Comments on using the nothon notebook

Zoltán Vörös

Sat Apr 13 2013 09:13:22 GMT+0200 (CEST)

1 Beginning with nothon

The most recent version of the source code resides at https://github.com/v923z/nothon

After starting nothon.py, the notebook can be accessed at address

http://127.0.0.1:8080/?name=somenotebook.note

If the notebook exists (this should be a JSON file), then its content will be rendered in the browser. If it doesn't exist, then a new notebook with that name will be created. When saving this notebook, its content will be written to disc under the name somenotebook.note.

Once in the browser, a new div can be added by clicking on the appropriate label under the + sign on the right hand side. The active div's parent (always indicated by a thick right-hand-side border) can be removed by clicking on remove. This operation moves the divide to the trash, which is part of the document, but is hidden. (The trash can be made visible by clicking on the trashbin icon on the lower right corner.) By clicking on recover, either the active block of the trashbin, or if that does not exist, then the last block in the trash is restored to the visible part of the document, and is inserted as the last divide. Blocks of the document can be moved up or down by clicking on the up or down arrow on the right hand side.

2 Usage in general terms

Headers in text, code, and head blocks are evaluated by pressing 'Enter'. This moves the cursor to the body of a text header, or send a request to the server to execute and return the output of the corresponding handler function.

Units can be "executed" by pressing 'Cntr+Enter'. This will render mathematical formulae in a text block, or execute a plot. If 'Shift+Enter' is pressed, a new block of the same type is also inserted into the notebook.

3 Customisation

The visual appearance of a notebook can be influenced by changing the appropriate parameters in the cascaded style sheet. These modifications do not change the behaviour of the server or the client.

Several aspects of the behaviour of the server can be customised in a resource file. (In future versions, this will be extended to the client, too.) By default, only png files are created, when the user plots in the notebook. By changing the value of the variable self.plot_pdf_output to True, the plotting backend will generate pdf output, too. This output can later be included in latex files.

The ordering of the directory tree can be customised. By default, the server sends the tree ordered linux style, i.e., in each folder, the files are listed first, and then the folders. By setting the variable self.dirlisting_style to "windows", folders will be shown first, and then files.

When working with code (see below), it might be useful to include only a function or a segment of a code file. This can be done by defining a start and end tag in the source code, and supplying that to the code handler.

In order for the code highlighter to know what amounts to the beginning/ending of a code segment, the tag is prepended and appended with a beginning/closing string. This can be defined in the variable self.code_delimiter. E.g., if self.code_delimiter = ('*-', '-*'), then the code should be enclosed between the tags

- some_tag .... code .... some_tag -

The user can define arbitrary tag combination.

4 Code segments 1.

Arbitrary code residing on the hard disc can conveniently be displayed. All one has to do is give the file name in a code cell.

helper.py
Created: Fri Mar 22 21:05:47 2013, modified: Fri Mar 22 21:05:47 2013

```
import os

def retreive_header(args):
        head = args.split('<br>')
        sp = head[0].split(' ')
        if not os.path.exists(sp[0]):
                return "File doesn't exist"
        if len(sp) == 1: n = 10
        # TODO: elif sp[1] == '#':
        else: n = int(sp[1])
        fin = open(sp[0], 'r')
        # *- function_something
        if n > 0:
                lines = []
                it = 0
                for line in fin:
                        lines.append(line.rstrip('\n\r'))
                        it += 1
                        if it >= n: break
        # function_something -*
        if n < 0:
                lines = fin.readlines()
                lines = lines[-10:]
        fin.close()
        return '<br>'.join([x.rstrip('\n\r') for x in lines])
```

5 Working with code 2.

Code lines can be numbered by adding -lineno on the command line. In addition, if a start and end tag are defined in the source file, the code highlighter can be made to display only the segment between the two strings. If -include is specified on the command line, the tags are also included in the highlit code. This can be useful when one wants to emphasise that we are dealing with a code segment only.

helper.py -lineno -tag function_something -include
Created: Fri Mar 22 21:05:47 2013, modified: Fri Mar 22 21:05:47 2013

```
        # *- function_something
        if n > 0:
                lines = []
                it = 0
                for line in fin:
                        lines.append(line.rstrip('\n\r'))
                        it += 1
                        if it >= n: break
        # function_something -*
```

6 Working with headers 1.

This functionality might be handy, if one wants to list the content of some file. We only have to enter the file name, and press enter. Additionally, an argument can be supplied, in which case, only the first or last n lines will be printed.

test.dat
Created: Sat Dec 15 23:54:22 2012, modified: Sat Dec 15 23:53:33 2012

```
# comment 1
# comment 2
12 3
23 22
2132 123
55 99
```

7 Working with headers 2.

If we supply an argument, we can print the first or last n lines as follows

test.dat 2
Created: Sat Dec 15 23:54:22 2012, modified: Sat Dec 15 23:53:33 2012

```
# comment 1
# comment 2
```

test.dat -2
Created: Sat Dec 15 23:54:22 2012, modified: Sat Dec 15 23:53:33 2012

```
2132 123
55 99
```

8 Adding plots

Plots can easily be included in a notebook. A plot cell has three usable subcells. The first one is the caption/title of the plot. This will be included in the table of contents, and this will also be used as the caption, when one converts the notebook to pdf via LaTeX.

The second cell is the matplotlib code that generates the plot. At the time of writing this, this code is going to be included in the pdf output, while in the notebook, it can be made hidden by clicking on the gray area next to it.

The third subcell is the plot itself. In case the plot cannot be generated (e.g., due to a syntax error in the code), the traceback will be returned instead.

Note that gnuplot also can be used as the plotting backend by adding #gnuplot or # gnuplot on the first line of the code.

```
plot(sin(x), 'ro')
xlabel('Time [s]')
ylabel('Displacement [a.u.]')
```

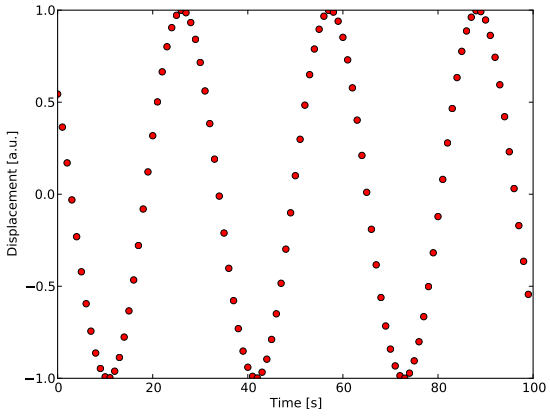

Figure 1: Time evolution of the displacement of a harmonic oscillator

9

10

11 This is a text box with some LaTeX code

In addition to containing raw text, courtesy of MathJaX, a text box can also deal with LaTeX code. This can be inserted by pressing Cntr-Alt-M (display style), or Alt-M (inline). If we wanted to solve the quadratic equation

$$a \cdot x^2 + b \cdot x + c = 0$$

then you would have to look up the solution formula,

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Some markup can be added to text. **Cntr-I** is going to make the text *italic*, **Cntr-B** turns it **boldface**, **Cntr-O** adds highlighting, while **Cntr-U** will underline the text.

Notebooks can be converted to pdf via LaTeX by calling python/latex.py with the notebook file as the single argument. Customisation of the latex file should be done through the templates in templates/. If pdf output is needed, self.plot_pdf_output has to be set to True in the resource file.

$$a = 123 \tag{1}$$
$$b = 122 \tag{2}$$