Practical Business Python

Taking care of business, one python script at a time

Mon 16 February 2015

Creating PDF Reports with Pandas, Jinja and WeasyPrint

Posted by Chris Moffitt in articles

Introduction

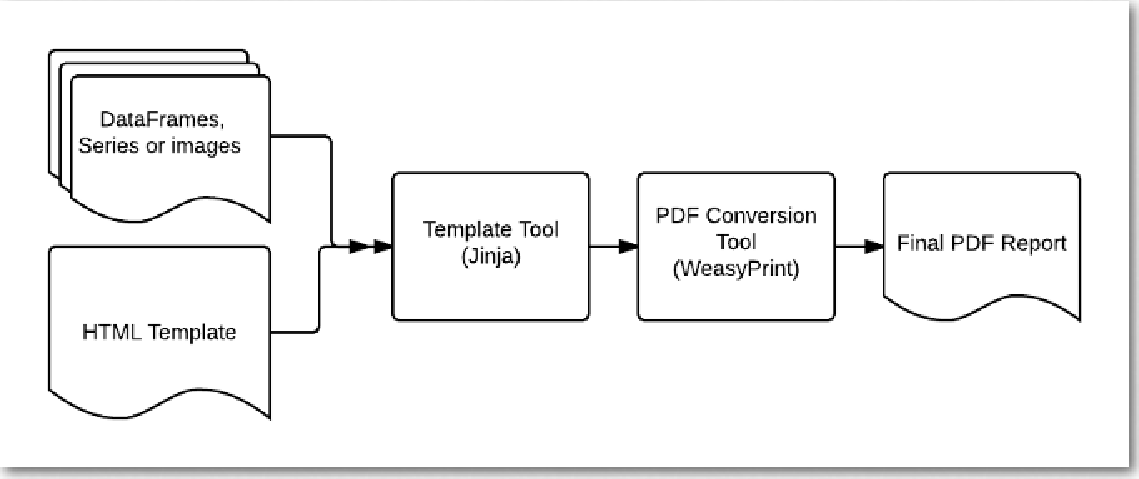
Pandas is excellent at manipulating large amounts of data and summarizing it in multiple text and visual representations. Without much effort, pandas supports output to CSV, Excel, HTML, json and more. Where things get more difficult is if you want to combine multiple pieces of data into one document. For example, if you want to put two DataFrames on one Excel sheet, you need to use the Excel libraries to manually construct your output. It is certainly possible but not simple. This article will describe one method to combine multiple pieces of information into an HTML template and then converting it to a standalone PDF document using Jinja templates and WeasyPrint.

Before going too far through this article, I would recommend that you review the previous articles on Pandas Pivot Tables and the follow-on article on generating Excel reports from these tables. They explain the data set I am using and how to work with pivot tables.

The Process

As shown in the reporting article, it is very convenient to use Pandas to output data into multiple sheets in an Excel file or create multiple Excel files from pandas DataFrames. However, if you would like to combine multiple pieces of information into a single file, there are not many simple ways to do it straight from Pandas. Fortunately, the python environment has many options to help us out.

In this article, I'm going to use the following process flow to create a multi-page PDF document.


DataFrames, Series or images
HTML Template
Template Tool (Jinja)
PDF Conversion Tool (WeasyPrint)
Final PDF Report

The nice thing about this approach is that you can substitute your own tools into this workflow. Don't like Jinja? Plug in mako or your templating tool of choice. If you want to use another type of markup outside of HTML, go for it.

The Tools

First, I decided to use HTML as the templating language because it is probably the simplest way to generate structured data and allow for relatively rich formatting. I also think everyone knows (or can figure out) enough HTML to generate a simple report. Also, I don't have the desire to learn a whole new templating language. However, if you choose to use other markup languages, the flow should work the same.

I chose Jinja because I have experience with Django and it closely mirrors Django's syntax. There are certainly other options out there so feel free to experiment with your options. I think for this approach there is nothing very complicated about our templates so any tool should work fine.

Finally, the most difficult part of this tool chain is figuring out how to render theHTML into PDF. I don't feel like there is an optimal solution yet but I chose WeasyPrint because it is still being actively maintained and I found that I could get it working relatively easily. There are quite a few dependencies for it to work so I'll be curious if people have any real challenges getting it to work on Windows. As an alternative, I have used xhtml2pdf in the past and it works well too. Unfortunately the documentation is a little lacking at this time but it has been around for a while and does generate PDF's effectively from HTML.

The Data

As discussed above, we'll use the same data from my previous articles. In order to keep this all a self-contained article, here is how I import the data and generate a pivot table as well as some summary statistics of the average quantity and price of the CPU and Software sales.

Import modules, and read in the sales funnel information.

```
from __future__ import print_function
import pandas as pd
import numpy as np
df = pd.read_excel("sales-funnel.xlsx")
df.head()
```

	Account	Name	Rep	Manager	Product	Quantity	Price	Status
0	714466	Trantow-Barrows	Craig Booker	Debra Henley	CPU	1	30000	presente
1	714466	Trantow-Barrows	Craig Booker	Debra Henley	Software	1	10000	presente
2	714466	Trantow-Barrows	Craig Booker	Debra Henley	Maintenance	2	5000	pending
3	737550	Fritsch, Russel and Anderson	Craig Booker	Debra Henley	CPU	1	35000	declined
4	146832	Kiehn-Spinka	Daniel Hilton	Debra Henley	CPU	2	65000	won

Pivot the data to summarize.

```
sales_report = pd.pivot_table(df, index=["Manager", "Rep", "Product"], values=["Price", "Quan
                              aggfunc=[np.sum, np.mean], fill_value=0)
sales_report.head()
```

			sum		mean	
			Price	Quantity	Price	Quantity
Manager	Rep	Product				
Debra Henley	Craig Booker	CPU	65000	2	32500	1
		Maintenance	5000	2	5000	2
		Software	10000	1	10000	1
	Daniel Hilton	CPU	105000	4	52500	2
		Software	10000	1	10000	1

Generate some overall descriptive statistics about the entire data set. In this case, we want to show the average quantity and price for CPU and Software sales.

```
print(df[df["Product"]=="CPU"]["Quantity"].mean())
print(df[df["Product"]=="CPU"]["Price"].mean())
print(df[df["Product"]=="Software"]["Quantity"].mean())
print(df[df["Product"]=="Software"]["Price"].mean())
```

```
1.88888888889
51666.6666667
1.0
10000.0
```

Ideally what we would like to do now is to split our data up by manager and include some of the summary statistics on a page to help understand how the individual results compare to the national averages.

DataFrame Options

I have one quick aside before we talk templates. For some quick and dirty needs, sometimes all you need to do is copy and paste the data. Fortunately a DataFrame has a `to_clipboard()` function that will copy the whole DataFrame to the clipboard which you can then easily paste into Excel. I have found this to be a really helpful option in certain situations.

The other option we will use later in the template is the `to_html()` which will generate a string containing a fully composed HTML table with minimal styling applied.

Templating

Jinja templating is very powerful and supports a lot of advanced features such as sandboxed execution and auto-escaping that are not necessary for this application. These capabilities however will serve you well as your reports grow more complex or you choose to use Jinja for your web apps.

The other nice feature of Jinja is that it includes multiple builtin filters which will allow us to format some of our data in a way that is difficult to do within Pandas.

In order to use Jinja in our application, we need to do 3 things:

- Create a template
- Add variables into the templates context
- Render the template into HTML

Here is a very simple template, let's call it **myreport.html** :

```
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>{{ title }}</title>
</head>
<body>
  <h2>Sales Funnel Report - National</h2>
   {{ national_pivot_table }}
</body>
</html>
```

The two keys portions of this code are the `{{ title }}` and `{{ national_pivot_table }}` . They are essentially placeholders for variables that we will provide when we render the document.

To populate those variable, we need to create a Jinja environment and get our template:

```
from jinja2 import Environment, FileSystemLoader
env = Environment(loader=FileSystemLoader('.'))
template = env.get_template("myreport.html")
```

In the example above, I am assuming that the template is in the current directory but you could put the full path to a template location.

The other key component is the creation of `env` . This variable is how we pass content to our template. We create a dictionary called `template_var` that contains all the variable we want to pass to the template.

Note how the names of the variables match our templates.

```
template_vars = {"title" : "Sales Funnel Report - National",
                 "national_pivot_table": sales_report.to_html()}
```

The final step is to render the HTML with the variables included in the output. This will create a string that we will eventually pass to our PDF creation engine.

```
html_out = template.render(template_vars)
```

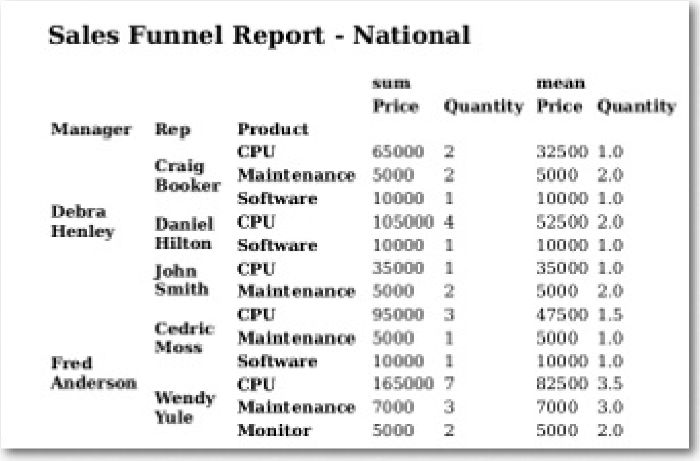
For the sake of brevity, I won't show the fullHTML but you should get the idea.

Generate PDF

The PDF creation portion is relatively simple as well. We need to do some imports and pass a string to the PDF generator.

```
from weasyprint import HTML
HTML(string=html_out).write_pdf("report.pdf")
```

This command creates a PDF report that looks something like this:



Sales Funnel Report - National

			sum		mean	
			Price	Quantity	Price	Quantity
Manager	Rep	Product				
Debra Henley	Craig Booker	CPU	65000	2	32500	1.0
		Maintenance	5000	2	5000	2.0
		Software	10000	1	10000	1.0
	Daniel Hilton	CPU	105000	4	52500	2.0
		Software	10000	1	10000	1.0
	John Smith	CPU	35000	1	35000	1.0
		Maintenance	5000	2	5000	2.0
Fred Anderson	Cedric Moss	CPU	95000	3	47500	1.5
		Maintenance	5000	1	5000	1.0
		Software	10000	1	10000	1.0
	Wendy Yule	CPU	165000	7	82500	3.5
		Maintenance	7000	3	7000	3.0
		Monitor	5000	2	5000	2.0

Ugh. It’s cool that it’s a PDF but it is ugly. The main problem is that we don’t have any styling on it. The mechanism we have to use to style is CSS.

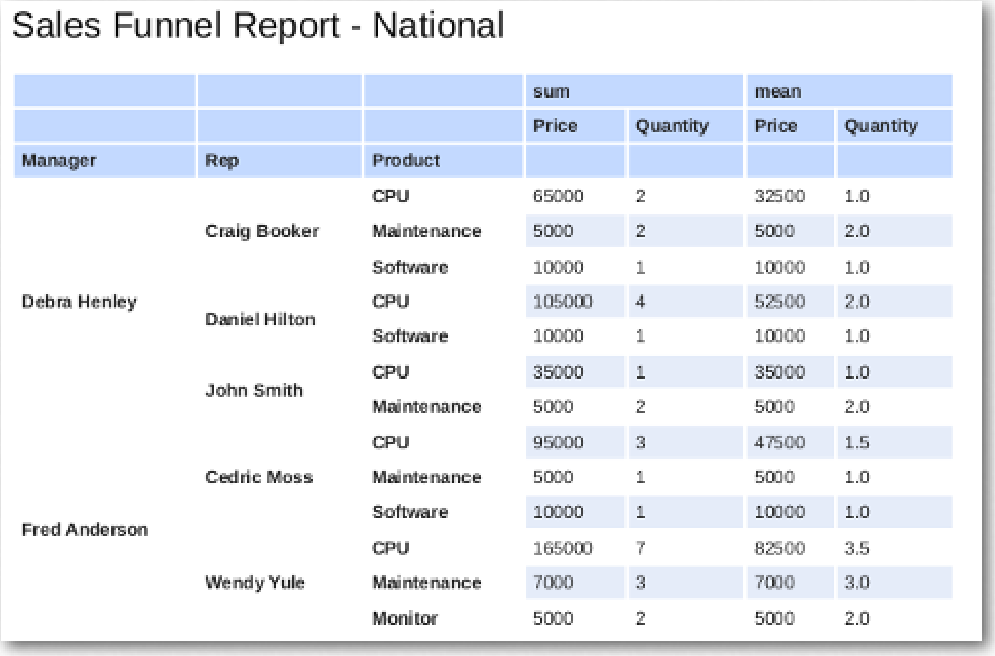
As an aside, I really don’t like CSS. Every time I start playing with it I feel like I spend more time monkeying with the presentation than I did getting the data summarized. I am open to ideas on how to make this look nicer but in the end, I decided to go the route of using a portion of blueprint CSS to have very simple styling that would work with the rendering engines.

For the rest of the article, I’ll be using blue print’stypography.css as the basis for my style.css shown below. What I like about this css is:

- It is relatively small and easy to understand
- It works will in the PDF engines without throwing errors and warnings
- It includes basic table formatting that looks pretty decent

Let’s try re-rendering it with our updated stylesheet:

```
HTML(string=html_out).write_pdf(args.outfile.name, stylesheets=["style.css"])
```



Sales Funnel Report - National

			sum		mean	
			Price	Quantity	Price	Quantity
Manager	Rep	Product				
Debra Henley	Craig Booker	CPU	65000	2	32500	1.0
		Maintenance	5000	2	5000	2.0
		Software	10000	1	10000	1.0
	Daniel Hilton	CPU	105000	4	52500	2.0
		Software	10000	1	10000	1.0
	John Smith	CPU	35000	1	35000	1.0
		Maintenance	5000	2	5000	2.0
Fred Anderson	Cedric Moss	CPU	95000	3	47500	1.5
		Maintenance	5000	1	5000	1.0
		Software	10000	1	10000	1.0
	Wendy Yule	CPU	165000	7	82500	3.5
		Maintenance	7000	3	7000	3.0
		Monitor	5000	2	5000	2.0

Just adding a simple stylesheet makes a huge difference!

There is still a lot more you can do with it but this shows how to make it at least serviceable for a start. As an aside, I think it would be pretty cool if someone that knew CSS way better than me developed an open sourced, simple CSS sheet we could use for report generation like this.

More Complex Templating

Up until now, we haven't done anything different than if we had just generated a simple Excel sheet using `to_excel()` on a DataFrame.

In order to generate a more useful report, we are going to combine the summary statistics shown above as well as break out the report to include a separate PDF page per manager.

Let's start with the updated template **(myreport.html)**:

```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>{{ title }} </title>
</head>
<body>
<div class="container">
    <h2>Sales Funnel Report - National</h2>
     {{ national_pivot_table }}
    {% include "summary.html" %}
</div>
<div class="container">
    {% for manager in Manager_Detail %}
        <p style="page-break-before: always" ></p>
        <h2>Sales Funnel Report - {{manager.0}}</h2>
        {{manager.1}}
        {% include "summary.html" %}
    {% endfor %}
</div>
</body>
</html>
```

The first thing you'll notice is that there is an `include` statement which mentions another file. The `include` allows us to bring in a snippet of HTML and use it repeteadly in different portions of the code. In this case the summary contains some simple national level stats we want to include on each report so that the managers can compare their performance to the national average.

Here is what **summary.html** looks like:

```
<h3>National Summary: CPUs</h3>
    <ul>
        <li>Average Quantity: {{CPU.0|round(1)}}</li>
        <li>Average Price: {{CPU.1|round(1)}}</li>
    </ul>
<h3>National Summary: Software</h3>
    <ul>
        <li>Average Quantity: {{Software.0|round(1)}}</li>
        <li>Average Price: {{Software.1|round(1)}}</li>
    </ul>
```

In this snippet, you'll see that there are some additional variables we have access to: `CPU` and `Software` . Each of these is a python list that includes the average quantity and price for CPU and Software sales.

You may also notice that we use a pipe `|` to `round` each value to 1 decimal place. This is one specific example of the use of Jinja's filters.

There is also a for loop that allows us to display the details for each manager in our report. Jinja's template language only includes a very small subset of code that alters the control flow. Basic for-loops are a mainstay of almost any template so they should make sense to most of you.

I want to call out one final piece of code that looks a little out of place:

```
<p style="page-break-before: always" ></p>
```

This is a simple CSS directive that I put in to make sure theCSS breaks on each page. I had to do a little digging to figure out the best way to make the pages break so I thought I would include it to help others out.

Additional Stats

Now that we have gone through the templates, here is how to create the additional context variables used in the templates.

Here is a simple summary function:

```
def get_summary_stats(df,product):
    """
    For certain products we want National Summary level information on the reports
    Return a list of the average quantity and price
    """
    results = []
    results.append(df[df["Product"]==product]["Quantity"].mean())
    results.append(df[df["Product"]==product]["Price"].mean())
    return results
```

We also need to create the manager details:

```
manager_df = []
for manager in sales_report.index.get_level_values(0).unique():
    manager_df.append([manager, sales_report.xs(manager, level=0).to_html()])
```

Finally, call the template with these variables:

```
template_vars = {"title" : "National Sales Funnel Report",
                 "CPU" : get_summary_stats(df, "CPU"),
                 "Software": get_summary_stats(df, "Software"),
                 "national_pivot_table": sales_report.to_html(),
                 "Manager_Detail": manager_df}
# Render our file and create the PDF using our css style file
html_out = template.render(template_vars)
HTML(string=html_out).write_pdf("report.pdf",stylesheets=["style.css"])
```

Here is the final PDF Report . I think it looks pretty decent for a simple report.

Ideas For Improvements

In the example above, we used the simple to_html() to generate our HTML I suspect that when you start to do more of these you will want to have finer grained control over the output of your table.

There are a couple of options:

- Pass a custom css class to_html using classes
- Use formatters to format the data
- Pass the data directly to your template and use iterrows to manually construct your table

Final Program

In order to pull it all together, here is the full program:

```
"""
Generate PDF reports from data included in several Pandas DataFrames
From pbpython.com
"""
from __future__ import print_function
import pandas as pd
import numpy as np
import argparse
from jinja2 import Environment, FileSystemLoader
from weasyprint import HTML


def create_pivot(df, infile, index_list=["Manager", "Rep", "Product"], value_list=["Price", "Quan
    """
    Create a pivot table from a raw DataFrame and return it as a DataFrame
    """
    table = pd.pivot_table(df, index=index_list, values=value_list,
                           aggfunc=[np.sum, np.mean], fill_value=0)
    return table

def get_summary_stats(df,product):
    """
    For certain products we want National Summary level information on the reports
    Return a list of the average quantity and price
    """
    results = []
    results.append(df[df["Product"]==product]["Quantity"].mean())
    results.append(df[df["Product"]==product]["Price"].mean())
    return results

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='Generate PDF report')
    parser.add_argument('infile', type=argparse.FileType('r'),
    help="report source file in Excel")
    parser.add_argument('outfile', type=argparse.FileType('w'),
    help="output file in PDF")
    args = parser.parse_args()
    # Read in the file and get our pivot table summary
    df = pd.read_excel(args.infile.name)
    sales_report = create_pivot(df, args.infile.name)
    # Get some national summary to include as well
    manager_df = []
    for manager in sales_report.index.get_level_values(0).unique():
        manager_df.append([manager, sales_report.xs(manager, level=0).to_html()])
    # Do our templating now
    # We can specify any directory for the loader but for this example, use current directory
    env = Environment(loader=FileSystemLoader('.'))
    template = env.get_template("myreport.html")
    template_vars = {"title" : "National Sales Funnel Report",
                     "CPU" : get_summary_stats(df, "CPU"),
                     "Software": get_summary_stats(df, "Software"),
                     "national_pivot_table": sales_report.to_html(),
                     "Manager_Detail": manager_df}
    # Render our file and create the PDF using our css style file
    html_out = template.render(template_vars)
    HTML(string=html_out).write_pdf(args.outfile.name,stylesheets=["style.css"])
```

You can also view the gist if you are interested amd download a zip file of **myreport.html**, **style.css** and **summary.html** if you find it helpful.

Thanks for reading all the way to the end. As always, feedback is appreciated.

← Generating Excel Reports from a Pandas Pivot Table

Generating fake data with barnum →

Tags pandas pdf jinja

Comments

Subscribe to the mailing list

email address

Subscribe

Support the site

Social

Github
Twitter
LinkedIn

Popular

- Pandas Pivot Table Explained
- Common Excel Tasks Demonstrated in Pandas
- Overview of Python Visualization Tools
- Web Scraping - It's Your Civic Duty
- Simple Graphing with IPython and Pandas

Tags

xlsxwriter ggplot sets s3 beautifulsoup plotly word process scikit altair notebooks scikit-learn stdlib csv analyze-this notebook github jinja python numpy oauth2 excel ipython google mlxtend vcs xlwings pdf pygal matplotlib pandas vega seaborn outlook barnum cases powerpoint pelican matplot plotting gui bokeh

Feeds

Atom Feed

Disclosure

We are a participant in the Amazon Services LLC Associates Program, an affiliate advertising program designed to provide a means for us to earn fees by linking to Amazon.com and affiliated sites.

Google Custom Searc

PSF SUPPORTING MEMBER