attaching-code-blocks-to-a-pdf

John Kitchin

September 30, 2013

Contents

1 Attaching code blocks to a pdf file during export

This post is a further exploration of using the export filters to modify construction of content exported from org-mode. In this post we look at some code that will save all of the code-blocks in an org-buffer to systematically named files, and then attach the files to an exported pdf file. We will use the attachfile LaTeX package to attach the scripts. We will build off of this post for the filters.

First, let us put in a gratuitous code block. In the rendered pdf, this script will be embedded in the pdf. I am not quite ready to build a filter that supports multiple backends, so in this post we just modify the latex export.

```
1 name = 'John'
2 print 'Hello {0}'.format(name)
```

 Double click me to open

```
Hello John
```

We are only going to attach the python code blocks in this example, and ignore all the other blocks. We will basically use the same kind strategy we have used before. We will initially parse the buffer to get a list of all the code blocks. Then we create a filter for the src-blocks that keeps a counter of src-blocks, and depending on the type of the nth src-block, we will save the file, and modify the text for that block. Here is our code for the list of code blocks.

```
(setq src-block-list
      (org-element-map (org-element-parse-buffer) 'src-block
        (lambda (src-block) src-block)))
```

Now we create the filter.

```
(defun ox-mrkup-filter-src-block (text back-end info)
  (catch 'return text)
  (let ((src-block (nth counter src-block-list)))
    (if (string= (org-element-property :language src-block) "python")
        (progn
          (setq scriptname (format "py-%d.py" counter))
          ;; save code block
          (with-temp-buffer
            (insert (org-element-property :value src-block))
            (write-region (point-min) (point-max) scriptname ))

          (setq output (format "%s\n\\attachfile{%s} Double click me to open" text scriptname)))
      ;; else
      (setq output text)))
  ;; increment counter no matter what so next block is processed
  (setq counter (+ counter 1))
  ;; return output
  output)
```

Finally, we export the document to LaTeX, and run pdflatex on it to generate the pdf.

```
(let ((counter 0)
      ;; these packages are loaded in the latex file
      (org-latex-default-packages-alist
       '(("utf8" "inputenc" nil)
         ("T1" "fontenc" nil)
         ("" "fixltx2e" nil)
         ("" "lmodern" nil)
         ("" "minted" nil) ;; for code syntax highlighting
         ;; customize how pdf links look
         ("linktocpage,
           pdfstartview=FitH,
           colorlinks,
           linkcolor=blue,
           anchorcolor=blue,
           citecolor=blue,
           filecolor=blue,
           menucolor=blue,
           urlcolor=blue" "hyperref" nil)))
      (org-export-filter-src-block-functions '(ox-mrkup-filter-src-block))
      (async nil)
      (subtreep nil)
      (visible-only nil)
      (body-only nil)
```

```
24          (ext-plist '()))
25    (org-latex-export-to-pdf async subtreep visible-only body-only ext-plist))
```