Paragraph designer with galley approach

Oleg Parashchenko

Abstract

The LaTeX package `paravesp.sty` controls the space above and below paragraphs.

The Python script `parades.py` generates paragraph styles with support of space above, space below and tabulators.

The system imposes the galley approach on the document.

1 Introduction

The goal was to support one layout specification defining the space above and below paragraphs. This is not how TeX works. To satisfy the requirement, the package `paravesp` (PARAgraph VErtical SPace) was developed.

The solution imposes the galley approach on the document. Paragraphs need to be wrapped by a tracking code, which controls how the material is added into the TeX vertical list.

The paragraph designer appeared as a generalization of the tracking code to other paragraph properties. The user describes the formatting options in a Python file. The program `parades.py` converts the definitions into TeX code.

The system works successfully in production, but so far is limited to my needs. A complete set of paragraph properties is not an immediate goal. Switching to the package `xgalley` from the LaTeX3 project might be a step in future development.

This article starts with the definition of the space between paragraphs and how it is implemented. The example demonstrates the use of the commands, which are then described using pseudocode.

The paragraph designer is first illustrated by a sample LaTeX fragment, which uses the paragraph styles. For each of the three types of styles, we give a sample definition in Python and the result of translating to TeX code, with explanations. Finally, a reference section lists all the supported paragraph properties and the commands of the Python `parades.py` tool.

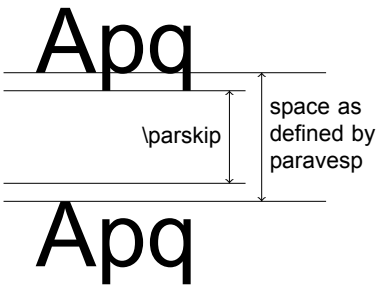
The article concludes with information on how to get the code and run it.

2 Space between paragraphs

The notion of "space between paragraphs" can be defined in various ways.

In one definition, the space between paragraphs is the amount of additional space relative to what happens inside a paragraph. This is what most typesetting engines implement, and what is named `parskip` in TeX.

The definition for `paravesp.sty` is: the space between paragraphs is the distance between the baseline of the preceding paragraph and the top of the next paragraph. The code ensures that this distance is larger than `prevdepth`.



2.1 Usage

The package `paravesp` imposes restrictions on how to construct a document. Otherwise it can't guarantee the desired space above or below paragraphs.

- Switches between the vertical and horizontal modes must be controlled. TeX's automatic switching is partially forbidden.
- The register `\parskip` belongs to the controlling code.
- The commands rely on the automatic insertion of `parskip` glue by TeX.

The guidelines for the controlling code are:

- At the end of a paragraph (after `\par`) use the command `\ParaSpaceBelow`.
- At the beginning of a paragraph, while still in the vertical mode, use `\ParaSpaceAbove`.
- At the beginning of block content, for which TeX will not insert `\parskip` automatically, use both `\ParaSpaceAbove` and `\IssueParaSpace`.

An example:

```
\ParaSpaceAbove{20pt}%
{\HeadingStyle Heading}\par
\ParaSpaceBelow{20pt}%
%
\ParaSpaceAbove{10pt}%
A paragraph of normal text...\par
\ParaSpaceBelow{10pt}%
%
\ParaSpaceAbove{10pt}%
Another paragraph of normal text...\par
\ParaSpaceBelow{10pt}%
%
\ParaSpaceAbove{20pt}\IssueParaSpace
\vbox{\fbox{Some info in a box}}%
\ParaSpaceBelow{20pt}%
```

2.2 Technical details

Below is a simplified version of what happens. Special cases are not shown.

After `\ParaSpaceBelow{`*length*`}`:

- vertical list is not changed
- `parskip` := *length* − `prevdepth`
- `prevdepth` is not changed

The command `\ParaSpaceBelow` splits its argument between two lengths, `prevdepth` and `parskip`. This is a precaution for the case when the next element in the vertical list is not controlled by the galley. Thanks to the retained `prevdepth`, a possible layout corruption is avoided.

After `\ParaSpaceAbove{`*length*`}`:

- vertical list: `vskip` −`prevdepth`, penalty as before `vskip`
- `parskip` := max(*length*, *old_ length*)
- `prevdepth` := −1000pt

The command `\ParaSpaceAbove`, which precedes a paragraph, can't know how much interline glue induced by `baselineskip` will be added. As a solution, the command disables this glue completely by setting `prevdepth` to minus infinity.

After `\IssueParaSpace`:

- vertical list: `vskip parskip`, penalty as before `vskip`
- `parskip` := 0pt
- `prevdepth` := −1000pt

You need the command `\IssueParaSpace` when TEX does not insert `\parskip` automatically, for example, before a box.

The command expects that it is called after `\ParaSpaceAbove`.

After `\IgnoreSpaceAboveNextPara`:

- vertical list is not changed
- `parskip` := −0.01pt
- `prevdepth` is not changed

The special case is `parskip` less than 0pt, which cancels the vertical spacing. It is useful when display content (image, list, etc.) is the first element inside a table cell.

3 Paragraph designer

The paragraph designer transforms Python objects with desired paragraph properties into TEX code which implements these properties.

The main benefit is that the paragraphs definitions can be constructed in such way that the repetitions (for example, font names) can be extracted into common settings.

The system proposes that every block-level element of a document should be wrapped into a command or an environment, which support the galley approach. The suggested sorts of the paragraphs:

- long body text paragraphs, wrapped by an environment,
- short paragraphs, wrapped by a command, and
- short paragraphs with tab stops, also wrapped by a command.

A document made using this approach looks structured. Here is an example.

```
\HeadI{Universal Declaration of Human Rights}
\HeadII{Preamble}
\begin{para}Whereas recognition...\end{para}
\begin{para}Whereas disregard
    and contempt...\end{para}
...
\HeadII{Article 14}
\begin{udhrlist}
\listitem{1}{Everyone has the right ...}
\listitem{2}{This right may not be invoked ...}
\end{udhrlist}
```

The sample is generated automatically from the XML source. The generation script, the paragraph styles as Python definition, the `.sty` code, and the PDF result are included in the package in the directory `example`.

3.1 Example: the command `\HeadI`

Commands are recommended for small paragraphs, such as headings and captions.

```
\HeadI{Universal Declaration of Human Rights}
```

A sample definition in Python:

```
add_style(ParagraphOptions(cmd='HeadI',
    space_above='20pt',
    space_below='20pt',
    fontsize='12pt', baseline='14pt',
    fontcmd=r'\fontseries{b}\selectfont',
    afterpar=r'\nobreak',
    ))
```

The properties of the paragraph are stored inside the object `ParagraphOptions`. As in many other programming languages, the backslash (\) is normally an escape character (not in the TEX sense!), and must be doubled inside strings (\\). An alternative in Python, as seen in the example here, is to prefix the string with `r`, which disables the escape.

The function `add_style` remembers the object in the global styles list. At the end of the Python script, the objects in the list are converted to TEX code.

The result of the conversion:

```
\newcommand{\HeadI}[1]{{%
```

```
\fontsize{12pt}{14pt}\fontseries{b}\selectfont%
\ParaSpaceAbove{20pt}%
\noindent #1\par}%
\nobreak\ParaSpaceBelow{20pt}}
```

The peculiarities are:

- The paragraph is created explicitly with `\noindent #1\par`.
- The text and the pre-paragraph settings are in a group. This way settings such as font changes affect only the given paragraph and not the rest of the document.

3.2 Example: the environment `para`

Environments are recommended for wrapping paragraphs in the text body.

```
\begin{para}Whereas recognition...\end{para}
\begin{para}Whereas disregard
    and contempt...\end{para}
```

A sample definition in Python:

```
add_style(ParagraphOptions(cmd='paracmd',
    env='para',
    space_above='10pt plus1pt minus1pt',
    ))
```

The result of the conversion, in a `.sty` file:

```
\newenvironment{para}{%
  \ParaSpaceAbove{10pt plus1pt minus1pt}%
  \noindent \ignorespaces}
{\par\global\def\pd@after@para{%
  \ParaSpaceBelow{0pt}}%
  \aftergroup\pd@after@para}
```

The paragraph is started explicitly with the command `\noindent`, followed by `\ignorespaces`, and finished, also explicitly, with `\par`.

The changes inside an environment, including post-paragraph settings, are again local and thus automatically discarded when the environment's group is finished. Therefore, using `\aftergroup`, the post-paragraph settings are applied after the end of the environment.

3.3 Example: tab stops in `listitem`

Paragraphs with tab stops are used to implement list items, captions, table of content entries and similar elements. The list paragraphs in the following example have one tab stop to store the list numbering.

```
\listitem{1}{Everyone has the right ...}
\listitem{2}{This right
    may not be invoked ...}
```

A sample definition in Python:

```
add_style(ParagraphOptions(cmd='listitem',
    moresetup='\\interlinepenalty=150\\relax',
    space_above='8pt',
    boxes=(('0cm', '0.5cm'),),
    leftskip='0.5cm'))
```

The argument `boxes` is a list of pairs. Each pair gives the offset of the tab stop from left and the width of the box. Due to peculiarities of Python, one-element lists of pairs need an extra comma inside.

The position of the paragraph text should be tuned manually to avoid overlapping with the tab stop boxes. In the example above, the left margin is set to `0.5cm` using `\leftskip`.

The result of the conversion, in a `.sty` file, is complicated:

```
\newcommand{\listitem}[2]{{%
  \ParaSpaceAbove{8pt}%
  \interlinepenalty=150\relax%
  \noindent \advance\pd@leftskip by 0.5cm %
  \hbox to 0pt{\hss\hbox to 0.5cm{#1\hss}%
    \dimen0=0.5cm %
    \advance\dimen0 by -0cm %
    \advance\dimen0 by -0.5cm \hskip\dimen0}%
    \the\everypar #2\par}%
  \ParaSpaceBelow{0pt}}
```

The skeleton of the list paragraph has these elements:

`\noindent` *tab stops* `\everypar` *text* `\par`

The use of `\noindent` and `\par` is clear. The paragraph starts with the tab stop boxes, therefore TEX does not insert `\everypar` automatically, therefore the code does it.

The token `\pd@leftskip` is a `\let`-synonym for `\leftskip`. In a right-to-left document one would set the token to `\rightskip`.

A tab stop is constructed from two nested boxes. The inner box gives the width of the tab stop and aligns the content to the left:

`\hbox to` *width*`{`*content* `\hss}`

The outer box puts the inner box at the specified offset.

```
\hbox to 0pt{\hss inner_box%
\dimen0=leftskip
\advance\dimen0 by -offset
\advance\dimen0 by -width
\hskip\dimen0}%
```

The calculation is not obvious. The illustration in figure 1 provides the source for it.

The image reflects how the boxes, glues and lengths are related. We see that `offset`+`width`+`x` is `leftskip`, therefore `x` (`\dimen0`) is `leftskip` minus `offset` minus `width`.

4 Paragraph designer reference

Denomination: `cmd`, `env`, `stylecmd`. These are the names for the generated commands and environments.

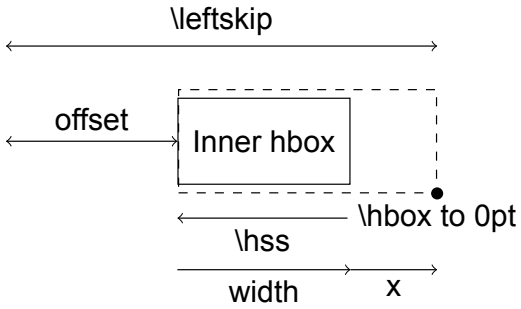


Figure 1: Calculation for tab stops.

Examples of `cmd` and `env` have already been given. The command for `stylecmd` makes a character style, which affects the font and does not set the paragraph properties (vertical spacing, tabulars, etc.).

A sample paragraph definition:

```
ParagraphOptions(cmd="Caption,
  stylecmd="UseCaption", ...)
```

In a LaTeX document you could then write:

```
{\UseCaption Article 1.} All human beings
are born free and equal in dignity ...
```

All the three denominators can be mixed together at once. You must specify `cmd` even if you don't need it.

Fonts: `fontsize`, `baseline`, `fontcmd`.

The only supported font properties are its size and line spacing. The other properties, such as width and series, need to be manually defined in `fontcmd`:

```
ParagraphOptions(...,
    fontcmd=r'\fontseries{b}\selectfont',
    ...)
```

Dimensions: `leftskip`, `hsize`, `space_above` and `space_below`.

The names are self-explanatory.

The default value for both `space_above` and `space_below` is `0pt`. This means that if you haven't given a value, then two consecutive paragraphs will touch each other, as if `\nointerlineskip` were given between them.

Use the special value `#natural` to disable the use of `\ParaSpaceAbove` or `\ParaSpaceBelow` and instead restore the default TeX behaviour.

```
ParagraphOptions(...,
  space_above='#natural',
  space_below='#natural', ...)
```

Tuning: `moresetup`, `afterpar`, `preamble_arg1`, `preamble_arg2`, `preamble_arg3`, `preamble_arg4`.

The content of `moresetup` is literally copied into the style definition at the end of the paragraph setup, just before `\noindent`. A few ideas what can be set in `moresetup`:

- A color for the paragraph text,
- `\penalty` to suggest a page break,
- `\interlinepenalty` for list item paragraphs, to avoid a page breaks inside.

The content of `afterpar` is literally copied into the style definition directly after `{...\par}`. This is a good place to put `\nobreak` or some other penalty.

The content of `preamble_argN` is copied literally into the style definition directly before `#N`. Possible applications:

- Add `\ignorespaces` if the text might contain spurious spaces at the beginning.
- For list item paragraphs, `\hfil` centers the tab box content, `\hfill` aligns to the right.

Tab stops. Tab stops are hboxes of a given width at given offset. All the offsets are relative to the left border of the text flow.

```
ParagraphOptions(...,
  boxes=(
    (offset1,width1),
    (offset2,width2),
    ...,
    (offset_n,width_n)),
  ...)
```

Due to Python peculiarities, a one-element list of lists needs an additional comma, otherwise Python unwraps one level of parentheses. Thus, the correct way is:

```
ParagraphOptions(...,
  boxes=((offset,width),), # comma inside
  ...)
```

The content of the boxes is left-aligned. To center or right-align the content, add `\hfil` or `\hfill` through the parameter `preamble_argN`.

Inheritance. The parameter `parent` uses an existing paragraph object as the starting point for the paragraph being defined. Properties not specified in the new paragraph definition are taken from the parent.

```
head_i = ParagraphOptions(
    cmd='HeadI',
    fontsize='12pt', baseline='14pt',
    fontcmd=r'\fontseries{b}\selectfont',
    ... )

ParagraphOptions(cmd='HeadII',
    parent=head_i, # Inheritance
    fontsize='11pt', baseline='13pt',
    ... )
```

In the example, the paragraph `HeadII` inherits `fontcmd` from `HeadI`, but uses the custom `fontsize` and `baseline` settings.

The infrastructure. A Python file with definitions: (1) starts by importing the support code; (2) continues with collecting the definitions; and (3) finishes with the command to dump the TEX result.

```
from parades import * # (1)

add_style(ParagraphOptions(...)) # (2)
add_style(ParagraphOptions(...))
...
add_style(ParagraphOptions(...))

main('paras') # (3)
```

The parameter of the function `main` (in this example `paras`) is the name of the generated sty-package as given by `\ProvidesPackage`.

5 Getting and running the code

All the files, including the example, are contained in the CTAN package `parades` (`http://ctan.org/pkg/parades`). Alternatively, you can get the source code from `github` in the repository `http://github.com/olpa/tex`, in the folder `paragraph_designer`.

Put the file `paravesp.sty` into a directory in which TEX will find it. Put the file `parades.py` into a directory in which Python will find it.

The paragraph generator runs from the command line.

```
$ python input-defs.py [output-defs.sty]
```

The script `input-defs.py` is the file with the Python definitions of the paragraphs. The optional argument is the name of a `.sty` file with the generated TEX definitions. If the output file is not specified, the code is dumped to the standard output.

The directory `example` contains a sample project. Refer to the file `README` in this directory for details how to use it.

6 Conclusion

The paragraph designer helps both on the technical and organization levels. On the technical level, it helps generating code for paragraph styles. It would be an unpleasant and error-prone task to write this code manually:

- Space above and below a paragraph.
- Paragraphs with tab stops such as list items, table of content entries, headers.

On the organizational level, the Python scripts allow one to have a common code base and adapt it to the needs of specific layouts.

The LATEX package `paravesp` can be used independently of the paragraph designer to implement vertical spacing.

There are problems with the package `paravesp` and the paragraph designer:

- Many features are not implemented and some need rework.
- The LATEX code written in the galley style is too verbose to be typeset manually.

The paragraph designer has been used in a production system for years. Thus the benefits can outweigh the problems.

◇ Oleg Parashchenko
bitplant.de GmbH
Fabrikstr. 15
89520 Heidenheim, Germany
`olpa (at) uucode dot com`
`http://uucode.com/`