Introduction

Background.

If the background doesn't interest you, and you just want to get started with Latexslides, jump to the 'Getting started' section.

For many people in the fields of natural sciences, LaTeX is the preferred typesetting tool. Therefore, it didn't take long before different packages where developed allowing people to also create slides with LaTeX. Out of many, Prosper and Beamer are probably the best known. The most popular package of these two is Prosper, while Beamer is the latest, the fanciest, and the most PowerPoint-like package for LaTeX slides. Unfortunately, there is a lot of LaTeX commands that you need to insert to make LaTeX slides. What if we could just write the basic slide contents and insert a minimum of tags? That is, it could be nice with a simpler syntax than what LaTeX slide packages require.

Another point is that it takes quite some boring work to update old presentations, written in Prosper or even older syntax, to a new package like Beamer. And what about the next super-package for LaTeX slides? It would be good if all your slides were written with a LaTeX-independent syntax such that the slides can easily be ported to a new (or old) format.

A third point is that sometimes we would like to automate the writing of slides, for example when creating animations where several slides with small differences are typically written by manual cut and paste operations. It would be better to have a program doing this.

From these points, it became natural to specify slides as program code and automate the generation of the specific syntax for different LaTeX packages. Our solution is Latexslides, which is written in Python and requires you to write your slides in Python. Typically, each slide is a `Slide` object, to which you assign a name, and then you can collect the names in lists to compose a particular presentation. It is easy to have your slides in modules such that new presentations can import slides from other presentations. This gives great flexibility in reusing slides from presentation to presentation without making a copy of the slides. The LaTeX knowledge required from the end-user is also less when using Latexslides than using plain LaTeX (then again, the Python knowledge needed is much larger). But some of the advanced arguments to Latexslides functions might still require a good LaTeX book.

Another important point is, of course, that whatever can be done in Python, should be done in Python :-) Also slides...

History.

Initially, `LaTeXSlides` was a module and script written by Hans Petter Langtangen and later improved by Åsmund Ødegård, both at Simula Research Laboratory, Oslo, Norway. `LaTeXSlides` soon became quite popular as it auto-generated both Beamer and Prosper slides from a minimum of Python commands specifying the slides. However, `LaTeXSlides` needed a complete redesign to better scale with future demands. Arve Knudsen, also from Simula, started this effort, while this author did the complete job of creating the new Latexslide package for slide generation in Python. The old API of `LaTeXSlides` is still supported for backward compatibility.

Getting started

If you want to get started with a more advanced example, it is recommended that you take a look at the file `doc/exampletalk.py` which demonstrates many of the functionalities that are available. Going through the source code of this talk at the same

time as reading the PDF generated from it, should get you started in less than half an hour.

Introductory example.

You can generate a new, empty presentation by typing

```
python -c 'import latexslides; latexslides.generate("myfile.py")'
```

Just invoke and edit the file `myfile.py`.

We will now look at a short example introducing the main functionalities of Latexslides. We look at the source code of two slides, and the result for Beamer can be viewed in the figures below. The first slide consists of two parts, the left part contains a figure, and the right part contains some text as well as a few lines of code, see Figures 1 and 2. The Python code used for generating these slides is as follows:

```
from latexslides import *

author_and_inst = [("John Doe", "Royal University of Nothing")]

slides = BeamerSlides(title="Short Introduction",
                      titlepage=False,
                      toc_heading=None,
                      author_and_inst=author_and_inst,)

slide1 = Slide("Slide 1",
               content=[TextBlock(r"""
Program for computing the height of a ball thrown up in the air:
$y=v_0t- {1\over 2} gt^2$""") ,
                        Code(file='code.py')],
               figure='brainhurts.ps',
               figure_pos='w',
               figure_fraction_width=0.4,
               left_column_width=0.3,
               )

slide2 = BulletSlide("Slide 2",
                     ["Latexslides is flexible:",
                      ["One can create slides based on Python elements",
                       r"\LaTeX~code can be inserted directly in the code"],
                      "Latexslides means less code and less time used:",
                      ["The code is generated automatically",
                       r"One does not need to learn \LaTeX"],
                      "The package is easily installed by typing" +
                      Code("python setup.py install"),],
                     block_heading="Advantages of latexslides",
                     )

collection = [slide1, slide2]
slides.add_slides(collection)

# Dump to file:
slides.write("intro.tex")
```

Before dissecting each individual slide, we will take a look at the general layout of this code. First, we need to import the Python module. Then, we define the parts of the presentation that are common for all slides, including the name and affiliation of the authors and the date. This is done by creating an instance of one of the subclasses of `Slides`, meaning either `BeamerSlides`, `ProsperSlides` or `HTMLSlides`, hereafter referred to as the main instance. Here we also define what both the title page and the rest of the presentation should look like, and how it should behave. The details on this can be found further down in this tutorial. For this introductory example, we have opted to show neither the title page nor the table of contents.

The next step is to define the objects that represent a slide. When we are finished with all slides, we have to collect them and add them to the main instance in the order we would like them to appear. Finally, we dump the whole presentation to file.

Our first slide is a general `Slide` object. The title is set to "Slide 1", and a figure is included on the slide. The figure is placed to the left of the slide, and set not to exceed more than 3/10 of the slide. It also had to be shrunk to 4/10 of its original size to fit on the slide. The right side of the slide consists of two elements, a `TextBlock` and some code. It is worth noticing that the Python code displayed on this slide is not part of the code defining the slide. Rather, we give the `Code` class an argument `file=code.py`, resulting in the Python code being read from the file `code.py`. Figure 1 displays the first slide.

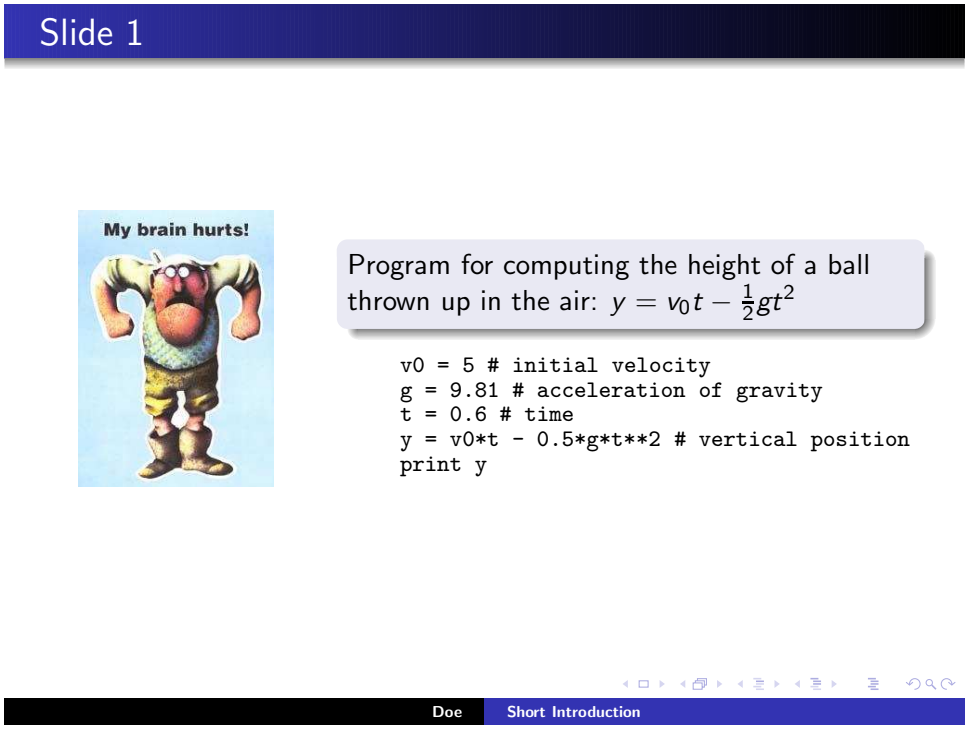

Figure 1: First slide of introductory example.

The second slide is a `BulletSlide`, which is less general than the super class `Slide`. This means that we have a less complicated interface at the cost of flexibility. The bullets are lists of string elements, but nested lists are allowed, as is shown. Finally, we opted to add a heading to the block making up the bullets. If omitted, no heading is displayed. Figure 2 displays the second slide.

Documentation of the classes

Keyword arguments.

Objects in Latexslides makes heavily use of keyword arguments. This means that when creating a new object, there are very few, if any, arguments that are mandatory. A lot of arguments are optional, and will be ignored if not present, or suitable default values will be used.

Initialization.

After an

```
from latexslides import *
```

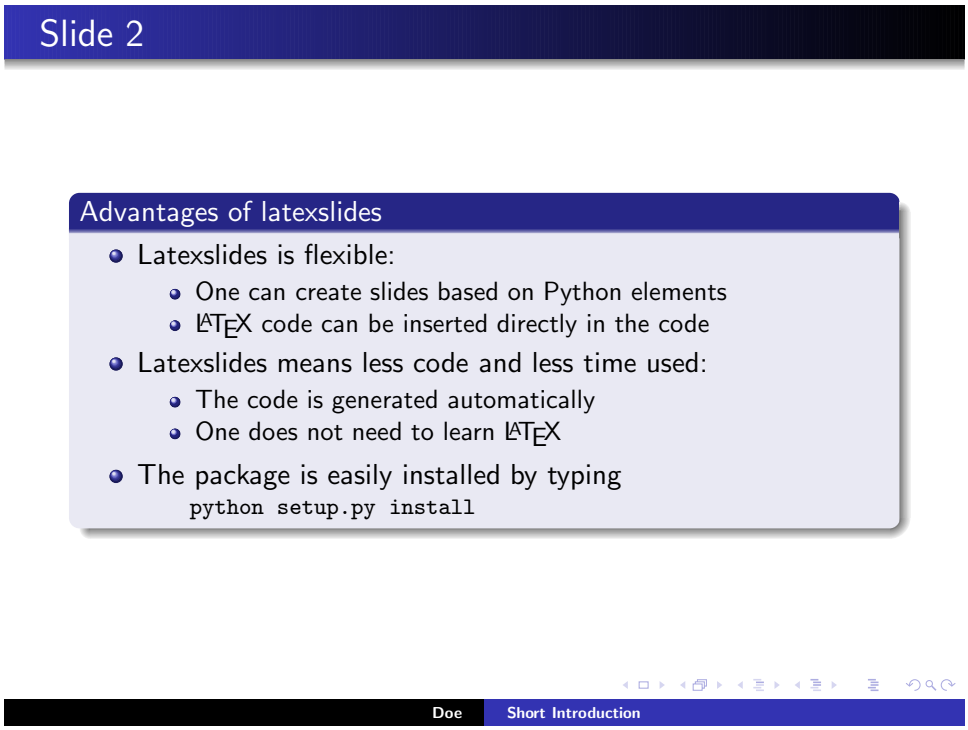

Figure 2: Second slide of introductory example.

the first instance we need to create is an instance of a subclass of the class `Slides`. This can be (at this time) either `ProsperSlides`, `BeamerSlides` or `HTMLSlides`. The default values and the Python data types for each keyword argument will be given below as 'Default' and 'Type'. If a keyword argument not works for all packages (subclasses), the name of the packages for which it works for will be given as well under 'Package'. Finally, necessary comments will be given under 'Comments'. It is worth noting the use of the keyword arguments `n`, `s`, `e`, and `w`. These correspond to north, south, east, and west, respectively. North indicates the top of a page, east the left of a page etc.

Here is a list of keyword arguments for the constructor of a `Slide` instance:

- `title`
  - Default: `"Here goes the title of the talk"`
  - Type: str
- `author_and_inst`
  - Default: `[('author1', 'inst1'), ('author2', 'inst2', 'inst3')]`
  - Type: list of tuples of str
- `date`
  - Default: `None`
  - Type: str
  - Comment: Today's date is chosen if not set.
- `titlepage`

  - Default: `True`
  - Type: bool
  - Comment: If set to `False`, the title page is skipped.

- `titlepage_figure`
  - Default: `None`
  - Type: str
  - Package: Beamer
  - Comment: Give name of figure file to appear on title page.

- `titlepage_figure_pos`
  - Default: `"s"`
  - Type: str
  - Package: Beamer
  - Comments: Position of figure on the title page. Values can be either `e`, or `s`. Really, anything but `s` results in `e`.

- `titlepage_figure_fraction_width`
  - Default: `1.0`
  - Type: float
  - Package: Beamer
  - Comment: Scales the picture, while keeping aspect ratio.

- `titlepage_left_column_width`
  - Default: `0.5`
  - Type: float
  - Package: Beamer
  - Comment: If the figure is positioned to the left or right, two columns are used. This value sets the relative size of the left column, and should be between `0` and `1`. The size of the right column is 1 minus this value.

- `short_title`
  - Default: `""`
  - Type: str
  - Package: Beamer
  - Comment: This value, if present, is used instead of the main title if it is too large to fit somewhere in the presentation.

- `short_author`
  - Default: `None`
  - Type: str
  - Package: Beamer
  - Comment: If the name of the authors(s) is too long to fit, this value, if present, is used instead. Else a string consisting of the first author's last name with 'et al.' added to it is used.

- `copyright_text`
  - Default: `None`
  - Type: str
  - Package: Prosper
- `toc_heading`
  - Default: `"Outline"`
  - Type: str
  - Package: Beamer
  - Comment: The heading of table of contents, shown between all sections and subsections. If set to `None` or an empty string, the table of contents is skipped. This might be preferable for shorter presentations.
- `toc_figure`
  - Default: `None`
  - Type: str
  - Package: Beamer
  - Comment: Give name of file containing the figure to be shown on the table of contents slide.
- `toc_figure_fraction_width`
  - Default: `1.0`
  - Type: float
  - Package: Beamer
  - Comment: Scales the picture, while keeping aspect ratio.
- `toc_left_column_width`
  - Default: `0.5`
  - Type: float
  - Package: Beamer
  - Comment: The figure is positioned to the left of the table of contents, hence two columns are used. This value sets the size of the left column, the size of the right column is 1 minus this value.
- `colour`
  - Default: `True`
  - Type: bool
  - Comment: A `False` value makes the slides more suited for printing. Table of contents is dropped, and colours are removed from the background. Different behaviour for the different packages. Beamer forces the use of the theme `seahorse` when the value is `False`.
- `handout`
  - Default: `False`
  - Type: bool
  - Package: Beamer

  - Comment: Makes the slides more suited for printing. The table of contents is dropped, and colours are removed from the background. Different behaviour for the different packages. The same as `colour`, but without `seahorse` as the forced colour theme.

- `beamer_theme`
  - Default: `"shadow"`
  - Type: str
  - Package: Beamer
  - Comment: All default Beamer themes are available. In addition the themes `simula` and `hpl1` are included with the Latexslides package. Different themes might lead to different output. The use of `simula` or `hpl1` is recommended. Read more about styles in the section "Styles".

- `beamer_colour_theme`
  - Default: `"default"`
  - Type: str
  - Package: Beamer
  - Comment: All default Beamer colour themes are available. For printing, the colour theme `seahorse` is recommended, as it is almost black and white. Take care not to use the `simula` theme with this one, as not all colours that are set there are overridden by this colour theme, resulting in some orange. If `colour` is set to `False`, this argument is overridden and the theme set to `seahorse`.

- `prosper_style`
  - Default: `"default"`
  - Type: str
  - Package: Prosper
  - Comment: all default Prosper styles are available. In addition the style `hplplainsmall` is included with the Latexslides package. The use of `hplplainsmall` is recommended, as it matches the font sizes of the Beamer themes better, making it less likely for text not to fit on the slides when switching from Beamer to Prosper than with `default`.

- `header_footer`
  - Default: `True`
  - Type: bool
  - Package: Beamer
  - Comment: Removes the top and bottom of the slide. On the top, the index over sections is removed, at the bottom the authors and title are removed. Theme dependent.

- `html`
  - Default: `False`
  - Type: bool
  - Package: Beamer

  - Comment: This enables the possibility to dump the slides to HTML. Setting this variable to `True` and `toc_heading` to an empty string gives the same result as using HTMLSlides instead of BeamerSlides (HTMLSlides simply sets this variable to `True` and `toc_heading` to an empty string). Requires the package TeX4ht to be installed.

- `newcommands`
  - Default: `[]`
  - Type: list or str
  - Comment: Adds the user's own commands to the top of the LaTeX file, useful because Latexslides supports raw LaTeX code. Can be given as a list of strings or a string separated by newline. Only `\newcommand` LaTeX commands can be given. The initial command (`\newcommand`) can be skipped.

- `latexpackages`
  - Default: `""`
  - Type: str
  - Comment: Adds additional packages to the list of packages at the beginning of the LaTeX file. In order to allow options to the packages, it is a simple multi-line string, e.g. the exact string otherwise written in the LaTeX file.

Adding slides.

After having created the `BeamerSlides` or `ProsperSlides` object, we are ready to start creating individual slides. There are five types of slides:

- `TextSlide`
- `BulletSlide`
- `Rawslide`
- `TableSlide`
- `Slide`

The last one can consist of several objects, whereas the first four slides are meant for plain text, plain bullet lists, creation of LaTeX tables based on Python lists, or plain LaTeX, respectively. For these, blocks are used when available.

`TextSlide` has the following keyword arguments:

- `title`
  - Default: `""`
  - Type: str
  - Comment: When empty string is given, no title is displayed.

- `text`
  - Default: `""`
  - Type: str
  - Comment: The text to be placed on the slide.

- `block_heading`
  - Default: `""`
  - Type: str
  - Comment: Additional title for the block. When empty string is given, no title is displayed.
- `hidden`
  - Default: `False`
  - Type: bool
  - Comment: If `True`, the slide is skipped. The properties `hide` and `unhide` can also be used, and will return the slide element itself, see the sections 'Organizing the objects' and 'Writing to file' further down.

`BulletSlide` has four keyword arguments:

- `title`
  - Default: `""`
  - Type: str
  - Comment: When empty string is given, no title is displayed.
- `bullets`
  - Default: `[]`
  - Type: list of strings
  - Comment: Each element in the list represents a bullet. The list can be nested, so that if one of the elements of the main list is a list, the items in that list will show up as sub-bullets.
- `block_heading`
  - Default: `""`
  - Type: str
  - Comment: Additional title for the block. When empty string is given, no title is displayed.
- `hidden`
  - Default: `False`
  - Type: bool
  - Comment: If `True`, the slide is skipped.
- `dim`
  - Default: `None`
  - Type: str
  - Package: Beamer
  - Comment: Available dimming settings are: `progressive` (one by one bullet), `single` (only one bullet at the time), `single_then_all` (only one bullet at the time, then all), and `blocks` (one block at the time, not relevant for `BulletSlide`).

`RawSlide` has two keyword arguments:

- `rawtext`
  - Default: `""`
  - Type: str
  - Comment: Enables the user to design her own slide using LaTeX commands directly. For example, one can include pure LaTeX code from old presentations. One should ensure that the string is a raw string preserving backslash and the like. If this class is used, changing between the different packages (e.g. Beamer and Prosper) would no longer be possible, as the code for these two differ quite a bit. As `HTMLSlides` is compatible with `BeamerSlides` these two, however, can still be interchanged.
- `hidden`
  - Default: `False`
  - Type: bool
  - Comment: If `True`, the slide is skipped.

`TableSlide` enables the generation of LaTeX tables based on nested Python lists, and has seven keyword arguments:

- `title`
  - Default: `""`
  - Type: str
  - Comment: When empty string is given, no title is displayed.
- `table`:
  - Default: `[[],[]]`
  - Type: Nested list of strings
  - Comment: The first entry in the outer list should contain the column heading for the LaTeX table to be generated. The remaining entries are the lines in the table. See also the documentation of `Table` further down.
- `column_heading_pos`
  - Default: `"c"`
  - Type: str
  - Comment: See documentation of `Table`.
- `column_pos`
  - Default: `"c"`
  - Type: str
  - Comment: See documentation of `Table`.
- `center`
  - Default: `False`
  - Type: bool
  - Comment: See documentation of `Table`.

- `block_heading`
  - Default: `""`
  - Type: str
  - Comment: Additional title for the block. When empty string is given, no title is displayed.
- `hidden`
  - Default: `False`
  - Type: bool
  - Comment: If `True`, the slide is skipped.

Slide.

A general slide can consist of several objects, and we will look at them now:

- `Text`
- `BulletList`
- `Code`
- `Table`
- `Block`
- `TextBlock`
- `BulletBlock`
- `CodeBlock`
- `TableBlock`

The difference between a plain object (like `BulletList`) and a block object (like `BulletBlock`), is that the latter is surrounded by a shadowed box (depending on the slide format, Beamer or Prosper, for instance), and has an optional heading for that box.

`Text` takes only one argument, and that is the text.
`BulletList` has `bullets` as argument and `dim` as keyword argument:

- `bullets`
  - Default: `[]`
  - Type: list of strings
  - Comment: Each element in the list represents a bullet. The list can be nested, so that if one of the elements of the main list is a list, the items in that list will show up as sub-bullets.
- `dim`
  - Default: `True`
  - Type: bool
  - Package: Beamer

  - Comment: This one is only used to override dimming if dimming is turned on for the whole slide, so that the whole block is displayed at once even though the rest of the bullet objects on the slide appear one by one.

`Code` has a number of keyword arguments, mainly because it supports the option of reading the code from a file, or even only reading the text in between a start and a stop expression:

- `code`
  - Default: `None`
  - Type: str
  - Comment: The code to be inserted. Is ignored if reading from file is chosen.
- `file`
  - Default: `None`
  - Type: str
  - Comment: The name of the file to include code from. If `from_regex` or `to_regex` is not set, the whole file is included.
- `from_regex`
  - Default: `None`
  - Type: str
  - Comment: Can only be used if `to_regex` is set as well. The file in inspected line for line, and if the regular expression matches, that line is included. All further lines are then included, until there is a match for `to_regex`. The line that matches `to_regex` itself is not included.
- `to_regex`
  - Default: `None`
  - Type: str
  - Comment: See `from_regex`
- `leftmargin`
  - Default: `"7mm"`
  - Type: str
  - Comment: Sets the left margin for the code.
- `fontsize`
  - Default: `"footnotesize"` (if the string does not start with a backslash, it is added automatically).
  - Type: str
  - Comment: Sets the size of the code font. Needs to be a valid LaTeX command ranging from `tiny` to `Huge` (if the string does not start with a backslash, it as added automatically).

`Table` gives us the possibility for creating LaTeX tables directly based on (nested) Python lists. Writing tables in LaTeX is a tedious task, so automatically generating it based on the Python lists saves us the effort of having to do it manually. Also, if we need to make changes to our table, we can simply change the Python lists, as the LaTeX table is built over again every time we run Latexslides. `Table` has `table` as argument in addition to two keyword arguments:

- `table`
  - Default: `[[],[]]`
  - Type: Nested list of strings
  - Comment: The first entry in the main list will be column headlines. The next entries in the list are the rows following the headline. All nested lists (i.e. each element in the main list) should have the same length. For example: `table=[[r"$h$", r"$\Omega$", "n"], [0.1, 0.001, 100]]`. If an element encountered in one of the sub-lists is not a string, the class will try to cast it to a string.
- `column_headline_pos`
  - Default: `"c"`
  - Type: str
  - Comment: The position of the columns for the headline row, valid values are `l`, `c`, and `r` for left, center, and right, respectively. If only a single character, it is the same for every column. Else, the characters in the string are used one-by-one for every column. For instance, the string `crl` would mean that the first headline column is centered, the second one right justified, and the third one left justified. Unless omitted or given as a string with length one, it should be the same length as the numbers of columns.
- `column_pos`
  - Default: `"c"`
  - Type: str
  - Comment: The position of the text for each column. If only a single character, it is the same for every column. Else, the characters in the string are used one-by-one for every column. For instance, the string `crl` would mean that the first ordinary column (not he headline) is centered, the second one right justified, and the third one left justified. Unless omitted or given with length one, it should be the same length as the numbers of columns given.

`Block` allows for the creation of block object consisting of a combination of the three previous types (`Text`, `BulletList`, and `Code`). Hence it is more general than the three block objects that follow:

- `heading`
  - Default: `""`
  - Type: str
  - Comment: If given, used as block title.
- `content`

  - Default: `[]`
  - Type: list
  - Comment: All objects to be placed within the block are sent as a list.

`TextBlock` is the same as `Text`, but has an additional keyword argument:

- `heading`
  - Default: `""`
  - Type: str
  - Comment: If given, used as block title.

`BulletBlock` is the same as `BulletList`, but has an additional keyword argument:

- `heading`
  - Default: `""`
  - Type: str
  - Comment: If given, used as block title.

`CodeBlock` is the same as `Code`, but has an additional keyword argument:

- `heading`
  - Default: `""`
  - Type: str
  - Comment: If given, used as block title.

`TableBlock` is the same as `Table`, but has these additional keyword arguments:

- `heading`
  - Default: `""`
  - Type: str
  - Comment: If given, used as block title.
- `center`
  - Default: `False`
  - Type: bool
  - Comment: If `True`, center the table within the block.

Creating the slide instance. Once we have all the objects we want a slide to consist of, we can create the slide instance itself. All arguments are keyword arguments:

- `title`
  - Default: `""`
  - Type: str
  - Comment: When empty string is given, no title is displayed.
- `content`
  - Default: `[]`

  - Type: list
  - Comment: All objects to be placed on the slide are sent as a list.

- `figure`
  - Default: `None`
  - Type: str or tuple of str
  - Comment: Give name of file or tuple of file names.

- `figure_pos`
  - Default: `s`
  - Type: str
  - Comments: Figure can be placed north (`n`), east (`e`), south (`s`), and west (`w`) on the slide. The same for all figures.

- `figure_fraction_width`
  - Default: `1.0`
  - Type: float
  - Comment: Scales the picture, keeps aspect ratio. `figure_size` can be used as well. In case both options are given, `figure_size` is chosen. If multiple figures are given with figure, the length of this argument is expected to be the same as the number of figures, or else the program exits with an error. The exception is if the length of the argument is 1 (or it is omitted), then this value (or the default of 1) will be used for all figures.

- `figure_angle`
  - Default: `None`
  - Type: int or str
  - Comment: Give angle in degrees in the counter-clockwise direction to rotate image. Negative values can be used. The same for all figures.

- `left_column_width`
  - Default: `0.5`
  - Type: float
  - Comment: If the figure is positioned to the left or right, two columns are used. This value sets the size of the left column, the size of the right column is 1 minus this value.

- `hidden`
  - Default: `False`
  - Type: bool
  - Comment: If `True`, the slide is skipped.

- `dim`
  - Default: `None`
  - Type: str
  - Package: Beamer

- Comment: Available dimming settings are: `progressive` (one by one bullet), `single` (only one bullet at the time), `single_then_all` (only one bullet at the time, then all), and `blocks` (one block at the time). Sub-items will always be shown together with their parent. If a specific block has the option `dim` set to `False`, the `dim` option for the slide is overridden. Setting `dim` for a block to `False` should only be used with `progressive`, as adding something to a slide that is not part of a block, can cuase the result to be unsatisfactory.

Adding sections.

One can insert special slides to mark sections in the talk. The section slides will automatically appear in a table of contents and optionally in the header in Beamer (but not in Prosper). Sections are added in the same way as normal slides, except they are of type `Section`. If no section is defined, normal slides are simply added to a list of slides. If a section is defined, all slides added after that object will be part of this section, until a new section is defined. It has two keyword arguments:

- `title`
  - Default: `""`
  - Type: str
  - Comment: The title of the section
- `short_title`
  - Default: `""`
  - Type: str
  - Comment: Used when there is no space for the main title, for instance in the header.

Adding subsections.

Class `SubSection` is added in the same way as `Section`. If a subsection is added before a section is added, an error is issued and the generation of slides terminated. If a subsection is defined, all slides added after that object will be part of this subsection, until a new section or subsection is defined. It has two keyword arguments:

- `title`
  - Default: `""`
  - Type: str
  - Comment: The title of the section
- `short_title`
  - Default: `""`
  - Type: str
  - Comment: Used when there is no space for the main title, for instance in the header.

Organizing the objects.

Instead of creating a slide and adding it immediately to the initial `Slides` object using the the function `add_slide` in `Slides`, it might be better to generate all the slide objects, and then add all the slides at the end of the file. This way, it is easier to change the order of the slides. This is done in both the introductory example and the `exampletalk.py` script. One can either use a 'for'-loop and call the function `add_slide` for each element in the list, or call the function `add_slides` which takes the whole list as argument. If the presentation exists of the objects `slide1`, `section1`, `slide2`, `subsection1`, `slide3`, one can add all those to a list at the end of the presentation. If `slides` is the initial `Slide` object, and `collection` is a list of slides, one can use any of the two following ways of adding the slides in the list to the `slides` object:

```
for slide in collection:
    slides.add_slide(slide)
```

or

```
slides.add_slides(collection)
```

The list of slide objects can be generated automatically, more about this in the next section.

Additional scripts

Naming slide objects.

The slide object that one creates need to be stored in a Python variable, and instead of naming these object `slide1`, `slide2` etc. it would be preferable to name them according to the slide title. When writing many slides, it would be easier to just create the slide objects without naming them, and to run a script to automatically name these objects when finished. The executable `create_slidenames` which is part of Latexslides does exactly that. Running

```
create_slidenames exampletalk.py
```

means the script searches the given file for lines starting with `= Slide`, `= Section`, or `= SubSection`, and adds a line consisting of the title of the slide before these lines. Special characters as well as spaces and Norwegian characters are substituted so the variable name is valid in Python. Spaces and the character `'='` are substituted by underscores, Norwegian characters are substituted by plain ASCII letters (ø by o, å by aa and æ by ae), and other characters are simply removed.

Let's look at an example. The line

```
= BulletSlide('Character test !"#%&/()[]^ØÆÅ',)
```

becomes

```
Character_test_OAEAA \
= BulletSlide('Character test !"#%&/()[]^ØÆÅ',)
```

and the line

```
= Section('More about {if "__name__" == __main__}',)
```

becomes

```
More_about_if___name________main__ \
= Section('More about {if "__name__" == __main__}',)
```

In this way, we don't need to worry about naming the Python objects. Note that one needs to use the exact same syntax, i.e. starting a line with the character `'='` followed by the name of the class to be used. The space between those two can be skipped. Also note that the created name of the object does not change automatically if the title is changed. When running, the script examines the previous line to see if we already have assigned the instance to a variable name, and if so, nothing is done. In this way, running the script several times should not result in multiple declarations of the variable name, however, this also results in the variable name not being updated when the title of the slide or section is changed. Deleting the line containing the variable name and running the script again should fix this. Any instance that already has a variable assigned to it manually on the same line, for instance when writing `myslide = Slide('This is the title',)`, is also skipped. This script changes the actual file, however a backup is made with the extension `.old~~`, so running

```
create_slidenames exampletalk.py
```

would result in `exampletalk.py` now containing the variable names as well, whereas `exampletalk.py.old~~` would contain the original file.

Extracting slide names.

Once you are finished writing the talk you need to add all the slide objects to the main instance of the `Slides` class used (e.g. `BeamerSlides`). The easiest way of keeping an overview is to do this at the end of the Python file, rather then after creating each slide, as discussed in the previous section. We included a script for extracting the variable names of all slides in the file in the script `extract_slidenames`. It runs through the file scanning for the variable names, and prints out a Python list in the terminal that can be copied directly in the the bottom of the Python file. So when running

```
extract_slidenames exampletalk.py
```

the file `exampletalk.py` remains the same. Instead of copying the test we could add the list to the bottom of the file automatically:

```
extract_slidenames exampletalk.py >> exampletalk.py
```

Note that if we add or remove slides, we need to regenerate this list.

Converting talks to the OpenOffice format.

Another script that comes as part of Latexslides is `pdf2odp`. It is used for converting PDF files in general to a format recognized by OpenOffice. This script is described in more detail in the section 'Using talks in OpenOffice' further down.

Writing to file

When finished writing the presentation, we can dump the LaTeX code to file. This is done by simply calling the function `write` for the initial `Slides` object (of class `BeamerSlides`, `ProsperSlides` etc.). The argument is the file name. The `write` function also outputs the necessary commands one needs to run for creating the slides for the specific package used (Prosper, Beamer, HTML). So if all slide instances are collected in a list `collection`, these are dumped to file by writing

```
for c in collection:
    slides.add_slide(c)
filename = 'mytalk.tex'
slides.write(filename)
```

or (even simpler)

```
slides.add_slides(collection)
filename = 'mytalk.tex'
slides.write(filename)
```

If there are some slides that you do not want to be part of the presentation, you do not have to delete them. You can either remove the slides you do not want to include from the list, or even simpler, mark them as hidden. This can be done in three ways.

1. Set the keyword argument `hidden` to `True` when creating the slide.
2. Set the attribute `hidden` to `True`. If you slide instance is called `titleslide`, it will not be printed if you write `titleslide.hidden=True`.
3. Add the property `hide` to an object when adding it. It returns the object with the attribute `hidden` set to `True`. So instead of writing

   ```
   slides.add_slide(titleslide)
   ```

you would write

```
slides.add_slide(titleslide.hide)
```

Compiling the talk

If the function `slides.write()` is used, the necessary commands for compiling the talk are output to screen. If not, one simply uses `latex mytalk.tex`. Please note that the generation of HTML slides requires running `tex4ht` and `t4ht` afterwards.

About figure support

Because Prosper cannot be used with `pdflatex`, `latex` is recommended. This means, however, that only .ps images can be used, not .png, if portability between Prosper and Beamer is desirable. It is recommended that you use the ImageMagick `convert` tool for converting the PNG image to PostScript, and including them as .ps or .eps files.

Using talks in OpenOffice

Latexslides includes a script for converting presentations in the PDF format to the OpenOffice Impress format (.odp). OpenOffice Impress is a presentation tool similar to Microsoft Office PowerPoint. OpenOffice is a free, open source alternative to Microsoft Office. One could convert the .odp file further to a PowerPoint file (.ppt), either by opening it in Impress and opting to save it as a .ppt file, or through a commandline script like PyODConverter (http://www.artofsolving.com/opensource/pyodconverter).

Sometimes when giving a presentation, you might experience that the organizers only accept .ppt or .odp files. `pdf2odp`, which comes as part of Latexslides, solves this problem. Of course, not accepting PDF files these days would be surprising, and hence this is not the main objective of the script. The reason it was written is that sometimes, one might want to add a few extras to the presentation. For instance, one could highlight certain words by adding a circle, drawing some arrows, or one could use some of the advanced possibilities provided by presentation tools like Impress and PowerPoint without having to write the whole presentation with this tool, something that we already agreed upon can be rather tedious. In this way,

we can combine the best from two worlds; the simplicity of Latexslides and the advanced interactive functionality of Impress or PowerPoint.

The script `pdf2odp` uses a Python module called `odfpy`. An error is given before exiting the script in case this module is unavailable. If the module is found, Ghostscript is used for converting the input file, which should be a PDF file, to a set of PNG files, one for each page in the input file. This conversion process can take some time. If Ghostscript is unavailable, the script will exit with an error. When the conversion is finished, `odfpy` is used to create the OpenOffice file. Each slide in the new presentation contains an image that corresponds to the slide in the input file, and this image covers the whole slide. Finally, the OpenOffice file is saved and the scripts exits.

As the slides from the original PDF file merely are images in the new .odp file, we loose some of the functionality from the PDF file, mainly the linking within the document. With Beamer, for instance, there are links in the presentation allowing one to navigate within the slides. These links will now simply be part of the image, and cannot be clicked. Also, as all the text is part of an image, it is no longer possible to copy the text or index the file in any way.

Package specifics

ProsperSlides.

Some of the keyword arguments are only available to BeamerSlides and HTMLSlides. If these keyword arguments are present when using ProsperSlides, they are simply ignored.

HTMLSlides.

HTMLSlides is really the same as BeamerSlides, with the following modifications:

- The `html` keyword for Slides is set to `True`.
- The `toc_heading` keyword for Slides is set to en empty string, removing the table of contents.
- All sections and subsections are removed.

After Latexslides has generated the LaTeX code, the commands `tex4ht` and `t4ht` are used to generate the actual HTML file.

Styles

Several LaTeX styles are included with this package. These are:

- beamerthemsimula.sty
- beamerthemehpl1.sty

for Beamer and:

- PPRhplplain.sty
- PPRhplplain2.sty
- PPRhplplainsmall.sty

for Prosper. These are located in the folder `styles`. If you want to use these, you might want to set the following in your shell start-up file (for Bash, this is `~/.bashrc`:

```
export TEXINPUTS=:.:/absolute/path/to/styles
```

Note the first `'.'`; it ensures that the system-wide directories are searched first. More information on how to install the style files to the correct directories is found in the README file.

Emacs bindings

The following Emacs command (Alt + up-arrow) starts a slide object without a variable name:

```
(global-set-key [ (meta up)] " = Slide('',
content=[BulletBlock(bullets=[
    '',")
```

Bullet points can then be written. The end of the `Slide` object is automatically generated by this Emacs command (Alt + down-arrow):

```
(global-set-key [ (meta down)] "
]),  # end bullets and BulletBlock
],   # end contents
)")
```

In order for these to work, the commands need to be included in the file `.emacs` in your home directory. Sorry, no vim bindings!

Oldlatexslides

The source code from the old `LaTeXSlides` package can still be used. However it needs a few modifications. We start by importing the module:

```
import latexslides.old as LaTeXSlides
```

Here, we assume that the import statement in your old presentations was `import LaTeXSlides`, not `from LaTeXSlides import *`. If you used the latter, you might want to try

```
from latexslides.old import *
```

The only difference is that the header, titlepage, and footer objects are to be omitted, meaning they should not be part of the list that is created at the end of the talk. Finally, the `header_footer` argument that was used for each bullet-slide in the old package is now a global variable set in the initialization of the main slides class, and is the same for all slides.

Support

Please contact `ilmarw@simula.no` for bug reports, feature requests and general help.