pylama: Make typesetting great again

Nils Bore

1 Introduction

Pylama can be best described as a wrapper around the `LaTeX` typesetting system. It attempts to provide a cleaner syntax for your standard document while still allowing you to add latex inline when needed. The pylama language is built on top of the Python interpreter and, as opposed to `LaTeX` the language is inspired by Python's indentation based grouping of code and text. To execute code within your pylama document, you start the line with a `>` , similar to command prompts like bash. Use one or multiple lines depending on the complexity of the task, all python code and libraries can be imported and used.

To start out your document, you typically type something like

```
>from pylama.common import *
>documentclass("proc")
```

. With this done, you can start building your document. Variables in the document are stored in a global context, so if you want to provide the environment with the basics and start out your document, you can type:

```
>title = "pylama: Make typesetting great again"
>author = "Nils Bore"
>begin_document()
>make_title(title, author)
```

You declare subsections with respect to the parent section, avoiding having to type "sub" too many times and helping you keep track of the hierarchy.

```
>maths = intro.subsection("Maths")
```

Too get more familiar with the language, let's look at how we can use pylama for maths.

1.1 Maths

Equations, like

$$A * A = A^2 \tag{1}$$

can be declared with the following syntax:

```
>equation1 = equation()
    A*A = A^2
```

Note the `equation1` assignment preceding the equation declaration. This gives us a handle that we can use to reference the equation later in the document using the syntax `>ref(equation1)` with the result: 1 . This has the great advantage that we can actually check if the variable is present in the current context and throw and error if that is not the case. In latex this would quietly fail with a question mark in the final document.

The other important thing to note is that there is an indent before the actual equation text. This declares that the text is a child of the `equation()` declaration. The equation environment takes care of processing the formula into text that can be inserted into the final `LaTeX` document. The indentation based syntax is a cornerstone of the pylama language and allows these so called contexts to be nested arbitrarily deep. The next section delves deeper into contexts.

1.2 Lists and Blocks

Item lists are a good example for illustrating nested text blocks. At the top level sits an `>itemize()` or `>enumerate()` declaration. In the indented level, as children of the `>itemize()` , sit several `>item()` which in turn have their own children, typically text, that declare the content of the respective items. A complete list can be declared as

```
>itemize()
    >item()
        Item 1!
    >item()
        Item 2!
    >item()
        Item 3!
```

which results in the following list:

- Item 1!
- Item 2!

- Item 3!

Note that the children block themselves can contain one or multiple text or code blocks and that they can be layered in several steps. This is illustrated in the next section, where we mix text and figures inside a table environment.

1.3 Tables and Variables

Usually, variables in pylama are declared in a global scope. For example, if you declare a variable `>a=3` anywhere in your document, even inside a block, it will be accessible anywhere after that declaration. This allows you to reference a nested block even when you are on a higher level in the block hierarchy, as we will see in the subfigure example. However, there is one exception to this in the form of local variables. Local variables are declared by providing keyword arguments to the block functions. In the example below, `rows, cols` and `caption` are local variables and can only be referenced within the child block. Notice how we can use them in the table definition to display the size of the table.

```
>table(rows=2, cols=3, caption="A table.")
    >centering()
    >setcell(0, 0)
        Figure:
    >setcell(1, 0)
        >includegraphics(image="birds.png",
            scale=0.3)
    >setcell(0, 1)
        Rows:
    >setcell(1, 1)
        >text(rows)
    >setcell(0, 2)
        Columns:
    >setcell(1, 2)
        >text(cols)
```

Listing 1: The tables are indexed by row col in a numpy matrix

The result can be viewed in Table 1 . Note that indexing the blocks like this might be a slightly clumsy way of defining the contents of your table. Instead, we might use python lists to define the table below. Looping through and setting the blocks appropriately allows us to express this more compactly.

```
>table_contents = [["(Observations)", "Finch", "
    Seagull", "Crow", "Pigeon", "Hawk"],\
>                    ["Sea", 2, 10, 1, 3, 4],\
>                    ["Park", 4, 2, 6, 8, 0],\
>                    ["Cliff", 2, 4, 1, 0, 2]]
>table(rows=4, cols=6, caption="Bird places.")
    >centering()
    >for r in range(0, rows):
    >    for c in range(0, cols):
    >        setcell(r=r, c=c)
        Obs:
        >text(table_contents[r][c])
```

Listing 2: Python and pylama interoperation.

The resulting table is shown in Table 2 .

2 Figures and References

Figures are defined in a way to let the most common types be very easy to define. Simply adding an image with a caption and a reference variable is as easy as:

```
>f1 = figure(caption="A lovely blue bird.",
    image="birds.png", placement="htpb")
    >center()
```

Listing 3: Declaring a figure is a oneliner.

Figure 1: A lovely blue bird.

All common code blocks that can be referenced return a reference variable, in this case `f1` . This can be easily referenced through `>ref(f1)` , resulting in: 1 . If you wish, you can also reference using string, but in that case you need to provide the `label` argument.

2.1 Subfigures

A common case in scientific writing is also the use of tables containing figures, often with captions for each individual figure. This is enabled with the `figuretable()` function block, which works in a way very similar to ordinary tables, with the difference that all cells should contain only `subfigure()` function blocks. This is showcased in the example below:

```
>figuretable(rows=1, cols=4, caption="A table
    with four lovely blue birds.")
    >center()
    >setcell(0, 0)
        >subfigure(caption="Jason.", image="
            birds.png", scale=0.4)
            >center()
    >setcell(0, 1)
        >subfigure(caption="Kenneth.", image="
            birds.png", scale=0.4)
            >center()
    >setcell(0, 2)
        >subfigure(caption="Olav.", image="birds
            .png", scale=0.4)
            >center()
    >setcell(0, 3)
        >sf1 = subfigure(caption="Makunde.",
            image="birds.png", scale=0.4)
            >center()
```

Listing 4: Figuretables are similar to tables with subfigures in the cells with the same index function.

Note the `sf1` assignment. This can be reference outside the figure table with the result: 2d .

Figure:	Rows:	Columns:
	2	3

Table 1: The table showcases how to assign table cells by indices and how you can include contexts within each others. The image showcased in all figures is created by BanzaiTokyo, `http://www.iconarchive.com/show/ugly-birds-icons-by-banzaitokyo/angor-icon.html` . See table example Listing 1 for code.

Obs: (Observations)	Obs: Finch	Obs: Seagull	Obs: Crow	Obs: Pigeon	Obs: Hawk
Obs: Sea	Obs: 2	Obs: 10	Obs: 1	Obs: 3	Obs: 4
Obs: Park	Obs: 4	Obs: 2	Obs: 6	Obs: 8	Obs: 0
Obs: Cliff	Obs: 2	Obs: 4	Obs: 1	Obs: 0	Obs: 2

Table 2: This example showcases how you can define a table in python and directly import it into pylama. For the code generating this table, look at Listing 2 .

3 Details

3.1 Lazy blocks

Pylama is designed to make the common things easy while making the hard things possible. In that spirit, e.g. captions of figures and tables are provided as strings to the block function of the figure or table. However, sometimes you would like the captions themselves to be defined using the pylama functions and blocks. This is where the `lazy()` block function comes in. It enables you to define a block like:

```
>lazy_1 = lazy()
    This is text that will only get added later.
```

This block will not get added in the text in that place of its declaration. Instead it will get added when you call `lazy_1.add()` . However, you could instead call `ss = lazy_1.string()` , resulting in a string `ss` with the block evaluated as text, which you can then provide as the caption argument to a figure, table, or other block function. Several of the more complex captions in the examples of this document were generated in this way.

3.2 Extending through Contexts

Inside each block you have access to a variety of introspection details. Below are some examples of what you can access for example inside an equation block.

```
>equation()
    A + A = 2A
    ># prints "4", the current indent level:
    >print Context.context.indent
    ># prints "A + A = 2A", the first child:
    >print Context.parent.children[0].text
    ># prints "0" as this block has no children:
    >print len(Context.children)
```

The `Context` class is the main interface for all of pylama. Through it, you can access details of the environment like in the example above, but it is also the mechanism through which the document is constructed. Importantly, no block gets added to the final document unless its parent block explicitly adds it. The `Context.context` object has a method, `Context.context.add()` which adds all of its child blocks into the document, which can then in turn add their children if they wish to. We will make our own version of the equation block function above to demonstrate this. Since all of our code should compile down to `LaTex` , we can simply write:

```
>def equation(label):
>    latex("\\begin{equation}")
>    Context.context.add()
>    latex("\label{%s}" % label)
>    latex("\end{equation}")
>    return label
```

Now this can be put in a python file and exported as any other library into any pylama document. The function simply puts the proper declarations around the equation block and then adds the child blocks through the `Context.context.add()` function.

4 Useful Things

4.1 Combining files

Splitting your text up between different `.pymd` or `.tex` files is generally considered good practice. A `.pymd` file can be included via the `input()` function while `.tex` files are imported via the `latexinput()` function. `.pymd` files get expanded within the larger environment, giving it access to declared variables there in order for it to e.g. reference figures or

(a) Jason. (b) Kenneth. (c) Olav. (d) Makunde.

Figure 2: The figure table example generated from code example Listings 4 .

other sections. So, for example, if you have a file `test/input.pymd` you can import it with:

```
>input("test/input.pymd")
```

If the contents of the file are

```
This was written in another file!
```

this will result in the following line: This was written in another file!

4.2 Citing

The following code cites a reference and adds a bibliography file to the document:

```
>cite("knuth1979tex")
>add_bibliography("main", "ieeetr")
```

This results in the reference [1] and a bibliography at the end of the document.

References

[1] D. E. Knuth, *TEX and METAFONT: New directions in typesetting.* American Mathematical Society, 1979.