generates LaTeX macro definitions for macros that require more powerful argument processing than `\newcommand` can handle. `newcommand.py` is intended for LaTeX advanced beginners (i.e., those who know how to use `\newcommand` but not internal LaTeX 2$_\varepsilon$ macros like `\@ifnextchar`) and for more advanced users who want to save some typing when defining complex macros.

1 Introduction

LaTeX's `\newcommand` is a rather limited way to define new macros. Only one argument can be designated as optional, it must be the first argument, and it must appear within square brackets. Defining macros that take multiple optional arguments or in which an optional argument appears in the middle of the argument list is possible but well beyond the capabilities of the casual LaTeX user. It requires using TeX primitives such as `\def` and `\futurelet` and/or LaTeX 2$_\varepsilon$ internal macros such as `\@ifnextchar`.

`newcommand.py` is a Python program that reads a specification of an argument list and automatically produces LaTeX code that processes the arguments appropriately. `newcommand.py` makes it easy to define LaTeX macros with more complex parameter parsing than is possible with `\newcommand` alone. Note that you do need to have Python installed on your system to run `newcommand.py`. Python is freely available for download from `http://www.python.org/`.

To define a LaTeX macro, one gives `newcommand.py` a macro description written in a simple specification language. The description essentially lists the required and optional arguments and, for each optional argument, the default value. The next section of this document describes the syntax and provides some examples, but for now, let's look at how one would define the most trivial macro possible, one that takes no arguments. Enter the following at your operating system's prompt:

---

[*] newcommand.py has version number 2.0, last revised 2010/06/01.

```
newcommand.py "MACRO trivial"
```

(Depending on your system, you may need to prefix that command with "`python`".) The program should output the following LaTeX code in response:

```
% Prototype: MACRO trivial
\newcommand{\trivial}{%
  % Put your code here.
}
```

Alternatively, you can run `newcommand.py` interactively, entering macro descriptions at the "`% Prototype:`" prompt:

```
% Prototype: MACRO trivial
\newcommand{\trivial}{%
  % Put your code here.
}
% Prototype:
```

Enter your operating system's end-of-file character (Ctrl-D in Unix or Ctrl-Z in Windows) to exit the program.

While you certainly don't need `newcommand.py` to write macros that are as trivial as `\trivial`, the previous discussion shows how to run the program and the sort of output that you should expect. There will always be a "`Put your code here`" comment indicating where you should fill in the actual macro code. At that location, all of the macro's parameters—both optional and required—will be defined and can be referred to in the ordinary way: `#1`, `#2`, `#3`, etc.

2 Usage

As we saw in the previous section, macros are defined by the word "`MACRO`" followed by the macro name, with no preceding backslash. In this section we examine how to specify increasingly sophisticated argument processing using `newcommand.py`.

2.1 Required arguments

Required arguments are entered as `#1`, `#2`, `#3`, . . . , with no surrounding braces:

```
% Prototype: MACRO required #1 #2 #3 #4 #5
\newcommand{\required}[5]{%
  % Put your code here.
  % You can refer to the arguments as #1 through #5.
}
```

Parameters must be numbered in monotonically increasing order, starting with `#1`. Incorrectly ordered parameters will produce an error message:

```
% Prototype: MACRO required #1 #3 #4
                               ^
newcommand.py: Expected parameter 2 but saw parameter 3.
```

2.2 Optional arguments

Optional arguments are written as either "`OPT[`⟨*param*⟩`={`⟨*default*⟩`}]`" or "`OPT(`⟨*param*⟩`={`⟨*default*⟩`})`". In the former case, square brackets are used to offset the optional argument; in the latter case, parentheses are used. ⟨*param*⟩ is the parameter number (`#1`, `#2`, `#3`, ...), and ⟨*default*⟩ is the default value for that parameter. Note that curly braces are required around ⟨*default*⟩.

```
% Prototype: MACRO optional OPT[#1={maybe}]

\newcommand{\optional}[1][maybe]{%
  % Put your code here.
  % You can refer to the argument as #1.
}
```

Up to this point, the examples have been so simple that `newcommand.py` is overkill for entering them. We can now begin specifying constructs that LaTeX's `\newcommand` can't handle, such as a parenthesized optional argument, an optional argument that doesn't appear at the beginning of the argument list, and multiple optional arguments:

```
% Prototype: MACRO parenthesized OPT(#1={abc})

\makeatletter
\newcommand{\parenthesized}{%
  \@ifnextchar({\parenthesized@i}{\parenthesized@i({abc})}%
}

\def\parenthesized@i(#1){%
  % Put your code here.
  % You can refer to the argument as #1.
}
\makeatother
```

```
% Prototype: MACRO nonbeginning #1 OPT[#2={abc}]

\makeatletter
\newcommand{\nonbeginning}[1]{%
  \@ifnextchar[{\nonbeginning@i{#1}}{\nonbeginning@i{#1}[{abc}]}%
}

\def\nonbeginning@i#1[#2]{%
  % Put your code here.
  % You can refer to the arguments as #1 and #2.
}
\makeatother
```

```
% Prototype: MACRO multiple OPT[#1={abc}] OPT[#2={def}]

\makeatletter
```

```
\newcommand{\multiple}[1][abc]{%
  \@ifnextchar[{\multiple@i[{#1}]}{\multiple@i[{#1}][{def}]}%
}

\def\multiple@i[#1][#2]{%
  % Put your code here.
  % You can refer to the arguments as #1 and #2.
}
\makeatother
```

The template for optional arguments that was shown on the preceding page stated that optional arguments contain a “⟨*param*⟩`={`⟨*default*⟩`}`” specification. In fact, optional arguments can contain *multiple* “⟨*param*⟩`={`⟨*default*⟩`}`” specifications, as long as they are separated by literal text:

```
% Prototype: MACRO multiopt OPT(#1={0},#2={0})

\makeatletter
\newcommand{\multiopt}{%
  \@ifnextchar({\multiopt@i}{\multiopt@i({0},{0})}%
}

\def\multiopt@i(#1,#2){%
  % Put your code here.
  % You can refer to the arguments as #1 and #2.
}
\makeatother
```

In that example, `\multiopt` takes an optional parenthesized argument. If omitted, it defaults to `(0,0)`. If provided, the argument must be of the form “`(`⟨x⟩`,`⟨y⟩`)`”. In either case, the comma-separated values within the parentheses are parsed into `#1` and `#2`. Contrast that with the following:

```
% Prototype: MACRO multiopt OPT(#1={0,0})

\makeatletter
\newcommand{\multiopt}{%
  \@ifnextchar({\multiopt@i}{\multiopt@i({0,0})}%
}

\def\multiopt@i(#1){%
  % Put your code here.
  % You can refer to the argument as #1.
}
\makeatother
```

The optional argument still defaults to `(0,0)`, but `#1` receives *all* of the text that lies between the parentheses; `\multiopt` does not parse it into two comma-separated values in `#1` and `#2`, as it did in the previous example.

The ⟨*default*⟩ text in an `OPT` term can reference any macro parameter introduced before the `OPT`. Hence, the following defines a macro that accepts a required argument followed by an optional argument. The default value of the optional argument is the value provided for the required argument:

```
% Prototype: MACRO paramdefault #1 OPT[#2={#1}]
\makeatletter
\newcommand{\paramdefault}[1]{%
  \@ifnextchar[{\paramdefault@i{#1}}{\paramdefault@i{#1}[{#1}]}%
}

\def\paramdefault@i#1[#2]{%
  % Put your code here.
  % You can refer to the arguments as #1 and #2.
}
\makeatother
```

2.3 Literal text

In addition to required and optional parameters, it is also possible to specify text that must appear literally in the macro call. Merely specify it within curly braces:

```
% Prototype: MACRO textual #1 { and } #2 {.}
\makeatletter
\newcommand{\textual}[1]{%
  \textual@i{#1}%
}

\def\textual@i#1 and #2.{%
  % Put your code here.
  % You can refer to the arguments as #1 and #2.
}
\makeatother
```

A macro such as `\textual` can be called like this:

```
\textual {Milk} and {cookies}.
```

Actually, in that example, because both `Milk` and `cookies` are delimited on the right by literal text, TEX can figure out how to split `\textual`'s argument into `#1` and `#2` even if the curly braces are omitted:

```
\textual Milk and cookies.
```

2.4 Starred macros

The names of some LaTeX macros can be followed by an optional “`*`” to indicate a variation on the normal processing. For example, `\vspace`, which introduces a given amount of vertical space, discards the space if it appears at the top of the page. `\vspace*`, in contrast, typesets the space no matter where it appears. `newcommand.py` makes it easy for users to define their own starred commands:

```
% Prototype: MACRO starred * #1 #2
\makeatletter
\newif\ifstarred@star
\newcommand{\starred}{%
  \@ifstar{\starred@startrue\starred@i*}{\starred@starfalse\starred@i*}%
}

\def\starred@i*#1#2{%
  \ifstarred@star
    % Put code for the "*" case here.
  \else
    % Put code for the non-"*" case here.
  \fi
  % Put code common to both cases here (and/or above the \ifstarred@star).
  % You can refer to the arguments as #1 and #2.
}
\makeatother
```

Note that unlike the generated code shown up to this point, the code for starred macros includes *multiple* placeholders for user code.

The “`*`” in a starred macro does not have to immediately follow the macro name; it can appear anywhere in the macro specification. However, `newcommand.py` currently limits macros to at most one asterisk.

Embedding an asterisk within curly braces causes it to be treated not as an optional character but as (required) literal text. Contrast the preceding example with the following one:

```
% Prototype: MACRO starred {*} #1 #2
\makeatletter
\newcommand{\starred}{%
  \starred@i%
}

\def\starred@i*#1#2{%
  % Put your code here.
  % You can refer to the arguments as #1 and #2.
}
\makeatother
```

The asterisk in that definition of `\starred` must be included in every macro invocation or TeX will abort with a “`Use of \starred@i doesn't match its definition`” error.

2.5 More than nine arguments

TEX imposes a limit of nine arguments per macro. Internally, "`#`" is expected to be followed by exactly one digit, which means that "`#10`" refers to argument `#1` followed by the character `0`. Fortunately, it's rare that a macro needs more than nine arguments and rarer still that those arguments are not better specified as a list of ⟨*key*⟩=⟨*value*⟩ pairs, as supported by the keyval package and many other LATEX packages.

If large numbers of arguments are in fact necessary, `newcommand.py` does let you specify them. The trick that the generated code uses is to split the macro into multiple macros, each of which takes nine or fewer arguments and stores the value of each argument in a variable that can later be accessed. Because digits are awkward to use in macro names, `newcommand.py` uses roman numerals to name arguments in the case of more than nine arguments: `\`⟨*name*⟩`@arg@i` for `#1`, `\`⟨*name*⟩`@arg@ii` for `#2`, `\`⟨*name*⟩`@arg@iii` for `#3`, `\`⟨*name*⟩`@arg@iv` for `#4`, and so forth. The following example takes 14 required arguments and one optional argument (which defaults to the string "`etc`"):

```
% Prototype: MACRO manyargs #1 #2 #3 #4 #5 #6 #7 #8 #9 #10 #11 #12 #13
    #14 OPT[#15={etc}]
\makeatletter
\newcommand{\manyargs}[9]{%
  \def\manyargs@arg@i{#1}%
  \def\manyargs@arg@ii{#2}%
  \def\manyargs@arg@iii{#3}%
  \def\manyargs@arg@iv{#4}%
  \def\manyargs@arg@v{#5}%
  \def\manyargs@arg@vi{#6}%
  \def\manyargs@arg@vii{#7}%
  \def\manyargs@arg@viii{#8}%
  \def\manyargs@arg@ix{#9}%
  \manyargs@i
}

\def\manyargs@i#1#2#3#4#5{%
  \def\manyargs@arg@x{#1}%
  \def\manyargs@arg@xi{#2}%
  \def\manyargs@arg@xii{#3}%
  \def\manyargs@arg@xiii{#4}%
  \def\manyargs@arg@xiv{#5}%
  \@ifnextchar[{\manyargs@ii}{\manyargs@ii[{etc}]}%
}

\def\manyargs@ii[#1]{%
  \def\manyargs@arg@xv{#1}%
  % Put your code here.
  % You can refer to the arguments as \manyargs@arg@i through \manyargs@arg@xv.
}
\makeatother
```

The current version of `newcommand.py` is limited to 4000 arguments, which should be more than enough for most purposes.

2.6 Summary

A macro is defined in `newcommand.py` with:

`MACRO` ⟨*name*⟩ ⟨*arguments*⟩

in which ⟨*name*⟩ is the name of the macro, and ⟨*arguments*⟩ is zero or more of the following:

Argument	Meaning	Example
#⟨*number*⟩	Parameter (required)	`#1`
{⟨*text*⟩}	Literal text (required)	`{+}`
OPT[#⟨*number*⟩={⟨*text*⟩}]	Parameter (optional, with default)	`OPT[#1={tbp}]`
OPT(#⟨*number*⟩={⟨*text*⟩})	Same as the above, but with parentheses instead of brackets	`OPT(#1={tbp})`
*	Literal asterisk (optional)	*

Within an `OPT` argument, #⟨*number*⟩={⟨*text*⟩} can be repeated any number of times, as long as the various instances are separated by literal text.

3 Further examples

3.1 Mimicking LaTeX's picture environment

The LaTeX `picture` environment takes two, parenthesized, coordinate-pair arguments, the second pair being optional. Here's how to define a macro that takes the same arguments as the `picture` environment and parses them into x_1, y_1, x_2, and y_2 (i.e., `#1`–`#4`):

```
% Prototype: MACRO picturemacro {(}#1{,}#2{)} OPT(#3={0},#4={0})
\makeatletter
\newcommand{\picturemacro}{%
  \picturemacro@i%
}

\def\picturemacro@i(#1,#2){%
  \@ifnextchar({\picturemacro@ii({#1},{#2})}{\picturemacro@ii({#1},{#2})({0},{0})}%
}

\def\picturemacro@ii(#1,#2)(#3,#4){%
  % Put your code here.
  % You can refer to the arguments as #1 through #4.
}
\makeatother
```

The first pair of parentheses and the comma are quoted because they represent required, literal text.

3.2 Mimicking LaTeX's `\parbox` macro

LaTeX's `\parbox` macro takes three optional arguments and two required arguments. Furthermore, the third argument defaults to whatever value was specified for the first argument. This is easy to express in LaTeX with the help of `newcommand.py`:

```
% Prototype: MACRO parboxlike OPT[#1={s}] OPT[#2={\relax}] OPT[#3={#1}]
    #4 #5

\makeatletter
\newcommand{\parboxlike}[1][s]{%
  \@ifnextchar[{\parboxlike@i[{#1}]}{\parboxlike@i[{#1}][{\relax}]}%
}

\def\parboxlike@i[#1][#2]{%
  \@ifnextchar[{\parboxlike@ii[{#1}][{#2}]}{\parboxlike@ii[{#1}][{#2}][{#1}]}%
}

\def\parboxlike@ii[#1][#2][#3]#4#5{%
  % Put your code here.
  % You can refer to the arguments as #1 through #5.
}
\makeatother
```

3.3 Dynamically changing argument formats

With a little cleverness, it is possible for a macro to accept one of two completely different sets of arguments based on the values provided for earlier arguments. For example, suppose we want to define a macro, `\differentargs` that can be called as either

```
\differentargs*[optarg]{reqarg}
```

or

```
\differentargs{reqarg}(optarg)
```

That is, the presence of an asterisk determines whether `\differentargs` should expect an optional argument in square brackets followed by a required argument or to expect a required argument followed by an optional argument in parentheses.

The trick is to create two helper macros: one for the "`*`" case (`\withstar`) and the other for the non-"`*`" case (`\withoutstar`). `\differentargs` can then invoke one of `\withstar` or `\withoutstar` based on whether or not it sees an asterisk. The following shows how to use `newcommand.py` to define `\differentargs`,

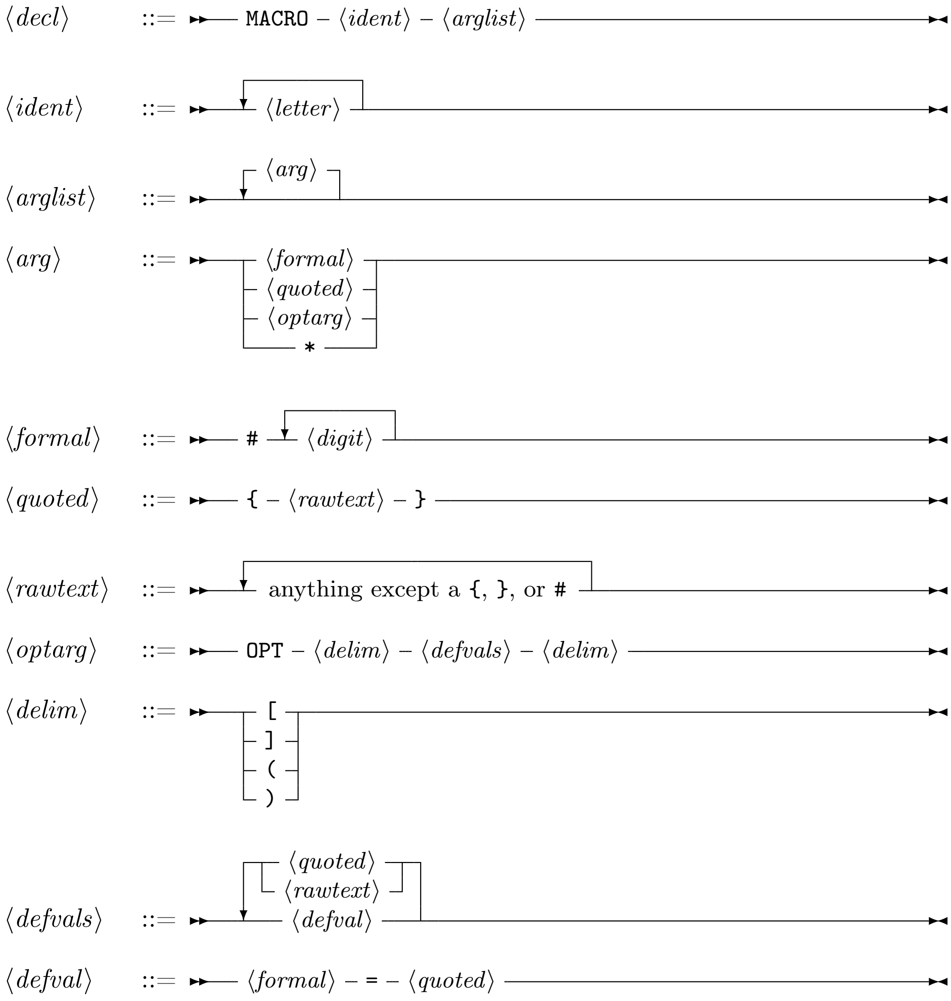
`\withstar`, and `\withoutstar` and how to edit `\differentargs` to invoke its helper macros:

```
% Prototype: MACRO differentargs *
\makeatletter
\newif\ifdifferentargs@star
\newcommand{\differentargs}{%
  \@ifstar{\differentargs@startrue\differentargs@i*}
          {\differentargs@starfalse\differentargs@i*}%
}

\def\differentargs@i*{%
  \ifdifferentargs@star
    % Put code for the "*" case here.
    \let\next=\withstar
  \else
    % Put code for the non-"*" case here.
    \let\next=\withoutstar
  \fi
  % Put code common to both cases here (and/or above the \ifdifferentargs@star).
  \next
}
\makeatother


% Prototype: MACRO withstar OPT[#1={starry}] #2
\newcommand{\withstar}[2][starry]{%
  % Put your code here.
  % You can refer to the arguments as #1 and #2.
}


% Prototype: MACRO withoutstar #1 OPT(#2={dark})
\makeatletter
\newcommand{\withoutstar}[1]{%
  \@ifnextchar({\withoutstar@i{#1}}{\withoutstar@i{#1}({dark})}%
}

\def\withoutstar@i#1(#2){%
  % Put your code here.
  % You can refer to the arguments as #1 and #2.
}
\makeatother
```

Note that we edited `\differentargs@i` to let `\next` be equivalent to either `\withstar` or `\withoutstar` based on whether an asterisk was encountered. `\next` is evaluated outside of the `\ifdifferentargs@star`…`\fi` control structure. This rigmarole is necessary because directly calling `\withstar` or `\withoutstar` would cause those macros to see `\ifdifferentargs@star`'s `\else` or `\fi` as their first argument when they ought to see the text following the `\differentargs` call.

4 Grammar

The following is the formal specification of `newcommand.py`'s grammar, written in a more-or-less top-down manner. Literal values, shown in a typewriter font, are case-sensitive. ⟨*letter*⟩ refers to a letter of the (English) alphabet. ⟨*digit*⟩ refers to a digit.


⟨decl⟩ ::= MACRO – ⟨ident⟩ – ⟨arglist⟩
⟨ident⟩ ::= ⟨letter⟩
⟨arglist⟩ ::= ⟨arg⟩
⟨arg⟩ ::= ⟨formal⟩ ⟨quoted⟩ ⟨optarg⟩ *
⟨formal⟩ ::= # ⟨digit⟩
⟨quoted⟩ ::= { – ⟨rawtext⟩ – }
⟨rawtext⟩ ::= anything except a {, }, or #
⟨optarg⟩ ::= OPT – ⟨delim⟩ – ⟨defvals⟩ – ⟨delim⟩
⟨delim⟩ ::= [ ] ( )
⟨defvals⟩ ::= ⟨quoted⟩ ⟨rawtext⟩ ⟨defval⟩
⟨defval⟩ ::= ⟨formal⟩ – = – ⟨quoted⟩

5 Acknowledgements

I'd like to say thank you to the following people:

- John Aycock for writing the Scanning, Parsing, and Rewriting Kit (SPARK)—the lexer and parser underlying `newcommand.py`—and making it freely available and redistributable.

- Hendri Adriaens for pointing out a bug in the code generated by `newcommand.py`. Previously, bracketed text within a mandatory argument could be mistaken for an optional argument.
- Tom Potts for reporting a spurious error message caused by the processing of `OPT`. This bug has now been fixed. Tom Potts also proposed the example used in Section 3.3 in which the starred and unstarred versions of a macro take different arguments.

6 Copyright and license