Nbconvert Refactor

Final 1.0

Jonathan Frederic

June 20, 2013

Part I

Introduction

IPython is an interactive *Python* computing environment[1]. It provides an enhanced interactive Python shell. The IPython Notebook is a browser based interface distributed with IPython. It enables the creation of richly formatted notebooks that contain embedded IPython code.

Notebooks are a collection of cells. Each cell is assigned a type, either by the user or notebook backend. The type of the cell determines how its contents will be handled. Code cells can be executed one at a time or in batch. If a code cell produces output, the notebook backend will automatically add output cell(s) upon execution of the code cell. Output cells are always inserted immediately after their parent code cell.

Text and heading cell types are included. If additional formatting is needed, Markdown, LaTeX, and HTML can be used. When saved, the notebook is written as a *JSON* text file.[2] Cells containing binary data (i.e. output cells with figures) are *base-64* encoded[3]. The IPython API contains functions that allow one to read and write from the notebook file type.

To export a notebook to something other than JSON two options exist. The first is to "print" the notebook to a PDF using the web browser[4]. The second option is to use **nbconvert**. With nbconvert, notebooks can be exported to various formats including, but not limited to, LaTeX, reveal.js, RST, and HTML. This is important for users that want to be able to share their work outside of IPython. nbconvert can be customized by the user to export to formats that are not supported by default. This senior project is an addition of a Sphinx Latex output format and a **refactor** of the nbconvert source code. Where *"**Refactoring** is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior[5]."*

Part II

Motivation

Nbconvert started out as experiment created by Fernando Perez[6]. Overtime contributions were made by various authors to enable the export of additional formats. Since nbconvert started as an experiment, it lacked a solid foundation. As the codebase continued to grow, it became apparent that a nbconvert needed a major re-engineering. As contributors pushed the existing nbconvert architecture to its limits, nbconvert's core classes needed to be extended. As the project became increasingly popular, the number of requests to *merge* changes into the *master branch* became unmanageable[7]. The IPython core development team agreed to implement *Jinja* as a template engine in attempt to mitigate the number of merge requests[8].

The original nbconvert idea was to define a base exporter classes and then subclasses corresponding to each export format. The new template powered nbconvert was implemented using the existing nbconvert structure. The template engine was implemented as a subclass of the base exporter. The result was a confusing codebase that was both a mix of the original nbconvert and a template powered nbconvert. This senior project fixed this by separating the original nbconvert code from the templating nbconvert code. In addition, the new code template powered exporter was completely refactored.

The LaTeX template included with nbconvert was capable of producing simple LaTeX documents. Many users were already interested in the sophisticated LaTeX output that Sphinx could produce. At the time, Sphinx documents could be produced by exporting notebooks to RST and importing that into Sphinx. This senior project added a Sphinx LaTeX template, which builds off of the existing Sphinx LaTeX output. The new Sphinx LaTeX template allows Sphinx LaTeX to be exported directly from nbconvert without exporting to RST first.

Part III

Details

1 Pre-refactor

Nbconvert's parent level directory structure prior to the refactor is seen below

In [1]:
```
%%bash
ls ../old_nbconvert/ -1 | grep -v "[.][a-z]"
```

```
converters
css
js
profile
reveal
rst2ipynblib
templates
tests
```

Where

- **converters** exporters, exporter base, Jinja filters, and notebook transformers. The last two are specific to the new template based exporter.
- **css** static style sheet for HTML exporter and style sheet for reveal.js exporter
- **js** mathjax javascript
- **profile** configuration files for template engine based exporter. Each configuration file corresponds to an output template
- **reveal** empty
- **rst2ipynblib** empty
- **templates** Jinja templates for new template based exporter
- **tests** Nose tests for testing the orignal nbconvert exporters

There were many unsorted files in the top level directory, some of which were depracated, as seen below

In [2]:
```
%%bash
ls ../old_nbconvert/ -1 | grep ".py"
```

```
custom_converter.py
dollarmath.py
__init__.py
nbconvert2.py
nbconvert.py
nbstripout.py
notebook_sphinxext.py
rst2ipynblib
rst2ipynb.py
```

2 Refactor

Nbconvert should serve two main functions:

1. Commandline notebook conversion utility.
2. Rich API for exporting and importing notebooks.

The old nbconvert code was moved into "*/nbconvert1*". The new nbconvert code was moved into "*/nbconvert*". The new folder structure is seen below.

In [3]:
```
%%bash
ls -1 | grep -v "[.][a-z]"
```

```
nbconvert
nbconvert1
```

The directory structure of the refractored nbconvert was designed to expose as much of the inner workings of the template exporter as possible (as seen below.)

In [4]:
```
%%bash
ls ./nbconvert/ -1 | grep -v "[.][a-z]"
```

```
exporters
filters
templates
transformers
utils
```

Where

- **exporters** base exporter class and light subclasses that define default options for each export format
- **filters** Jinja filters that are accessible within the templates
- **templates** templates for each export format
- **transformers** notebook transformers
- **utils** collection of utility functions

The nbconvert export process is a multistep process:

1. Load notebook file using IPython API
2. Preprocess the notebook using **Transformer(s)**. A Transformer is a class that acts on the notebook as a whole or on a cell-by-cell basis prior to export.
3. The **filters** are passed into the Jinja templating engine. A **Filter** (Jinja specific) is a function that takes one or more arguments and returns a string. The filters are passed into Jinja so they are accessible to the templates in the next conversion step.
4. Notebook is converted using Jinja
5. *(optional)* Conversion results and exported figures are written to the user's hard disk.

The first implementation of the template based nbconvert used *profiles* to configure a single Jinja exporter class to export to different formats. This is not how the IPython config system was originally designed to be used. It was decided to replace the the *profile* design with lightweight subclasses.

The success of the refactor cannot be measured; however, code metrics can give an idea of how much the code was changed. Code lines can be broken up into three categories, blank, comments, and SLOC (source lines of code). By counting the number of blank lines and comment lines we know the remaining lines are SLOC.

$$SLOC = Total - (blanks + comments)$$

In the following python block the blank and comment lines of the pre-refactored nbconvert, refactored nbconvert, and *achived* nbconvert files are counted. The *archived* nbconvert files (*/nbconvert1/*) are the original nbconvert class/sub-class design.

In [5]:
```python
import os

def directory_traverser(path, endswith=".py"):
    for root, dirs, files in os.walk(path):
        for name in files:
            if name.endswith(endswith):
                yield os.path.join(root, name)

def python_file_metrics(filename):
    file = open(filename, "r")

    total_count = 0
    comment_count = 0
    blank_count = 0

    multiline_string_delim = "\"" * 3
    in_docstring = False

    for line in file:
        total_count+=1
        line = line.strip()
        if in_docstring:
            comment_count += 1
            if multiline_string_delim in line:
                in_docstring = False
        else:
            if line.startswith(multiline_string_delim) and not (len(line)
                comment_count += 1

                #only start doc-string block if terminator isn't in same
                if not multiline_string_delim in line[3:]:
                    in_docstring = True

            elif line.startswith("#"):
                comment_count += 1
            elif line=="":
                blank_count+=1
    file.close()
    return (total_count, comment_count, blank_count)

def python_project_metrics(filenames):
    total_count = 0
    comment_count = 0
    blank_count = 0

    for filename in filenames:
        (total,comments,blank) = python_file_metrics(filename)
        total_count += total
        comment_count += comments
        blank_count += blank

    comments_and_blanks = blank_count + comment_count
    print("""
    SLOC : {0}
    Comments : {1}
    Blanks : {2}
    Total: {3}

    Ratio of Comments per SLOC: {4:0.3}
    """.format(total_count - comments_and_blanks, comment_count, blank_cou

print("1. Pre-refactored")
python_project_metrics(directory_traverser("../old_nbconvert/"))
print("\n2. Refactored")
python_project_metrics(directory_traverser("./nbconvert/"))
print("\n3. Archived")
python_project_metrics(directory_traverser("./nbconvert1/"))
```

```
1. Pre-refactored

    SLOC : 2741
    Comments : 2787
    Blanks : 1199
    Total: 6727

    Ratio of Comments per SLOC: 1.02


2. Refactored

    SLOC : 896
    Comments : 1304
    Blanks : 496
    Total: 2696

    Ratio of Comments per SLOC: 1.46


3. Archived

    SLOC : 2143
    Comments : 2324
    Blanks : 962
    Total: 5429
```

```
            Ratio of Comments per SLOC: 1.08
```

The code metric above does not recognize line continuations. The larger comment to code ratio for both the archived and refactored content means the pre-refactored template based exporter actually had a comment to code ratio less than 1.02. One an a half comments for every line of code may sound daunting, but a large portion of the comments are due to the IPython coding stardard doc string format.

3 Sphinx LaTeX

The Sphinx LaTeX template was designed provide the best possible output with minimal configuration. Instead of hardcoding the Sphinx material into the template, the template references the Sphinx installation on the user's machine. Because of this, the Sphinx LaTeX template is only 356 SLOC[9]. If the user has Sphinx installed on his or her machine, a PDF can be created from the nbconvert output using PdfLatex.

The Sphinx Latex template provides two output document styles

1. **HowTo** For short documents
2. **Manual** For longer documents, notebook Heading 1s are treated as *parts* and Heading 2s as chapters. Each part starts on a new page.

It also allows the user to choose how IPython code cells are rendered

1. **Simple** A thin horizontal break is used to separate code from text. At the top left of the horizontal break, in small font, the input/output prompt is visible.
2. **Notebook** Code cells are rendered like they are in the notebook. The are rendered in light gray tables with slightly rounded corners.

Part IV
Conclusion

In conclusion, the refactor improved the code's readability by increasing the comment to code ratio and separating the files that were no longer in use from the code base. The refactored nbconvert and the orignal pull request can be found on GitHub[10]. The Sphinx LaTeX template added the ability to export beautiful LaTeX documents directly from nbconvert. It is in the master nbconvert repository and can be used with no modification to nbconvert. The Sphinx LaTeX template was used to export this notebook (senior project paper.) The source code for the template can be viewed on GitHub[9]. This paper is also available on GitHub as a gist[11].

Part V
References

1. What is Python? Executive Summary, http://www.python.org/doc/essays/blurb.html
2. Introducing JSON, http://www.json.org/
3. Base64 - Online sample of a base64 poperty, http://www.motobit.com/util/base64-decoder-encoder.asp

4. What tools are available to export an ipython notebook to a PDF file?, http://stackoverflow.com/questions/14132213/what-tools-are-available-to-export-an-ipython-notebook-to-a-pdf-file
5. Refactoring Home Page, http://www.refactoring.com/
6. Fernando Perez, IPython PI, https://github.com/fperez
7. Wikipedia, Revision control, http://en.wikipedia.org/wiki/Revision_control
8. Jinja homepage, http://jinja.pocoo.org/
9. sphinx template source with SLOC count, on GitHub, https://github.com/ipython/nbconvert/blob/master/nbconvert/templates/latex/sp
10. nbconvert refactor, original pull request on GitHub, https://github.com/ipython/nbconvert/pull/137
11. gist for this paper, on GitHub, https://gist.github.com/jdfreder/5816002. Can also be view in nbviewer at http://nbviewer.ipython.org/5816002