The options package

Convenient key-value options for LaTeX package writers

Daan Leijen
2015-12-05

Contents

1. Introduction

The `options` package provides easy to use key-value options for LaTeX package writers. It has a similar interface as `pgfkeys` with path options but comes with more built-in data types and more convenient support for families and searching.

The main features of `options` are:

- *Declare your options only once*: in most packages you usually need to declare both a new command, and the option that sets it. In the `options` package you declare the option just once. For example, `\options{/my/len/.new length}` and then use it anywhere else as `\option{/my/len}`.
- *Use paths for keys*: just like `pgfkeys`, the `options` package uses paths (instead of families) to declare options and prevent name clashes between different packages. Paths are convenient for complex options, like `border/left/width`, and are also convenient to specify searches.
- *Many built-in data types*: the `options` library comes with many useful data types like `choice`, `list`, `toggle`, `num`, `dim`, `length`, `glue`, commands, and plain values, and it is easy to add your own (Section 3.1. Also you can hook into existing definitions like an `if` or `counter` (Section 3.3).
- *Value options*: You can define value-only options that start with a special character, like `"Georgia"` for a font option, or `!8080FF` for a color option (Section 6.3).
- *Convenient searches*: you can specify paths that should be searched from other paths and redirect even from absolute paths. For complex packages this is very useful to inherit common options (Section 2.2).
- *Easy filtering*: it is easy to collect unknown options and process them later. Combined with the search mechanism this makes it easy to do custom processing (Section 2.2).
- *It is fast*: for simple user options, the `options` package is a bit faster than `pgfkeys` and if searches or filters are involved it is usually about twice as fast as `pgfkeys` (and about six times faster as `xkeyval`) (Section 8).
- *Handles class and package options*: use the same option declarations to handle the options passed to a class or package (Section 5.1).

2. Overview

Defining options is very easy. As an example, we will make an `\mybox` command that takes named options. Our goal is that the user can invoke `\mybox` like:

```
\mybox{padding=1ex, border/width=0.4pt, font/style=italic}{text}
```

2.1. An example

We can define the options for our `\mybox` command as:

```
\options{
  /mybox/.new family,
  /mybox/padding/.new length       = \fboxsep,
  /mybox/border/width/.new length = \fboxrule * 2,
  /mybox/border/color/.new color,
  /mybox/font/style/.new choice    = {normal,italic,small-caps},
}
```

The options are all specified as a *path* where names that start with a dot, like `.new length` are called *handlers*. In this case, the handlers create the options for our box command. To parse the options passed by the user, we use the same `\options` command:

```
\newcommand\mybox[2]{%
  {\options{/mybox,#1}%
   \myboxdisplay{#2}%
  }%
}
```

Here we start the option list with `/mybox` which makes that the default path (because it was declared with `.new family`) so the user can give relative option names instead of prefixing all options with `/mybox/`. Options are always set local to the current group.

Finally, we can use `\option{`⟨*option*⟩`}` to get the value of an option. This command expands directly to the command name (i.e. `\optk@`⟨*option*⟩) and is very efficient. So, our implementation for displaying our box may look like:

```
\newcommand\myboxdisplay[1]{%
  \setlength\fboxsep{\option{/mybox/padding}}%
  \setlength\fboxrule{\option{/mybox/border/width}}%
  \colorlet{currentcolor}{.}%
  \color{\option{/mybox/border/color}}%
  \fbox{%
    \color{currentcolor}%
    \ifcase\option{/mybox/font/style/@ord}\relax
    \or\itshape
    \or\scshape
    \fi
```

```
    #1%
  }%
}
```

Here, instead of using the `font/style` directly, we use the automatically generated `font/style/@ord` that gives the ordinal of the choice on which we can match more efficiently. Here is our new command in action:

```
Here is a \mybox{padding=1ex, border/color=blue!70, font/style=italic}{boxed} text.
```

Here is a *boxed* text.

There are many ways to refine our implementation, for example, we can use styles to make it easier to set multiple options at once:

```
\options{ /mybox/border/normal/.new style* =
            {/mybox/border/width=0.4pt, /mybox/border/color=black }}
```

The `*` after the `style` signifies that this option does not expect an argument, and we can use it as:

```
A \mybox{border/normal}{normal} border.
```

A normal border.

Another improvement is in our choice definition for the font. Instead of using a case over the choice ordinal, we can set the required font command directly as the value of the choice options, we show this in Section 3.1.

2.2. Paths and searches

The option paths are mainly used to prevent name clashes between different packages but they can also be used for searches. In particular, we can specify for any path that if a sub-path cannot be found, we should look under another path. Suppose we define a new `\fancymybox` command that takes some extra options. In that case, we would like to re-use the `/mybox` options and look for any undefined options under `/mybox` instead:

```
\options{
  /fancymybox/.new family = {/mybox}, % search also /mybox
  /fancymybox/border/radius/.new length,
  /fancymybox/rounded/.new style = [5pt]{/fancymybox/border/radius = #1},
}
```

Note how the `rounded` style takes an argument which is defaulted to `5pt`. In the `.new family` declaration, we can provide a list of search paths: here we just gave `/mybox` such that any options not found under `/fancymybox` will be looked up under `/mybox`:

```
\options{/fancymybox, rounded=10pt, border/normal}
```

In the previous sample, `/mybox/border/normal` is invoked. In general, we can add comma separated search paths either in a `.new family` declaration or using the `.search also` handler. Search paths can be added to any path and will apply recursively. For example, if we set:

```
\options{
  /a/foo/x/.new cmd* = x,
  /b/bar/y/.new cmd* = y,
  /c/a/.search also = {/a/foo},
  /c/b/.search also = {/b},
  /c/.new family    = {/a,/b},
}
```

Then all of the following options are resolved:

```
\options{ /c/foo/x, /c/a/x, /c/b/bar/y, /c/bar/y}
```

xxyy

Note that the `options` package will even search if an absolute path is given, and always searches with the relative sub-path.

This is important for modularity since it allows us for example to combine options of different sub packages. For example, if I want to handle options under `/package A` and `/package B` together, I can just define a new family:

```
\options{ /packageAB/.new family = {/package A,/package B}}
```

and start processing options using `\options{/packageAB,..}`. Even when the user now uses absolute paths like `/packageAB/`⟨*name*⟩ the option search will route it automatically to the sub packages.

Also note that for efficiency, the basic `\option` command does *not* search and always expands directly to the command name. Therefore, in implementation code we still need to use `\option{/a/foo/x}` and cannot use `\option{/c/foo/x}` for example.

2.3. Handling unknown options

It is possible to handle only options under some path and ignore any unknown options. For example, give our previous options, we can only process the options under `/c/a` as:

```
\options{/options/collectunknown, /c/a/.cd, x, bar/y }\\
\options{/options/remaining/.show}
```

x
/options/remaining=⟨ bar/y ⟩

Here we used the `/options/collectunknown` style to signify that we want to collect unknown options into the `/options/remaining` list. We used the `.cd` handler to change the default path to `/c/a` such that only `x` is found (as `/a/foo/x`) but the `bar/y` option is put in the remaining list.

Any remaining options can be processed eventually using the `\optionswithremaining` command:

```
\optionswithremaining{/c/b/.cd}
y
```

The command takes a list of options that are processed before the options in the remaining list. It is allowed to pass in `/options/collectunknown` right away again to collect any new remaining options.

The `/options/remaining` is list is only cleared when using the `/options/collectunknown` style. This can be useful to collect unknown options using multiple passes of `\optionsalso`.

Besides using the remaining list, you can also define general `@unknown` handlers on any path. When an option not found, the library looks bottom-up for `@unknown` handlers and invokes it with the path and its argument if found. The general handler `/@unknown` will raise a package error.

You can customize this behavior by installing an `@unknown` handler yourself:

```
\options{
  /mybox/@unknown/.new cmd 2 =
    {I don't know option "#1" (=#2).},
  /mybox/silly = hi
}
```

I don't know option "/mybox/silly" (=hi).

As an aside, note that we needed to put braces around `@unknown` handler since it uses the `=` character.

3. Defining options

3.1. Basic data types

⟨*option*⟩`/.new value =` `[`⟨*default*⟩`]`⟨*initial value*⟩

Defines a new ⟨*option*⟩ that with an ⟨*initial value*⟩. A value option just contains the value that was provided by the user. The ⟨*default*⟩ value is optional. If it is given, it is used when the user does not provide an argument when using this option.

⟨*option*⟩`/.new toggle` [`=` ⟨*bool*⟩]

Define a new *toggle*. These are boolean values and have the advantage over the standard `\newif` (see Section 3.3) that they only require one macro instead of three. The initial value (if not given) is `false` and the default value is always `true`. A toggle can be tested using `\iftoggle{`⟨*toggle*⟩`}`.

```
\options{/test/condition/.new toggle}
\options{/test/condition}% default sets to true
Toggle is \iftoggle{/test/condition}{true}{false}
```

Toggle is true

Besides assigning a new value of `true` or `false`, you can also flip a condition as:

```
\options{/test/condition/.flip}
```

⟨*option*⟩`/.new choice = [`⟨*default*⟩`]{`⟨*choice*$_1$⟩`[=`⟨*value*$_1$⟩`]`,...,⟨*choice*$_n$⟩`[=`⟨*value*$_n$⟩`]}`

Defines a new choice option where the user can provide ⟨*choice*$_1$⟩ to ⟨*choice*$_n$⟩ as arguments. The initial value is always ⟨*choice*$_1$⟩. We always need to enclose the choice list with braces to distinguish the comma used to separate the choices from the comma used to separate the options. For convenience, the `new choice` handler also defines ⟨*option*⟩`/@ord` that contain the ordinal of the current choice (starting at 0), and the ⟨*option*⟩`/@name` that contains the current choice name. The ordinal is a number and can be tested efficiently using `ifcase` and `ifnum`. For example:

```
\options{/test/program/.new choice={latex,xelatex,luatex,pdftex}}
\options{/test/program = xelatex}
\noindent\options{/test/program/.show}

% case on ordinal
\ifcase\option{/test/program/@ord}%
  latex
\or
  xelatex
\else
  other
\fi

% ifnum on ordinal
\ifnum\option{/test/program/@ord}<2\relax
  latex or xelatex
\else
  luatex or pdftex
\fi
```

/test/program=⟨xelatex⟩ (@ord=1), choices=⟨latex,xelatex,luatex,pdftex⟩
xelatex
latex or xelatex

Another powerful feature is to define the values that each choice implies. By default this is the name of the choice but we can assign anything else. For example, for our `\mybox` command, we could have specified the font style as:

```
\options{/test/font/style/.new choice=
            {normal={}, italic=\itshape, small-caps=\scshape}}
```

The value of `\option{/test/font/style}` is not the choice name now, but the command that we assigned to it. We can now use the option directly instead of doing a case on the `@ord` value:

```
\options{/test/font/style=italic}
This is {\option{/test/font/style}in italics}.
```

This is *in italics*.

⟨*option*⟩`/.new list [=[`⟨*default*⟩`]{`⟨$elem_1$⟩`,...,`⟨$elem_n$⟩`}]`

Define a new comma separated list. The initial value if not given is the empty list. There are various operations to work with lists:

- `\letoptionlist{`⟨*option*⟩`}\`⟨*macro*⟩: stores the list in `\`⟨*macro*⟩.

- `\optionlistdo{`⟨*option*⟩`}{`⟨*cmd*⟩`}`: invokes ⟨*cmd*⟩ on each element. The iteration can be stopped by ending ⟨*cmd*⟩ with `\listbreak`.
- `\ifoptioncontains{`⟨*option*⟩`}{`⟨*elem*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`: test if ⟨*elem*⟩ occurs in the list and invokes either the ⟨*true*⟩ or ⟨*false*⟩ branch.

```
\options{/test/list/.new list = {banana,milk,eggs}}
\optionlistdo{/test/list}{%
  Element “\textsf{#1}”.
  \ifstrequal{#1}{milk}{\listbreak}{}%
}
```

Element “banana”. Element “milk”.

There are also two special handlers for manipulating lists in the options,

- ⟨*list option*⟩`/.push =` ⟨*elem*⟩: pushes `<elem` on the end of the list.
- ⟨*list option*⟩`/.concat = {`⟨*list*⟩`}`: concatenates ⟨*list*⟩ to the end of the list.

For example,

```
\options{/test/list 2/.new list = {banana}}
\options{/test/list 2/.push = milk, /test/list 2/.show}
```

/test/list 2=⟨banana,milk⟩

⟨*option*⟩`/.new length [= [`⟨*default*⟩`]`⟨*dimexpr*⟩`]`

Defines a new length option ⟨*option*⟩. This option stores its value in a new length register, and its value be used directly where a length is expected, e.g., `\hspace{\option{`⟨*option*⟩`}}`. If no initial value is given, the initial length is `0pt`. The user can assign any length expressions (*dimexpr*), e.g., `\options{`⟨*option*⟩ `= 1pt + \fboxsep}`.

⟨*option*⟩`/.new dim [= [`⟨*default*⟩`]`⟨*dimexpr*⟩`]`

Defines a new dimension option ⟨*option*⟩. This option stores its value as an unevaluated `dimexpr`, and its value be used directly where a dimension is expected, e.g., `\hspace{\option{`⟨*option*⟩`}}`. The main difference with a length option is that a dimension option is not evaluated at the time the key is assigned, but rather when it is *used*. This may be important when relying on the contents of other registers. For example, if we declare:

```
\setlength\fboxrule{1pt}
\options{/test/width/.new dim=\fboxrule,/test/length/.new length=\fboxrule}
\setlength\fboxrule{10pt}
```

This will show `10pt` for the `width`, but `1pt` for the length:

```
\options{/test/width/.show}\\\options{/test/length/.show}
```

/test/width=⟨10.0pt⟩ (=⟨\fboxrule ⟩)
/test/length=⟨1.0pt⟩

⟨*option*⟩`/.new num` [`=` [⟨*default*⟩]⟨*numexpr*⟩]

Defines a new number option ⟨*option*⟩. The assigned value is evaluated as a *numexpr*. Can be used directly in any context that expects a *numexpr*. For example:

```
\options{/test/num/.new num = 2+4}
Is it 6? \ifnum\option{/test/num}=6\relax Yes.\else No!\fi
```

Is it 6? Yes.

⟨*option*⟩`/.new glue` [`=`[⟨*default*⟩]⟨*glueexpr*⟩]

Defines a new glue option. Can be assigned any valid glue expression and used in any context that expects a glue expression. If no initial value is given, `0pt` is used.

3.2. Command options

The options in this section are not values by themselves but are only invoked for their side effect.

⟨*option*⟩`/.new family` [`=` ⟨*search list*⟩]

Defines a new family, this is a shorthand for

```
⟨option⟩/.search also = ⟨search list⟩,
⟨option⟩/.new style*  = {⟨option⟩/.cd},
⟨option⟩/.type        = family
```

⟨*option*⟩`/.new style = {`⟨*options*⟩`}`

⟨*option*⟩`/.new style* = {`⟨*options*⟩`}`

Defines a new style; when invoked it will also set the specified ⟨*options*⟩ using `\optionsalso`. The `.new style*` variant does not take an argument itself. The plain `.new style` can use `#1` for the argument value, e.g.

```
/border/width/.new style = {/border/top/width=#1,/border/bottom/width=#1}
```

⟨*option*⟩`/.new cmd = [`⟨*default*⟩`]`⟨*code*⟩

⟨*option*⟩`/.new cmd*` = ⟨*code*⟩

Define a new command that is invoked when the options is given. `.new cmd*` takes no argument while `.new cmd` takes a single argument.

Use of these options is generally discouraged and if you can you should try to use a data type directly together with `\option{`⟨*option*⟩`}` commands.

```
\options{ /test/cmd/.new cmd = You said “#1”,
          /test/cmd = hi,
}
```

You said “hi”

⟨*option*⟩`/.new cmd 0` = ⟨*code*⟩

⟨*option*⟩`/.new cmd 1 = [`⟨*default*⟩`]`⟨*code*⟩

⟨*option*⟩`/.new cmd 2 = [`⟨*default*⟩`]`⟨*code*⟩

⟨*option*⟩`/.new cmd 3 = [`⟨*default*⟩`]`⟨*code*⟩

⟨*option*⟩`/.new cmd 4 = [`⟨*default*⟩`]`⟨*code*⟩

Define commands with multiple arguments, where `.new cmd 0` is equal to `.new cmd*` and `.new cmd 1` to `.new cmd`. Each argument needs to be enclosed in braces and if not all arguments are given, they will be empty.

```
\options{
  /test/cmd2/.new cmd 2 = [{x}{y}]{I got (#1,#2)\\},
  /test/.cd,
  cmd2,
  cmd2 = hi,
  cmd2 = {hi},
  cmd2 = {{hi}},
  cmd2 = {hi}{there},
}
```

I got (x,y)
I got (h,i)
I got (h,i)
I got (hi,)
I got (hi,there)

Note how `{hi}` and `hi` had the same effect since the `options` processing peels of a single layer of braces. If we want to preserve braces we need to double up.

⟨*option*⟩`/.new cmd tuple =` ⟨*code*⟩

⟨*option*⟩`/.new cmd triple =` ⟨*code*⟩

These are variants of `.new cmd 2` and `.new cmd 3` that take exactly 2 or 3 arguments separated by commas.

⟨*option*⟩`/.new cmdx = {`⟨*pattern*⟩`}{`⟨*postfix*⟩`}{`⟨*code*⟩`}`

This defines a plain TEX command with the specified ⟨*pattern*⟩. Also, when invoking the command, it will append ⟨*postfix*⟩ to the argument which is often necessary to ensure the command pattern will always match. For example, here is how we defined `.new cmd tuple` which matches with exactly 2 arguments:

```
/handlers/new cmd tuple/.new handler/.defaults = \optionsalso{
  #1/.new cmdx = {##1,##2,##3}{„}% match comma separated
    {\ifstrequal{##3}{,}%
       {#2}%
       {\optionerror{#1}{expecting a 2 argument tuple}}%
    },
  #1/.type=cmd tuple,
}
```

3.3. Using existing definitions

Use these declarations if you need to work with existing definitions and are not able to declare your own using the data types in Section 3.1.

⟨*option*⟩`/.is if =` ⟨*if name*⟩

Declare a new option that directly sets a defined LaTeX `if` declared with `\newif`. The name should not start with the `if` part, e.g.

```
\options{ /twocolumn/.newif = @twocolumn }
```

⟨*option*⟩`/.is counter =` ⟨*counter name*⟩

Declare an option that directly sets or gets a LaTeX counter declared with `\newcounter`. You can use `.inc`, `.step`, and `.refstep` operations on counters.

⟨*option*⟩`/.is def =` `[`⟨*default*⟩`]`⟨*macro name*⟩

Declare an option that directly sets or gets a defined definition. This is basically equivalent to the `\define@key` operation of the `keyval` package.

```
\def\mytest{}
\options{
  /mytest/.is def = \mytest,
  /mytest = “hi”,
}
\mytest{} there.
```

“hi” there.

⟨*option*⟩`/.is edef =` `[`⟨*default*⟩`]`⟨*macro name*⟩

Same as the `.is def` but will use `\edef` to redefine the macro definition.

4. Option transformers

4.1. Operations on option paths

⟨*path*⟩`/.cd`

Change the directory to make ⟨*path*⟩ the default option prefix.

⟨*option*⟩`/.show`

Show the current option (in the document)

⟨*option*⟩`/.typeout`

Show the current option in the console.

4.2. Operations on arguments

⟨*option*⟩`/.expand once =` ⟨*value*⟩

Expand the argument once (using `\expandafter`)

⟨*option*⟩`/.expand twice =` ⟨*value*⟩

Expand the argument twice.

⟨*option*⟩`/.expanded =` ⟨*value*⟩

Fully expand the argument. Defined as:

```
/handlers/expanded/.new transformer =
  \protected@edef\optionvalue{\optionvalue}
```

⟨*list option*⟩`/.push =` ⟨*element*⟩

Push an element on the end a list option.

⟨*list option*⟩`/.concat =` ⟨*list*⟩

Concatenate a list to the end of a list option.

⟨*option*⟩`/.inc` [`=` ⟨*numexpr*⟩]

Increment a counter or number (`.new num`) with the given amount (or 1 if no argument is given)

⟨*counter option*⟩`/.step`

Step a counter option by 1.

⟨*counter option*⟩`/.refstep`

'Ref step' a counter option by 1.

4.3. Predefined options

`/options/exec` = ⟨*code*⟩

Directly execute the provided ⟨*code*⟩

```
\options{ /options/exec=hi there}
```
hi there

`/options/ignoreunknown` [= ⟨*bool*⟩]

If set to true, this will ignore any subsequent unknown options. Such ignored options will be added to the `/options/remaining` list.

`/options/allowsearch` [= ⟨*bool*⟩]

If set to `false`, it will no longer search along search paths provided by `.search also` or `.new family`.

`/options/unknownwarnonly` [= ⟨*bool*⟩]

If set to true, this only issues a warning when finding an unknown option (instead of raising an error).

`/options/remaining` [= ⟨*options*⟩]

A list of remaining options to be processed. This list should not be used directly but mostly in conjunction with `/options/collectunknown` and `\optionswithremaining`.

`/options/collectunknown`

A style. If given, it will clear the `/options/remaining` list and set `/options/ignoreunknown` to true.

5. Option commands

This section defines the command macros available on option values.

5.1. Processing options

`\options{`⟨*options*⟩`}`

Process the ⟨*options*⟩ list. This is a comma separated list of ⟨*option*⟩ [=⟨*value*⟩] elements. Both the ⟨*option*⟩ and ⟨*value*⟩ are trimmed of whitespace (at the end and start). The list may have empty elements and also end or start with commas. When extracting the ⟨*value*⟩ a single layer of braces is removed – this is done such that the user can specify values that contain commas or equal signs themselves. Any option defines are always local to the current group.

When invoking `\options` the initial path is always the root (`/`) and the flag `/option/ignoreunknown` is false, and `/option/allowsearch` is true.

`\optionsalso{`⟨*options*⟩`}`

Just like `\options` except it will use the current default path and option flags.

`\optionswithremaining{`⟨*options*⟩`}`

Like `\options` but also processes any options in the list `/options/remaining` after processing ⟨*options*⟩. Will start by making a copy of the `/options/remaining` list so you can call `/options/collectunknown` in ⟨*options*⟩ to immediately start collecting further remaining options.

`\options@ProcessOptions{`⟨*family*⟩`}`

Call this in a package (`.sty`) or class (`.cls`) file to handle the options passed to it. In a package file it will also consider the known options that are passed to the class besides the options passed to it directly. The ⟨*family*⟩ should be the root path for your options. For example,

```
\NeedsTeXFormat{LaTeX2e}[1995/12/01]

\ProvidesPackage{mypkg}[2015/01/01,By Me]
\RequirePackage{options}

\options{
  /mypkg/.new family,
  /mypkg/columns/.new num = [1]{2},
  /mypkg/10pt/.new cmd* = \typeout{use 10pt font size},
}
\options@ProcessOptions{/mypkg}
```

Others can now pass options to your package as:

```
\usepackage[10pt,columns=2]{mypkg}
```

5.2. Using options

`\option{`⟨*option*⟩`}`

This uses the current value of ⟨*option*⟩. It directly expands to the command `\optk@`⟨*option*⟩ and is very efficient. Usually that command contains the value of the option, but sometimes it expands to something different depending on the type of the options. For example, it may be a length register.

`\letoption{`⟨*option*⟩`}\`⟨*macro*⟩

If you need to use an option in loop or need careful expansion control, it is sometimes more efficient to `\let` bind the option value into a macro, e.g.

```
\letoption{/my/option}\myoptionvalue
```

`\edefoption{`⟨*option*⟩`}\`⟨*macro*⟩

Same as `\letoption` but fully expands the current value.

5.3. Testing options

All these test take true and false branch as final arguments.

`\ifoptiondefined{`⟨*option*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Is the option defined?

`\ifoptionvoid{`⟨*option*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Is the option undefined or is its value blank (empty or spaces)?

`\ifoptionblank{`⟨*option*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Is the option value blank (empty or spaces)?

`\ifoptionequal{`⟨*option*⟩`}{`⟨*value*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Is the option value equal to the provided value?

`\ifoptionanyof{`⟨*option*⟩`}{`⟨*list*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Does the option value occur in the comma separated value list ⟨*list*⟩?

`\ifoptiontype{`⟨*option*⟩`}{`⟨*type*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Does the option have type ⟨*type*⟩?

`\ifoptionnil{`⟨*list option*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Is list option value empty?

`\ifoptioniscode{`⟨*option*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Is this option just an action? i.e. it has no associated value.

5.4. List options

These are utility functions for list options.

`\optionlistdo{`⟨*list option*⟩`}{`⟨*action*⟩`}`

Perform ⟨*action*⟩ on every element of the current value of ⟨*list option*⟩. The element is available as `#1` in ⟨*action*⟩ and the iteration can be stopped by ending with `\listbreak`.

`\letoptionlist{`⟨*list option*⟩`}\`⟨*macro*⟩

`\let` bind the current list value as a list of comma separated values in \⟨*macro*⟩. This is sometimes needed since the internal representation of lists uses another representation internally (in order to contain commas itself).

`\ifoptioncontains{`⟨*list option*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Check if a list option contains a certain element.

5.5. Showing options

`\optionshow{`⟨*option*⟩`}`

Show the value of ⟨*option*⟩.

```
Ignore unknown = \optionshow{/options/ignoreunknown}
```

Ignore unknown = ⟨opt@ignoreunknown⟩ (=⟨false⟩)

`\optionshowall[`⟨*bool*⟩`]`

Show all the options that are defined. If you pass `true` in the optional argument it also shows all the internal values under the `/handlers/` which can be a big list.

`\optionshowpath{`⟨*option*⟩`}`

Show all the options defined under a certain path. Useful for debugging:

```
\optionshowpath{/mybox}
/mybox/
  padding=⟨3.0pt⟩, initial=⟨\fboxsep ⟩
  border/
    width=⟨0.4pt⟩, initial=⟨\fboxrule * 2⟩
    color=⟨black⟩, initial=⟨black⟩
    normal=style
  font/
    style=⟨normal⟩ (@ord=0), choices=⟨normal,italic,small-caps⟩, initial=⟨normal⟩
```

6. Advanced topics

This section gives an overview of more advanced mechanisms, like defining handlers and new data types.

6.1. Handlers

Names that start with a dot are *handlers*. These are commands that are called with the current option path and argument, and are used for example to declare new options (e.g. `.new choice`), to change the environment (e.g. `.cd`), or to transform the argument (e.g. `.expanded`).

You can define your own handler `.`⟨*name*⟩ by adding a command option under `/handlers/`⟨*name*⟩. For example, let's make a handler that transforms a color argument from a HTML format `#XXXXXX` to a named color:

```
\options{%
  /handlers/htmlcolor/.new transformer = {%
    % we need to change the \optionvalue
    % #1 = current path, #2 = current \optionvalue
    \definecolor{#1/@color}{HTML}{#2}%
    \def\optionvalue{#1/@color}%
  }
}
```

We can use our new handler directly with our previous box command:

```
A \mybox{border/color/.htmlcolor = 800080}{purple} box.
```

A purple box.

There are various kinds of handlers:

- `.new handler`: a general handler that does not chain nor invoke the original option.
- `.new operation`: chains with other handlers but does not invoke the original option.
- `.new transformer`: transforms the `\optionvalue` and chains with other handlers.

6.2. Defining new data types

Handlers are also used to define the standard data types and can be used to define new data types yourself. Here is for example how the `value` data type is defined:

```
\option{
  /handlers/new value/.new handler = []\optionsalso{%
    #1/.new cmd = \option@set{#1}{##1},
    #1/.type    = value,
    #1/.initial = {#2},
  },
}
```

The `.new handler` receives the current path (`#1`) and the initial value (`#2`). The default value provided at definition time is empty (`[]`). When a user defines a new value, we simply set more options based on the path. When the defined option is set, the `.new cmd` is called with the argument (`##1`) and we use that set the actual option value directly: `\option@set{#1}{##1}`. Usually, we do some additional processing here, for example, for choice values we would check if the choice is valid.

6.3. Single option values

Finally, we can also specify handlers that are invoked when the option name starts with special character. This allows you to handle, say, a quoted value like `"Georgia"` as a shorthand for `/font/family=Georgia`. As an example, we will handle options that start with a bang (`!`) as an HTML color. Handling a special character ⟨*char*⟩ can be done simply by installing a handler under `/handlers/special/`⟨*char*⟩:

```
\options{%
  /handlers/special/!/.new handler = {%
    \optionsalso{ /mybox/border/color/.expanded/.htmlcolor = \@gobble#2 }%
  },
}
A \mybox{!008080}{teal} box.
```

A teal box.

Here we used chaining to first expand the argument (`.expanded`) and then invoking the `.htmlcolor` handler. We needed to use `\@gobble` to remove the exclamation mark from the provided argument (i.e. `#2` will be equal `!008080`).

6.4. Search algorithm

When `\options` parses the options and finds an option and argument, it will assign argument to `\optionvalue` and search for the option:

1. If the option name is not absolute (i.e. does not start with `/`) then prepend the default path (`\opt@defaultpath`).
2. If the option exist we are done; invoke ⟨*option*⟩`/@code`.
3. Otherwise, if the option has handlers `.`⟨*name*⟩ look for the first handler and invoke `/handlers/`⟨*name*⟩`/@code` if it exists with `\option@handlerpath` set to the option path.
4. If there is no handler, perform a search bottom-up through the sub-paths of the option path. If ⟨*path*⟩`/@searchalso` exists where ⟨*option*⟩ = ⟨*path*⟩`/`⟨*subpath*⟩, then invoke the search also handler with ⟨*subpath*⟩ as its argument, and keep searching recursively.
5. If we still don't find the option, search bottom-up through the sub-paths of the option path for `/@unknown` handlers and invoke it the first one that is found. If none are defined, this will invoke `/@unknown` with ⟨*option*⟩ as its path which will raise a package error.

This may seem like quite a bit of work but there has been much attention to efficiency. For example, we can match in a single TEX definition for handlers since we evaluate chained handlers from left to right. Also, we never search for handlers but just match directly etc.

7. Advanced options and commands

This section introduces option handlers and commands that are useful when extending the `options` package itself.

7.1. Transforming an option

These operations transform an option itself. Usually only used when defining new handlers but should not be necessary in general.

⟨*option*⟩`/.expands`

When applied to an option it ensures the option argument is always fully expanded (as if using `.expanded` at every invocation).

⟨*option*⟩`/.default =` ⟨*default value*⟩

Set the default argument if the user does not provide one. The default argument is stored in ⟨*option*⟩`/@def`.

⟨*option*⟩`/.initial =` `[`⟨*default*⟩`]`⟨*initial*⟩

Set the default argument (if provided) and the initial value of an option. The initial value is stored in ⟨*option*⟩`/@ini` and used when invoking `.reset`.

⟨*option*⟩`/.reset`

Sets the option back to its original value.

⟨*option*⟩`/.undef`

Undefine an option. This can be used to redefine existing options of other packages.

⟨*option*⟩`/.search also =` ⟨*search paths*⟩

Specify additional search paths for the given ⟨*option*⟩ as explained in Section 2.2.

⟨*option*⟩`/.type =` ⟨*type*⟩

Specify the type of an option. This is used for example by the default `.show` function to display values correctly. The type can be queried using `\letoptiontype`, `\ifoptiontype`, and `\ifoptioniscode`.

7.2. Compatibility

Some compatibility functions for `pgfkeys` users.

⟨*option*⟩`/.is family`

Equivalent to `.new family` (but without a search argument)

⟨*option*⟩`/.code =` ⟨*code*⟩

Equivalent to `.new cmd`.

⟨*option*⟩`/.style =` ⟨*style*⟩

Equivalent to `.new style`.

7.3. Special paths

`/handlers/`⟨*name*⟩ `=` ⟨*code*⟩

Define such path to install a new handler ⟨*name*⟩. The ⟨*code*⟩ takes two arguments (the current path and argument) and should be defined using `.new handler`, `.new operation`, or `.new transformer`.

`/handlers/show/`⟨*type*⟩ `=` ⟨*code*⟩

Provide custom show functions for options of type ⟨*type*⟩. The ⟨*code*⟩ takes a single argument, namely the option name. For example:

```
/handlers/show/length/.new cmd = \option@showvalue{\the\option{#1}}
```

Using `\option@showvalue` ensures option values are shown consistently.

`/handlers/special/`⟨*char*⟩ `=` ⟨*code*⟩

Provide handlers for options that start with a special character, see Section 6.3.

7.4. Miscellaneous commands

`\optionerror{`⟨*option*⟩`}{`⟨*message*⟩`}`

Raise a package error for ⟨*option*⟩ with a certain message.

`\optionwarning{`⟨*option*⟩`}{`⟨*message*⟩`}`

Output a package warning for ⟨*option*⟩ with a certain message.

`\optionerror@expect{`⟨*option*⟩`}{`⟨*expecting*⟩`}`

Raise a package error for ⟨*option*⟩ with an ⟨*expecting*⟩ message.

`\letoptiontype{`⟨*option*⟩`}\`⟨*type*⟩

Bind the type of an option to `\`⟨*type*⟩. Properly takes care of command options.

7.5. Setting options directly

`\optionname{`⟨*option*⟩`}`

Expand to the internal name of an options, i.e. `optk@`⟨*option*⟩.

`\option@set{`⟨*option*⟩`}{`⟨*value*⟩`}`

Set the `\optk@`⟨*option*⟩ macro to ⟨*value*⟩.

`\option@eset{`⟨*option*⟩`}{`⟨*value*⟩`}`

Set the `\optk@`⟨*option*⟩ macro to a single expansion of ⟨*value*⟩ (using `\expandafter`).

`\option@def{`⟨*option*⟩`}`⟨*pattern*⟩`{`⟨*code*⟩`}`

Define the `\optk@`⟨*option*⟩ as a macro that takes parameters defined in ⟨*pattern*⟩.

`\option@let{`⟨*option*⟩`}\`⟨*macro*⟩

`\let` the `\optk@`⟨*option*⟩ macro to `\`⟨*macro*⟩. This can be more efficient than `\options@set` if the value happens to be a macro itself.

7.6. Invoking an option directly

Internally, every option ⟨*option*⟩ comes with ⟨*option*⟩`/@code` that is invoked when an option is set with the `\optionvalue` containing the current argument. The `\optionvalue` will equal `\optionnovalue` when no argument is given. A single expansion of `\optionvalue` will yield the exact value provided by the user (minus one layer of outer braces).

The ⟨*option*⟩`@code` usually starts with an argument check. If no argument is expected (the $*$ options on commands) and error is raised if there was an argument. If an argument is expected, and there is no argument, the `\optionvalue` is set to the default argument (in ⟨*option*⟩`/@def`). If there is no default, an error is raised.

`\option@invoke{`⟨*option*⟩`}{`⟨*value*⟩`}`

`\option@xinvoke{`⟨*option*⟩`}{`⟨*value*⟩`}`

`\option@einvoke{`⟨*option*⟩`}{`⟨*value*⟩`}`

`\option@invokedefault{`⟨*option*⟩`}`

Various ways to invoke a option handler directly. The `\option@invoke` macros directly call ⟨*option*⟩`/@code`. The `einvoke` does a single expansion of its argument, and `xinvoke` does a full expansion. The `\option@invokedefault` call the ⟨*option*⟩`/@code` with `\optionvalue` set to `\optionnovalue`.

7.7. Defining a new code value

⟨*option*⟩`/.new code =` `[`⟨*default*⟩`]`⟨*code*⟩

Define a new ⟨*option*⟩`/@code` handler. This is invoked when the option ⟨*option*⟩ is set with the macro `\optionvalue` set to the argument. This will equal `\optionnovalue` if no argument was provided. This is the most primitive command handler and it is recommended to use `.new cmd` instead.

⟨*option*⟩`/.new code*` `=` ⟨*code*⟩

Just like `.new code` but checks that ⟨*option*⟩ did not get an argument, i.e. that `\optionvalue` equals `\optionnovalue`.

7.8. Defining new handlers

To define handlers, we cannot use `.new code` since the handler needs access to the option path it is handling. The `.new handler` gets also passed this path as an argument.

`/handlers/`⟨*name*⟩`/.new handler =` `[`⟨*default*⟩`]`⟨*code*⟩

Define a new handler ⟨*name*⟩. If ⟨*default*⟩ is provided, this will be the default argument for the newly defined handler. The ⟨*code*⟩ takes two arguments, the option path (`#1`) when `.`⟨*name*⟩ is invoked, and the provided argument (`#2`). For example:

```
/handlers/new style*/.new handler = \optionsalso{
  #1/.new cmd* = \optionsalso{#2},
  #1/.type     = style,
},
```

/handlers/⟨*name*⟩/.new handler* = ⟨*code*⟩

Similar to .new handler but for handlers that take on argument.

⟨*option*⟩/.new transformer = [⟨*default*⟩]⟨*code*⟩

A form of handler that transforms a provided argument. Should redefine \optionvalue which will be passed to the next handler. Takes the option path and current option argument as arguments. For example:

```
/handlers/expanded/.new transformer=\protected@edef\optionvalue{\optionvalue}
```

⟨*option*⟩/.new operation = [⟨*default*⟩]⟨*code*⟩

A form of handler that is invoked for its side effect. Takes the option path and current option argument as arguments. In contrast with a transformer after this handler, the original option will not be set and it only chains with other handlers. For example:

```
/handlers/concat/.new operation = \option@concat{#1}{#2}
```

⟨*option*⟩/.new operation* = ⟨*code*⟩

Same as .new operation but for operations that take no argument.

⟨*option*⟩/.new cmd transformer = [⟨*default*⟩]⟨*code*⟩

A transformer that changes the option value of a command. For example:

```
/handlers/expands/.new cmd transformer={\protected@edef\optionvalue{\optionvalue}}
```

7.9. Defining options directly

\optionprependcode{⟨*option*⟩}{⟨*code*⟩}

Add some code to ⟨*option*⟩/@code right after the argument check. This is used for example by the .expands handler to always expand the \optionvalue.

\optionnewcode{⟨*option*⟩}{⟨*code*⟩}

Define a new ⟨*option*⟩/@code handler that expects an argument.

\optionnewcode*{⟨*option*⟩}{⟨*code*⟩}

Define a new ⟨*option*⟩/@code handler that expects no argument.

`\optionnewhandler{`⟨*option*⟩`}{`⟨*code*⟩`}`

Define a new ⟨*option*⟩`/@code` handler for defining a handler that will take an argument. This will insert some code that calls ⟨*code*⟩ with the option path that is handled in `#1` and the argument in `#2`.

`\optionnewhandler*{`⟨*option*⟩`}{`⟨*code*⟩`}`

Define a new ⟨*option*⟩`/@code` handler for defining a handler that takes no argument. This will insert some code that calls ⟨*code*⟩ with the option path that is handled in `#1` and the argument in `#2`.

8. Performance

There are some performance numbers of the `options`, `pgfkeys` and `xkeyval` packages. To test the performance, we performed 100.000 invocations of `\options`, `pgfkeys`, and `setkeys` respectively. For each library, we defined two options in the family `bar` and `foo`, as:

```
\options{
  /bar/bar-test/.new value = hi,
  /foo/foo-test/.new value = world,
  /foo/.new family={/bar},
}
```

We then tested two queries. The first one is *simple* and set an option that can be directly found:

```
\options{/foo,foo-test=a test}
```

and a *complex* one that needs a one-level search:

```
\options{/foo,bar-test=a test}
```

We measured the time of a run without any testing (the *baseline*) and then ran each benchmark 100.000 times and picked the best time out of three runs, subtracting the baseline time. The benchmarks were run on an Intel Core2 Quad CPU @ 3ghz with 4Gb memory running XeLaTeX 3.1415926 from TeX Live 2013/W32TeX. The results were:

	simple		
package	relative	total	100.000 reps.
`options`	1.0x	3.41s	25ms per 1000
`pgfkeys`	1.3x slower	4.34s	34ms per 1000
`xkeyval`	8.1x slower	27.77s	268ms per 1000

complex (one level search)			
package	relative	total	100.000 reps.
`options`	1.0x	5.32s	44ms per 1000
`pgfkeys`	2.1x slower	11.10s	101ms per 1000
`xkeyval`	5.7x slower	30.34s	294ms per 1000

So both `options` and `pgfkeys` are quite a bit faster than `xkeyval`, and `options` performs quite well when searches are involved. We also tested against the basic `keyval` package but this is a bit tricky since `keyval` does not support features like searching in the first place. It was about 1.5 times faster on simple queries though.