Jeremy Mulcahey's Senior Project: The IPython Notebook for Data Analysis

Contents

Chapter 1
Introduction

This notebook is for anyone that has felt a tinge of excitement by the mention of terms such as: Data Science, Big Data, Python, etc. The IPython Notebook makes it easy to add another data analysis tool to your kit. This document contains installation and set up instructions for many of the tools that enable you to initially make the most of your own IPython Notebooks. This document will also provide examples using Python packages and coding in Python. As you grow more comfortable with Python, there are many alternatives to the IPython Notebook such as the IPython console, text editors, and IDEs, but those are not my focus. Dr. Granger and his team are striving to make the IPython Notebook capable of analyzing and presenting data for all situations that will arise.

1.1 Background Requirements

Patience and an open mind.

Ideally, this guide will be useful for the full range of statistics students from "I have never programmed before" to "I know what i'm doing. I just want to know which programs I need and where to get them."

1.2 Goal

To pass along the struggles, successes, and code I learned while analyzing data in the IPython Notebook. Some of the packages and programs I will provide information on are:

- Anaconda
- Python
- IPython Notebook
- GitHub & Sharing IPython Notebooks
- Pandas
- Numpy
- Seaborn
- NBViewer
- Statsmodels
- Requests
- JSON
- ASCII
- SciPy
- Rpy2
- R in IPython Notebook
- urllib2
- and more!

1.3 Disclaimer

From the moment I started this project with Dr. Doi, I have had to learn everything I am sharing in this document. I have no prior experience with Python, any of its packakges, LaTeX, the IPython

Notebook, etc. Many of the approaches and work arounds are that of a novice. Much of the code I provide is not the only way to accomplish the task at hand and most of it might not be the best way either. If you feel a section of code can be improved, or you find packages that do the same work as some of my functions, I encourage you to use them or write your own improvements.

I hope you gain as much using this notebook as I gained writing it.

Chapter 2
Starting with Python

2.1 Why Python?

The obvious answer is the IPython Notebook. IPython Notebook is a one-stop shop for data analysis, widgets, homework, and projects. The IPython notebook can take the place of an IDE, text editor, and/or console. Working in the IPython Notebook enables you to write code, analyze data in Python and R, format using LaTeX and HTML, and produce graphs - all in the same document! The notebooks can be converted to HTML, LaTeX, PDF, and more. The notebooks can also be shared, stored, and backed up using NBViewer and GitHub, which we will discuss later.
Two important points to keep in mind as we introduce Python are:

- Python is Executebale Pseudocode,
- & Python is an object oriented language.

If this is the first time you have heard these terms, there is no reason to be intimiated. Each of these bullet points is effectively contributing to the same idea, "[T]he Python language is easy to fall in love with." (McKinney)

Python is Executeable Pseudocode.
This is a spoiling characteristic of the Python language. As you learn about writing code, or for those with coding experience, Python will surprise you time and time again as code you write executes with minimal syntax errors. With little understanding of programming logic, users can write what they think should work and, more often than not, it will work.

Python is an object oriented language.
Many smarter and more experienced programmers are working tirelessly to make analyzing data as painless as possible. A large portion of what we need Python for has already been coded into modules, libraries, methods, objects, etc. Thanks to these objects containing their own functions/methods and data,various examples in this notebook will require very few lines of code to produce a lot of information.

If you would like a more indepth introduction of the IPython Notebook, here is what CO-founder Dr. Granger has to say about it: http://bit.ly/1rVyrpi

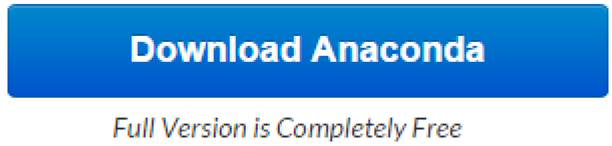
2.2 Installing Python

During this project, Dr. Doi has put together a document for "Installation and Configuration of Python on PC***** (mainly based on Sheppard's Inrtro to Python for Econometrics, Statisics, and Data Analysis - IPESDA)". Most of this section will be directly from his document.
*****[see IPESDA for install instructions for Mac/Linux]

The first and largest step is the installation of Anaconda. Anaconda is a "Completely free enterprise-ready Python distribution for large-scale data processing, predictive analytics, and scientific computing" with "195+ of the most popular Python packages for science, math, engineering, and data analysis". Thanks to Anaconda, beginning in Python is a relatively painless process.

• Download Anaconda (https://store.continuum.io/cshop/anaconda/)

Follow the link above then click on "Download Anaconda" in the upper right.



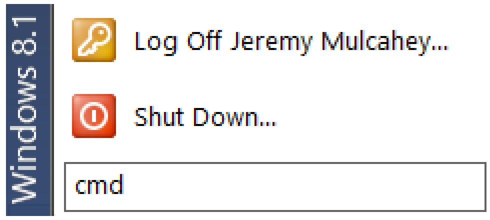
Click on the picture that matches your operating system.

For this project, everything is coded in Python 2.7, not 3.4. Additionally, the books and resources Dr. Doi and I used to learn python and build this project suggest the use of Python 2.7, for now. So, download Python 2.7.

Open the installer and follow the steps for the steps for the default/recommended installation. During installation, be sure to install in default directory(C:/Anaconda). If Anaconda is not installed there, be sure that target directory contains no unicode characters or spaces. Otherwise subsequent steps may not work well.

2.3 Updating Conda, Anaconda, and Anaconda Packages

• After installation is complete, open the command prompt. This can be done by opening the windows start menu by clicking in the lower left of your desktop. Just above the icon there will be a search box. Type letters **cmd** into the search box and hit enter.



A black window will open on the screen. Type the following lines one at time, hitting enter after each one and allowing the program to finish before submitting the next line:

```
conda update conda
conda update anaconda
```

The statements above can be used at any point to ensure that Anaconda is up to date with the latest packages.

I recommend repeating this update process periodically with any packages you use. Simply open **cmd**, type conda update and the name of the package you want to check for update and hit enter. Do it now.

```
conda update pandas
```

The statement above will update the pandas package and the packages it is built on (ie Numpy). This is helpful since the individual packages update more frequently than Anaconda. A few times during this project I was attempting to access functions in a packages that I found in online documentation, but they did not exist. Executing these update commands solved the problem every time.

2.4 Installing and Updating Additional Packages

The command prompt (**cmd**) should still be open at this point. If it is not, then open it.

Let's install our first package using the command prompt the same way we used it to update a package. Type the following line and hit enter:

```
pip install seaborn
```

This statement installs the "Seaborn: statistical data visulation" package. This will be used in a later section.

This process can be repeated for any packages you come across and want to try.

```
pip install [name of package]
```

Note: Packages installed using pip have a different update command:

```
pip install --upgrade [name of package]
```

TIP: For pip packages that fail to update, this has worked for me:

```
pip uninstall seaborn
pip install seaborn
```

2.5 Anaconda Add-Ons (Optional)

I intend to use these Add-Ons in the future, but I did not use them for this project. If you want to move forward with the installations, skip to the next section. Otherwise, here are the descriptions and instructions from Dr. Doi's installation guide:

• Get Anaconda Add-Ons (https://store.continuum.io/cshop/anaconda/).

```
The Accelerate and IOPro Add-Ons speed up Python (Accelerate contain
s MKL optimizations)
```

- `Select "All Product are Free for Academic Use"`

- `Get license by email by filling out form (select "Anaconda Academic License")`

```
This license is good for one year.
```

- `Copy the license file in the .continuum folder found in the root USER directory of Windows.`

```
At the command prompt submit:

conda update conda
conda install accelerate
conda install iopro
```

- `If you look in the license file, it seems to grant access to numbapro, mkl, and iopro. So, the 30-day trial for mkl should now automatically be upgraded to a 1-year trial. If you remove the license file from the .continuum directory, you should see a message in the console when launching IPython Notebook that mkl is only good for xxx days if you're still within the 30-day trial window. When the license file is in the directory the warning message disappears.`

Chapter 3
Setting up the IPython Notebook

3.1 Install MathJax

Whether or not you intend to use LaTeX, it is recommended that MathJax is installed before using IPython Notebook. The package that enables us to use LaTeX code in the IPython Notebook is called MathJax. Since MathJax is included in the Anaconda package list (http://docs.continuum.io/anaconda/pkg-docs.html), it can be managed with conda install and conda update (not pip). Running this conda install will add MathJax 2.2-0 (or later) as a "NEW" package. So, install it now:

```
conda install mathjax
Hit y, then enter.
```

```
The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    mathjax-2.2                |                0         7.1 MB
    requests-2.4.0             |            py27_0        572 KB
    ------------------------------------------------------------
                                           Total:         7.6 MB

The following NEW packages will be INSTALLED:

    mathjax:  2.2-0

The following packages will be UPDATED:

    requests: 2.3.0-py27_0 --> 2.4.0-py27_0

Proceed ([y]/n)? y
```

3.2 Accessing the IPython Notebook

Everything, from a Python standpoint, is ready to use. There are several ways to access the IPython Notebook for regular use. I would use one of the following three options:

- Command Prompt,
- Windows start up,
- or creating a shortcut.

Command Prompt: Since the Anaconda installation has the IPython Notebook open in a restrictive directory, the command prompt is the most versatile way to your notebook kernel. This approach gives you the option to work out of any directory you choose. I prefer to use **C: as my home directory since all of my notebooks and files are readily accessible from this location. To do this I type, type cd ../../. Then type ipython notebook.**

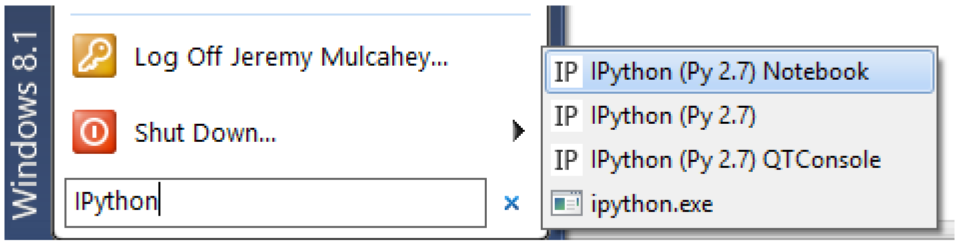
```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\WINDOWS\System32>cd ../../

C:\>ipython notebook
```

If you choose to use a different directory, or have more directories to back out of to reach C:

, simply use cd ../ for each directory you need to back out of, then cd [directory name] to change to the desired directory (ie. cd desktop/myfiles/calpoly)

```
Note: The following two methods open IPython Notebook in your IPytho
n Notebook directory and only allows access to the IPython Notebook
directory and its subdirectories.
```

Windows start up: Click the Windows start up menu and type IPython into the search box. You will see these options:



Select IPython Notebook from the list and it will open in your Web-browser.

Creating a shortcut: Do eveything from the "Windows start up" instructions above, except for selecting "IPython Notebook from the list". Instead, right-click and hold on IPython Notebook and drag it to your desktop. Release the right-click and select Create shortcuts here.

From there, you can move the shortcut anywhere you desire.

3.3 Your first IPython Notebook

Once IPython Notebook opens in the browser, you will see a relatively blank page with the IP[y]: Notebook heading.

Click on New Notebook.

Change the title by clicking on the word Untitled by the IP[y]: Notebook header.

"Enter a new notebook name:", hit ok, and you will have your first IPython notebook.

You're ready to start using Python, but that's only part of our process. The last big step we need to take is sharing and backing-up your notebooks on GitHub.

Note:After these installation chapters, if you would like to learn more about the IPython Notebook from Co-founder Dr. Granger: http://bit.ly/Zsoiqc

Chapter 4
Starting with GitHub

If you haven't heard of GitHub yet, you will. Several people I have asked about getting a data science job have given me the same order, "Get a GitHub". GitHub is the easiest way to share your work with potential employers, back-up your projects/assignments, and work simultaneously with other students/colleagues on different sections of the same project. Aside from how essential GitHub is for aspiring data scientists and programmers, it's required to use NBViewer (how we currently share the IPython Notebooks).

4.1 Create an account

Creating an account with GitHub is very straightforward. Go to https://github.com/ and create one now.

4.2 Learn what GitHub is and how to use it

GitHub has greatly simplified every process we will need to set up and share Notebooks. We can work around many of the issues I struggled with over the past year.

Thoroughly read this GitHub introduction (it's brief) and you will almost be done with setting up and using your GitHub.

https://guides.github.com/activities/hello-world/

4.3 Installing and understanding GitHub Desktop

This miraculous tool is what now makes GitHub so simple that anyone can use it.

Install GitHub Desktop using this link: https://windows.github.com/

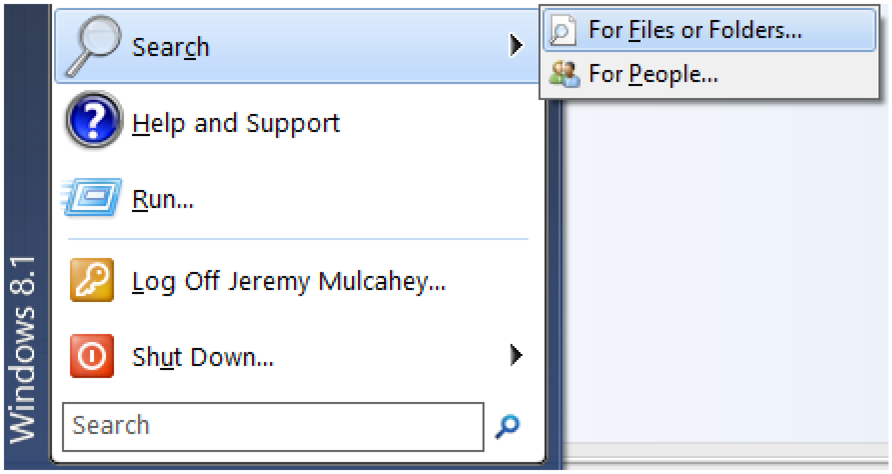
Read this brief tutorial on GitHub desktop: https://guides.github.com/introduction/getting-your-project-on-github/index.html

If you are new to programming, I highly recommend using the desktop tool from this point on.

4.4 Using GitHub Desktop

Open GitHub desktop.

Locate the folder that stored your IPython Notebook from earlier. If you are not sure where it is stored, you can perform a windows search to find it.



Type the name of your notebook into the search bar, or search for "IPython Notebooks".

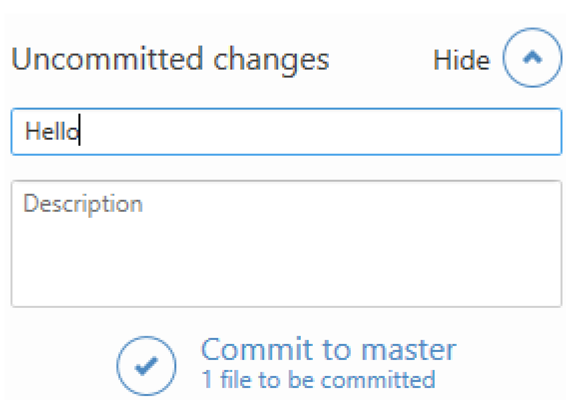
Once you have located the folder containing your notebook, drag it into GitHub desktop (as done in the previous tutorial).

Now the repository can be viewed online in your GitHub account.

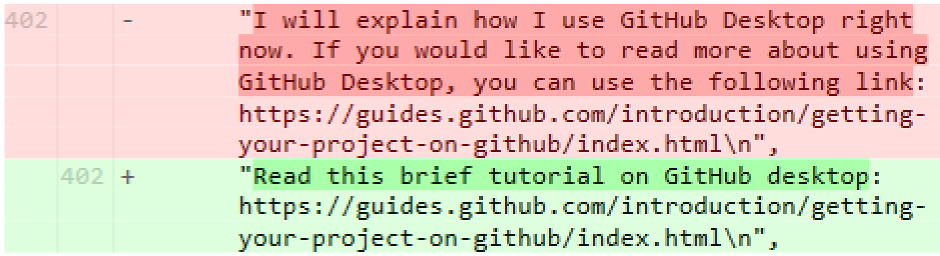
Since the tutorial uses GitHub Desktop for Mac, I'll show you how I use it for windows.

Return to your IPython Notebook in your web browser. Type print('Hello World') into the box and hit Shift+Enter. The notebook will execute this line of code. Hit the save icon on the left side of the icon bar.

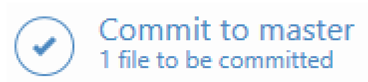
Having completed a change to your Notebook, return to GitHub desktop. GitHub desktop will now say you have Uncommitted changes. In the summary box, write what you feel summarizes the change you made. For this case, type Hello into the Summary box. You can add a detailed description in the Description box, if you choose.



The right side of the program will show you what has been removed and added from the file with the Uncommitted Changes. Below is an example of a change I made to this section.



After reviewing the changes, click Commit to master.



Finally, in the upper right corner of the program, click the Sync button.

From now on, your notebooks will be automatically updated in GitHub desktop. Repeat this commit process whenever you want to back them up online or share the changes you have made.

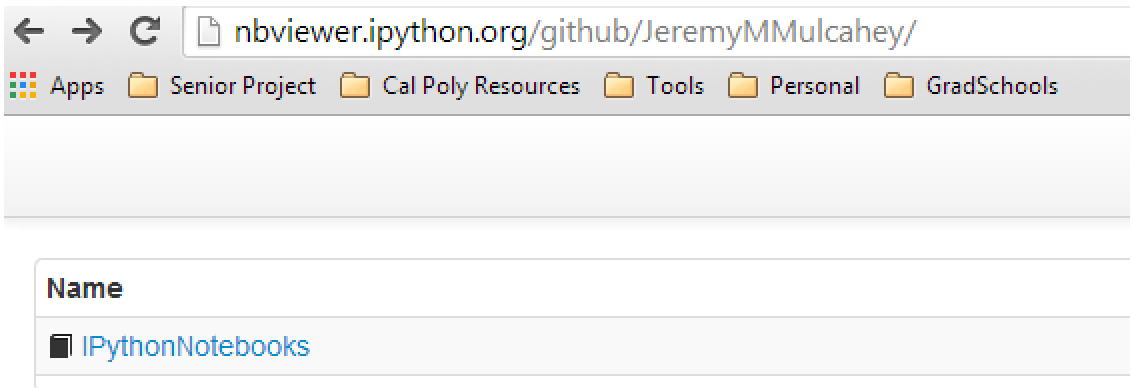
Chapter 5
Sharing IPython Notebooks with NBViewer

The hard parts are over.

To share notebooks, we need to use NBViewer. "IPython Notebook Viewer is a free webservice that allows you to share static html versions of hosted notebook files. If a notebook is publicly available, by giving its url to the Viewer, you should be able to view it."

To obtain a url for sharing your notebook, go to http://nbviewer.ipython.org/ Simply type your GitHub username (Section 4.1) into the box provided and hit "Go!".

Save the URL from this page. This is the page that allows others to view your committed IPython Notebooks on GitHub using NBViewer!

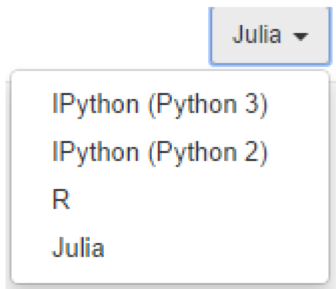


Chapter 6
R for IPython Notebook

Is it time to code yet?

It could be... As statistics majors, we have a required course in R. Since some of you might have prior experience with R, we should set R up right now. You can familiarize yourself with your IPython Notebook by running your pre-existing R scripts, or code snippets from your introductory statistics courses.

NOTE: This is the most important note yet. Some time, some day, you can expect to see Python 3 and R support built-in to the IPython Notebook itself. It will be as simple as a dropdown menu in the upper-right corner of the notebook.



Until then, if you really want R now (like I did), buckle-up! It's going to be a bumpy ride.

6.1 Installing Rpy2

Rpy2 is the package we need to run R through the IPython Notebook. Unfortunately, it's not as simple as pip install rpy2.

I have it on good authority that this entire section is completely unnecessary for Mac users. Rpy2 installs and works on Macs with zero issues. For us Windows users, you are benefiting from many meetings between Cal Poly's Dr. Brian Granger and myself, and many failed attempts to install Rpy2 on Windows. Luckly, there is now a native R kernel for IPython.

To obtain Rpy2, click on the link. Hit control+f and type rpy2 into the search bar. http://www.lfd.uci.edu/~gohlke/pythonlibs/

Hitting enter a couple times should take you to the Rpy2 section. Download version 2-2.4.3 for the correct operating system.

Rpy2 (experimental) provides access to the R software environment for statistical computing and graphics. Built with Rtools against msvcrt.dll and R 3.1.

- rpy2-2.3.9.win-amd64-py3.2.exe
- rpy2-2.3.9.win32-py3.2.exe
- rpy2-2.4.3.win-amd64-py2.7.exe
- rpy2-2.4.3.win-amd64-py3.3.exe
- rpy2-2.4.3.win-amd64-py3.4.exe
- rpy2-2.4.3.win32-py2.7.exe
- rpy2-2.4.3.win32-py3.3.exe
- rpy2-2.4.3.win32-py3.4.exe

Follow the next steps carefully. In my experience, here's where it gets tricky. The wrong combination of Rpy2 and version of R crashes the kernel.

If you have R installed, I recommend backing up your scripts before proceeding (and anything else you want to save).

If you do not have R installed, and you are using Windows 8.1, I recommend installing R version 3.0.2. http://cran.r-project.org/bin/windows/base/old/3.0.2/

For those without R installed:

- **Follow the link above. Download and install R version 3.0.2.**
- **Run the Rpy2 installation file.**

For those with R installed:

- **Run the Rpy2 installation file.**

6.2 Testing the Rpy2 installation

Go to your IPython Notebook. In the next cell (the empty one below Hello World), type these lines in their own cells (hitting shift+enter after each line):

```
In [1]: import rpy2
```

```
In [2]: %load_ext rpy2.ipython
```

```
In [3]: %R install.packages("lattice")
```

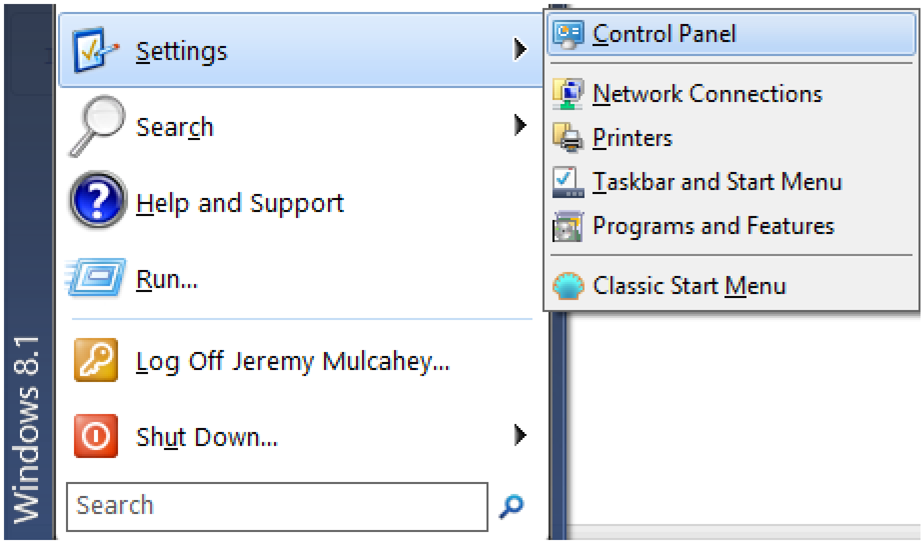
Scroll down to USA and select one of the USA portals.

```
In [4]: %R library(lattice)

Out[4]: <StrVector - Python:0x000000000CDD5E48 / R:0x0000000022F81408>
        [str, str, str, ..., str, str, str]
```

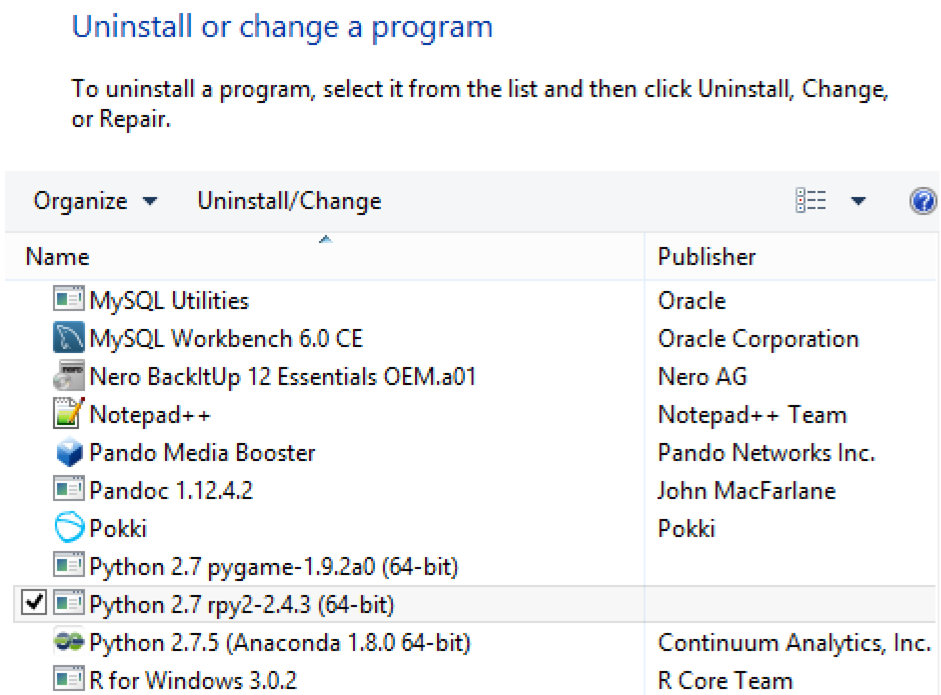
If your output matches mine, then R is working. It is successfully downloading and installing packages. Skip ahead to Chapter 7.

6.3 If the Kernel Crashes

Go to your windows control panel.



Scroll down to Programs and Features.



Uninstall both R for Windows and Python 2.7 rpy2-2.4.3.

Download a different version of R (ie 3.0.1, 3.0.3, 3.1.0, etc). Install the new version of R and install Rpy2 again.

Return to the "Testing Rpy2 Installation" section above and repeat the process.

There should be a combination that enables R to work in the IPython Notebook.

After Rpy2 is working as intended, install any version of R you prefer and add your backed up files. Rpy2 will work with the version it needs to and when you work exclusively in R, you can use your preferred version.

6.4 R installation Final Notes

Dr. Granger and his team are constantly working to improve functionality in all areas of the IPython Notebook. R is one of those areas. Originally, getting Rpy2 to work on a PC was almost impossible. Then, it was very hard. Now, it's a bit touchy. Soon, as previously mentioned, it will be a drop down option in the notebook itself.

Chapter 7
Analyzing NIST datasets in the IPython Notebook

Before introducing Python packages as an alternative to R and SAS, Dr. Doi and I felt it wise to investigate the precision of Python's data analysis capabilities compared to R's and SAS's.

The NIST (National Institute of Standards and Technology) "is the federal technology agency that works with industry to develop and apply technology, measurements, and standards."

In this section I will analyze datasets from the NIST's Dataset Archives. I will compare the NIST's "Certified Values" to the values obtained from analyzing the data in R, SAS, and Python.

This section will provide an introduction to coding in Python, using statistical packages for analyzing and visualizing data, and establish ways to extract precise values in R, SAS, and Python.

Note: If you would like to learn more about the Python syntax, visit Dr. Granger's notebook (http://bit.ly/1y8h6hS). Otherwise, you can copy and paste sections of my code and change the arguments as needed.

7.1 Python Packages for data analysis and the Import Cell

Packages can be imported for use in any cell at any time. My preference is to import relevant packages and commands into a common cell at the beginning of the notebook, or section of the notebook. This will provide a collection of packages, with their abbreviations, in one convenient location. In the event that you cannot remember if you have imported a package, or what you imported it as, you can jump to your import cell.

Below is the list of packages we will need for this section. Please go to your cmd and pip install urllib2 before moving on. Then, copy and execute the cell of packages by pasting them into your notebook cell and hitting shift+enter.

In [2]:
```
import urllib2 as ul
import pandas as pd
import numpy as np
import matplotlib
import scipy as sp
from statsmodels.formula.api import ols
from statsmodels.stats.anova import anova_lm
import matplotlib.pyplot as plt
from IPython.core.display import Image
from matplotlib.gridspec import GridSpec
import seaborn as sns

#This line allows the graphs to show up in the notebook cells
%matplotlib inline
```

7.2 Linear Regression Analysis: Norris Dataset

7.2.1 Object Oriented Language Introduction: Reading in and preparing data from an ASCII webpage

In [3]:
```
#create a variable for the web address
url = 'http://www.itl.nist.gov/div898/strd/lls/data/LINKS/DATA/Norris.dat'

#open creates a file object named Norris.dat
#wb allows us to write to the file object
#file objects have a built-in function to write to the file
#use the urllib2 as ul package to write the website to the file
#the .read function is built-in to the ul object
open('data/Norris.dat','wb').write(ul.urlopen(url).read())
```

This is our first cell that takes advantage of Python being an object oriented language. It might take some time to wrap your head around it, or it might not.

What just happened is... as we create objects, which are exactly what you might intuitively think of as objects (ie: a ball), those objects have their own functions (methods) and characteristics (data) we can immediately use.

Let's continute with the ball example. If you have a ball, you do not determine its color, size, or inertia. You do not show the ball how to roll. The ball looks how it looks and if you set it on a decline, it rolls away.

The ball knows what it knows, and you know how to use it to get what you want. You can roll the ball at pins, throw it, roll it up hill, hit it with your hand, hit it with a racket, etc.

That is what we are doing here. A file knows how to write to itself, and the open function knows how to read the content of the page. By passing arguments (such as the filename and url) we are using what the objects know to accomplish what we need to accomplish.

In [4]:
```
#creates a np array named NorrisData
#uses the loadtxt method from the NumPy package (np) to read in
#the data from line 60 to the end of the file
NorrisData = np.loadtxt('data/Norris.dat',skiprows=60)
```

Since the NIST datasets are provided in ASCII and start on line 60 of the webpage, I found the best way to import them was to create the file (which we just completed), import them as a NumPy array, then convert them to a Pandas DataFrame.

Having data in a Pandas DataFrame provides the greatest amount of flexbility for analyzing and manipulating data. My goal through-out the project was to make sure I could get any data I was working with into this format.

In [5]:
```
#create a DataFrame object using the Pandas package (pd)
#the columns can be named during the conversion from ndarray to pd.df
NorrisFrame = pd.DataFrame(NorrisData,columns=['y','x'])
```

Always check to make sure the data was read in correctly. There are many ways to do this.

In [6]:
```
#the dataframe object displays its first five observations
NorrisFrame.head()
```

Out[6]:

	y	x
0	0.1	0.2
1	338.8	337.4
2	118.1	118.2
3	888.0	884.6
4	9.2	10.1

In [7]:
```
#the dataframe object displays its last five observations
NorrisFrame.tail()
```

Out[7]:

	y	x
31	117.6	118.3
32	228.9	229.2
33	668.4	669.1
34	449.2	448.9
35	0.2	0.5

It's clear to me that we have all 36 observations (indicies 0 to 35). The first value matches the first value of the webpage and the last value macthes the last value of the webpage. If you're uncomfortable with the indexing, you can check the number of observations with:

In [8]:
```
#number of observations
len(NorrisFrame)
```

Out[8]: 36

Now is a good time to introduce the dir() function and explain why the previous two functions required no arguments.

In Python, accessing functions/methods of an object always passes the object as the first argument. We do not see it, but what the code is really doing is executing NorrisFrame.head(NorrisFrame), which returns the head of the NorrisFrame object.

The dir() function is how I knew to use the .head() and .tail() functions.

To access the full list of data and methods an object has, simply type: dir(object name)

In [10]:
```
#execute this code in your notebook
#it is too much output for this document
dir(NorrisFrame)
```

As you can see, the Pandas DataFrame is a robust and versatile object.

Before moving to the next section, it's worth summarzing and acknowledging what we've done. We created a data file from an ASCII webpage, read the data file into an array, and converted it to a DataFrame with named columns - in 4 lines of code.

7.2.2 NIST Certified Values

The Certified values we are trying to match can be found at: http://www.itl.nist.gov/div898/strd/anova/SiRstv_cv.html

For easy reference, I have included them:

Certified Regression Statistics

Parameter	Estimate	Standard Deviation of Estimate
β_0	-0.262323073774029	0.232818234301152
β_1	1.00211681802045	0.429796848199937E-03

Residual Standard Deviation	0.884796396144373
R-Squared	0.999993745883712

Certified Analysis of Variance Table

Source of Variation	Degrees of Freedom	Sums of Squares	Mean Squares	F Statistic
Regression	1	4255954.13232369	4255954.13232369	5436385.54079785
Residual	34	26.6173985294224	0.782864662630069	

The Certified values are quite precise. It's important to view them so we can match the number of decimal places using SAS, R, and Python.

In [11]:
```
#saving the values as strings (ie in quotes '') helps us with future
#steps in this process, we will need their len() and to compare them digit
 by
#digit, this also solves a problem I had with trailing zeros being ignored
B0 = '-0.262323073774029'
B1 = '1.00211681802045'
STDofEstB0 = '0.232818234301152'
STDofEstB1 = '0.000429796848199937'
resstd = '0.884796396144373'
Rsq = '0.999993745883712'
ModSS = '4255954.13232369'
ModMSE = '4255954.13232369'
ModSSResid = '26.6173985294224'
ModMSEResid = '0.782864662630069'
Fstat = '5436385.54079785'

#creates an array object named CertVals containing the certified values
CertVals = np.array([B0, B1, STDofEstB0, STDofEstB1,resstd,Rsq,ModSS,
                     ModMSE, ModSSResid,ModMSEResid,Fstat])
```

We now have the NIST Certified values in an array for our later comparisons.

7.2.3 Linear Regression and ANOVA values in Python

The package we will use for data analysis in this notebook is Statsmodels.

Their help documentation can be found at: http://statsmodels.sourceforge.net/

In [12]:
```
#we imported ols (Oridnary Least Squares) in our import cell
#we named our columns x and y, now we need to tell the OLS function
#what we want to regress. The second argument is our DataFrame.
NorrisLM = ols('y ~ x', NorrisFrame).fit()
print NorrisLM.summary()
```

```
                            OLS Regression Results

==========================================================================
====
Dep. Variable:                      y   R-squared:                       1
.000
Model:                            OLS   Adj. R-squared:                  1
.000
Method:                 Least Squares   F-statistic:                 5.436
e+06
Date:                Sun, 12 Oct 2014   Prob (F-statistic):           4.65
e-90
Time:                        20:57:41   Log-Likelihood:                -45
.647
No. Observations:                  36   AIC:                             9
5.29
Df Residuals:                      34   BIC:                             9
8.46
Df Model:                           1

==========================================================================
====
                 coef    std err          t      P>|t|      [95.0% Conf. I
nt.]
--------------------------------------------------------------------------
----
Intercept     -0.2623      0.233     -1.127      0.268        -0.735     0
.211
x              1.0021      0.000   2331.606      0.000         1.001     1
.003
==========================================================================
====
Omnibus:                        2.696   Durbin-Watson:                   1
.272
Prob(Omnibus):                  0.260   Jarque-Bera (JB):                1
.566
Skew:                          -0.450   Prob(JB):                        0
.457
Kurtosis:                       3.485   Cond. No.
855.
==========================================================================
====
```

```
Tips: Don't forget to use dir() on unfamiliar objects. The linear mo
del object has a wealth of information such as:

.get_inflence().summary_table()  (for Cook's, student redisuals, h,
fitted values, etc)
.resid()   (for residuals)

The anova_lm() function provides just the anova table.

print NorrisLM.summary() and executing just NorrisLM.summary() (with
out the print command) provide the table in different formats.

I recommend trying these before moving on to familiarize yourself wi
th some of the features statsmodels offers.
```

As you can see from the output table, the results provided by OLS are far from the precision we need. Like R and SAS, we can extract more decimal places. Some of these values can be extracted directly from the linear model object and others have to be accessed through the EstimatedParameters object.

In [13]:
```
#create an Estimated parameters object
NorrisParams = NorrisLM.params
```

In [14]:
```
#repr() converts our values to strings without losing truncating them
PB0 = repr(NorrisParams[0])
PB1 = repr(NorrisParams[1])
PSTDofEstB0 = repr(NorrisLM.bse[0])
PSTDofEstB1 = repr(NorrisLM.bse[1])
Presstd = repr(np.sqrt(NorrisLM.mse_resid))
PRsq = repr(NorrisLM.rsquared)
PModSS = repr(NorrisLM.ess)
PModMSE = repr(NorrisLM.mse_model)
PModSSResid = repr(NorrisLM.ssr)
PModMSEResid = repr(NorrisLM.mse_resid)
PFstat = repr(NorrisLM.fvalue)

PyVals = np.array([PB0, PB1, PSTDofEstB0, PSTDofEstB1,Presstd,PRsq,
                     PModSS,PModMSE, PModSSResid,PModMSEResid,PFstat])
```

Unforunately, there's no shortcut or easier explanation to what happened in the cell above. Determining how to extract those values was the result of a lot of time, dir() usage, and help documentation. The great part is, once you've done it, you'll know how to repeat the process (as we've done here by replicating my extractions).

7.2.4 Linear Regression and ANOVA values in R (using Rpy2)

Here's our first real look at R in the IPython Notebook. We can use this notebook to extract the necessary values for our precision comparison (SAS will be a different story).

To prepare for working with R, I exported our Pandas DataFrame as a csv file.

In [15]:
```
NorrisFrame.to_csv('C:/Users/flunk_000/Desktop/CalPoly/IPythonNotebook/SeniorProject/data/NorrisFrame.txt')
```

As i'm sure you noticed during the Rpy2 installation, we need import and load rpy2 to use it.

In [16]:
```
import rpy2
%load_ext rpy2.ipython
```

Since this notebook is about using Python and the IPython Notebook, the R and SAS data extractions will be brief.

To run R code in your notebook, start the line with %R. If you want execute an entire cell of R code, start the cell with %%R.

In [15]:
```
%%R
#read in data
setwd("C:/Users/flunk_000/Desktop/CalPoly/IPythonNotebook/SeniorProject/")
;
Norris = read.csv('data/NorrisFrame.txt', header=T);
NorrisFrame = as.data.frame(Norris);
```

In [16]:
```
%R head(NorrisFrame)
```

Out[16]:

	X	y	x
0	0	0.1	0.2
1	1	338.8	337.4
2	2	118.1	118.2
3	3	888.0	884.6
4	4	9.2	10.1
5	5	228.1	226.5

In [17]:
```
%R tail(NorrisFrame)
```

Out[17]:

	X	y	x
0	30	10.2	11.1
1	31	117.6	118.3
2	32	228.9	229.2
3	33	668.4	669.1
4	34	449.2	448.9
5	35	0.2	0.5

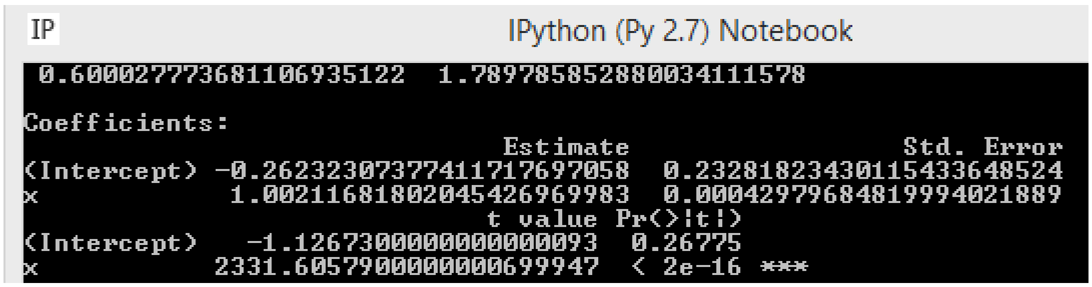
We have verified the data was read in correctly.

In [18]:
```
%%R
#analysis in R
```

```
NorrisLM = lm(y ~ x, data=NorrisFrame);
NorrisResid = NorrisLM$residuals;
NorrisCoef = NorrisLM$coefficients;
NorrisANOVA = aov(y ~ x, data=NorrisFrame);
```

In [19]:
```
%%R
#obtaining precise numbers in R
print(summary(NorrisANOVA), digits =20);
print(summary(NorrisLM), digits =20);
print(sqrt(deviance(NorrisANOVA)/df.residual(NorrisANOVA)), digits = 20);
```

Currently, these values print to the R kernel in your IPython Notebook console:


IP IPython (Py 2.7) Notebook

```
 0.600027773681106935122  1.789785852880034111578

Coefficients:
                               Estimate                  Std. Error
(Intercept) -0.262323073774117176970 58  0.23281823430115433648524
x            1.00211681802045426969983  0.00042979684819994021889
                             t value Pr(>|t|)
(Intercept)   -1.12673000000000000093  0.26775
x           2331.60579000000000699947  < 2e-16 ***
```

There are different tricks you can experiment with in order to print to the notebook instead of the console, but they're not the focus of this section (and will soon be unnecessary). If you want to be able to copy and paste the values we need, copy and paste my R code into R and run it.

In [17]:
```
RB0 = '-0.262323073774117 18'
RB1 = '1.00211681802045427'
RSTDofEstB0 = '0.23281823430115431'
RSTDofEstB1 = '0.00042979684819994'
Rresstd = '0.884796396144379437 84'
RRsq = '0.999993745883712'
RModSS = '4255954.13232369348406792'
RModMSE = '4255954.13232369348406792'
RModSSResid = '26.61739852942280038'
RModMSEResid = '0.78286466263010001665'
RFstat = '5436385.5407999996096'

RVals = np.array([RB0, RB1, RSTDofEstB0, RSTDofEstB1,Rresstd,RRsq,
                    RModSS,RModMSE, RModSSResid,RModMSEResid,RFstat])
```

7.2.5 Linear Regression and ANOVA values in SAS

Here is my code for the SAS analysis and value extractions:

In [33]:
```
OPTIONS NODATE NONUMBER CENTER LS=160;
*Removes the header information and centers output;
OPTIONS FORMDLIM="~";

data NorrisData;
infile "C:/Users/flunk_000/Desktop/CalPoly/IPythonNotebook/SeniorProject/data/NorrisFrameSAS.txt" DLM=',';
input subject y x;
```

```
ODS TRACE ON;
proc glm data= NorrisData;
        model y = x;
                output out = linear_norris;
                ODS output ParameterEstimates = pe;
                ODS output Overallanova = anova;
                ODS output fitstatistics = fs;
run;
ODS Trace off;

proc print data=pe;
        format estimate 20.19
                stderr 20.19
                probt pvalue20.19;
        run;

proc print data=anova;
        format SS 20.19
                MS 20.19
                fvalue 20.19;
        run;

proc print data=fs
        format
                rsquare 20.19
                rootmse 20.19;
        run;
```

In [18]:
```
SB0 = '-0.262323073773836000'
SB1 = '1.002116818020450000'
SSTDofEstB0 = '0.232818234313772000'
SSTDofEstB1 = '0.0004297968482232346'
Sresstd = '0.884796396192334000'
SRsq = '0.9999937458837110000'
SModSS = '4255954.132323680000'
SModMSE = '4255954.132323680000'
SModSSResid = '26.617398532308000000'
SModMSEResid = '0.782864662714941000'
SFstat = '5436385.54020847000'

SASVals = np.array([SB0, SB1, SSTDofEstB0, SSTDofEstB1,Sresstd,SRsq,
                    SModSS,SModMSE, SModSSResid,SModMSEResid,SFstat])
```

7.2.6 Testing Python's Precision against NIST, R, and SAS

I prefer to write functions for any task I have to repeat (and any task I can get away with writing a function for). Let's look at the functions we'll use to compare the precision of our programs.

In [19]:
```
#def allows us to define a function
#cert_val_lengths is the name of the function
#it creates an array of the lengths of NIST values for later precision com
parisons
#the len() function determines the length of a string
#the dtype argument here returns an array of integer values
def cert_val_lengths(CertifiedValuesArray):
```

```
    #creates an array of zeros to store the lengths
    CertValLengths = np.zeros(len(CertifiedValuesArray),dtype=int)

    for val in range(len(CertifiedValuesArray)):
        CertValLengths[val]= len(CertifiedValuesArray[val])
    return CertValLengths
```

In [20]:
```
#function requires three values, one I provide (from R, SAS, Py, etc)
#another from the NIST certified values
#third is the precision we are looking for
def nist_compare(MyValue, NISTValue, CertValLength):

    #converts the value to a list
    MyValueList = list(MyValue)

    NISTValueList = list(NISTValue)

    counter = 0

    #checks to see how similar the values are
    #the CertValLength allows us to ignore the extraneous precision
    #added to the arrays by NumPy
    for val in range(CertValLength):
        if MyValueList[val] == NISTValueList[val]:
            counter+=1
        else:
            return counter

    #returns how many values matches
    return counter
```

We have a way to compare the values, now let's write a function to compare the arrays.

In [21]:
```
def array_compare(MyArray,NISTArray,LabelArray):

    #create an empty array for value comparisons
    ValMatches = np.zeros(len(NISTArray),dtype=int)

    #uses our first function to create the lengths of the certified values
    CertValLengths = cert_val_lengths(NISTArray)

    for val in range(len(LabelArray)):

        #compares the values using the previous function
        ValMatch = nist_compare(MyArray[val],NISTArray[val],CertValLengths
[val])

        #prints the comparison and uses our pre-determined precision
        print(LabelArray[val], ValMatch,'of',CertValLengths[val])

        #stores the values in our empty array
        ValMatches[val] = ValMatch

    #returns the precision we were looking for
    return ValMatches
```

In [22]:
```
#create an array of the values we want to compare
NorrisLabels = np.array(['Beta0:','Beta1:','STDofEstimateB0:',
                          'STDofEstimateB1','resstd:','R-sq:',
                          'Model SS:','Model MS:','Model SSResid:',
                          'Model MSResid:','F-stat:'])
```

Let's look at how each program's output compared to the NIST Certified Values.

In [23]:
```
R = array_compare(RVals,CertVals,NorrisLabels)
```

```
('Beta0:', 15, 'of', 18)
('Beta1:', 16, 'of', 16)
('STDofEstimateB0:', 16, 'of', 17)
('STDofEstimateB1', 18, 'of', 20)
('resstd:', 16, 'of', 17)
('R-sq:', 17, 'of', 17)
('Model SS:', 16, 'of', 16)
('Model MS:', 16, 'of', 16)
('Model SSResid:', 15, 'of', 16)
('Model MSResid:', 14, 'of', 17)
('F-stat:', 13, 'of', 16)
```

In [24]:
```
SAS = array_compare(SASVals,CertVals,NorrisLabels)
```

```
('Beta0:', 14, 'of', 18)
('Beta1:', 16, 'of', 16)
('STDofEstimateB0:', 12, 'of', 17)
('STDofEstimateB1', 14, 'of', 20)
('resstd:', 12, 'of', 17)
('R-sq:', 16, 'of', 17)
('Model SS:', 15, 'of', 16)
('Model MS:', 15, 'of', 16)
('Model SSResid:', 10, 'of', 16)
('Model MSResid:', 11, 'of', 17)
('F-stat:', 11, 'of', 16)
```

In [25]:
```
Py = array_compare(PyVals,CertVals,NorrisLabels)
```

```
('Beta0:', 16, 'of', 18)
('Beta1:', 16, 'of', 16)
('STDofEstimateB0:', 16, 'of', 17)
('STDofEstimateB1', 18, 'of', 20)
('resstd:', 16, 'of', 17)
('R-sq:', 16, 'of', 17)
('Model SS:', 16, 'of', 16)
('Model MS:', 16, 'of', 16)
('Model SSResid:', 15, 'of', 16)
('Model MSResid:', 15, 'of', 17)
('F-stat:', 14, 'of', 16)
```

In [26]:
```
R,SAS,Py,cert_val_lengths(CertVals)
```

Out[26]:
```
(array([15, 16, 16, 18, 16, 17, 16, 16, 15, 14, 13]),
 array([14, 16, 12, 14, 12, 16, 15, 15, 10, 11, 11]),
```

```
array([16, 16, 16, 18, 16, 16, 16, 16, 15, 15, 14]),
array([18, 16, 17, 20, 17, 17, 16, 16, 16, 17, 16]))
```

We can do as little or as much as we want with this data (such as write functions to compare these integers and provide a rating, or determine if there are common values, such as SSE, that were rounded and used in other calculations yielding less a lower precision).

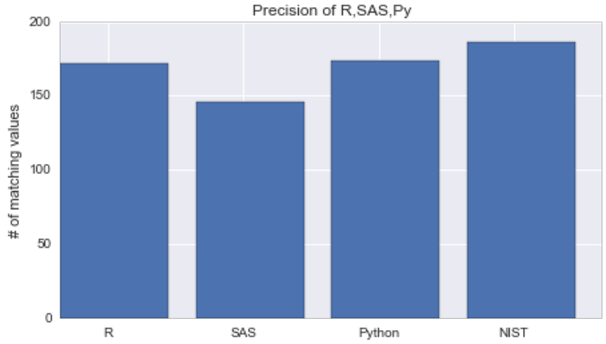
We can also use this information to pick the program that best matches the task at hand. For instance, R or Python did a great job of matching 18 values of Standard Deviation of Estimate for Beta1, while SAS ony matched 14 values. If 18 is your target precision, you could use either R or Python.

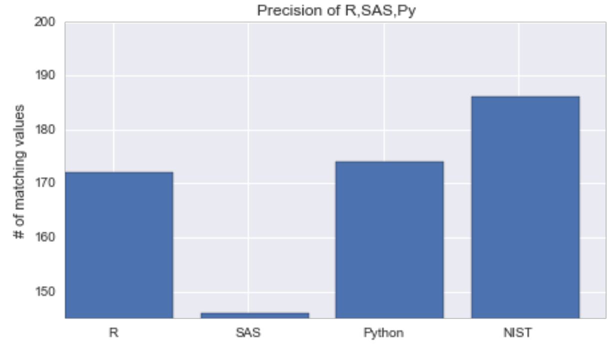
The important part is that upon visual inspection, Python appears to be the closest to the Certified values for Linear Regression, with R close behind, and SAS in third place.

We can use Numpy and matplotlib to verify if my visual inspection is correct:

In [28]:
```
#open a figure and add the axes
hist = plt.figure(figsize=(16,4))
gs = GridSpec(1,2)
axis = hist.add_subplot(gs[0,0])

programs=4
#create array of bar values
Matches = [np.sum(R),np.sum(SAS),np.sum(Py),np.sum(cert_val_lengths(CertVa
ls))]

#location of bars on plot
loc = np.arange(programs)

bars = axis.bar(loc, Matches)

axis.set_ylim(0,200)
axis.set_ylabel('# of matching values')
axis.set_title('Precision of R,SAS,Py')
axis.set_xticks(loc+.35)
XNames = axis.set_xticklabels(['R', 'SAS', 'Python','NIST'])

axis2 = hist.add_subplot(gs[0,1])
bars2 = axis2.bar(loc, Matches)

axis2.set_ylim(145,200)
axis2.set_ylabel('# of matching values')
axis2.set_title('Precision of R,SAS,Py')
axis2.set_xticks(loc+.35)
XNames = axis2.set_xticklabels(['R', 'SAS', 'Python','NIST'])
Matches
```

Out[28]:
```
[172, 146, 174, 186]
```





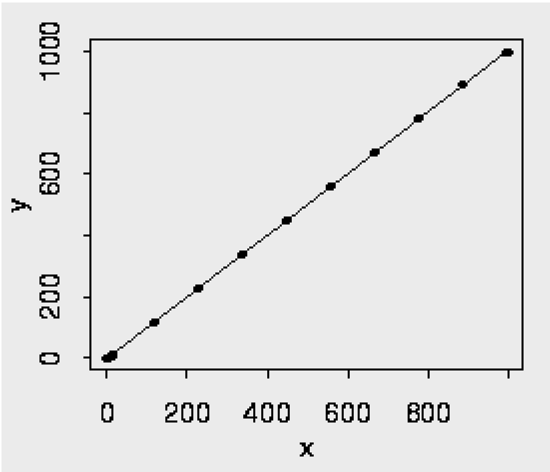
Over the 11 values of concern, Python was more precise than R by 2 decimal places. SAS was 26,28 decimals less precise than R,Py (respectively).

Important Note: This does not mean that SAS miscalculated the values. If you return to the SAS output, you will see that each SAS value has trailing zeros instead of additonal digits. SAS simply doesn't provide the level of precision we sought to completely match the certified values in this exercise.

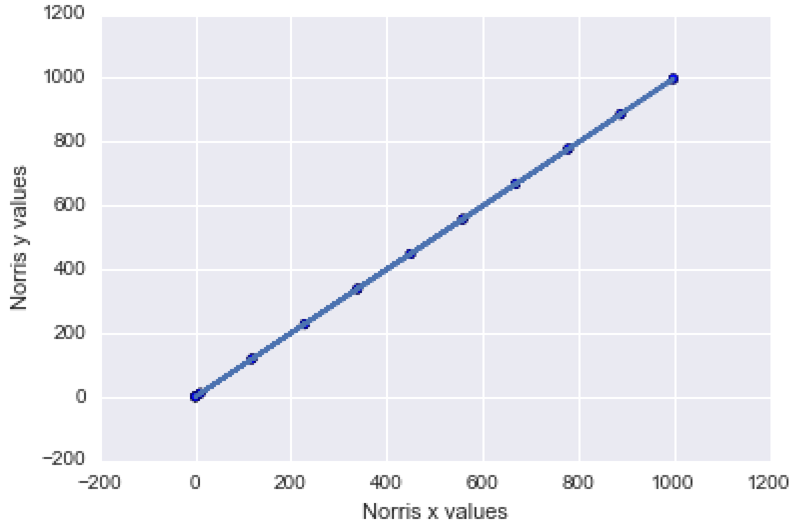
7.2.7 Plotting Norris data with Mathplotlib

The NIST graphics aren't as detailed as I would like, but they give us something to compare the Python graphics to.

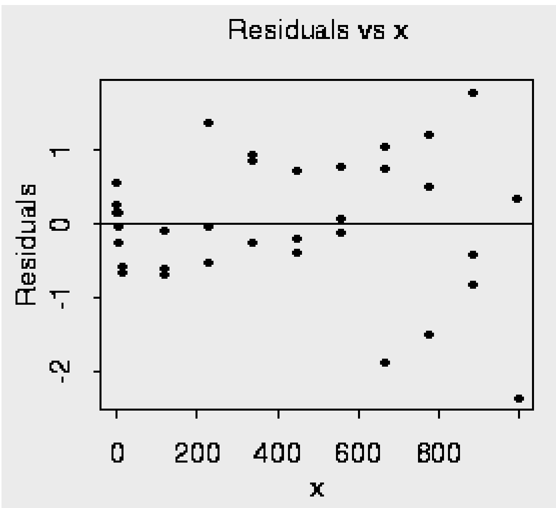
Norris Regression:



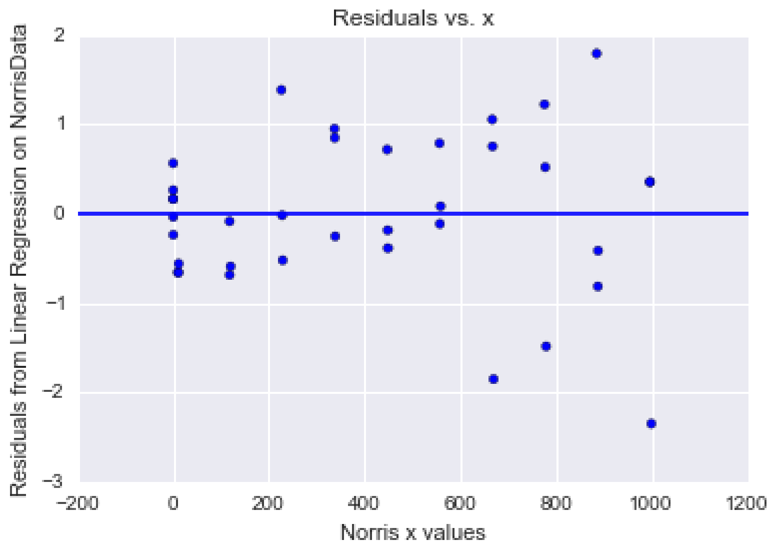
In [70]:
```
#using matplotlib
#plots the points
plt.scatter(NorrisFrame['y'],NorrisFrame['x'])

#draws the lines
plt.plot(NorrisFrame['y'],NorrisFrame['x'])

#labels the X axis and Y axis
plt.xlabel('Norris x values')
plt.ylabel('Norris y values');
```



Looks good. Let's check the residual plot:



In [71]:
```
#residual plot
plt.scatter(NorrisFrame['x'],NorrisLM.resid)

#adds a horizontal line at y=0
plt.axhline()
plt.xlabel('Norris x values')
plt.ylabel('Residuals from Linear Regression on NorrisData')

#adds a title
plt.title('Residuals vs. x');
```



As much as we can tell, it appears the graphs are the same. We have successfully created both of the provided NIST graphics. This is the end of the Linear Regression Analysis on the Norris dataset.

7.3 ANOVA: SiR Dataset

7.3.1 Reading in and preparing data from an ASCII webpage

For a detailed explanation of this process, refer to Section 7.2.1.

In [83]:
```
SiRurl = 'http://www.itl.nist.gov/div898/strd/anova/SiRstv.dat'
open('data/SiRstv.dat','wb').write(ul.urlopen(SiRurl).read())
```

In [84]:
```
SiRData = np.loadtxt('data/SiRstv.dat',skiprows=60)
SiRFrame = pd.DataFrame(SiRData, columns=['Instrument','Resistance'])
```

In [85]:
```
SiRFrame.head()
```

Out[85]:

	Instrument	Resistance
0	1	196.3052
1	1	196.1240
2	1	196.1890
3	1	196.2569
4	1	196.3403

In [86]:
```
SiRFrame.tail()
```

Out[86]:

	Instrument	Resistance
20	5	196.2119
21	5	196.1051

22	5	196.1850
23	5	196.0052
24	5	196.2090

Data was read in correctly.

7.3.2 NIST Certified Values

For a detailed explanation of this process, refer to Section 7.2.2.

The Certified values we are trying to match can be found at: http://www.itl.nist.gov/div898/strd/anova/SiRstv_cv.html

In [87]:
```
Certresstd = '0.104076068334656'
CertRsq = '0.190999039051129'
CertModSS = '0.0511462616000000'
CertModMSE = '0.0127865654000000'
CertModSSResid = '0.216636560000000'
CertModMSEResid = '0.0108318280000000'
CertFstat = '1.18046237440255'

SiRCertVals = np.array([Certresstd,CertRsq,CertModSS,CertModMSE,
                        CertModSSResid,CertModMSEResid,CertFstat])
```

7.3.3 ANOVA values in Python

For a detailed explanation of this process, refer to Section 7.2.3.

In [88]:
```
#In our formula argument, since we are analyzing Resistence by intstrument
#we need to treat instrument as a categorical variable with C()
SiRLM = ols('Resistance ~ C(Instrument)', SiRFrame).fit()

#you can chose to use the anova_lm function
#SiRANOVA = anova_lm(SiRLM)
```

In [89]:
```
Pyresstd = repr(np.sqrt(SiRLM.mse_resid))
PyRsq = repr(SiRLM.rsquared)
PyModSS = repr(SiRLM.ess)
PyModMSE = repr(SiRLM.mse_model)
PyModSSResid = repr(SiRLM.ssr)
PyModMSEResid = repr(SiRLM.mse_resid)
PyFstat = repr(SiRLM.fvalue)

SiRPyVals = np.array([Pyresstd, PyRsq, PyModSS, PyModMSE,PyModSSResid,
                      PyModMSEResid,PyFstat])
```

7.3.4 ANOVA values in R

For a detailed explanation of this process, refer to Section 7.2.4.

In [90]:
```
SiRFrame.to_csv('C:/Users/flunk_000/Desktop/CalPoly/IPythonNotebook/Senior
```

```
Project/data/SiRFrame.txt')
```

In [91]:
```
import rpy2
%load_ext rpy2.ipython
```

```
The rpy2.ipython extension is already loaded. To reload it, use:
  %reload_ext rpy2.ipython
```

In [92]:
```
%%R
#read in data
setwd("C:/Users/flunk_000/Desktop/CalPoly/IPythonNotebook/SeniorProject/")
;
SiR = read.csv('data/SiRFrame.txt', header=T);
SiRFrame = as.data.frame(SiR);
```

In [93]:
```
%R head(SiRFrame)
```

Out[93]:

	X	Instrument	Resistance
0	0	1	196.3052
1	1	1	196.1240
2	2	1	196.1890
3	3	1	196.2569
4	4	1	196.3403
5	5	2	196.3042

In [94]:
```
%R tail(SiRFrame)
```

Out[94]:

	X	Instrument	Resistance
0	19	4	195.9885
1	20	5	196.2119
2	21	5	196.1051
3	22	5	196.1850
4	23	5	196.0052
5	24	5	196.2090

Data was read in correctly.

In [95]:
```
%%R
#analysis in R
SiRFrame$Instrument = factor(SiRFrame$Instrument);
SiRANOVA = aov(Resistance ~ Instrument, data=SiRFrame);
SiRLM = lm(Resistance ~ Instrument, data=SiRFrame);
```

In [192]:
```
%%R
#obtaining precise numbers in R
```

```
print(summary(SiRANOVA), digits =17);
print(sqrt(deviance(SiRANOVA)/df.residual(SiRANOVA)), digits = 17);
print(summary(SiRLM), digits =17);
```

In [33]:
```
SiRRresstd = '0.10407606833466272189'
SiRRRsq = '0.19099903905112250446'
SiRRModSS = '0.051146261600009186588'
SiRRModMSE = '0.012786565400002294912'
SiRRModSSResid = '0.216636560000027678097'
SiRRModMSEResid = '0.010831828000001384252'
SiRRFstat = '1.1804600000000000648'

SiRRVals = np.array([SiRRresstd, SiRRRsq, SiRRModSS, SiRRModMSE,
                     SiRRModSSResid,SiRRModMSEResid,SiRRFstat])
```

7.3.5 ANOVA values in SAS

In [ ]:
```
OPTIONS NODATE NONUMBER CENTER LS=160;
*Removes the header information and centers output;
OPTIONS FORMDLIM="~";

data SiRData;
infile "C:/Users/flunk_000/Desktop/CalPoly/IPythonNotebook/SeniorProject/d
ata/SiRFrameSAS.txt" DLM=',';
input Instrument $ Resistance;
proc print data=SiRData;
run;

proc glm data= SiRData;
        class Instrument;
        model Resistance = Instrument;
                ODS output Overallanova = SiRanova;
                ODS output fitstatistics = SiRfs;
run;

proc print data=SiRanova;
        format SS 17.16
                MS 17.16
                fvalue 17.16;
        run;

proc print data=SiRfs;
        format rsquare 17.16
                rootmse 17.16;
        run;
```

In [34]:
```
SiRSASresstd = '0.1040760683346600'
SiRSASRsq = '0.1909990390511160'
SiRSASModSS = '0.0511462615999996'
SiRSASModMSE = '0.0127865653999999'
SiRSASModSSResid = '0.2166365600000160'
SiRSASModMSEResid = '0.0108318280000008'
SiRSASFstat = '1.180462374402440'

SiRSASVals = np.array([SiRSASresstd,SiRSASRsq,SiRSASModSS,SiRSASModMSE
```

```
                         ,SiRSASModSSResid,SiRSASModMSEResid,SiRSASFstat])
```

7.3.6 Testing Python's Precision against NIST, R, and SAS

For a detailed explanation of this process, refer to Section 7.2.6.

Like 7.2.6, we need a label array to print out the values of interest:

In [96]:
```
SiRLabels = np.array(['resstd:','R-sq:','Model SS:','Model MS:',
                      'Model SSResid:','Model MSResid:','F-stat:'])
```

In [97]:
```
SiRR = array_compare(SiRRVals,SiRCertVals,SiRLabels)
```

```
('resstd:', 15, 'of', 17)
('R-sq:', 16, 'of', 17)
('Model SS:', 16, 'of', 18)
('Model MS:', 16, 'of', 18)
('Model SSResid:', 15, 'of', 17)
('Model MSResid:', 16, 'of', 18)
('F-stat:', 7, 'of', 16)
```

In [98]:
```
SiRSAS = array_compare(SiRSASVals,SiRCertVals,SiRLabels)
```

```
('resstd:', 15, 'of', 17)
('R-sq:', 15, 'of', 17)
('Model SS:', 11, 'of', 18)
('Model MS:', 11, 'of', 18)
('Model SSResid:', 15, 'of', 17)
('Model MSResid:', 17, 'of', 18)
('F-stat:', 14, 'of', 16)
```

In [99]:
```
SiRPy = array_compare(SiRPyVals,SiRCertVals,SiRLabels)
```

```
('resstd:', 15, 'of', 17)
('R-sq:', 15, 'of', 17)
('Model SS:', 11, 'of', 18)
('Model MS:', 11, 'of', 18)
('Model SSResid:', 15, 'of', 17)
('Model MSResid:', 17, 'of', 18)
('F-stat:', 14, 'of', 16)
```

In [100]:
```
SiRR, SiRSAS, SiRPy, cert_val_lengths(SiRCertVals)
```

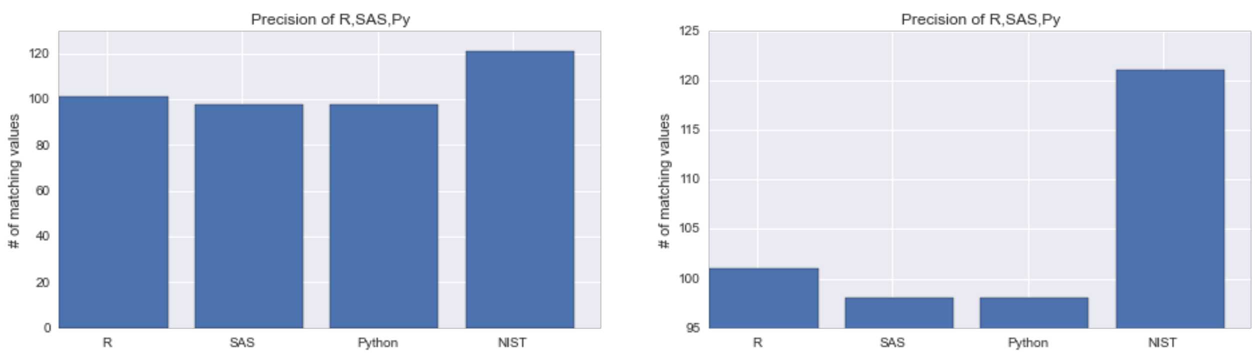
Out[100]:
```
(array([15, 16, 16, 16, 15, 16,  7]),
 array([15, 15, 11, 11, 15, 17, 14]),
 array([15, 15, 11, 11, 15, 17, 14]),
 array([17, 17, 18, 18, 17, 18, 16]))
```

With the exception of rounding the F-statistics early, R appears to have handled this dataset well. Going back and examining the array values, you can see R has the trailing zeros that match the ceritified values while Python and SAS did not. Interestingly enough, extracting large enough R values added additional digits after 4 or 5 trailing zeros. Did R caluculate those digits or were they randomly produced since we forced R to provide the extract

precision?

In [101]:
```
#open a figure and add the axes
SiRhist = plt.figure(figsize=(16,4))
gs=GridSpec(1,2)
SiRaxis = SiRhist.add_subplot(gs[0,0])

SiRprograms=4
#create array of bar values
SiRMatches = [np.sum(SiRR),np.sum(SiRSAS),np.sum(SiRPy),
              np.sum(cert_val_lengths(SiRCertVals))]

#location of bars on plot
SiRloc = np.arange(SiRprograms)

SiRbars = SiRaxis.bar(SiRloc, SiRMatches)

SiRaxis.set_ylim(0,130)
SiRaxis.set_ylabel('# of matching values')
SiRaxis.set_title('Precision of R,SAS,Py')
SiRaxis.set_xticks(SiRloc+.35)
SiRXNames = SiRaxis.set_xticklabels(['R', 'SAS', 'Python','NIST'])

SiRaxis2 = SiRhist.add_subplot(gs[0,1])
SiRbars2 = SiRaxis2.bar(SiRloc, SiRMatches)
SiRaxis2.set_ylim(95,125)
SiRaxis2.set_ylabel('# of matching values')
SiRaxis2.set_title('Precision of R,SAS,Py')
SiRaxis2.set_xticks(SiRloc+.35)
SiRXNames = SiRaxis2.set_xticklabels(['R', 'SAS', 'Python','NIST'])
SiRMatches
```

Out[101]:
```
[101, 98, 98, 121]
```

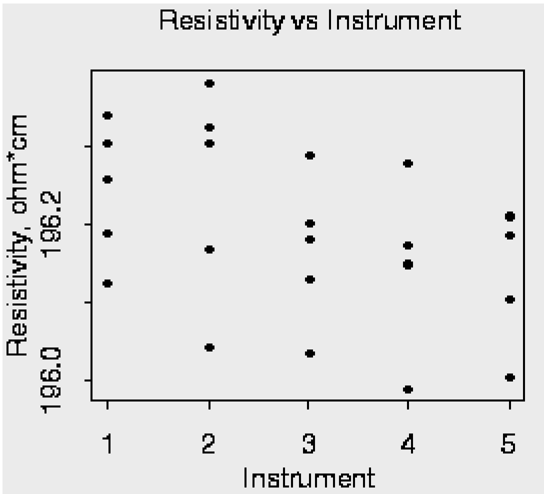


Over the 7 values of concern, R was more precise than SAS and Python by 2 decimal places. The programs appear to have about the same precision overall, but this is due to big mismatches on some values. R rounded the F-stat early (7 decimal places before SAS and Python), yet R was 5 decimal places more precise than SAS and Python for calculating Model SS and Model MSE.

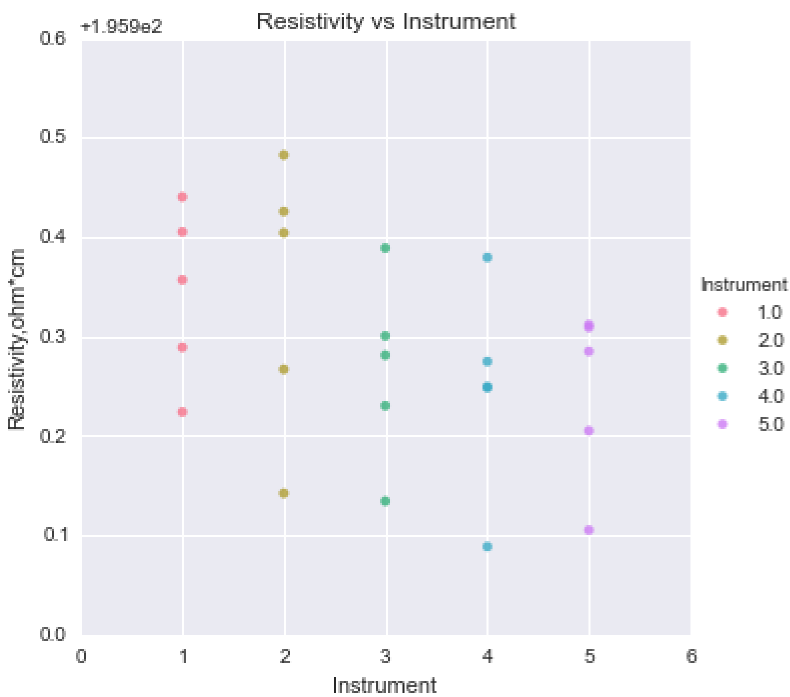
7.3.7 Plotting SiR data with Seaborn

The NIST graphics aren't as detailed as I would like, but they give us something to compare the Python graphics to. This time we will Seaborn to try to match the NIST plots.

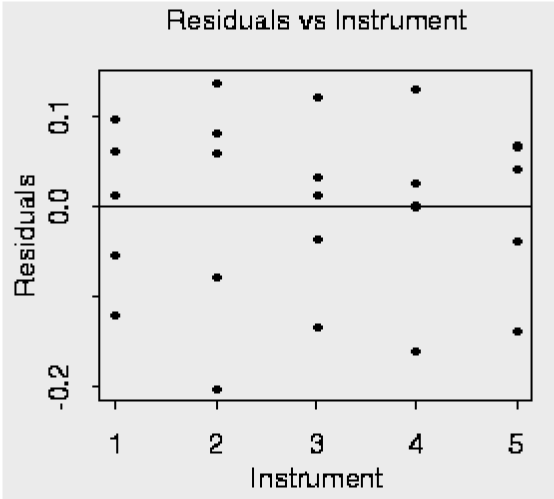
SiR ANOVA:



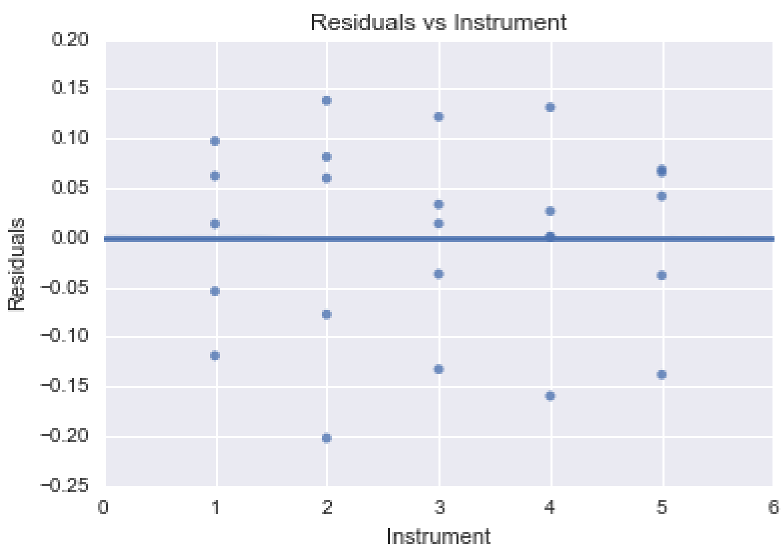
In [102]:
```python
#plot with seaborns linear model plot
#use the column names are the first and second argument
#and the data frame as the thid argument
SiRANOVA = sns.lmplot('Instrument','Resistance',data=SiRFrame,
           ci=False, hue='Instrument',fit_reg=False)
SiRANOVA.set(title ='Resistivity vs Instrument')
SiRANOVA.set(ylabel = 'Resistivity,ohm*cm');
```



SiR Residuals vs. Instrument:



In [101]:
```
SiRPlot = sns.regplot(SiRFrame['Instrument'],SiRLM.resid, ci=False)
SiRPlot.set(ylabel = 'Residuals')
SiRPlot.set(title='Residuals vs Instrument');
```



In sections 7.2 and 7.3 we have determined Python can perform as well as SAS and R on these NIST data sets.

7.4 Univariate Summary Statistics: PiDigits

7.4.1 Reading in and preparing data from an ASCII webpage

For a detailed explanation of this process, refer to Secton 7.2.1.

In [103]:
```
PiUrl = 'http://www.itl.nist.gov/div898/strd/univ/data/PiDigits.dat'
open('data/Pi.dat','wb').write(ul.urlopen(PiUrl).read())
```

In [104]:
```
PiData = np.loadtxt('data/Pi.dat',skiprows=60,dtype=int)
PiFrame = pd.DataFrame(PiData)
```

In [105]:
```
PiFrame.head()
```

Out[105]:

	0
0	3
1	1
2	4
3	1
4	5

In [106]:
```
PiFrame.tail()
```

Out[106]:

	0
4995	6
4996	0
4997	4
4998	7
4999	2

Data looks good. I think we can all agree that Pi starts with 31415...

7.4.2 NIST Certified Values

For a detailed explanation of this process, refer to Section 7.2.2.

The Certified values we are trying to match can be found at: http://www.itl.nist.gov/div898/strd/univ/certvalues/pidigits.html

In [98]:
```
PiMean = '4.53480000000000'
PiSigma = '2.86733906028871'

PiCertVals = np.array([PiMean,PiSigma])
```

7.4.3 Univariate Summary Statistics in Python

NIST only provides the mean and standard deviation for comparison. I'm going to show you a couple things that might be helpful as well as extract the values for comparison.

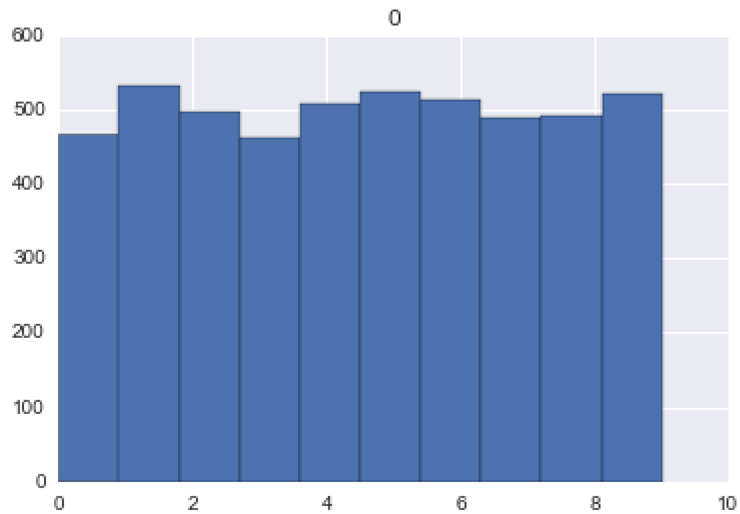
In [99]:
```
#dont forget to check dir(PiFrame) to see what a DataFrame can offer
PiFrame.describe()
```

Out[99]:

	0
count	5000.000000
mean	4.534800
std	2.867339
min	0.000000

25%	2.000000
50%	5.000000
75%	7.000000
max	9.000000

In [100]:
```
PiFrame.hist();
```



Back to the desired values. As you can see, the DataFrame has the values we want already built in. Since we desire a more precise value for this exercise, the simplest way is to use NumPy.

In [399]:
```
PyPiMean = repr(np.mean(PiData))
PyPistd = repr(np.std(PiData))

PyPiVals = np.array([PyPiMean, PyPistd])
```

7.4.4 Testing Python's Precision against NIST values

For a detailed explanation of this process, refer to Section 7.2.6.

In [394]:
```
#specify titles for the output values
PiLabelArray = np.array(['Mean:','Sigma:'])
```

In [395]:
```
PiPy = array_compare(PyPiVals,PiCertVals,PiLabelArray)
```

```
('Mean:', 5, 'of', 16)
('Sigma:', 5, 'of', 16)
```

That's unforunate... Let's look at the values and see if we feel our previous comparison method might be a bit misleading in this instance.

In [401]:
```
print(PyPiVals[0])
print(PiCertVals[0])
```

```
4.5347999999999997
```

```
4.53480000000000
```

These means are essentially the same. Not nearly as bad as matching 5 of 16 digits would lead us to believe.

In [403]:
```
print(PyPiVals[1])
print(PiCertVals[1])
```

```
2.8670523120445486
2.86733906028871
```

The NumPy standard deviation is way off. Lucky for us, we still have a Pandas DataFrame all set to go. Here's a work around:

In [416]:
```
PyMean = PiFrame.mean()
print '%17.14f' % (PyMean)
print(PiCertVals[0])

PySTD = PiFrame.std()
print '%17.14f' % (PySTD)
print(PiCertVals[1])
```

```
 4.53480000000000
4.53480000000000
 2.86733906028871
2.86733906028871
```

Success! This method might be a few more lines of code, but it appears to be a more precise approach.

Chapter 8
Streaming Data in the IPython Notebook

An interest in working with Dynamic data is what brought Dr. Doi and I together on this project. Dynamic data is data that is always changing. Some data sets might change only a few times in a 5 minute interval, others might change 100 times a second. The fascinating thing about dynamic data is any time it is analyzed, the analysis is potentionally behind the data. Consider LADWP's (Los Angeles Department of Water and Power) water data. In 2013, the United States Census Bureau estimated the population of the city of Los Angeles was about 3.9 millon people. With 3.9 million people, it seems impossible that the water consumption in Los Angeles could ever stop. This implys that as we analyze water consumption in LA, we are immediately missing new data. In the future, I intend to use dynamic environmental data streams to provide people with accurate, analyzed in real time, information that enables them to make choices that best support sustainability in their area.

8.1 Python Packages for Streaming and the Import Cell

In [4]:
```
import requests
import json
import pandas as pd
from mpl_toolkits.basemap import Basemap
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

8.2 Streaming Data from USA.gov

USA.gov describes the data as, "We provide a raw pub/sub feed of data created any time anyone clicks on a 1.USA.gov URL. The pub/sub endpoint responds to http requests for any 1.USA.gov URL and returns a stream of JSON entries, one per line, that represent real-time clicks."

A few years ago USA.gov "held a nationwide 1.USA.gov Hack Day... to encourage people to explore the 1.USA.gov data." The projects and code resulting from this are more sophicated than what we're doing here and can be found at: http://www.usa.gov/About/developer-resources/1usagov.shtml

In [5]:
```
url = "http://developer.usa.gov/1usagov"
```

In [87]:
```
#url argument is the live datastream
r = requests.get(url, stream=True)

#after grabbing n data values, the datastream stops
n = 500
data = []

#looks at each line of the request individually and adds it to the list "d
ata"
for i, line in enumerate(r.iter_lines()):
    data.append(line)
```

```
    #this is a dirty little trick that should be avoided in larger functio
ns and
    #programs, but works great for this quick line fectching function
    if i > n:
        break
```

In [88]:
```
#load the json lines from the list
jdata = [json.loads(item) for item in data[1:]]
```

In [89]:
```
#create a DataFrame
USAGovFrame = pd.DataFrame(jdata)
```

We built a DataFrame from USAGov's live stream. Let's take a peek at it and see what we ended up with.

In [90]:
```
USAGovFrame.head()
```

Out[90]:

	heartbeat	_id	a	al	c	ckw	cy	dp	g	gr
0	NaN	543ae232-002b9-0416e-cf1cf10a	Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKi...	es-419,es;q=0.8	MX	NaN	NaN	NaN	15r91	N
1	NaN	543ae232-003ac-038db-301cf10a	Mozilla/5.0 (iPhone; CPU iPhone OS 7_1_2 like ...	en-us	US	NaN	Ponder	NaN	1v02m1T	T.
2	NaN	543ae233-00395-07c0e-361cf10a	Mozilla/5.0 (iPhone; CPU iPhone OS 8_0_2 like ...	fr-fr	FR	NaN	NaN	NaN	ZzdKp8	N
3	NaN	543ae234-00165-07334-261cf10a	Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKi...	en-US,en;q=0.8	US	NaN	Spirit Lake	NaN	1qfk3VH	IA
4	NaN	543ae234-00364-06587-2a1cf10a	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_5)...	es-es	ES	NaN	Vigo	NaN	K6Cor	5:

5 rows × 21 columns

After looking at the descriptions of the variables provided from 1USA.gov, I don't know what I would do with several of them. Let's reduce the DataFrame to variables we are interested in playing with.

In [91]:
```
#drop all the columns that I don't know anything about
```

```
#ckw and dp aren't consistently collected, especially with smaller sample sizes
#this will check for them and drop them if they are present
if 'ckw' and 'dp' in USAGovFrame.columns:
    USAGovFrame.drop(['_heartbeat_','_id','al','ckw','nk','g','h','kw','hc','dp'],inplace=True, axis=1)
else:
    USAGovFrame.drop(['_heartbeat_','_id','al','nk','g','h','kw','hc'],inplace=True, axis=1)
```

In [92]:
```
#add user friendly names for the remaining variables
USAGovFrame.columns=['User_Agent','Country_Code','Geo_city_name',
                     'Geo_Region','Short_url_Cname','Encoding_user_login',
                     '[Latitude,Longitude]','Referring_URL','Timestamp',
                     'Timezone','Long_URL']
```

In [93]:
```
#look at the new DataFrame
USAGovFrame.head()
```

Out[93]:

	User_Agent	Country_Code	Geo_city_name	Geo_Region	Short_url_Cname	Encoding_
0	Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKi...	MX	NaN	NaN	j.mp	pontifier
1	Mozilla/5.0 (iPhone; CPU iPhone OS 7_1_2 like ...	US	Ponder	TX	ift.tt	ifttt
2	Mozilla/5.0 (iPhone; CPU iPhone OS 8_0_2 like ...	FR	NaN	NaN	1.usa.gov	tweetdecl
3	Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKi...	US	Spirit Lake	IA	1.usa.gov	theusnav
4	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_5)...	ES	Vigo	58	1.usa.gov	anonymo

8.3 Plotting on a world map

With the DataFrame we have, let's plot all the coordinates on a world map to get an idea of who just visited .gov websites when we excuted the code above.

First, we need to learn about the coordinates in the Latitude/Longitude column.

In [101]:
```python
#check the type of data at the first index
type(USAGovFrame['[Latitude,Longitude]'][0])
```

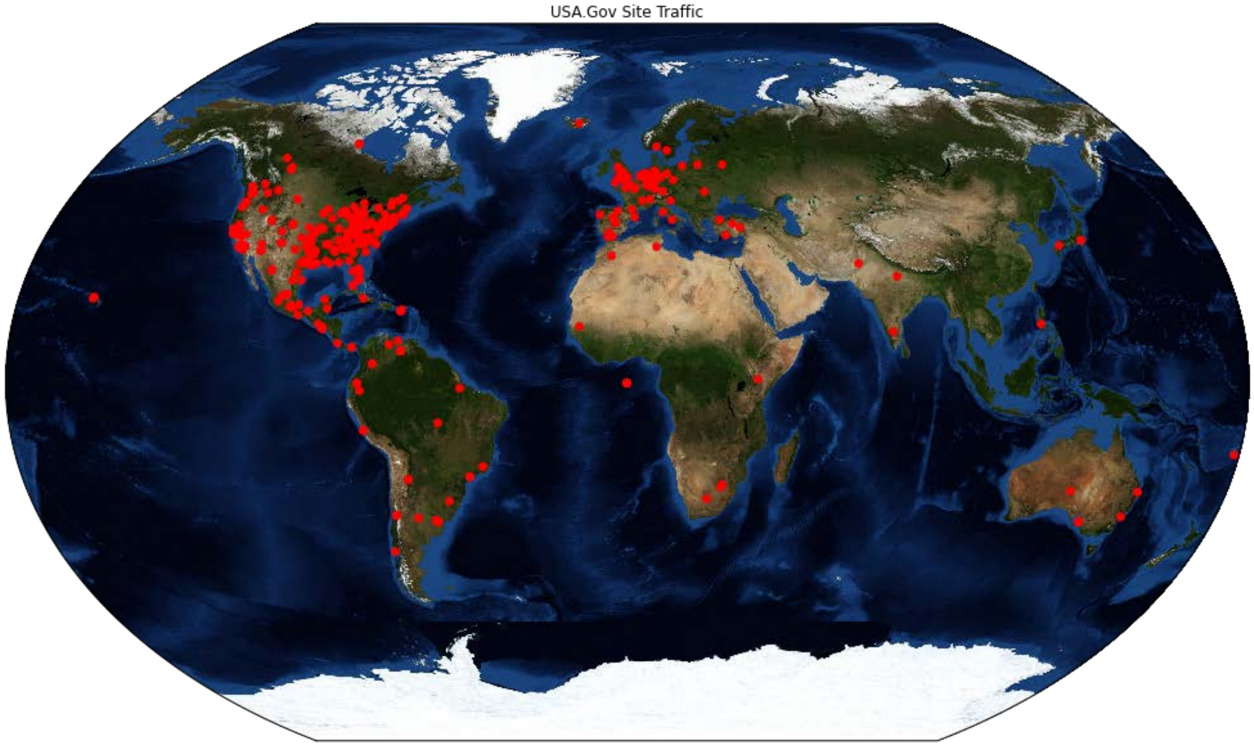
Out[101]: list

I tried to zip and unpack this data the short way, but I received a "too many values" error. As you might have figured out by now, it's time for another function to help us with this process.

The goal is to pass the column of the DataFrame to a function that will separate the latitude and the longitude into their own variables in order to plot them as x and y variables on a world map.

In [94]:
```python
#function for unpacking a list in each row of a column into two separate l
ists
def lat_lon(column):

    #create two empty lists
    lat=[]
    lon=[]

    #at each row in the column, add first index to lat, second to lon
    for i in column:
        lat.append(i[0])
        lon.append(i[1])

    #return the new lists
    return lat,lon
```

In [97]:
```python
#open a larger figure so we have a better since of the global web traffic
plt.figure(figsize=(20,10))

# lon_0 is central longitude of projection.
# resolution = 'c' means use crude resolution coastlines.
map = Basemap(projection='kav7',lon_0=0,resolution='c')

#bluemarble is a built in function to basemap
map.bluemarble()

#drop the NaN values from the data frame
coords = USAGovFrame['[Latitude,Longitude]'].dropna()

#run the function from the cell above
lat, lon = lat_lon(coords)

#plot the points on the bluemarble basemap
# '.' is the marker type, c is the color
x,y = map(lon, lat)
map.plot(x,y,'.', c='red',markersize=12)
plt.title("USA.Gov Site Traffic");
```


USA.Gov Site Traffic

Chapter 9
Time Series, More with Data Frames, and Advanced Plotting in the IPython Notebook

9.1 Pendulum Data

In [103]:
```
import numpy as np
import pandas as pd
import math
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
import pylab as pl
```

With a pendulum and a web camera, Dr. Hughes used the Matlab routine pendulum_data.m to obtain the pendulum dataset we will be working with. Additionally, the matlab routine also provided us with the center of the least-squares best fit circle for the data (1152.57606607623, 394.773399557239), which we will need in the following section.

9.1.1 Converting x and y values to angular position (Θ)

In [104]:
```
#read the csv file
PendData = np.loadtxt('data/pend_data.csv', skiprows=1, delimiter=',')

#convert the numpy structure into a pandas DataFrame
PendFrame = pd.DataFrame(PendData,columns=['time(sec)','x(pixels)','y(pix
els)'])
PendFrame.head()
```

Out[104]:

	time(sec)	x(pixels)	y(pixels)
0	0.00000	347.397	11.801
1	0.15452	722.760	60.761
2	0.26375	748.770	66.665
3	0.32808	598.070	28.983
4	0.40682	384.457	11.128

Using the provided data set and center of the circle, let's make some changes to our data frame.

First, we need to convert x and y to Θ to obtain a time series in this format:

$$\{(t_i,\theta_i):i=0,\ldots,n\}$$

Where:

$$\theta_i = \frac{180}{\pi} \cdot atan2\left(\frac{y_i - 1152.57606607263}{x_i - 394.773399557239}\right) + 90$$

Let's write a fucntion that uses this equation to create a new column in the data frame that contains the computed Θ values.

In [105]:
```
#function takes y and x, then returns the solution to the above equation
def calc_theta(y,x):
    return ((180/math.pi)*math.atan2((y-1152.57606607263),(x-394.77339955
7239)))+90

#the left side creates a new column in the data frame names "theta"
#the right side uses information from the existing columns and the theta
function above
#to create the "theta" column
PendFrame['theta']= PendFrame.apply(lambda row: calc_theta(row['y(pixels)
'],row['x(pixels)']), axis=1)
PendFrame.head()
```

Out[105]:

	time(sec)	x(pixels)	y(pixels)	theta
0	0.00000	347.397	11.801	-2.378128
1	0.15452	722.760	60.761	16.720526
2	0.26375	748.770	66.665	18.055472
3	0.32808	598.070	28.983	10.255820
4	0.40682	384.457	11.128	-0.517825

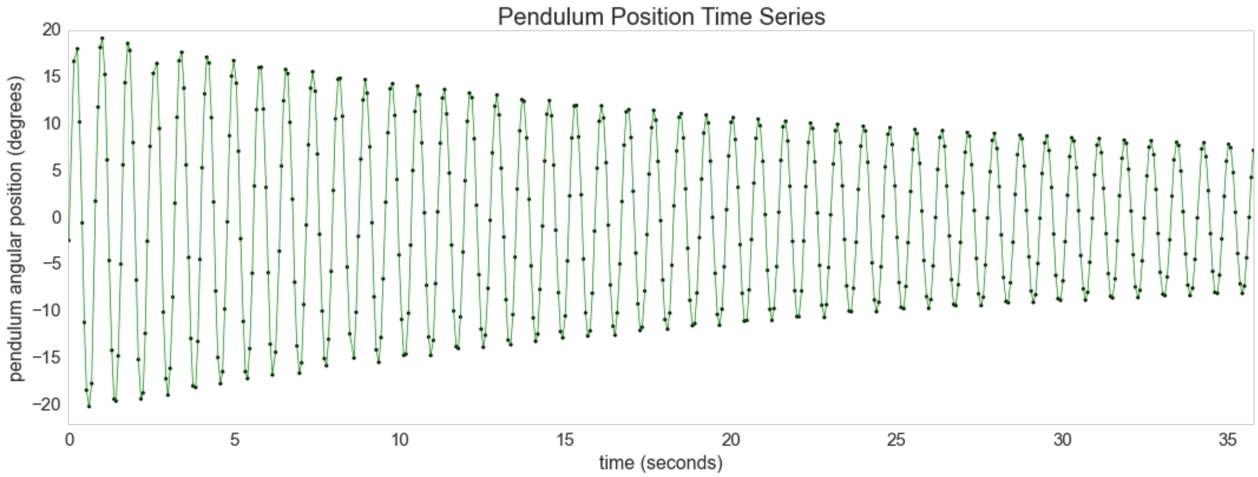
9.1.2 Graphing the Time series: Angular position vs. Time

Now that the data frame has a time column and a Θ column, let's graph the time series:

$$\{(t_i, \theta_i) : i = 0, \ldots, n\}$$

In [106]:
```
#open a figure window
plt.figure(figsize=(18,6))

#plot the scatterplot first to keep the markers in the foreground
#s is the size of the markers, and black is the color of the markers
plt.scatter(PendFrame['time(sec)'],PendFrame['theta'], s=8, c='black')

#built-in DataFrame function to plot time as x and theta as y, with custo
m y limits, line width, and color
TSPlot = PendFrame.plot(x='time(sec)',y='theta',ylim=(-22,20),linewidth=.
7, c='green')

#change background of the plot to white
TSPlot.set_axis_bgcolor('white')

#set x and y labels, title, and adjust thier sizes according
```

```
TSPlot.set_ylabel('pendulum angular position (degrees)', fontsize=16)
TSPlot.set_xlabel('time (seconds)',fontsize=16)
TSPlot.set_title('Pendulum Position Time Series',fontsize=20)

#increase the font size of the x and y ticks
plt.tick_params(axis='both', labelsize=15)
```



9.1.3 Summary Statistics

Now that we have a visualization of our dataset, let's look at some other useful information.

In [107]:
```
PendFrame.describe()
```

Out[107]:

	time(sec)	x(pixels)	y(pixels)	theta
count	512.000000	512.000000	512.000000	512.000000
mean	18.088852	389.999988	24.999988	-0.243205
std	10.262232	184.034433	14.569337	9.307719
min	0.000000	2.140000	9.884000	-20.105156
25%	9.346825	235.100000	13.858000	-8.029286
50%	18.119500	394.293500	20.843000	-0.024068
75%	26.951250	551.912500	30.556750	7.897347
max	35.785000	770.170000	79.954000	19.179884

From this table we can see that our dataset took measurement of the pendulums position for a total of 35.785 seconds (since the timespan is equal to the max-min of the time column). What we do not know is whether or not the data points are evenly spaced in time. We can look at the graph above and note there are about 6-7 peaks per 5 second interval. We can also look at the spacing between the quantiles, the first 25% of the data values taking place in a 9.346825 second span, with the following 25% taking place in an 8.772675 second span (Q2-Q1), the following 25% taking place in an 8.83175 second span (Q3-Q2), and the final 25% taking place in an 8.83375 second span. Overall, it looks pretty close.

We can apply a smoother to obtain equally spaced t_i's, but let's write a function to assess the time spacing of the data.

In [108]:
```
#this function will create an array with the difference between
#each point so we can examine if the time intervals are consistent
#throughout the data, returns a pandas series of differences
def spacing_check(vector):

    #create a series with one less value than the vector (since the first
 entry
    #is 2nd-1st of the vector) to store the difference values
    diff = np.zeros(len(vector)-1,dtype=float)
    difference = pd.Series(diff)

    for time in range(len(vector)-1):
        difference[time]=vector[time+1]-vector[time]
    return difference
```

Let's create a series of differences using our function.

In [109]:
```
TimeSpans = spacing_check(PendFrame['time(sec)'])
```

There are a couple ways to test that our function is working properly.

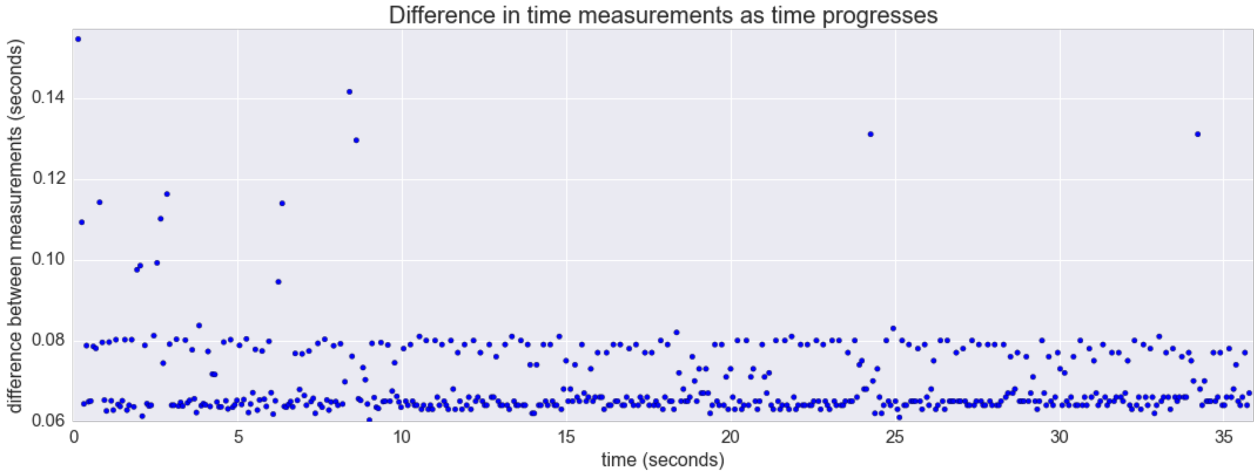
In [110]:
```
#this should be equal to 511, since we start by subtracting the second va
lue from the first
print(len(TimeSpans))==511

#subtracting the first value of our time column from the second value sho
uld equal the first value of our difference array
(PendFrame['time(sec)'][1]-PendFrame['time(sec)'][0])==TimeSpans[0]
```

```
True
```

Out[110]:
```
True
```

Everything appears to be in order. Let's look at a plot of the difference to see if there is any increasing or decreasing trend in the time spacing of the data points.

In [111]:
```
#the first argument is a slice that takes the last 511 time
#values so we can compare them to our 511 differences
#to see if there is a change over time
fig = plt.figure(figsize=(18,6))
TimeSpacingPlot = fig.add_subplot(1,1,1)
plt.scatter(x=PendFrame.ix[1:511,'time(sec)'],y=TimeSpans)

#set x and y labels, title, and adjust thier sizes according
TimeSpacingPlot.set_ylabel('difference between measurements (seconds)', f
ontsize=16)
TimeSpacingPlot.set_xlabel('time (seconds)',fontsize=16)
TimeSpacingPlot.set_title('Difference in time measurements as time progre
sses',fontsize=20)
TimeSpacingPlot.set_ylim(.06,.157)
TimeSpacingPlot.set_xlim(0,35.9)
plt.tick_params(axis='both', labelsize=15)
```



9.1.4 Modeling a Sine Wave

In section 9.1.2, we graphed the pendulum data and saw a damped sine wave in the output. To build a model for this, we need a sinusoid with an exponential decay term.

$$\hat{\theta}(t) = [A \cdot \sin(2\pi \cdot f \cdot t\ -\ \varphi)\,] \cdot e^{-\lambda \cdot t} +\ B$$

Where:
t = Time, in seconds
A = Peak Amplitude
f = Frequency, in Hz
Φ = Phase Shift
B = Vertical shift

Now, we need to write a python function. This is one of the easier functions we have worked with. Simply translate the above function to code using the math package where necessary (such as math.pi).

In [114]:
```
#function for sinwave
def sin_wave(time,amp,freq,phi,vertShift,damp):
    return ((math.exp(-1*damp*time))*(amp*math.sin(2*math.pi*freq*time-ph
i)+vertShift))
```

With this function, we can change the individual values of our Parameter estimates to find the "best" estimate by minimizing SSE. This is measured by the SSE printed below the next box of code.

In [115]:
```
#changing these values and excuting this code after each change
#will alter the value.
amp = 19.649
freq = 1.26082
phi = .18205
vertShift = -.4705
damp = .02695

#creates a new column in the DataFrame and populates it with sine wave va
lues
PendFrame['SineValues']= PendFrame.apply(lambda row: sin_wave(row['time(s
```

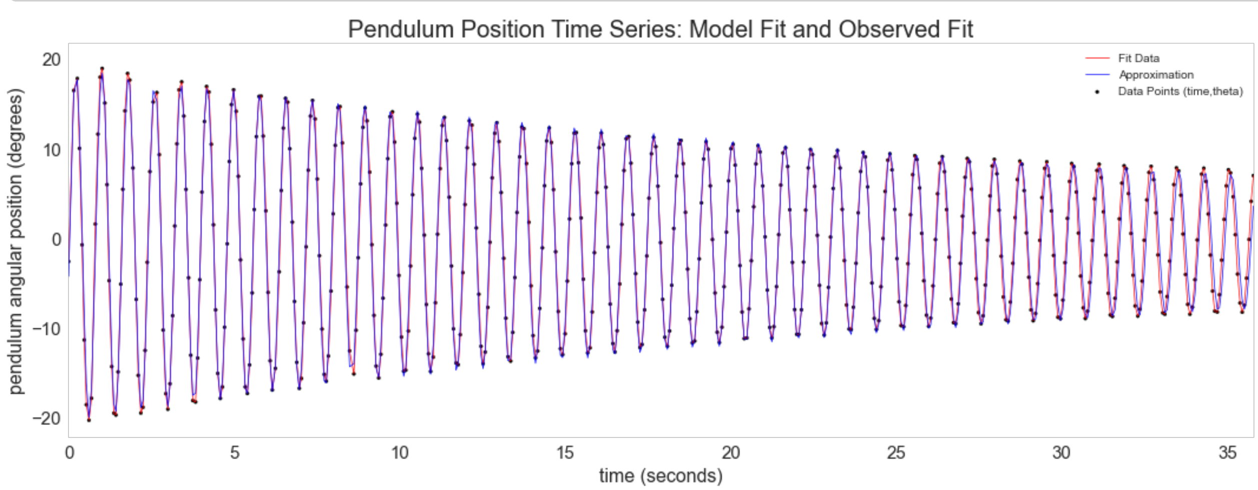
```
ec)'],amp,freq,phi,vertShift,damp), axis=1)

#check sum of squares to see if error is decreasing
SSE = np.sum((PendFrame['theta']-PendFrame['SineValues'])**2)
print("SSE:",SSE)
```

```
('SSE:', 983.71108227071329)
```

As you can see above, my estimated Parameters are fairly precise. Playing around with them might yield an even lower SSE, but it shouldn't be much different from my printed value.

Let's plot the model we just made against the graph from Section 9.1.2 and see how they compare.

In [119]:
```
#open a figure window and color it white
plt.figure(figsize=(18,6))

#original points
plt.scatter(PendFrame['time(sec)'],PendFrame['theta'], s=8, c='black')

#original line
TSPlot = PendFrame.plot(x='time(sec)',y='theta',ylim=(-22,20),linewidth=.
7, c='red')

#my model line
PendFrame.plot(x='time(sec)',y='SineValues',ylim=(-22,22),linewidth=.7, c
='blue')

#change background to white
TSPlot.set_axis_bgcolor('white')

#set x and y labels, title, and adjust thier sizes according
TSPlot.set_ylabel('pendulum angular position (degrees)', fontsize=16)
TSPlot.set_xlabel('time (seconds)',fontsize=16)
TSPlot.set_title('Pendulum Position Time Series: Model Fit and Observed F
it',fontsize=20)
TSPlot.legend(['Fit Data','Approximation','Data Points (time,theta)'],1)
plt.tick_params(axis='both', labelsize=15)
```

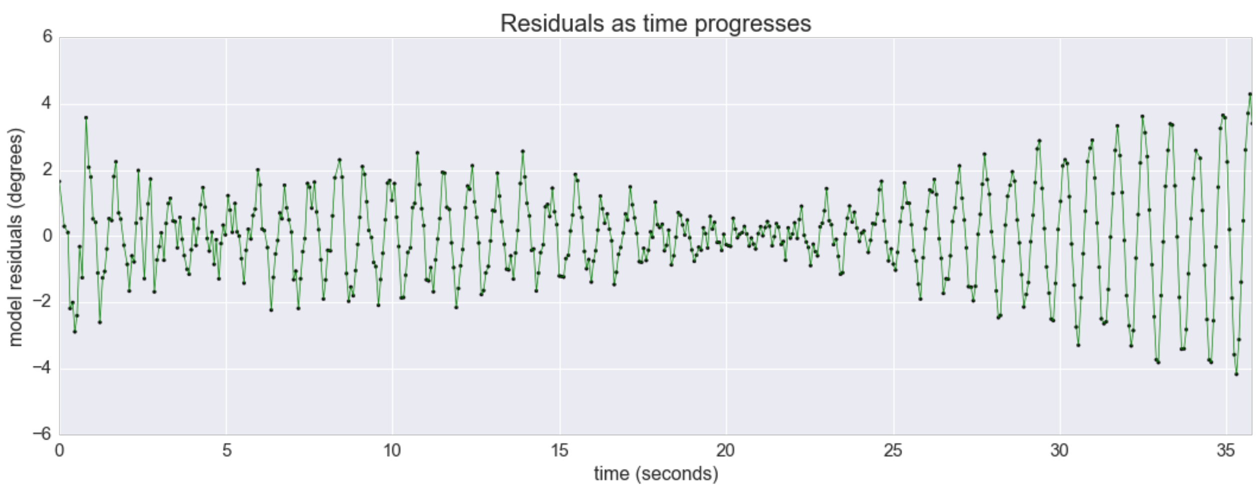


Not too shabby. They appear to match the least toward the end of the time period.

We can look at the residuals to see where the model matches and fails to match the pendulum data.

In [121]:
```
#residuals are the observed values at ti - the values from model at ti
def calc_resid(obs,pred):
    return obs-pred

#creates a new column in the DataFrame and populates it with residuals
PendFrame['Residuals']= PendFrame.apply(lambda row: calc_resid(row['theta
'],row['SineValues']), axis=1)
```

In [122]:
```
#open a figure window and color it white
plt.figure(figsize=(18,6))

#plot the scatterplot first to keep the markers in the foreground
plt.scatter(PendFrame['time(sec)'],PendFrame['Residuals'], s=8, c='black'
)

#built-in DataFrame function to plot time as x and theta as y, with custo
m y limits, line width, and color
TSPlot = PendFrame.plot(x='time(sec)',y='Residuals',linewidth=.7, c='gree
n')

#set x and y labels, title, and adjust thier sizes according
TSPlot.set_ylabel('model residuals (degrees)', fontsize=16)
TSPlot.set_xlabel('time (seconds)',fontsize=16)
TSPlot.set_title('Residuals as time progresses',fontsize=20)
plt.tick_params(axis='both', labelsize=15)
```



My visual inspection was confirmed by the residual plot. Additionally, we can see other areas where the model fails to capture the exact movement of the pendulum (such as the first 3 seconds).

9.2 Geiger Counter Data

This is another data set provided by Dr. Hughes. It is the result of putting radioactive material next to a geiger counter. The data was collected using Matlab. The variables are the number of detections in a 5 second period, for over 22 hours, and the timestamp.

In [129]:
```python
import matplotlib.dates as mdates
from datetime import datetime
```

9.2.1 Converting time and date stamps for plotting

In [130]:
```python
gcFrame = pd.read_table("data/source_radioactivity.txt",names=['Date/Time
','Detections'])

#check to make sure we're working with the "entire" data set
len(gcFrame.Detections)
```

Out[130]: 16384

In [131]:
```python
gcFrame.head()
```

Out[131]:

	Date/Time	Detections
0	09/23/2013 17:26:27	5
1	09/23/2013 17:26:32	9
2	09/23/2013 17:26:37	8
3	09/23/2013 17:26:42	7
4	09/23/2013 17:26:47	9

In [135]:
```python
#convert the timestamp to a time date2num can use
#the % and punctuation is written exactly how the information appears in
the df
gcFrame['Time'] = [datetime.strptime(t, "%m/%d/%Y %H:%M:%S") for t in gcF
rame['Date/Time']]

#the scatter column is a conversion of the time column to a number we can

#use to graph the points on the plot of the time series
gcFrame['Scatter'] = mdates.date2num(gcFrame['Time'])
```

In [136]:
```python
gcFrame.head()
```

Out[136]:

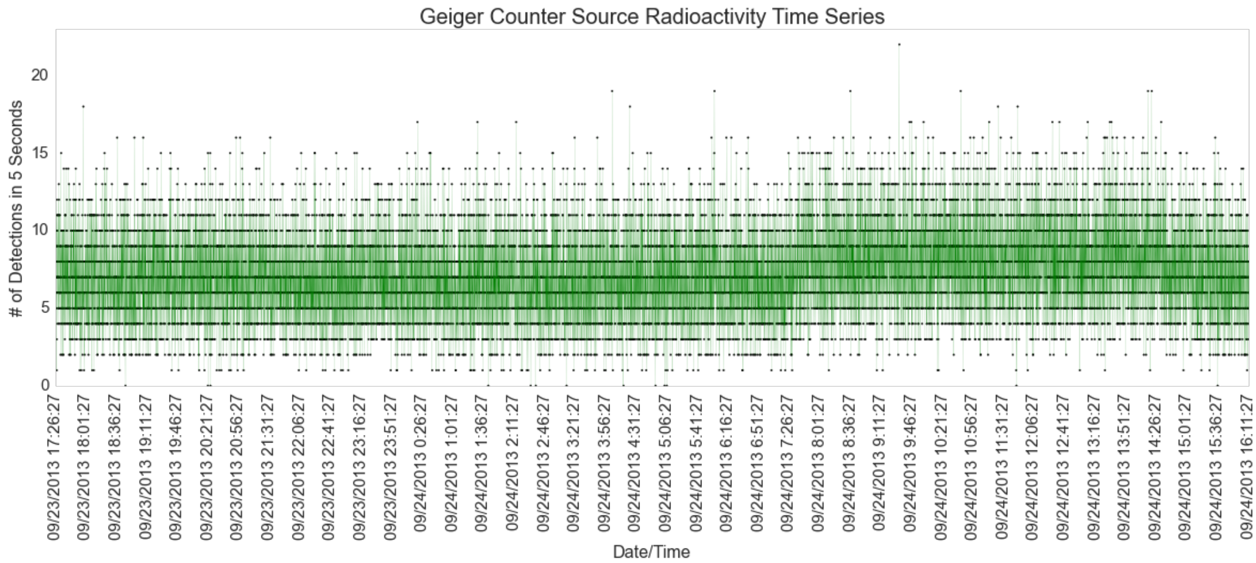
	Date/Time	Detections	Time	Scatter
0	09/23/2013 17:26:27	5	2013-09-23 17:26:27	735134.726701
1	09/23/2013 17:26:32	9	2013-09-23 17:26:32	735134.726759
2	09/23/2013 17:26:37	8	2013-09-23 17:26:37	735134.726817

3	09/23/2013 17:26:42	7	2013-09-23 17:26:42	735134.726875
4	09/23/2013 17:26:47	9	2013-09-23 17:26:47	735134.726933

9.2.2 Plotting the Geiger counter time series with points

There are many plotting options, including ones specifically for time series data. I continued to have the best luck with the Panda's DataFrame and adjusted my plotting accordingly. Here's the code for a plot of the time series with the individual 16384 points. Change the line widths, colors, and sizes in the code below to customize your own time series plot.

In [138]:
```
#open a figure window and color it white
figure = plt.figure(figsize=(20,6))

#built-in DataFrame function to plot time as x and theta as y, with custo
m y limits,
#line width, and color
TSPlotG = gcFrame.plot(x='Scatter',y='Detections',linewidth=.1, ylim=(0,2
3), c='green')

TSPlotG.set_axis_bgcolor('white')   #change background to white

#set x and y labels, title, and adjust thier sizes according
TSPlotG.set_ylabel('# of Detections in 5 Seconds', fontsize=16)
TSPlotG.set_xlabel('Date/Time',fontsize=16)
TSPlotG.set_title('Geiger Counter Source Radioactivity Time Series',fonts
ize=20)
plt.tick_params(axis='both', labelsize=15)   #change the font size of the
 axes ticks

xlabels = []
xticks = []
for i in range(40):
    #populate a list of 41 date/times with even time intervals over our 1
6834 points
    xlabels.append(gcFrame['Date/Time'][i*420])

    #an array to place the date/time labels at the corresponding x value
    xticks.append(gcFrame['Scatter'][i*420])

#place the date/time labels and rotate them
TSPlotG.set_xticklabels(xlabels, rotation=90, fontsize=15)
TSPlotG.set_xticks(xticks)

#add the points to the graph
plt.scatter(gcFrame['Scatter'],gcFrame['Detections'], s=2.5, color='black
');
```



9.2.3 Moving Averages with Geiger Counter Data

When we looked at the Pendulum data, we were worried about a constant Δt. I provided an, arguably, uncessary way to evalute the change from one t_i to the next. If the time intervals are not equally spaced, you can create your own intervals in specified n-sized windows using a smoother.

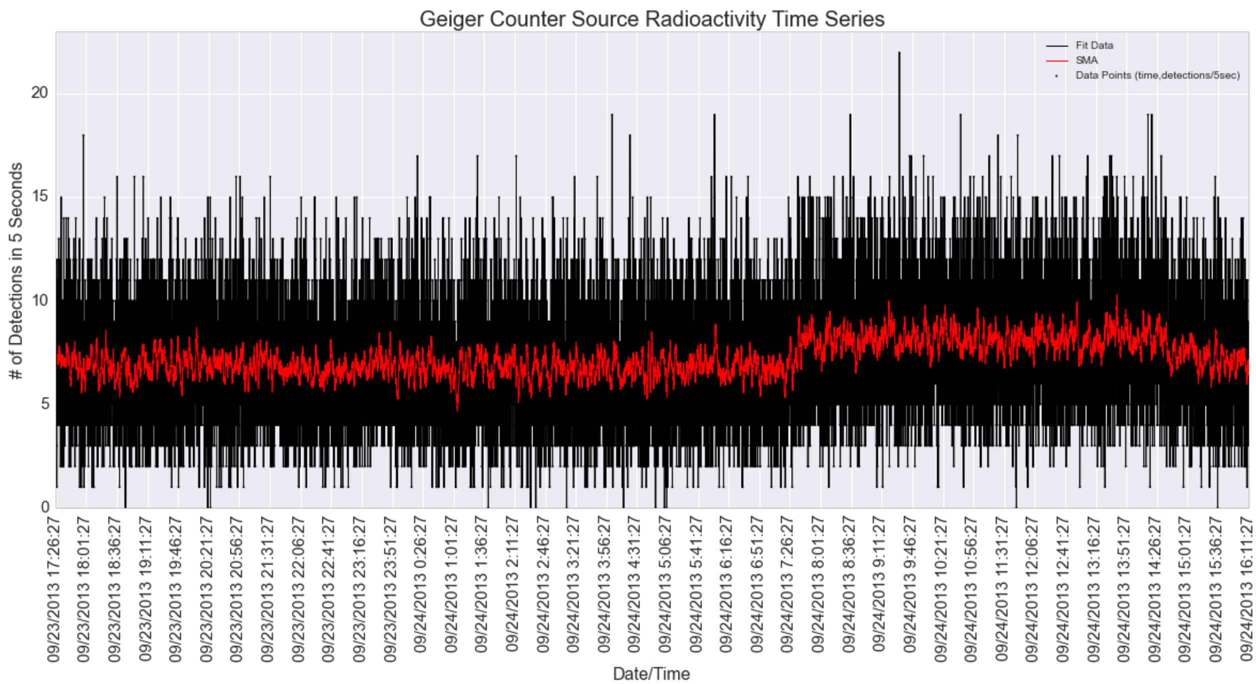
Below is the simple moving average, or as Pandas calls it the "roling_mean".

In [141]:
```
#pandas has a built-in rolling means function
gcFrame['SMA'] = pd.rolling_mean(gcFrame.Detections,window=24)
```

Plot the rolling mean over the original geiger counter data.

In [143]:
```
#open a figure window and color it white
figure = plt.figure(figsize=(20,8))

#built-in DataFrame function to plot time as x and theta as y, with custo
m y limits,
#line width, and color
TSPlotG2 = gcFrame.plot(x='Scatter',y='Detections',linewidth=1, ylim=(0,2
3), c='black')

Line2 = gcFrame.plot(x='Scatter',y='SMA',linewidth=1, c='red')

#set x and y labels, title, and adjust thier sizes according
TSPlotG2.set_ylabel('# of Detections in 5 Seconds', fontsize=16)
TSPlotG2.set_xlabel('Date/Time',fontsize=16)
TSPlotG2.set_title('Geiger Counter Source Radioactivity Time Series',font
size=20)
plt.tick_params(axis='both', labelsize=15)   #change the font size of the
 axes ticks

xlabels = []
xticks = []
for i in range(40):
    #populate a list of 41 date/times with even time intervals over our 1
```

```
6834 points
    xlabels.append(gcFrame['Date/Time'][i*420])

    #an array to place the date/time labels at the corresponding x value
    xticks.append(gcFrame['Scatter'][i*420])

#place the date/time labels and rotate them
TSPlotG2.set_xticklabels(xlabels, rotation=90, fontsize=15)
TSPlotG2.set_xticks(xticks)

#add the points to the graph
Scat = plt.scatter(gcFrame['Scatter'],gcFrame['Detections'], s=2.5, color
='black')
TSPlotG2.legend(('Fit Data','SMA','Data Points (time,detections/5sec)'),
loc=1);
```



I have heard the centered moving average is also built-in to Pandas, but it was just as fast to write my own function for it.

In [144]:
```
#function for CMA since it wasn't built into pandas
def CMA(vec, n):
    c = n/2
    cma = np.zeros([16384,1])  #created an empty array to store values
    for i in range(c, len(vec)-c):    #look at values from n/2 to 16384-n
/2
        window = vec[i-c:i+c+1]        #taking average over these values
        cma[i] = (np.mean(window))     #store average in array
    cma[:c] = None                     #fill first n/2 values with None
    cma[(len(vec)-c):] = None          #fill last n/2 values with None
    return cma
```

In [145]:
```
gcFrame['CMA'] = CMA(gcFrame.Detections,24)    #add column to the data fra
me
```

This time, let's look at only a section of the data. For this, we will slice the data from 7am to

8am on September 24th.

In [147]:
```
print(gcFrame['Date/Time'][9763],gcFrame['Date/Time'][10483])
```

```
('09/24/2013 7:00:02', '09/24/2013 8:00:02')
```

With the indicies above, i'm creating a new data frame with all the varibles I want from 7am to 8am.

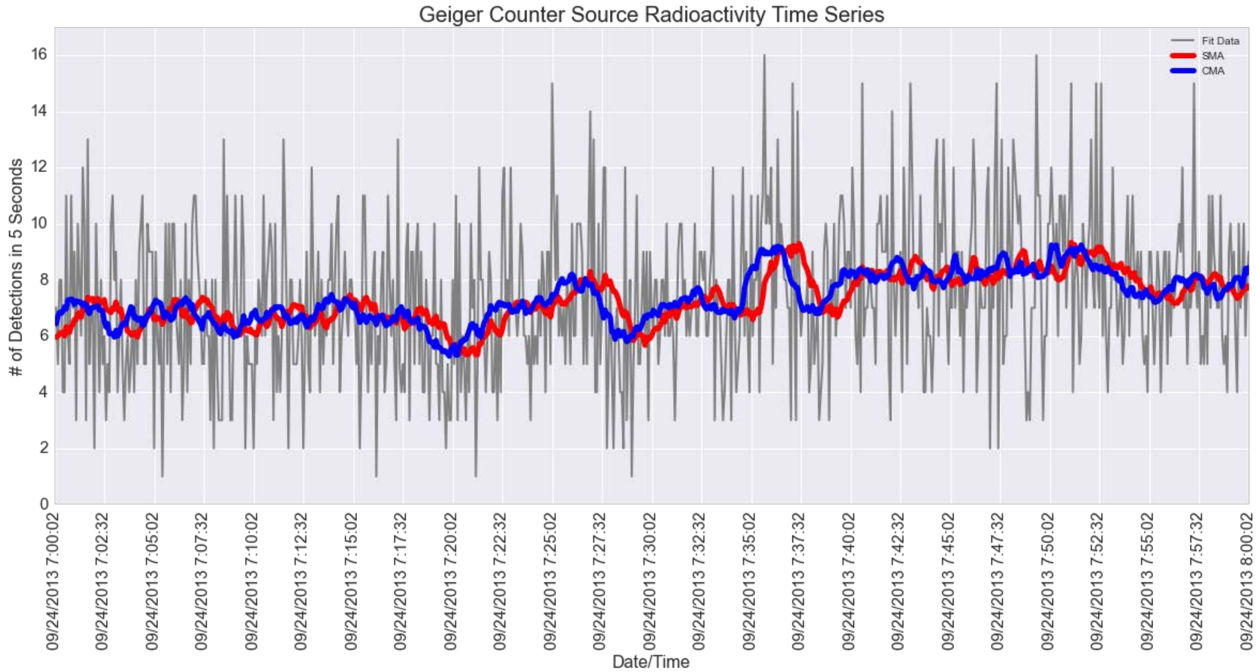
In [148]:
```
#slice the data
sliceFrame = pd.DataFrame(gcFrame.CMA[9763:10484], columns=['CMA'])
sliceFrame['SMA'] = gcFrame.SMA[9763:10484]
sliceFrame['Detections'] = gcFrame.Detections[9763:10484]
sliceFrame['Scatter'] = gcFrame.Scatter[9763:10484]
sliceFrame['Date/Time'] = gcFrame['Date/Time'][9763:10484]
len(sliceFrame.CMA)
```

Out[148]:
```
721
```

Now there are only 721 observations. The way I wrote the axis function, it will need to be updated. There are time locators that will determine the spacing automatically, but I enjoy functions. If I had to do this more, I would write a general fuction for determining the time ticks for the x axis. After adjusting the x axis fuction, we'll see how the SMA and CMA compare in the sliced time window.

In [153]:
```
#open a figure window and color it white
figure = plt.figure(figsize=(20,8))


#built-in DataFrame function to plot time as x and theta as y, with custo
m y limits,
#line width, and color
TSPlotG3 = sliceFrame.plot(x='Scatter',y='Detections',linewidth=2, ylim=(
0,17), c='grey')

#set x and y labels, title, and adjust thier sizes according
TSPlotG3.set_ylabel('# of Detections in 5 Seconds', fontsize=16)
TSPlotG3.set_title('Geiger Counter Source Radioactivity Time Series',font
size=20)
plt.tick_params(axis='both', labelsize=15)   #change the font size of the
 axes ticks

xlabels = []
xticks = []
for i in range(25):
    #populate a list of 25 date/times with even time intervals over our 7
21 points
    xlabels.append(sliceFrame['Date/Time'][9763+i*30])

    #an array to place the date/time labels at the corresponding x value
    xticks.append(sliceFrame['Scatter'][9763+i*30])     #an array to plac
e the date/time labels at the corresponding x value

TSPlotG3.set_xticklabels(xlabels, rotation=90, fontsize=15)    #place the
 date/time labels and rotate them
```

```
TSPlotG3.set_xticks(xticks)
Line2 = sliceFrame.plot(x='Scatter',y='SMA',linewidth=5, c='red')
Line3 = sliceFrame.plot(x='Scatter',y='CMA',linewidth=5, c='blue')
TSPlotG3.set_xlabel('Date/Time',fontsize=16)
TSPlotG3.legend(('Fit Data','SMA','CMA'), loc=1);
```



Chapter 10
Formatting and Coverting IPython Notebooks

In [ ]: