PySD-Cookbook Documentation

Release 0.1.0

James Houghton

Mar 08, 2017
# Contents

1 Quickstart

2 How to use this cookbook

3 Contents
   3.1 Getting Started .......................................................... 7
   3.2 Data Handling .................................................................. 23
   3.3 Visualization ................................................................... 30
   3.4 Model Fitting ................................................................... 43
   3.5 Geographic Analyses ...................................................... 59
   3.6 Surrogating Functions ..................................................... 77
   3.7 Realtime Data Incorporation ............................................ 81
   3.8 Model Development Workflow ....................................... 87
   3.9 Data Used in this Cookbook ............................................. 88
   3.10 Chapters to be Written .................................................. 98
   3.11 End Notes ....................................................................... 99

4 Extra Resources .................................................................. 101

5 Indices and tables ................................................................ 103
Simple recipes for powerful analysis of system dynamics models
CHAPTER 1

Quickstart

1. Download and install Anaconda Python 3.6+
2. Install PySD with pip using the command `pip install pysd` at your command prompt
3. Download this cookbook and unzip it to a working directory
4. Navigate your command prompt to the working directory just created
5. Launch ipython notebook with the command `ipython notebook` at your command prompt
6. In the ipython browser, open `source\analyses\getting_started\Hello_World_Teacup.ipynb`
How to use this cookbook

Every recipe in this cookbook is an executable ipython notebook. Because of this, I recommend that you download a copy of the cookbook, and follow along by executing the steps in each notebook as you read, playing with the parameters, etc.

If you want to implement a recipe, you can then make a copy of the notebook you are interested in, and modify it to analyze your own problem with your own data.

To download the cookbook in its entirety, use this link or visit the cookbook’s Github Page and select one of the options in the righthand panel.
Chapter 2. How to use this cookbook
Dear Reader,

This cookbook is intended to be a resource for system dynamicists interested in applying the power of python analytics to their simulation models.

Every recipe in this cookbook is an executable Jupyter Notebook (a.k.a. iPython Notebook). Because of this, I recommend that you download a copy of the cookbook, open the source files and follow along by executing the steps in each recipe as you read, playing with the parameters, etc. If you want to implement a recipe, you can make a copy of the notebook you are interested in, and modify it to analyze your own problem with your own data.

Workbook versions of many recipes are available for use in a classroom setting. In these workbooks, components of the code have been removed for students to fill in on their own. These are identified with a trailing _Workbook in the filename.

You can download the cookbook as a zip file from this link, or, if you prefer, create a fork of the repository from its github page, and clone it to your computer.

If you come up with an interesting recipe of your own, I’d love to have your contributions. If you find a bug in the scripts used in this cookbook, or have a suggestion for improvements, I’d encourage you to submit an issue on the project’s issue tracker. If you find a bug in the way PySD itself behaves, the PySD Issue Tracker is a great place to raise it.

Thanks for being interested and supportive of this project. I hope you find the tools presented here to be useful.

Happy Coding!

James Houghton

Installation and Setup of Python and PySD
Using Anaconda

The simplest way to get started with Python is to use a prepackaged version such as Anaconda from Continuum analytics.

PySD works with Python 2.7+ or 3.5+. I recommend using the latest version of python 3 from the following link: https://www.continuum.io/downloads

Installing PySD

With a python environment set up, at the command prompt you can type:

```
pip install pysd
```

this should install PySD and all of its dependencies. In some cases you may need to prepend the command with ‘sudo’ to give it administrative priveledges:

```
sudo pip install pysd
```

You’ll be prompted for your administrator password.

Manually installing PySD’s dependencies

On the off chance that pip fails to install the dependencies, you can do the task yourself.

There are a few packages which form the core of the python scientific computing stack. They are:

1. numpy - a library for matrix-type numerical computation
2. scipy - a library which extends numpy with a bunch of useful features, such as stats, etc.
3. matplotlib - a basic plotting library
4. pandas - a basic data manipulation library
5. ipython - an environment for interacting with python and writing clever text/documentation documents, such as this one.

PySD also requires a few additional package beyond the basic data science stack:

1. parsimonious - a text parsing library
2. yapf - a python code formatting library

To install these packages, use the syntax:

```
pip install numpy
```

Run the command once for each package, replacing numpy with scipy, matplotlib, etc.

Launch Jupyter Notebook, and get started

If you used the anaconda graphical installer, you should have a ‘Anaconda Launcher’ user interface installed. Opening this program and clicking the ‘Jupyter Notebook’ will fire up the notebook explorer in your browser.

Alternately, at your command line type:
jupyter notebook

Your browser should start, and give you a document much like this one that you can play with.

Upgrading PySD

PySD is a work in progress, and from time to time, we’ll upgrade its features. To upgrade to the latest version of PySD (or any of the other packages, for that matter) use the syntax:

```
pip install pysd --upgrade
```

Getting Started with Python

Why should system dynamicists learn to code?

There is a whole world of computational and analysis tools being developed in the larger data science community. If system dynamicists want to take advantage of these tools, we have two options:

1. we can **replicate each of them within one of the system dynamics modeling tools we are familiar with**
2. we can **bring system dynamics models to the environment where these tools already exist**.

PySD and this cookbook are an outgrowth of the belief that this second path is the best use of our time. Bringing the tools of system dynamics to the wider world of data science allows us to operate within our core competency as model builders, and avoids doubled effort. It allows those who are familiar with programming to explore using system dynamics in their own projects, and ensures that the learning system dynamicists do to use these external tools will have application in the wider world.

Why Python?

Python is a high-level programming language that allows users to write code quickly and spend less time mucking about with the boring bits of programming. As a result, it is becoming increasingly popular and is the focus for development of a wealth of data science tools.

In the pen-strokes of xkcd:

**A (very) brief intro to programming in python**

**Basic Python Data Structures**

- Everything in python is an object.
- Objects have different ‘types’.
- Objects can be made up of other objects.
- Variable names are just labels assigned to point to specific objects.
Variables

Variables are created by assigning a value to a label.

```python
a = 3  # this will be an integer
b = "bob"  # this will be a string
c = 23.987  # this will be a float

print a, b, c
```

3 bob 23.987

Lists

Lists are ordered collections of objects (and are objects themselves).

```python
my_list = [1, 2, a]
print my_list
```

[1, 2, 3]

Elements of the list can be accessed or modified by position, with the first element having the index 0.

```python
print my_list[2]
```

3

```python
my_list[2] = 4
print my_list
```

[1, 2, 4]

Tuples

A tuple is an ordered list of python objects that is immutable, meaning that once defined they can’t be added to or changed. They are useful for things like sets of coordinates, where it doesn’t make sense to ‘add another dimension’.

From a pragmatic point of view, it’s mostly important to understand that they are created with (parentheses) and are often used in function calls and returns.

```python
my_tuple = (3, 4, 'hi')
my_tuple = (2, 4, 6)
print my_tuple[2]
```

6

```python
my_tuple[2] = 'bye'
```

```
TypeError                      Traceback (most recent call last)
    12 Chapter 3. Contents
Dictionaries

Dictionaries are named collections of objects which can be accessed by their label:

```python
my_dictionary = {'key 1': 1, 'key 2': 'b'}
print my_dictionary['key 2']
```

```
bob
```

You can add elements to a dictionary by assigning to an undefined element

```python
my_dictionary['key 3'] = 27
print my_dictionary
```

```
{'key 1': 1, 'key 2': 'bob', 'key 3': 27}
```

Python Control Flow

**if statements**

The body of an `if` statement must be indented - standard practice is 4 spaces.

```python
if True:
    print 'Inside the if statement'
```

```
Inside the if statement
```

```python
if 5 < 3:
    print 'In the if'
else:
    if 5 > 3:
        print 'in the elif'
    else:
        print 'In the else'
```

```
in the elif
```

```python
if 5 < 3:
    print 'In the if'
elif 5 >= 3:
    print 'in the elif'
else:
    print 'in the else'
```

```
in the else
```
This runs instead

for loops

For loops allow you to iterate over lists.

```python
my_list = [1, 2, 3, 'bob']
for emile in my_list:
    print emile
```

2
3
bob

If we want to iterate over a list of numbers, as is often the case with a for loop, we can use the `range` function to construct the list for us:

```python
for i in range(0, 10):
    if i > 3:
        print i,
    else:
        print 'bob',
```

bob bob bob bob 4 5 6 7 8 9

Python Functions

Functions are defined using the syntax below. As with `if` and `for`, indentation specifies the scope of the function.

```python
def my_function(param1, param2):
    result = param1 + param2
    return result

print my_function(3, 4)
```

7

Functions can have default arguments, making them optional to use in the function call:

```python
def my_other_function(param1=5, param2=10):
    return param1 * param2

print my_other_function(param2=4)
```

20

```
NameError
Traceback (most recent call last)
<ipython-input-35-1b17a9a8d97e> in <module>()
```
```python
4 print my_other_function(param2=4)
5 ---> 6 print param2
NameError: name 'param2' is not defined
```

### Methods and Attributes of Objects

Many python objects have their own methods, which are functions that apply specifically to the object, as in the string manipulation functions below:

```python
my_string = 'How about a beer?'
print my_string.lower()
print my_string.upper().rjust(30) # chained call to method
print my_string.replace('?', '!!')
```

how about a beer?

HOW ABOUT A BEER?

How about a beer!

Some objects have attributes which are not functions that act upon the object, but components of the object’s internal representation.

In the example below, we define a complex number, which has both a real part and a complex part, which we can access as an attribute:

```python
my_variable = 12.3 + 4j
print my_variable
print my_variable.real
print my_variable.imag
```

(12.3+4j)

12.3

4.0

### Resources for learning to program using Python.

- To get started learning python, an excellent collection of resources is available in The Hitchhiker’s Guide to Python.
- To try Python in the browser visit learnpython.org.
- Check out this overview of Python for computational statistics
- Online course on python for data science

and finally...

```python
import this
```

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than right now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

Using the Jupyter Notebook

The Jupyter (formerly iPython) Notebook is a browser-based interactive coding environment that allows cells of text (such as this one) to be interspersed with cells of code (such as the next cell), and the output of that code (immediately following)

```python
string = 'Hello, World!
print string
```

Hello, World!

Code Cells

To add cells to the notebook, click the [+ ] button in the notebook’s tool pane. A new cell will appear below the one which is selected. By default this cell is set for writing code snippets.

To execute code in a cell such as this, select it and press either the play [>] button in the tool pane, or press <shift><enter>. The results of the cell’s computation will be displaced below.

Once a cell has been run, the variables declared in that cell are available to any other cell in the notebook:

```python
print string[:6] + ' Programmer!'
```

Hello, Programmer!

Text Cells

To format a cell as text, with the desired cell highlighted, click the dropdown in the tool pane showing the word Code, and select Markdown.

Markdown is a simple syntax for formatting text. Headings are indicated with pound signs #:

```markdown
### Heading
```
**Italic**s and **bold** are indicated with one and two preceeding and following asterisks respectively: *Italic*: Italics, **Bold**: Bold

Code is blocked with tick fences (same key as tilde, not single quotes):

```python
...
Code goes here
```

and quotes are preceeded with the right-pointy chevron ‘>` > “This is not my quote.” - *Benjamin Franklin*

**Tab-completion**

While you’re typing code, it’s often possible to just start the word you want, and hit the `<tab>` key. iPython will give you a list of suggestions for how to complete that term.

For example, in the box below, if you place the cursor after `my`, ipython will show the above two variable names as options, which you can select and enter.

```python
my_very_long_variable_name = 2
my_fairly_long_variable_name = 3
my
```

**Context help**

It is sometimes hard to remember what arguments a function takes (or what order they come in).

If you type the name of the function and the open parenthesis, and then press `<shift><tab>`, a tooltip will come up showing you what arguments the function expects.

```python
sum()
```

**References**

- A comprehensive tutorial to using the Jupyter Notebook

**Working with git and github**

*git* is a version control system used widely by software developers to manage their codebase, as it has excellent support for collaboration, allowing different individuals to work on the same codebase, and combine their efforts.

*Github* is an online service that uses the *git* version control system to store files and facilitate collaboration on software projects. The code for PySD is hosted with github, as is that for much of the open source software community.

For system dynamicists, Github is a useful tool for tracking changes to a model or analysis code, documenting the development process, and facilitating model collaboration. For academics, it can serve as a permanent, citable repository for scripts used to prepare an academic paper, thus facilitating replication and extension of the research.

While the workflow may seem intimidating, once mastered it can prevent a lot of headaches.
Resources

Rather than include a tutorial of the github workflow in this cookbook, I’ll point to several resources for getting started with git and github:

- Interactive tutorial for using git and github at the command line
- Getting Started with Github video
- Github Desktop GUI Tool

Hello World: The Teacup Model

This notebook demonstrates the basic capability of PySD using a model of a cup of tea cooling to room temperature.

Our model simulates Newton’s Law of Cooling, which follows the functional form:

\[
\frac{dT}{dt} = k(T - T_{ambient})
\]

This model has all of the canonical components of a system dynamics model: a stock, a flow, a feedback loop, a control parameter, and exhibits dynamic behavior. The model equations are:

<table>
<thead>
<tr>
<th>Characteristic Time= 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Units: Minutes</td>
</tr>
<tr>
<td>Heat Loss to Room=</td>
</tr>
<tr>
<td>(Teacup Temperature - Room Temperature) / Characteristic Time</td>
</tr>
<tr>
<td>Units: Degrees/Minute</td>
</tr>
<tr>
<td>This is the rate at which heat flows from the cup into the room.</td>
</tr>
<tr>
<td>Room Temperature= 70</td>
</tr>
<tr>
<td>Units: Degrees</td>
</tr>
<tr>
<td>Teacup Temperature= INTEG (</td>
</tr>
<tr>
<td>-Heat Loss to Room, 180</td>
</tr>
<tr>
<td>Units: Degrees</td>
</tr>
</tbody>
</table>

Load the model

We begin by importing the PySD module using the python standard import commands. We then use PySD’s Vensim model translator to import the model from the Vensim model file and create a model object. We see that PySD translates the vensim component names into acceptable python identifiers.
%pylab inline
import pysd
model = pysd.read_vensim('../../models/Teacup/Teacup.mdl')

Populating the interactive namespace from numpy and matplotlib

The `read_vensim` command we have just run does two things. First it translates the model into a python module which is stored `../../models/Teacup/Teacup.py` in the same directory as the original file, with the filename changed to `.py`. You can go and have a look at the file and compare it to the vensim model file that it came from to get a sense for how the translation works.

The second thing the function does is load that translated python file into a PySD object and return it for use.

**Run with default parameters**

To understand the general behavior of the model, we can run a simulation using the default parameters specified by the Vensim model file. The default behavior of the run function is to return the value of all variables as a pandas dataframe:

```python
values = model.run()
values.head(5)
```

Pandas has some simple plotting utility built in which allows us to easily visualize the results.

```python
values.plot()
plt.ylabel('Degrees F')
plt.xlabel('Minutes')
plt.legend(loc='center left', bbox_to_anchor=(1,.5));
```

**Return specific model components**

If we wish to see values of only certain variables, we can pass a list of component names with the keyword argument `return_columns`. This will change the columns of the returned dataframe such that they contain samples of the requested model components. This is (very) slightly faster, but often cleaner:
Return values at a specific time

Sometimes we want to specify the timestamps that the run function should return values. For instance, if we are comparing the result of our model with data that arrives at irregular time intervals. We can do so using the `return_timestamps` keyword argument. This argument expects a list of timestamps, and will return values at those timestamps.

```python
values = model.run(return_columns=['Teacup Temperature', 'Room Temperature'])
values.plot()
plt.ylabel('Degrees F')
plt.xlabel('Minutes')
values.head()
```

```python
stocks = model.run(return_timestamps=[0, 1, 3, 7, 9.5, 13.178, 21, 25, 30],
                   return_columns=['Teacup Temperature'])
stocks.plot(linewidth=0, marker='o')
plt.ylabel('Degrees F')
plt.xlabel('Minutes')
stocks.head()
```
Modify parameter values

We can specify changes to the parameters of the model in the call to the run function. Here we set the room temperature to the constant value of 20 degrees before running the simulation.

```python
values = model.run(params={'Room Temperature': 50},
                   return_columns=['Teacup Temperature', 'Room Temperature'])
values.plot()
plt.ylabel('Degrees F')
plt.xlabel('Minutes');
```
We can also specify that a parameter be set with a time-varying input. In this case, we raise the room temperature from 20 to 80 degrees over the course of the 30 minutes. We can see that once the room temperature rises above that of the tea, the tea begins to warm up again.

```python
import pandas as pd

temp_timeseries = pd.Series(index=range(30), data=range(20,80,2))
values = model.run(params={'Room Temperature':temp_timeseries},
                   return_columns=['Teacup Temperature', 'Room Temperature'])
values.plot()
plt.ylabel('Degrees F')
plt.xlabel('Minutes');
```
Note that when you set a variable equal to a value, you overwrite the existing formula for that variable. This means that if you assign a value to a variable which is computed based upon other variable values, you will break those links in the causal structure. This can be helpful when you wish to isolate part of a model structure, or perform loop-knockout analysis, but can also lead to mistakes. To return to the original model structure, you’ll need to reload the model.

### Specifying model initial conditions

In addition to parameters, we can set the initial conditions for the model, by passing a tuple to the argument `initial_condition`. In this case, the first element of the tuple is the time at which the model should begin its execution, and the second element of the tuple is a dictionary containing the values of the stocks at that particular time.

Note again that this is a different syntax from merely sending a new value to the stock using the `params` syntax, which could cause unintended behavior as previously described.

```python
stocks = model.run(params={'room_temperature':75},
                   initial_condition=(0, {'teacup_temperature':33}),
                   return_columns=['Teacup Temperature', 'Room Temperature'])
stocks.plot()
plt.ylabel('Degrees F')
plt.ylim(30,80)
plt.xlabel('Minutes');
```
Once a model has been run, we can choose to run it forwards again from its current state. To do this we specify a new set of timestamps over which we would like the model to run, and pass the `initial_condition` argument the string "current".

```python
values = model.run(initial_condition='current',
                    return_columns=['Teacup Temperature', 'Room Temperature'],
                    return_timestamps=list(range(31,45)))
values.plot()
plt.ylabel('Degrees F')
plt.ylim(30,80)
plt.xlabel('Minutes');
```
Data Handling

Data handling with Pandas

Pandas is a library optimized for handling one or two dimensional data sources [1]. One dimensional data is stored in a *Series* object, and two dimensional data is stored in a *DataFrame* object.

Loading the library

It is customary to give the library a short handle ‘*pd*’ at import time:

```python
import pandas as pd
pd.options.display.max_rows = 10 #this line aids in displaying the data concisely
```

Loading data from CSV files

Pandas gives us a comprehensive set of tools for loading data from a *variety of sources*, including CSV, Excel, SQL, JSON, and Stata, amongst others. In this demonstration, we’ll read a comma separated value file of global emissions data from the year 1751 until 2011.
The `.read_csv` `method <>` gives us options for how we want to format the data as we read it in. In reading in our data file, we want to skip the second row (indexed as 1!) and use the column `Time` as the index of our resulting `DataFrame`.

```python
emissions = pd.read_csv('..../data/Climate/global_emissions.csv',
                         skiprows=[1], index_col='Year')
emissions  # Display the resulting DataFrame in the notebook
```

**Selecting rows of data by name**

Both `DataFrame` and `Series` objects have an `index` attribute which is used to identify their rows. We can access rows of data according to this index, using the `.loc[..., ]` syntax.

Between the brackets, we can select individual rows:

```python
emissions.loc[1875]
```

or ranges of dates:

```python
emissions.loc[1908:1920]
```

or ranges beginning or ending at a specific point:

```python
emissions.loc[1967:]
emissions.loc[:1805]
```

Give these a try and become comfortable selecting index ranges.

```python
emissions.loc[1985:1987]
```

**Selecting rows of data by position**

In addition to selecting by row names, we can select by the row position using the `.iloc` syntax.

This syntax lets us select the first n rows:

```python
emissions.iloc[:5]
```

or, if we wish, the last n, using a minus sign to indicate counting from the end of the `DataFrame`:

```python
emissions.iloc[-5:]
```

or rows in the middle:

```python
emissions.iloc[10:20]
emissions.iloc[1:30]
```

**Renaming columns**

The column names given in the CSV file are too long to use conveniently in dealing with data. We can assign new column names from a list of strings, that will be applied in order as the new column names:

```python
emissions.columns = ['Total Emissions', 'Gas Emissions', 'Liquid Emissions',
                     'Solid Emissions', 'Cement Emissions', 'Flare Emissions',
                     'Per Capita Emissions']
emissions.iloc[-3:]
```
Accessing specific columns

Each of the columns in the DataFrame can be accessed as its own Series object, using the same syntax we would use to access members of a python dictionary:

emissions['Total Emissions']

---

KeyError Traceback (most recent call last)
<ipython-input-9-dc76c55d0847> in <module>()
----> 1 emissions[['Total Emissions']]

/Users/houghton/anaconda/lib/python2.7/site-packages/pandas/core/frame.pyc in __getitem__(self, key)
   1984         # either boolean or fancy integer index
   1985     elif isinstance(key, (Series, np.ndarray, Index, list)):
-> 1986         return self._getitem_array(key)
   1987     elif isinstance(key, DataFrame):
   1988         return self._getitem_frame(key)

/Users/houghton/anaconda/lib/python2.7/site-packages/pandas/core/frame.pyc in _getitem_array(self, key)
   2028         return self.take(indexer, axis=0, convert=False)
   2029     else:
-> 2030         indexer = self.ix._convert_to_indexer(key, axis=1)
   2031         return self.take(indexer, axis=1, convert=True)

/Users/houghton/anaconda/lib/python2.7/site-packages/pandas/core/indexing.pyc in _convert_to_indexer(self, obj, axis, is_setter)
   1208             raise KeyError(''.join(['%s' % objarr[mask]]))
   1211         return _values_from_object(indexer)

KeyError: "['Total Emissions'] not in index"

Passing a list of column names into this syntax returns a subset of the dataframe:

emissions[['Gas Emissions', 'Liquid Emissions']]

Element-wise Arithmetic

We can perform element-wise arithmetic on DataFrame columns using natural syntax.

emissions['Gas Emissions'] + emissions['Liquid Emissions']

Year
1751   0
Array Operations

A number of simple operations are built into Pandas to facilitate working with the data. For example, we can show descriptive statistics such as the maximum value of each column:

```
print emissions.idxmax(), emissions.max()
```

<table>
<thead>
<tr>
<th>Year</th>
<th>Total Emissions</th>
<th>Gas Emissions</th>
<th>Liquid Emissions</th>
<th>Solid Emissions</th>
<th>Cement Emissions</th>
<th>Flare Emissions</th>
<th>Per Capita Emissions</th>
<th>dtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>2007</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>int64</td>
</tr>
<tr>
<td>2008</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>int64</td>
</tr>
<tr>
<td>2009</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>int64</td>
</tr>
<tr>
<td>2010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>int64</td>
</tr>
<tr>
<td>2011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>int64</td>
</tr>
</tbody>
</table>

The year in which this maximum value occurred:

```
emissions.idxmax()
```

<table>
<thead>
<tr>
<th>Year</th>
<th>Total Emissions</th>
<th>Gas Emissions</th>
<th>Liquid Emissions</th>
<th>Solid Emissions</th>
<th>Cement Emissions</th>
<th>Flare Emissions</th>
<th>Per Capita Emissions</th>
<th>dtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>int64</td>
</tr>
</tbody>
</table>

Or the sum of each column:

```
emissions.sum()
```

<table>
<thead>
<tr>
<th></th>
<th>Total Emissions</th>
<th>Gas Emissions</th>
<th>Liquid Emissions</th>
<th>Solid Emissions</th>
<th>Cement Emissions</th>
<th>Flare Emissions</th>
<th>Per Capita Emissions</th>
<th>dtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Emissions</td>
<td>373729.0</td>
<td></td>
<td>49774.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>float64</td>
</tr>
</tbody>
</table>
Merging Datasets

The dataset we have currently is missing data for per capita consumption before 1950. We have another dataset which gives us estimates of the world population which we can use to try and fill in some missing data. It too, however, has some missing values: before 1900, the data comes at 50 year intervals.

What we need to do is first merge the two datasets together. Pandas gives us a merge function which allows us to align the datasets on their index values.

Interpolating missing values

The merge operation creates NaN values in the rows where data is missing from the world population column. We can fill these using a cubic spline interpolation from the surrounding points:

Calculating per capita emissions

Now we can calculate a new value for per capita emissions. We multiply by 1,000,000 to convert from units of ‘Million Metric Tons’ as the Total Emissions are expressed, to merely ‘Metric Tons’, as the existing, incomplete estimate of per capita emissions is expressed.
interpolated['Per Capita Emissions 2'] = interpolated['Total Emissions'] /
interpolated['World Population'] * 1000000
interpolated.loc[1751:2011]

Pandas and PySD

By default, PySD will return the results of model simulation as a Pandas DataFrame, with the column names representing elements of the model, and the index (row names) as timestamps in the model.

```python
import pysd
model = pysd.read_vensim('..../models/Predator_Prey/Predator_Prey.mdl')
sim_result_df = model.run()
sim_result_df
```

In this case, may want to downsample the returned data to make it more manageable:

```python
sim_result_df.loc[range(50)]
```

Notes

[1]: While pandas can handle dimensions larger than two, it is clunky. Xarray is a package for handling multidimensional data that interfaces well with Pandas.

Resources

- Basic introduction to Pandas constructs
- More advanced usage of Pandas syntax
- Cookbook of Pandas Applications

Saving Simulation Results to a Database

There are a number of reasons why you might want to save simulation results to a database:

- Observing changes in model output over the course of model development
- Cacheing runs of a model to speed later analysis or display, especially in large models
- Creating a traceable record of your work

It’s relatively easy to set up a sequel database and commit runs output to it. This demo uses sqlite, which creates a database in a local file.

```python
import sqlite3
import numpy as np
import pysd
```

Ingredients
Model

We’ll use the simple teacup model for this experiment, and we’ll ask for the value at integer times from \([0..29]\).

```python
model = pysd.read_vensim('..../models/Teacup/Teacup.mdl')
tseries = range(30)
```

A database

In this example, we’ll create a database which will be saved in the working directory as `example.db`. We populate its columns with two columns for storing the parameter values that we’ll change from run to run, and then a column for each timestamp value we intend to save:

```python
conn = sqlite3.connect('example.db')
c = conn.cursor()
# Create table
c.execute('''CREATE TABLE simulations
geroom_temperature real, init_teacup_temperature real,
% real)'''
% ',
'.join([f't{i} real' for i in tseries]));
```

Parameters

We want to save the output of our model when driven with a variety of parameters. For demonstration, we’ll set these randomly:

```python
room_temps = np.random.normal(75, 5, 100)
init_tea_temps = np.random.normal(175, 15, 100)
```

The Recipe

We’re now ready to simulate our model with the various parameters. After execution, we construct a SQL insert query containing each of the returned values, and commit it to the database.

```python
for room_temp, init_tea_temp in zip(room_temps, init_tea_temps):
    output = model.run(params={'room_temperature':room_temp},
                        initial_condition=(0,{'teacup_temperature':init_tea_temp}),
                        return_timestamps=tseries)
    c.execute("INSERT INTO simulations VALUES (%s, %s, '%s')",
               (room_temp, init_tea_temp, ', '.join(output['teacup_temperature'].
                                                 apply(str)))
    conn.commit()
```

We can see that the result was added properly by fetching a record:

```python
c.execute('SELECT * FROM simulations')
c.fetchone()
```
Finally, we must remember to close our connection to the database:

```python
conn.close()
```

#remove the database file when we are finished with it.
!rm example.db

## Visualization

### Basic Visualization with matplotlib

Python's most commonly used plotting library is matplotlib. The library has an interface which mirrors that of Mathworks' Matlab software, and so those with matlab familiarity will find themselves already high up on the learning curve.

### Loading matplotlib and setting up the notebook environment

The matplotlib plotting library has a magic connection with the iPython shell and the notebook environment that allows static images of plots to be rendered in the notebook. Instead of using the normal `import ...` syntax, we'll use this iPython 'magic' to not only import the library, but set up the environment we'll need to create plots.
%pylab inline

Populating the interactive namespace from numpy and matplotlib

Load data to plot

We’ll use the emissions data we saw before in the Pandas tutorial, as it’s familiar:

```python
import pandas as pd
emissions = pd.read_csv('..../data/Climate/global_emissions.csv',
    skiprows=2, index_col='Year',
    names=['Year', 'Total Emissions',
           'Gas Emissions', 'Liquid Emissions',
           'Solid Emissions', 'Cement Emissions',
           'Flare Emissions', 'Per Capita Emissions'])
emissions.head(3)
```

Basic plotting

The basic plot command takes as its first two arguments the x and y values of the points which we wish to plot:

```python
plt.plot(emissions.index, emissions['Total Emissions']);
```

3.3. Visualization
Labeling axes and title

Following our plot command we can submit commands to add text to the figure, such as adding labels to the x and y axes, and a title to the figure.

```python
plt.plot(emissions.index, emissions['Total Emissions'])
plt.xlabel('Year')
plt.ylabel('Million Metric Tons CO2/Year')
plt.title('Historical CO2 Emissions', fontsize=18);
```

![Graph of Historical CO2 Emissions](image)

Changing line properties

We can include various elements into the plot command to specify how the line will look:

```python
plt.plot(emissions.index, emissions['Total Emissions'], 'ro', alpha=.5);
```
Specifying axis bounds

We can specify that we want our plot to be bounded by various x and y values:

```python
plt.plot(emissions.index, emissions['Total Emissions'])
plt.xlim(1950,2000)
plt.ylim(1500,7500);
```
Multiple lines

We can add lines to our plot simply by adding additional calls to the plot function. Passing the plot function an argument called ‘label’ allows us to format a legend with appropriate references to each line:

```python
plt.plot(emissions.index, emissions['Liquid Emissions'], 'r', label='Liquid')
plt.plot(emissions.index, emissions['Solid Emissions'], 'b', label='Solid')
plt.plot(emissions.index, emissions['Gas Emissions'], 'g', label='Gas')
plt.legend(loc='upper left');
```

Other plot types

There are a number of other plot types available, such as histograms, radial plots, plots with logarithmic axes, or stackplots:

```python
plt.stackplot(emissions.index, emissions[['Liquid Emissions'], emissions['Gas Emissions'], emissions['Solid Emissions']],
            labels=['Liquid', 'Gas', 'Solid'])
plt.legend(loc='upper left');
```
Saving figures

We can save a figure to the disk by calling matplotlib's savefig function:

```python
plt.plot(emissions.index, emissions['Total Emissions'])
plt.savefig('Figure_1_Total_Emissions.png')
```
Matplotlib and Pandas

Pandas uses matplotlib to provide a basic plotting interface of its own. The dataframe we have been working with has a convenience method called `.plot()`, which assumes some basic format for how you would like your data presented, and tries to do so for you.

This is handy when you are just interested in having a quick look at your data, without going to the trouble to create finished plots.

```
emissions.plot();
```

The Dataframe’s wrapper of matplotlib gives us a number of basic options for how our plots are shown:

```
emissions.plot(subplots=True, figsize=(10,6));
```
Matplotlib and PySD

As PySD returns a Pandas Dataframe, we can either use the plotting interface directly, or Pandas’s convenience wrapper. Here we’ll load a model which produces a chaotic output in three dimensions to use in our demonstration.

```python
import pysd
model = pysd.read_vensim('../models/Roessler_Chaos/roessler_chaos.mdl')
res = model.run()
res.head()
```

Plotting vs. time.

```python
plt.plot(res.index, res['x'], 'r')
plt.plot(res.index, res['y'], 'b')
plt.plot(res.index, res['z'], 'g');
```
Plotting variables against one another

```python
plt.plot(res['x'], res['y']);
```

While so far I have shown mostly basic, 2d plots, we can also call on matplotlib's 3d plotting engine

```python
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```
Resources

- Gallery of different matplotlib graphics, showing what types of plots are possible.
- Getting started with matplotlib video series

Other Visualization Packages in Python

There are a number of packages which interact with matplotlib, or work on their own, to extend python’s plotting abilities.

We’ll use the Roessler chaos model to generate data for plotting in this notebook:

```python
%pylab inline
import pysd
model = pysd.read_vensim('../models/Roessler_Chaos/roessler_chaos.mdl')
res = model.run()
```

**Seaborn**

Seaborn is a package which aims to make plots more stylish. To use it, we simply have to import it into the python environment and continue plotting as with matplotlib. Seaborn provides a host of new functions and customizations that are illustrated here.

```python
import seaborn
plt.plot(res['x'], res['y']);
```
MPLD3

MPLD3 translates matplotlib plotting syntax into a d3-rendered interactive javascript object.

Note that in the figure below, zoom and pan tools allow you to interact with the image.

Syntax here is very similar to before, with the addition of a call to the display function. For further examples see the mpld3 gallery.

```python
import mpld3
plt.plot(res['x'], res['y']);
mpld3.display()
```

Phase Portraits

In this notebook we’ll look at how to generate phase portraits. A phase diagram shows the trajectories that a dynamical system can take through its phase space. For any system that obeys the markov property we can construct such a diagram, with one dimension for each of the system’s stocks.

Ingredients

Libraries

In this analysis we’ll use:
• **numpy** to create the grid of points that we’ll sample over
• **matplotlib** via ipython’s pylab magic to construct the quiver plot

```
%pylab inline
import pysd
import numpy as np
```

Populating the interactive namespace from numpy and matplotlib

**Example Model**

A single pendulum makes for a great example of oscillatory behavior. The following model constructs equations of motion for the pendulum in radial coordinates, simplifying the model from one constructed in cartesian coordinates. In this model, the two state elements are the angle of the pendulum and the derivative of that angle, its angular velocity.

```
model = pysd.read_vensim('../models/Pendulum/Single_Pendulum.mdl')
```

**Recipe**

**Step 1: Define the range over which to plot**

We’ve defined the angle $0$ to imply that the pendulum is hanging straight down. To show the full range of motion, we’ll show the angles ranging from $-1.5\pi$ to $+1.5\pi$, which will allow us to see the points of stable and unstable equilibrium.

We also want to sample over a range of angular velocities that are reasonable for our system, for the given values of length and weight.

```
angular_position = np.linspace(-1.5*np.pi, 1.5*np.pi, 60)
angular_velocity = np.linspace(-2, 2, 20)
```

---

3.3. Visualization 41
Numpy’s \texttt{meshgrid} lets us construct a 2d sample space based upon our arrays

\begin{verbatim}
apv, avv = np.meshgrid(angular_position, angular_velocity)
\end{verbatim}

**Step 2: Calculate the state space derivatives at a point**

We’ll define a helper function, which given a point in the state space, will tell us what the derivatives of the state elements will be. One way to do this is to run the model over a single timestep, and extract the derivative information. In this case, the model’s stocks have only one inflow/outflow, so this is the derivative value.

As the derivative of the angular position is just the angular velocity, whatever we pass in for the \texttt{av} parameter should be returned to us as the derivative of \texttt{ap}.

\begin{verbatim}
def derivatives(ap, av):
    ret = model.run(params={'angular_position':ap,
        'angular_velocity':av},
    return_timestamps=[0,1],
    return_columns=['change_in_angular_position',
        'change_in_angular_velocity'])

    return tuple(ret.loc[0].values)

derivatives(0,1)
\end{verbatim}

\[ (1.0, -0.0) \]

**Step 3: Calculate the state space derivatives across our sample space**

We can use numpy’s \texttt{vectorize} to make the function accept the 2d sample space we have just created. Now we can generate the derivative of angular position vector \texttt{dapv} and that of the angular velocity vector \texttt{davv}. As before, the derivative of the angular position should be equal to the angular velocity. We check that the vectors are equal.

\begin{verbatim}
vderivatives = np.vectorize(derivatives)
dapv, davv = vderivatives(apv, avv)
(dapv == avv).all()
\end{verbatim}

\texttt{True}

**Step 4: Plot the phase portrait**

Now we have everything we need to draw the phase portrait. We’ll use matplotlib’s \texttt{quiver} function, which wants as arguments the grid of x and y coordinates, and the derivatives of these coordinates.

In the plot we see the locations of stable and unstable equilibria, and can eyeball the trajectories that the system will take through the state space by following the arrows.

\begin{verbatim}
plt.figure(figsize=(18,6))
plt.quiver(apv, avv, dapv, davv, color='b', alpha=.75)
plt.box('off')
plt.xlim(-1.6*np.pi, 1.6*np.pi)
plt.xlabel('Radians', fontsize=14)
\end{verbatim}
plt.ylabel('Radians/Second', fontsize=14)
plt.title('Phase portrait for a simple pendulum', fontsize=16);

Model Fitting

Fitting a model’s parameters with run-at-a-time optimization

In this notebook, we’ll fit a simple compartmental model to disease propagation data. We’ll use a standard optimizer built into the python scipy library to set two independent parameters to minimize the sum of squared errors between the model’s timeseries output and data from the World Health Organization.

About this technique

A run-at-a-time optimization runs the simulation forward from a single starting point, and so only requires an a-priori full state estimate for this initial condition. This makes it especially appropriate when we only have partial information about the state of the system.

Ingredients:

We’ll use the familiar pandas library along with pysd, and introduce the optimization functionality provided by scipy.optimize.

%pylab inline
import pandas as pd
import pysd
import scipy.optimize

Populating the interactive namespace from numpy and matplotlib

The model that we’ll try to fit is simple ‘Susceptible-Infectious’ model. This model assumes that everyone is either susceptible, or infectious. It assumes that there is no recovery, or death; and doesn’t account for changes in behavior due to the presence of the disease. But it is super simple, and so we’ll accept those limitations for now, until we’ve seen it’s fit to the data.
We’ll hold infectivity constant, and try to infer values for the total population and the contact frequency.

```python
model = pysd.read_vensim('..../models/Epidemic/SI Model.mdl')
```

We’ll fit our model to data from the WHO patient database for Sierra Leone. We see the standard S-Shaped growth in the cumulative infections curve. As our model has no structure for representing recovery or death, we will compare this directly to the Population Infected with Ebola. We format this dataset in the notebook ‘Ebola Data Loader’

```python
data = pd.read_csv('..../data/Ebola/Ebola_in_SL_Data.csv', index_col='Weeks')
data.plot();
```
Recipe

Step 1: Construct an ‘error’ function

We’ll begin by constructing a function which takes the model parameters that we intend to vary, and returns the sum of the squared error between the model’s prediction and the reported data.

Our optimizer will interact with our parameter set through an ordered list of values, so our function will need to take this list and unpack it before we can pass it into our model.

With PySD we can ask directly for the model components that we’re interested in, at the timestamps that match our data.

```python
def error(param_list):
    # unpack the parameter list
    population, contact_frequency = param_list
    # run the model with the new parameters, returning the info we're interested in
    result = model.run(params={'total_population':population,
                               'contact_frequency':contact_frequency},
                       return_columns=['population_infected_with_ebola'],
                       return_timestamps=list(data.index.values))
    # return the sum of the squared errors
    return sum((result['population_infected_with_ebola'] - data['Cumulative Cases →'])**2)
```

```
error([10000, 10])
```

157977495.47574666
Step 2: Suggest a starting point and parameter bounds for the optimizer

The optimizer will want a starting point from which it will vary the parameters to minimize the error. We’ll take a guess based upon the data and our intuition.

As our model is only valid for positive parameter values, we’ll want to specify that fact to the optimizer. We know that there must be at least two people for an infection to take place (one person susceptible, and another contagious) and we know that the contact frequency must be a finite, positive value. We can use these, plus some reasonable upper limits, to set the bounds.

\[
\begin{align*}
\text{susceptible\_population\_guess} &= 9000 \\
\text{contact\_frequency\_guess} &= 20 \\
\text{susceptible\_population\_bounds} &= (2, 50000) \\
\text{contact\_frequency\_bounds} &= (0.001, 100)
\end{align*}
\]

Step 3: Minimize the error with an optimization function

We pass this function into the optimization function, along with an initial guess as to the parameters that we’re optimizing. There are a number of optimization algorithms, each with their own settings, that are available to us through this interface. In this case, we’re using the L-BFGS-B algorithm, as it gives us the ability to constrain the values the optimizer will try.

\[
\begin{align*}
\text{res} &= \text{scipy.optimize.minimize(} \text{error, } [\text{susceptible\_population\_guess,} \\
&\quad \text{contact\_frequency\_guess}], \\
&\quad \text{method='L-BFGS-B',} \\
&\quad \text{bounds=[susceptible\_population\_bounds,} \\
&\quad \text{contact\_frequency\_bounds]})
\end{align*}
\]

Result

If we run the simulation with the parameters suggested by the optimizer, we see that the model follows the general behavior of the data, but is too simple to truly capture the correct shape of the curve.
plt.ylabel('Cumulative Infections')
plt.title('Model fit to Sierra Leone Ebola historical infections data')
plt.legend(loc='lower right')
plt.text(2,9000, 'RMSE: 7.5% of Max', color='r', fontsize=12)

res

fun: 22200247.95370693
hess_inv: <2x2 LbfgsInvHessProduct with dtype=float64>
jac: array([ 0. , -1666.3223505])
message: 'CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH'
nfev: 66
nit: 10
status: 0
success: True
x: array([ 8.82129606e+03, 8.20459019e+00])

sqrt(res.fun/len(data))/data['Cumulative Cases'].max()

0.075054691238299831

Step-at-a-time optimization

One of the consequences of forcing our models to obey the markov property is that the dynamic behavior of the model can be represented entirely in the transition from one state of the system to the next. This means that if we have full measurement of the state of the system, we can separate the model’s timeseries behavior into a series of

3.4. Model Fitting
independent timesteps. Now we can fit the model parameters to each timestep independently, without worrying about errors compounding throughout the simulation.

We’ll demonstrate this fitting of a model to data using PySD to manage our model, pandas to manage our data, and scipy to provide the optimization.

**About this technique**

We can use this technique when we have full state information measurements in the dataset. It is particularly helpful for addressing oscillatory behavior.

```python
%pylab inline
import pandas as pd
import pysd
import scipy.optimize
```

Populating the interactive namespace from numpy and matplotlib

```python
/Users/houghton/anaconda/lib/python2.7/site-packages/pandas/computation/__init__.py:19: UserWarning: The installed version of numexpr 2.4.4 is not supported in pandas and will be not be used
  UserWarning)

Ingredients

Model

In this demonstration we’ll fit the Lotka–Volterra Model model:

```python
model = pysd.read_vensim('../../../models/Predator_Prey/Predator_Prey.mdl')
```

We will apply this model to a predator/prey system consisting of Didinium and Paramecium, that was described in:

There are four parameters in this model that it will be our task to set, with the goal of minimizing the sum of squared errors between the model’s step-at-a-time prediction and the measured data.

**Data**

The data we’ll use was compiled from this work by Christian Jost.

```python
data = pd.read_csv('..//data/Predator_Prey/Veilleux_CC_0.5_Predator_Prey.txt', sep='\s+', header=4)
data[['prey(#ind/ml)','predator(#ind/ml)']].plot();
data.head(2)
```

![Graph showing prey and predator population over time](image)

**The Recipe**

**Step 1: Shape the dataset such that each row contains the start and end of a 'step'**

As we are analyzing the model a step at a time, it makes sense to restructure our dataframe so that each row contains both the starting and final state of each step. We can do this by merging the dataset with itself, offset by one row. We’ll add suffixes to the columns to differentiate between the start and end of each step.

While this method increases the burden of data that we have to carry, it allows us to use pandas’s `apply` functionality to increase computational speed over a `for` loop.

```python
data_steps = pd.merge(data.iloc[:-1], data.iloc[1:].reset_index(drop=True),
                        left_index=True, right_index=True, suffixes=('_s','_f'))
data_steps.head()
```

3.4. Model Fitting
**Step 2: Define a single-step error function**

We define a function that takes a single step and calculates the sum squared error between the model’s prediction of the final datapoint and the actual measured value. The most complicated parts of this function are making sure that the data columns line up properly with the model components.

Note that in this function we don’t set the parameters of the model - we can do that just once in the next function.

```python
def one_step_error(row):
    result = model.run(return_timestamps=[row['time(d)_f']],
                      initial_condition=(row['time(d)_s'],
                                      {'predator_population':row['predator(#ind/ml)_s'],
                                       'prey_population':row['prey(#ind/ml)_s']}))

    sse = ((result.loc[row['time(d)_f']]['predator_population'] - row['predator(#ind/ml)_f'])**2 +
           (result.loc[row['time(d)_f']]['prey_population'] - row['prey(#ind/ml)_f'])**2)

    return sse
```

**Step 3: Define an error function for the full dataset**

Now we define a function that sets the parameters of the model based upon the optimizer’s suggestion, and computes the sum squared error for all steps.

```python
def error(parameter_list):
    parameter_names = ['predation_rate', 'prey_fertility', 'predator_mortality',
                       'predator_food_driven_fertility']
    model.set_components(params=dict(zip(parameter_names, parameter_list)))

    errors = data_steps.apply(one_step_error, axis=1)
    return errors.sum()
```

```python
error([.005, 1, 1, .002])
```

545152.61053738836

Now we’re ready to use scipy’s built-in optimizer.

```python
res = scipy.optimize.minimize(error, x0=[.005, 1, 1, .002], method='L-BFGS-B',
                                bounds=[(0,10), (0,None), (0,10), (0,None)])
```

**Result**

We can plot the behavior of the system with our fit parameters over time:

```python
predation_rate, prey_fertility, predator_mortality, predator_food_driven_fertility =
res.x
values = model.run(params={'predation_rate':predation_rate,
                            'prey_fertility':prey_fertility,
                            'predator_mortality':predator_mortality,
                            'predator_food_driven_fertility':predator_food_driven_fertility})
```
Parallel Model Fitting

In this notebook, we’ll fit a simple ageing model to all of the counties in the United States. As before, we’ll use `scipy.optimize` to perform the fitting, but we’ll use Python’s `multiprocessing` library to perform these optimizations in parallel.

When to use this technique

This technique is appropriate when we are modeling a large number of entirely independent but structurally identical systems. In this example, we’re conceptualizing the population of counties to be influenced by aging and exogenous migration patterns. If we were to attempt to link the models together, for instance by specifying that the outmigration from one county needed to be accounted for as the immigration to another county, we would need to perform a single large-scale optimization, or some form of hybrid.
Ingredients

Data

The first ingredient that we’ll use is census data from the 2000 and 2010 census:

```python
data = pd.read_csv('..\..\data\Census\Males by decade and county.csv', header=[0,1],
                   skiprows=[2])
data.head()
```

Model

The model will be a simple ageing chain that groups individuals into 10 year cohorts.

```python
model = pysd.read_vensim('..\..\models\Aging_Chain\Aging_Chain.mdl')
```

The Recipe

As in our other optimization problems, we’ll construct an error function that calculates the sum of squared errors between our model prediction and the measured data. We also construct a helper function called `fit` which basically makes the call to the optimizer and formats the result in something that we can aggregate into a Pandas DataFrame.

```python
param_names = ['dec_%i_loss_rate'%i for i in range(1,10)]
def error(param_vals, measurements):
    predictions = model.run(params=dict(zip(param_names, param_vals)),
                             initial_condition=(2000,measurements['2000']),
                             return_timestamps=2010,
                             rtol=1).loc[2010]
    errors = predictions - measurements['2010']
    return sum(errors.values[1:]**2)  #ignore first decade: no birth info

def fit(row):
    res = scipy.optimize.minimize(error, args=row,
                                  x0=[.05]*9,
                                  method='L-BFGS-B');
    return pd.Series(index=['dec_%i_loss_rate'%i for i in range(1,10)], data=res['x'])
```

At this point, fitting the model is a simple matter of applying the `fit` function to the data:

```python
%%capture
county_params = data.apply(fit, axis=1)
```

On my 2014 era machine, this optimization takes about half an hour.

We can plot the distributions of the fit parameters for each of the counties in a histogram, to get a sense of the result. (Here we’re ignoring the first decade, which will not have reasonable parameters, as we have no information about births to the system.)

```python
df2 = county_params.drop('dec_1_loss_rate',1)
df2.plot(kind='hist', bins=np.arange(-.15,.4,.01), alpha=.4, histtype='stepfilled')
plt.xlim(-.15,.4)
plt.title('Fit yearly loss rates from each US county\nby age bracket from 2000 to 2010', fontsize=16)
```
We can take advantage of the multicore nature of most modern machines by using python’s multiprocessing module to distribute the various counties between each of the cores we have available for the calculation. The basic structure for this piece of code comes from this gist. We are essentially creating a helper function that will apply the fit function to a subset of the census DataFrame, and calling this function once on each of our worker nodes.

```python
%%capture
def _apply_df(args):
    df, func, kwargs = args
    return df.apply(func, **kwargs)
def apply_by_multiprocessing(df, func, workers, **kwargs):
    pool = multiprocessing.Pool(processes=workers)
    result = pool.map(_apply_df, [(d, func, kwargs) for d in np.array_split(df, workers)])
    pool.close()
```
Fitting a model with Markov Chain Monte Carlo

Markov Chain Monte Carlo (MCMC) is a way to infer a distribution of model parameters, given that the measurements of the output of the model are influenced by some tractable random process. In this case, performs something akin to the opposite of what a standard Monte Carlo simulation will do. Instead of starting with distributions for the parameters of a model and using them to calculate a distribution (usually related to uncertainty) in the output of the simulation, we start with a distribution of that output and look for input distributions.

Ingredients

For this analysis we’ll introduce the python package PyMC which implements MCMC algorithms for us. Another project which performs similar calculations is PyStan.

```
%pylab inline
import pysd
import pymc
import pandas as pd
```

For this example, we’ll revisit the ebola case, only assuming that our data has some noise, and we’ll use MCMC to estimate distributions for the parameters for the model. For a more detailed description of this model and the dataset see the recipe Fitting with Optimization.

We’ll assume that the model simulates an underlying process of disease propagation, but that the data is noisy - perhaps it represents admittance rates at a hospital, and so will be missing some cases, and may include some false positives through misdiagnosis.

```python
return pd.concat(list(result))
```

```python
county_params = apply_by_multiprocessing(data[:10], fit, axis=1, workers=4)
```
model = pysd.read_vensim('../../../models/SI Model/SI Model.mdl')

data = pd.read_csv('../../../data/Ebola/Ebola_in_SL_Data.csv', index_col='Weeks')
data.plot();
Recipe

Step 1: Establish prior distributions for the input parameters

Our first step is to establish prior distributions for the parameters for which we will be trying to infer posterior distributions. As in the optimization case, we’ll modify the effective population and the contact frequency. In real epidemiological modeling, we’d have a more complex model, and we’d have some information about the population, etc. but this makes for a relatively tight example.

```python
population = pymc.Uniform('total_population', lower=2, upper=50000, value=10000)
contact_frequency = pymc.Exponential('contact_frequency', beta=5, value=5)
```

Step 2: Perform the simulation

We now need to define a function that will run the model given a draw from the input distributions, and return the values that we will take as those of the fundamental underlying phenomena, before measurement noise is included.

```python
@pymc.deterministic
def sim(population=population, contact_frequency=contact_frequency):
    result = model.run(params={'total_population':population,
                               'contact_frequency':contact_frequency},
                       return_columns=['infection_rate'],
                       return_timestamps=list(data.index.values))
    return result['infection_rate'].values
```
Step 3: Include noise terms

There are several ways we could include noise. If we expected no false positives, we could use a Binomial distribution, such that of \( n \) possible cases that could be reported, only a fraction \( p \) would be reported, and other cases missed. If we only want to model false positives, we could assume that there was an average rate of false positives, with the data following a poisson distribution. The full rate would be the sum of these two processes.

For now, however, we’ll simplify the analysis by only looking at the Poisson noise component. The mean of the poisson process will be the results of our simulation.

This is where we include our measured data into the model. PyMC will know how to calculate the log likelihood of seeing the observed data given the assumption that the simulation result represents the underlying process, subject to Poisson noise.

```python
admittances = pymc.Poisson('admittances', mu=sim,
       value=data['New Reported Cases'], observed=True)
```

Step 4: Perform the MCMC Sampling

Now that we have set up the problem for PyMC, we need only to run the MCMC sampler. What this will do, essentially, is take a trial set of points from our prior distribution, simulate the model, and evaluate the likelihood of the data given those input parameters, the simulation model, and the noise distribution. It will then use bayes law to decide whether to keep the trial points or throw them away. It will then choose a new set of points and start over. (There is a lot more cleverness happening than this, of course. If you want to know how it works, I recommend Bayesian Methods for Hackers.

First we assemble the various pieces of the data flow that we built up into a model that pymc can recognize, and instantiate a sampler MCMC to run the algorithm for us.

Then we’ll ask the MCMC algorithm to run until it has kept 20000 points. We’ll throw out the first 1000 of these, as they are likely to be biased towards the initial values we set up and not representative of the overall distribution.

```python
mcmdl = pymc.Model([population, contact_frequency, sim, admittances])
mcmc = pymc.MCMC(mcmdl)
mcmc.sample(20000,1000)

[-----------------100%-----------------] 20000 of 20000 complete in 67.1 sec
```

Step 5: Look at the distribution

We can now evaluate the results by looking at the series of points we ‘kept’. These are stored as traces within the population and contact frequency objects we built earlier.

```python
plt.figure(figsize=(6,2))
plt.hist(population.trace[:], bins=100, histtype='stepfilled', normed=True, alpha=.6);
plt.title('Fit distribution for the population parameter');
```
```python
def fit_distribution(contact_frequency, parameter):
    bins = 100
    histtype = 'stepfilled'
    normed = True
    alpha = 0.6
    plt.figure(figsize=(6, 2))
    plt.hist(contact_frequency.trace[:], bins, histtype=histtype, normed=normed, alpha=alpha);
    plt.title(f'Fit distribution for the {parameter} parameter');
```

```python
def fit_distribution(contact_frequency, parameter):
    bins = 100
    histtype = 'stepfilled'
    normed = True
    alpha = 0.6
    plt.figure(figsize=(6, 1))
    plt.hist(contact_frequency.trace[:], bins, histtype=histtype, normed=normed, alpha=alpha);
    plt.yticks([])
    plt.title(f'Fit distribution for the {parameter} parameter');
    plt.xlabel('Minutes');
```

```python
def fit_distribution(contact_frequency, parameter):
    bins = 100
    histtype = 'stepfilled'
    normed = True
    alpha = 0.6
    plt.figure(figsize=(6, 1))
    plt.hist(contact_frequency.trace[:], bins, histtype=histtype, normed=normed, alpha=alpha);
    plt.yticks([])
    plt.title(f'Fit distribution for the {parameter} parameter');
    plt.xlabel('Minutes');
```
Step 6: Understand the correlation between our parameter distributions

PyMC has calculated for us the input distributions for the parameters we have based upon our data. This isn’t the whole story, however, as these values are not independent of one another. We can see by plotting them against one another that there is correlation:

```python
plt.plot(population.trace[:], contact_frequency.trace[:], '.', alpha=.1)
plt.xlabel('Effective Population')
plt.ylabel('Contact Frequency');
```

If we use these distributions in the future to propagate uncertainty through the model (say in doing a policy test) we should make sure that we’re including this correlation. The simplest way to do this is just to use the traces themselves as inputs into a normal Monte Carlo error propagation calculation, making sure that we use the same index in each trace for any given run. This will automatically include the correlation for us. Clever!

Resources:


A great resource for learning about MCMC is Bayesian Methods for Hackers

Geographic Analyses

Installation
conda install -c ioos geopandas=0.2.0.dev0

Using SD to understand the SD Fever

In this script, we will use georeferenced data at a national level to simulate a multiregional infectious disease model. We’ll then present the advantages to project spatial data produced by our simulation back on a map.

```python
%matplotlib inline
import pandas as pd
import pysd

import pandas as pd

/Users/houghton/anaconda/lib/python2.7/site-packages/pandas/computation/__init__.py:19: UserWarning: The installed version of numexpr 2.4.4 is not supported in pandas and will be not be used
  UserWarning)

The model we’ll use to represent the dynamics of the disease is a simple SIR model, with individuals aggregated into stocks by their health condition S-Susceptible, I-Infectious, R-Recovered. We assume that the complete population is susceptible, therefore the initial value of susceptible stock is equal to the total population. In addition, we built on the hypothesis that from now all infected person are reported.

```python
from IPython.display import Image
Image(filename='../../models/SD_Fever/SIR_model.png')
```

In Vensim our model was parameterized with 1000 susceptible, 5 infectious and 0 recovered individuals, a recovery period of 5 days and a contact infectivity of 70%.

When we do not specify anything else, the parameters and setting (e.g. timestep, simulation time) from the Vensim model are used.

```python
model = pysd.read_vensim('..//..//models/SD_Fever/SIR_Simple.mdl')
result = model.run()
result.plot();
```
Modify parameter values

As we have seen before, we can specify changes to the parameters of the model in the call to the run function. Here we set the contact infectivity to 30% before running the simulation again. If you like, try what happens when you change some of the other parameters.

```python
result = model.run(params={ 'total_population':1000,
                            'contact_infectivity':.3,
                            'recovery_period': 5
                          })
result.plot();
```
Change Model settings

We can also change in a very simple manner the simulation time and timestep of the model. An easy way to do it is to use numpy linspace which returns evenly spaced numbers over a specified interval.

\[
\text{np.linspace(Start, Stop, Number of timestamps)}
\]

```python
import numpy as np
sim_time = 10
np.linspace(0, sim_time, num=sim_time*4+1)
```

\[
\begin{array}{cccccccc}
0. & 0.25 & 0.5 & 0.75 & 1. & 1.25 & 1.5 & 1.75 \\
2. & 2.25 & 2.5 & 2.75 & 3. & 3.25 & 3.5 & 3.75 \\
4. & 4.25 & 4.5 & 4.75 & 5. & 5.25 & 5.5 & 5.75 \\
6. & 6.25 & 6.5 & 6.75 & 7. & 7.25 & 7.5 & 7.75 \\
10. &
\end{array}
\]

We can use the return_timestamps keyword argument in PySD. This argument expects a list of timestamps, and will return simulation results at those timestamps.

```python
model.run(return_timestamps=np.linspace(0, sim_time, num=sim_time*2+1))
```

Geographical Information

Geospatial information as area on a map linked to several properties are typically stored into shapefiles.

For this script, we will use geopandas library to manage the shapefiles, and utilize its inherent plotting functionality.

```python
import geopandas as gp
shapefile = '/path/to/shapefile.shp'
```
geo_data = gp.GeoDataFrame.from_file(shapefile)
geo_data.head(5)

---

Traceback (most recent call last)

<ipython-input-7-17f6f8f07da0> in <module>()
----> 1 import geopandas as gp
         2 shapefile = '../data/SD_Fever/geo_df_EU.shp'
         3 geo_data = gp.GeoDataFrame.from_file(shapefile)
         4 geo_data.head(5)

/Users/houghton/anaconda/lib/python2.7/site-packages/geopandas/__init__.py in <module>()
                   2 from geopandas.geodataframe import GeoDataFrame
                   3
----> 4 from geopandas.io.file import read_file
                   5 from geopandas.io.sql import read_postgis
                   6 from geopandas.tools import sjoin

/Users/houghton/anaconda/lib/python2.7/site-packages/geopandas/io/file.py in <module>()
                   1 import os
                   2 ----> 3 import fiona
                   4 import numpy as np
                   5 from shapely.geometry import mapping

/Users/houghton/anaconda/lib/python2.7/site-packages/fiona/__init__.py in <module>()
                   71 from fiona._drivers import driver_count, GDALEnv, supported_drivers
                   72 from fiona.ogrext import Iterator, ItemsIterator, KeysIterator
                   73 from fiona._drivers import Session, WritingSession
                   74 from fiona.ogrext import Session, WritingSession

/Users/houghton/anaconda/lib/python2.7/site-packages/fiona/_drivers.py in <module>()
                   1 from six import string_types
                   2 ----> 3 from fiona.ogrext import Iterator, ItemsIterator, KeysIterator
                   4 from fiona._drivers import Session, WritingSession
                   5

/Users/houghton/anaconda/lib/python2.7/site-packages/fiona/ogrext.so: Library not loaded: /libtiff.5.dylib
  Referenced from: /Users/houghton/anaconda/lib/libgdal.1.dylib
  Reason: Incompatible library version: libgdal.1.dylib requires version 8.0.0 or later, but libtiff.5.dylib provides version 7.0.0

Then we can project the geographic shape of the elements on a map.
import matplotlib.pyplot as plt

geo_data.plot()
plt.xlim([-28, 50])
plt.ylim([30, 75])
plt.show()

And plot one of the georeferenced property (e.g. population)

geo_data.plot(column='population', scheme='fisher_jenks', alpha=0.9, k=9, linewidth=0.1,
               cmap=plt.cm.YlOrRd, legend=False)
plt.xlim([-28, 50])
plt.ylim([30, 75])
plt.show()
Run the model for each country

We want to run the core SD model for each country, with country specific parameterization. Thus, we formulate a function that based on each row parameterizes the model with the value from geodata, performs the simulation and finally returns the number of infectious individuals over time.

```python
def runner(row):
    sim_time = 200
    params= {'total_population':row['population'],
             'contact_infectivity' : row['inf_rate']}
    res = model.run(params=params,
                    return_timestamps=np.linspace(0, sim_time, num=sim_time*2+1))
    return res['infectious']
```

Apply function along rows of the Dataframe.

We want to apply the function row-wise (by country) therefore we set axis to 1 (row) instead of default 0 (column). The result is a new dataframe with the produced simulation for each country.

```python
res = geo_data.apply(runner, axis=1)
res.head()
```

Transpose simulation results for plotting

The pandas line plot assumes that rows represent the timeseries and columns the different objects. Since our data is not yet in this form, we have to transpose the data. In pandas all we have to do is add an .T at the end.

```python
import pandas as pd
df = pd.DataFrame(res).T
df.head(2)
```
Comparative Analysis

Next let's try to compare how severe a country is hit by the SD fever.

Rather than looking at the number of infectious persons over time, a better indicator for comparative analysis are the cumulative cases as percentage of population in each country.

We can reuse our code from before but instead of returning the number of infectious we return the cumulative cases.

```python
def runner(row):
    sim_time = 200
    params= {'total_population':row['population'],
             'contact_infectivity' : row['inf_rate']}
    res = model.run(params=params,
                     return_timestamps=range(0,sim_time))
    return res['cumulative_cases']
```

#TIP: Ensure you are using lower case letters and the character _ not space

```python
res = geo_data.apply(runner, axis=1)
res.head()
```

The answer is a simple matrix operation: divide row-wise the elements of our computed values by the column of the original geo data set where we had the population in each country.

Let's try to perform this type of operation on a minimal example.

```python
# Create arbitrary column
column = pd.Series([10, 0])
column
```
# Create arbitrary pandas dataframe
df = pd.DataFrame(np.random.randint(1,5,size=(2, 3)), columns=list('ABC'))
df

Now we can translate this operation on our actual problem.

res = pd.DataFrame(res.T/geo_data['population'])
res.plot(legend=False);

![Graph](image)

## Analysis of results

For example, we could study the impact of contact infectivity on the cumulative cases at the end of the simulation

geo_data['totalcases%pop'] = res.loc[199] # Slice the final value at the end of the simulation

df_scatter = pd.DataFrame(geo_data) # Geopandas dataframe to pandas Dataframe

df_scatter.plot.scatter(x='inf_rate', y='totalcases%pop'); # Plot infectivity versus cumulative cases at the end of the simulation

---

3.5. Geographic Analyses 67
How Spatial Analysis Leads to Insight

Finally, we present slightly advanced Python scripts to get our simulation results projected on the map.

We merge the complete simulation results with our original georeferenced information just as we did in the step before.

```python
geo_data.head(2)

res.head(2)

go_data_merged = geo_data.merge(res.T, left_index=True, right_index=True)
go_data_merged.head()
```

Plotting simulation results on map with Ipywidgets

Ipywidgets are interactive HTML widgets for IPython notebooks. Users gain control of their data and can visualize changes in the data.

```python
import matplotlib as mpl
from ipywidgets import interact, FloatSlider, IntSlider, RadioButtons, Dropdown
sim_time = 200
slider_time = IntSlider(description = 'Time Select',
                        min=0, max=sim_time-1, value=1)

@interact(time = slider_time) # Scenario = select_scenario,
def update_map(time):
    # Scenario
    ax = geo_data_merged.plot(column=time, scheme='fisher-jenks', alpha=0.9, k=9, linewidth=0.1,
                               cmap=plt.cm.Reds, legend=True, figsize=(20, 30))
```
Using SD models to understand the differences between population measures at varying levels of geographic disaggregation

In this recipe, we will use data at a national level to infer parameters for a population aging model. We’ll then try two different ways of using this trained model to understand variation between the behavior of each of the states.

About this technique

Firstly, we’ll use the parameters fit at the national level to predict census data at the disaggregated level, and compare these predicted state-level outputs with the measured values. This gives us a sense for how different the populations of the states are from what we should expect given our understanding of the national picture.

Secondly, we’ll fit parameters to a model at each of the state levels actual measured census data, and then compare the differences in fit parameters to each other and to the national expectation. This is a helpful analysis if the parameter itself (and its inter-state variance) is what we find interesting.

```python
%pylab inline
import pandas as pd
```
import pysd
import scipy.optimize
import geopandas as gpd

Populating the interactive namespace from numpy and matplotlib

Ingredients

Population data by age cohort

We start with data from the decennial census years 2000 and 2010, for the male population by state and county. We have aggregated the data into ten-year age cohorts (with the last cohort representing everyone over 80 years old). The data collection task is described here. In this analysis we will only use data at the state and national levels.

data = pd.read_csv('../../data/Census/Males by decade and county.csv', header=[0,1], index_col=[0,1])
data.head()

A model of an aging population

The model we’ll use to represent the population is a simple aging chain, with individuals aggregated into stocks by decade, to match the aggregation we used for the above data. Each cohort is promoted with a timescale of 10 years, and there is some net immigration, outmigration, and death subsumed into the loss flow associated with each cohort. This loss is controled by some yearly fraction that it will be our task to understand.

model = pysd.read_vensim('..//..//models/Aging_Chain/Aging_Chain.mdl')

Our model is initialy parameterized with 10 individuals in each stock, no births, and a uniform loss rate of 5%. We’ll use data to set the initial conditions, and infer the loss rates. Estimating births is difficult, and so for this analysis, we’ll pay attention only to individuals who have been born before the year 2000.

model.run().plot();
Geography Information

This information comes to us as a shape file .shp with its associated .dbf and .shx conspirator files. Let's check the plotting functionality:

```python
state_geo = gpd.read_file('..../data/Census/US_State.shp')
state_geo.set_index('StateFIPSN', inplace=True)
state_geo.plot();
state_geo.head(2)
```
Recipe Part A: Predict state-level values from national model fit

Step 1: Initialize the model using census data

We can aggregate the county level data to the national scale by summing across all geographies. This is relatively straightforward.

```python
country = data.sum()
country
```

<table>
<thead>
<tr>
<th></th>
<th>2000</th>
<th>2010</th>
</tr>
</thead>
<tbody>
<tr>
<td>dec_1</td>
<td>20332536</td>
<td>20703935</td>
</tr>
<tr>
<td>dec_2</td>
<td>20909490</td>
<td>21878666</td>
</tr>
<tr>
<td>dec_3</td>
<td>19485544</td>
<td>21645336</td>
</tr>
<tr>
<td>dec_4</td>
<td>21638975</td>
<td>20033352</td>
</tr>
<tr>
<td>dec_5</td>
<td>21016627</td>
<td>21597437</td>
</tr>
<tr>
<td>dec_6</td>
<td>15115009</td>
<td>20451686</td>
</tr>
<tr>
<td>dec_7</td>
<td>9536197</td>
<td>13926846</td>
</tr>
<tr>
<td>dec_8</td>
<td>6946906</td>
<td>7424945</td>
</tr>
<tr>
<td>dec_9</td>
<td>3060483</td>
<td>4083435</td>
</tr>
</tbody>
</table>

dtype: float64

If we run the model using national data from the year 2000 as starting conditions, we can see how the cohorts develop, given our arbitrary loss rate values:
Step 2: Fit the national level model to the remaining data

We’ve used half of our data (from the year 2000 census) to initialize our model. Now we’ll use an optimization routine to choose the loss rate parameters that best predict the census 2010 data. We’ll use the same basic operations described in the previous recipe: Fitting with Optimization.

To make this simple, we’ll write a function that takes a list of potential model parameters, and returns the model’s prediction in the year 2010

```python
def exec_model(paramlist):
    params = dict(zip(['dec_%i_loss_rate' % i for i in range(1, 10)], paramlist))
    output = model.run(initial_condition=(2000, country['2000']),
                        params=params, return_timestamps=2010)
    return output
```

Now we’ll define an error function that calls this executor and calculates a sum of squared errors value. Remember that because we don’t have birth information, we’ll only calculate error based upon age cohorts 2 through 9.

```python
def error(paramlist):
    output = exec_model(paramlist)
    errors = output - country['2010']
    # don't tally errors in the first cohort, as we don't have info about births
    return sum(errors.values[0, 1:]**2)
```

Now we can use the minimize function from scipy to find a vector of parameters which brings the 2010 predictions into alignment with the data.
res = scipy.optimize.minimize(error, x0=[.05]*9,         
                           method='L-BFGS-B')

country_level_fit_params = dict(zip(
                                    ['dec_%i_loss_rate'%i for i in range(1,10)], res['x']))
country_level_fit_params

{'dec_1_loss_rate': 0.021183432598200467,       
 'dec_2_loss_rate': -0.052101419562612286,     
 'dec_3_loss_rate': -0.0014091019293939956,   
 'dec_4_loss_rate': 0.0088436979759478375,    
 'dec_5_loss_rate': -0.0072046351581388701,   
 'dec_6_loss_rate': -0.011046250905098235,     
 'dec_7_loss_rate': 0.017228650364514753,      
 'dec_8_loss_rate': 0.063195268137886118,      
 'dec_9_loss_rate': 0.16077452197707129}

If we run the national model forwards with these parameters, we see generally good behavior, except for the 0-9yr demographic bracket, from whom we expect less self-control. (And because we don’t have births data.)

model.run(params=country_level_fit_params,         
          return_timestamps=range(2000,2011),
          initial_condition=(2000, country['2000'])).plot();

---

**Step 3: Make state-level predictions**

If we want to look at the variances between the states and the national level, we can try making state-level predictions using state-specific initial conditions, but parameters fit at the national level.

states = data.sum(level=0)
states.head()
We can now generate a prediction by setting the model’s initial conditions with state level data, and parameters fit in the national case. I’ve created a `model_runner` helper function to make the code easier to read, but this could be conducted in a single line if we so chose.

```python
def model_runner(row):
    result = model.run(params=country_level_fit_params,
                       initial_condition=(2000, row['2000']),
                       return_timestamps=2010)
    return result.loc[2010]

state_predictions = states.apply(model_runner, axis=1)
state_predictions.head()
```

**Step 4: Compare model predictions with measured data**

Comparing the state level predictions with the actual data, we can see where our model most under or overpredicts population for each region/cohort combination.

```python
diff = state_predictions-states['2010']
diff.head()
```

This is a little easier to understand if we weight it by the actual measured population:

```python
diff_percent = (state_predictions-states['2010'])/states['2010']
diff_percent.head()
```

**Step 5: Merge with geo data to plot**

I’m using geopandas to manage the shapefiles, and it has its own plotting functionality. Unfortunately, it is not a particularly well defined functionality.

```python
geo_diff = state_geo.join(diff_percent)
geo_diff.plot(column='dec_4')
geo_diff.head()
```
Recipe Part B: fit state-by-state models

Now let's try optimizing the model's parameters specifically to each state, and comparing with the national picture.

Step 1: Write the optimization functions to account for the state

We'll start as before with functions that run the model and compute the error (this time with a parameter for the information about the state in question) and add a function to optimize and return the best fit parameters for each state.

```python
def exec_model(paramlist, state):
    params = dict(zip([\'dec_%i_loss_rate\' % i for i in range(1, 10)], paramlist))
    output = model.run(initial_condition=(2000, state[\'2000\']),
                        params=params, return_timestamps=2010).loc[2010]
    return output

def error(paramlist, state):
    output = exec_model(paramlist, state)
    errors = output - state[\'2010\']
    # don't tally errors in the first cohort, as we don't have info about births
    sse = sum(errors.values[1:]**2)
    return sse
```

Step 2: Apply optimization to each state

We can wrap the optimizer in a function that takes census information about each state and returns an optimized set of parameters for that state. If we apply it to the states dataframe, we can get out a similar dataframe that includes optimized parameters.
%%capture
def optimize_params(row):
    res = scipy.optimize.minimize(lambda x: error(x, row),
                                   x0=[0.05]*9,
                                   method='L-BFGS-B');
    return pd.Series(index=['dec_\%i\_loss_rate' % i for i in range(1,10)],
                     data=res['x'])

state_fit_params = states.apply(optimize_params, axis=1)
state_fit_params.head()

Step 3: Merge with geographic data

As we’re looking at model parameters which themselves are multiplied by populations to generate actual flows of people, we can look at the difference between parameters directly without needing to normalize.

geo_diff = state_geo.join(state_fit_params)
geo_diff.plot(column='dec_4\_loss_rate')
geo_diff.head(3)

---

Surrogating Functions

Surrogating a function with a machine learning estimator

System dynamics generally represents the relationships between model elements as either analytical equations, or lookup tables. However, in some situations we may be presented with relationships that are not well estimated by equations, but involve more than a single input leading to a single output. When confronted with this situation, other paradigms
```python
%pylab inline
import pysd
import numpy as np
import pandas as pd

Populating the interactive namespace from numpy and matplotlib

/Users/houghton/anaconda/lib/python2.7/site-packages/pandas/computation/__init__.py:19: UserWarning: The installed version of numexpr 2.4.4 is not supported in pandas and will be not be used
  UserWarning)

model = pysd.read_vensim('../../models/Manufacturing_Defects/Defects.mdl')
data = pd.read_csv('../../data/Defects_Synthetic/Manufacturing_Defects_Synthetic_Data.csv')
data.head()

plt.scatter(data['Workday'], data['Time per Task'], c=data['Defect Rate'],
            linewidth=0, alpha=.6)
plt.ylabel('Time per Task')
plt.xlabel('Length of Workday')
plt.xlim(0.15, .45)
plt.ylim(.01, .09)
plt.box('off')
plt.colorbar()
plt.title('Defect Rate Measurements')
plt.figtext(.88, .5, 'Defect Rate', rotation=90, verticalalignment='center');```
from sklearn.svm import SVR

Factors = data[['Workday', 'Time per Task']].values
Outcome = data['Defect Rate'].values
regression = SVR()
regression.fit(Factors, Outcome)

SVR(C=1.0, cache_size=200, coef0=0.0, degree=3, epsilon=0.1, gamma=0.0,
    kernel='rbf', max_iter=-1, shrinking=True, tol=0.001, verbose=False)

def new_defect_function():
    """ Replaces the original defects equation with a regression model""
    workday = model.components.length_of_workday()
    time_per_task = model.components.time_allocated_per_unit()
    return regression.predict([workday, time_per_task])[0]

model.components.defect_rate = new_defect_function

model.components.defect_rate()

0.05949975783815001

model.run().plot();
Surrogating a function with a machine learning estimator

System dynamics generally represents the relationships between model elements as either analytical equations, or lookup tables. However, in some situations we may be presented with relationships that are not well estimated by equations, but involve more than a single input leading to a single output. When confronted with this situation, other paradigms

```python
%pylab inline
import pysd
import numpy as np
import pandas as pd

model = pysd.read_vensim('..../models/Manufacturing_Defects/Defects.mdl')

data = pd.read_csv('..../data/Defects_Synthetic/Manufacturing_Defects_Synthetic_Data.csv')
data.head()

plt.<<...>>(data['Workday'], data['Time per Task'], c=data['Defect Rate'],
linewidth=0, alpha=.6)
plt.ylabel('Time per Task')
plt.xlabel('Length of Workday')
plt.xlim(0.15, .45)
plt.ylim(.01, .09)
plt.box('off')
plt.colorbar()
plt.title('Defect Rate Measurements')
plt.figtext(.88, .5, 'Defect Rate', rotation=90, verticalalignment='center');

from sklearn.svm import <<...>>
Factors = data[['Workday', 'Time per Task']].values
```
```
Outcome = data['Defect Rate'].values
regression = SVR()
regression.(Factors, Outcome)

def new_defect_function():
    """ Replaces the original defects equation with a regression model""
    workday = model.<>.length_of_workday()
    time_per_task = model.components.time_allocated_per_unit<<...>>
    return regression.predict([[<...>]][0])

model.components.defect_rate = <<...>>
print model.components.defect_rate()

model.run().plot();
```

## Realtime Data Incorporation

### Nth Order Delay Demo

This is a fun demonstration designed to build intuition around the idea that balancing feedback loops with delays lead to oscillation. It uses a vensim model as the ‘system’ but provides a way for a user to interact with the simulation in realtime - essentially acting as the controller - a balancing feedback loop around the model output.

### About this Technique

This is a way to interact with the models in realtime using your keyboard.

### Ingredients

**The Game**

The student is asked to use the ‘up’ and ‘down’ arrow keys to bring a blue line (the system output) to the value of the dashed red line (the target). However, the inputs from the keyboard go through a delay process (here using either the ‘first order delay’ model, or the ‘third order delay’ model).

When we run this cell, the student will have 60 seconds to bring the blue line to the level of the red line.

```%pylab
import pysd
from matplotlib import animation
import numpy as np
```

Using matplotlib backend: MacOSX
Populating the interactive namespace from numpy and matplotlib

```#import the model (need to import each time to reinitialize)
#choose one of the following lines:
#model = pysd.read_vensim('..../models/Basic_Structures/First_Order_Delay.mdl')
model = pysd.read_vensim('..../models/Basic_Structures/Third_Order_Delay.mdl')```
```python
# set the delay time in the model
model.set_components({'delay':5})

# set the animation parameters
fps=4
seconds=60
dt=1./fps

# set up the figure axes
fig, ax = plt.subplots()
ax.set_xlim(0,1)
ax.set_ylim(-10, 20)
ax.set_xticks([])
title = ax.set_title('Time %.02f' %0)

# draw the target line
ax.plot([0,1], [10,10], 'r--')

# draw the moving line, just for now. We'll change it later
line, = ax.plot([0,1], [0,0], lw=2)

# set up variables for simulation
input_val = 1
model.components.input = lambda: input_val

# capture keyboard input
def on_key_press(event):
global input_val
if event.key == 'up':
    input_val += .25
elif event.key == 'down':
    input_val -= .25
sys.stdout.flush()

fig.canvas.mpl_connect('key_press_event', on_key_press)

# make the animation
def animate(t):
    # run the simulation forward
    time = model.components.t+dt
    stocks = model.run(return_columns=['input', 'delay_buffer_1', 'delay_buffer_2',
                                      'delay_buffer_3', 'output'],
                        return_timestamps=[time],
                        initial_condition='current', collect=True)

    # make changes to the display
    level = stocks['output']
    line.set_data([0,1], [level, level])
    title.set_text('Time %.02f' %time)

    # call the animator.
anim = animation.FuncAnimation(fig, animate, repeat=False,
                                frames=seconds*fps, interval=1000./fps,
                                blit=False)

record = model.get_record()
record.head()
```
Display student input vs model output

To show how we did, we can plot the input and output over time. Here we start to see the oscillatory behavior (for higher order and longer delays)

```python
plt.plot(x, input_collector, label='Your Input')
plt.plot(x, y, label='Model Response')
plt.legend(loc='lower right')
plt.xlabel('Time [Seconds]')
plt.ylabel('Value');
```

Display the value of each of the buffer stocks over time

If we plot the stock levels over time, we can see (especially for the third order case) how the delay works to smooth out the input values.

```python
import pandas as pd
delay_stock_values = pd.DataFrame(stocks_collector)
delay_stock_values.plot()
plt.xlabel('Time [Seconds]')
plt.ylabel('Stock Level');
```

Feeding models with realtime streaming data

In this example, we’ll construct a very simple model of the number of posts relating to a given topic on twitter timelines around the world. We’ll feed the model with live streaming data from the twitter API and run the model in real time.

Ingredients

Libraries

In addition to our standard data analytics stack, we’ll take advantage of the `tweepy` library for interacting with the twitter API, the standard python library for dealing with JSON objects, and the standard python `threading` library, to give us access to both the stream and the plots in real time.

```
%pylab
import tweepy
import json
import pysd
```
import threading
from matplotlib import animation

Using matplotlib backend: MacOSX
Populating the interactive namespace from numpy and matplotlib

Twitter Credentials

If you want to execute this recipe, you’ll need to sign up for a twitter developer account and create a file containing your credentials.

Name this file _twitter_credentials.py and save it in the working directory. It should contain something similar to:

```python
cconsumer_key = 'sdjfhkdghsk8u09wejne4vdj8j34'
cconsumer_secret = 'nvjsdv8wp43neri'
caccess_token = 'vndisoerihosfbuttonmashingjfdlasnvei'
caccess_token_secret = 'navdjewrjodfjkmorebuttonmashingnjkasdoinv'
```

We can load these into our environment using the import * syntax.

```python
from _twitter_credentials import *
```

Model

Our model is a simple delay process. The inflow of the process is the rate at which twitter users post messages containing our keywords, and the level decays over time as posts fall out of the top of users timelines.

We’ll explicitly set the timescale in this demo, to show the behavior of the system in a short timeperiod.

```python
model = pysd.read_vensim('..///models/Twitter/Twitter.mdl')
model.set_components({"displacement_timescale":30})
```

The Recipe

Listening to twitter

The first thing we’ll do is to create a variable that will track the total number of posts recieved in the last model iteration timeperiod. This counter will be reset after every model timestep.

```python
counter = 0
```

Next, we’ll construct a function that will be run whenever a tweet is recieved from the API. This function will increase the counter value, and format the tweet to print to the screen.

```python
class TweetListener(tweepy.StreamListener):
    def on_data(self, data):
        global counter
        counter += 1
        
        # Twitter returns data in JSON format - we need to decode it first
        decoded = json.loads(data)
```
# Also, we convert UTF-8 to ASCII ignoring all bad characters sent by users
print '@%s %s
' % (decoded['user']['screen_name'], decoded['text'].encode('ascii', 'ignore'))
return True

def on_error(self, status):
    print status

The tweepy library manages formatting our credentials for the API request:

```python
auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_token_secret)
```

Finally we create the object that will parse the twitter stream, and start it within its own thread.

```python
stream = tweepy.Stream(auth, TweetListener())
t = threading.Thread(target=stream.filter, kwargs={'track':['ISDC', 'PySD', 'ISDC15', 'Trump']})
t.daemon = True
t.start()
```

### Animation

First we create a function that will be called at every step in the integration:

```python
# make the animation
def animate(t):
    global counter
    # run the simulation forward
    time = model.components.t+dt
    model.run({'tweeting':counter},
              return_timestamps=time,
              return_columns=['tweeting', 'posts_on_timeline'],
              initial_condition='current',
              collect=True)
    out = model.get_record()
    ax.plot(out['tweeting'], 'r', label='Tweeting')
    ax.plot(out['posts_on_timeline'], 'b', label='Posts on Timeline')
    counter = 0
```

@alisonjp: RT @queenfeminist: Retweet for Bernie Sanders fav for Hillary Clinton
Ignore and Donald Trump wins

@MrTommyCampbell: RT @swhammerhead: #WhenTrumpIsElected the letter H will be removed from the American lexicon as Trump doesn't pronounce it anyways. It will

@JBRichard50: RT @BradThor: "It's impossible for Perry to have stayed gov of TX for so long if he really is the person we saw in those debates." http://t.co/

@AdeboyeOriade: RT @politicususa: President Obama Rips Donald Trump, Mike Huckabee, and The Entire Republican Party http://t.co/krMZwVV1u0 via @politicususa

@1baldeagle77: The one thing that makes me take Donald Trump seriously as a candidate Rush Limbaugh http://t.co/vxDAy08xw7 via @voxdotcom
Lastly we set the parameters for the animation, set up the figure, reset the counter (which has been accumulating posts since we ran the first part of the code) and start the animation.

```python
# set the animation parameters
fps=1
seconds=60*30
dt=1./fps

# set up the plot
fig, ax = plt.subplots()
ax.set_xlim(0,seconds)
title = ax.set_title('Expected Twitter Messages on First Page of Feed')
ax.set_xlabel('Seconds')
ax.set_ylabel('Posts, Posts/second')

# reset the counter to start fresh.
counter=0

# call the animator.
animation.FuncAnimation(fig, animate, repeat=False,
                      frames=seconds*fps, interval=1000./fps,
                      blit=False)
```

@shehelby: RT @ProBirdRights: How can they say me a bird can not be run for Presidency when Donal Trump a giant talking corn can??? #birb2016

@LacrosseUpdate: Hope Hicks flies quietly in the eye of the Trump storm http://t.co/SSUZkcyyiG http://t.co/3MlnhhsEwc

@thedancingqueer: RT @queenfeminist: Retweet for Bernie Sanders fav for Hillary Clinton

Ignore and Donald Trump wins


@david_alman: Fucking leftist media spends so much time covering Donald Trump’s statements from 20 years ago that it neglects like...anything relevant.

@kcasa7131: RT @pdacosta: #WhenTrumpIsElected, he might appoint himself as Chairman of the Federal Reserve, and begin to issue Trump dollars. http://t.co/QzrzwiQo8N

@presidentrumped: President Trump: Mr Trump is a Gemini this means he is TOTALLY unpredictable! http://t.co/tP51raAyUH

@MicheleUpdate: Nicolle Wallace: Trump Is 'Doing One Thing Really, Really Well' http://t.co/kLfGNkCqyh

@jjyorktown: RT @ConanOBrien: Im on vacation. Please keep Trump popular while Im gone, I need him in the debates.

@MisaelDiazMLM: .@amazonmex @jcqs68 @amazon Amazon: Dejen de vender los libros de Donald Trump! - Firm la Peti... https://t.co/UMSTq5AxY2 va @Change_M

@StrongerForce: RT @BradThor: "It's impossible for Perry to have stayed gov of TX for so long if he really is the person we saw in those debates." http://t.co/QzrzwiQo8N
Model Development Workflow

Unit Testing

In this notebook we’ll demonstrate some basic unit testing for models.

We build on the standard python unittest library, which is documented here. Other testing suites are available, with their own advantages, but this is the most basic, so we’ll use it for our demonstration.

Unit testing us usually done not in the ipython notebook, but in a standalone python file. Tests for this demo are found in the accompanying file testsite.py:

```python
import pysd
import unittest

class TestTeacupModel(unittest.TestCase):
    """ Test Import functionality ""
    @classmethod
    def set_up_class(cls):
        cls.model = pysd.read_vensim('..../models/Teacup/Teacup.mdl')

    def test_initialization(self):
        self.assertEqual(self.model.components.state['teacup_temperature'], 180.0)

    def test_heatflow_calc(self):
        self.assertEqual(self.model.components.heat_loss_to_room(), 11.0)

    def test_output(self):
        self.assertAlmostEqual(self.model.run()['teacup_temperature'].iloc[-1], 75, delta=1)

if __name__ == '__main__':
    unittest.main()
```

3.8. Model Development Workflow
\%run testsuite.py

... Ran 3 tests in 0.053s

Data Used in this Cookbook

All of the data used in these notebooks was either gathered from public sources, or is created synthetically for the purpose of this cookbook. The notebooks listed here show how each set was collected or generated.

Baby Name Data

This notebook formats data from the social security administration baby names database into a format that is easy for the cookbook to deal with. It expects the zip file to be unpacked into a subfolder called ‘names’.

```python
import pandas as pd
import glob

filenames = glob.glob('names/yob*')

females = pd.DataFrame()
males = pd.DataFrame()

for filename in filenames:
    data = pd.read_csv(filename, header=None, names=['Name','Gender',year], index_col='Name')
    females = females.join(data[data['Gender']=='F'].drop('Gender', axis=1), how='outer')
    males = males.join(data[data['Gender']=='M'].drop('Gender', axis=1), how='outer')

females.to_csv('female_names_timeseries.csv')
males.to_csv('male_names_timeseries.csv')
```

Collecting US decennial census data

In this notebook, we’ll collect demographic data from the US decennial census, by county.

The census website has an API, which is good, because everything else about the census website is close to unusable. The api is described here: http://www.census.gov/data/developers/data-sets/decennial-census-data.html

As a quick demonstration, we can use the API to get population data for every county in the US:

```python
import pandas as pd

df = pd.read_json('http://api.census.gov/data/2010/sf1?get=P0120001&for=county:*')
df.columns = df.iloc[0]
```
df.drop(df.index[0], inplace=True)
df.head()

The census code descriptions can also be accessed via the API. A listing of the field names is available here: http://api.census.gov/data/2010/sf1/variables.html

pd.read_json('http://api.census.gov/data/2010/sf1/variables/P0120001.json', typ='ser')

concept P12. Sex By Age [49]
lable Total population
name P0120001

dtype: object

Collect data on male population by age, county

For now I’m only going to look at males. This is probably a bad idea in general.

Start with the 2010 census

The male population is broken down into some somewhat arbitrary cohorts, each with its own name. We want all of the fields between P0120003 and P0120025.

We’ll do some data munging to get it in numeric format, and to take care of the labels and indicies.

fields = ['P012000%i' for i in range(3,26)]
url = 'http://api.census.gov/data/2010/sf1?get=%s&for=county:*' %','.join(fields)
print url
pops2010 = pd.read_json(url)
pops2010.columns = pops2010.iloc[0]
pops2010.drop(pops2010.index[0], inplace=True)
pops2010 = pops2010.applymap(float)
pops2010.set_index(['state', 'county'], inplace=True)
pops2010.head()

Get data from 2000

The 2000 census (logically) has different codes for its data, and (even more logically) breaks the cohorts down differently. In this case, we can get data for each age year with codes PCT012003 through PCT012104. The api limits us to only 50 columns at a time, so we’ll do it in chunks and stitch them together.

fields = ['PCT01200%i' for i in range(3,105)]
dflist = []
chunkSize = 40
for i in range(0, len(fields), chunkSize):
    chunk = fields[i:i+chunkSize]
    url = 'http://api.census.gov/data/2000/sf1?get=%s&for=county:*' %','.join(chunk)
```
print url
df_chunk = pd.read_json(url)
df_chunk.columns = df_chunk.iloc[0]
df_chunk.drop(df_chunk.index[0], inplace=True)
df_chunk.set_index(['state', 'county'], inplace=True)
dflist.append(df_chunk)
pops2000 = pd.concat(dflist,axis=1)
pops2000 = pops2000.applymap(float)
pops2000.head()
```

Align the datasets

As they have different cohorts, we need to do some summation before we can merge the two census years into a single table. I’ll break the data down into 10-year cohorts by selecting columns to stitch together. We’ll set breakpoints by the last few digits of the field name, and label our new cohorts according to which decade of your life they are. We’re using 1-based indexing here for the cohort names.

```
pops2010d = pd.DataFrame(index=pops2010.index)
decades = ['dec_%i' % i for i in range(1,10)]
breakpoints_2010 = [3, 5, 8, 12, 14, 16, 18, 22, 24, 26]
for dec, s, f in zip(decades, breakpoints_2010[:-1], breakpoints_2010[1:]):
    pops2010d[dec] = pops2010["P0120%03i" % i for i in range(s,f)].sum(axis=1)
pops2010d.head()
```

```
pops2000d = pd.DataFrame(index=pops2000.index)
decades = ['dec_%i' % i for i in range(1,10)]
breakpoints_2000 = [3, 13, 23, 33, 43, 53, 63, 73, 83, 104]
for dec, s, f in zip(decades, breakpoints_2000[:-1], breakpoints_2000[1:]):
    pops2000d[dec] = pops2000["PCT012%03i" % i for i in range(s,f)].sum(axis=1)
pops2000d.head()
```

Now that the data have been formatted in the same way, we’ll concatenate them. We also drop any rows that don’t show up in both datasets.
frame = pd.concat([pops2000d, pops2010d], keys=[2000, 2010], axis=1)
frame.dropna(inplace=True)
frame.head()

I’m happy with this format, so we’ll save it to csv:

frame.to_csv('Males by decade and county.csv')

As our dataframe has a MultiIndex we have to take care when re-importing from the csv to get the index and header columns correct.

pd.read_csv('Males by decade and county.csv', header=[0,1], index_col=[0,1])

**Carbon Data Sources**

**Carbon Emissions**


**Atmospheric Carbon**

Data for atmospheric carbon concentrations is the Scripps CO2 program Mauna Loa Monthly measurements, available at: http://scrippsc02.ucsd.edu/data/atmospheric_co2.

**World Population**

Data for the world population estimates comes from OurWorldInData.org: https://ourworldindata.org/world-population-growth/

**Manufacturing Defects Synthetic Data**

In this notebook we generate some data that will represent measurements of defects in a manufacturing setting.

```python
import numpy as np
import pandas as pd

#generate synthetic data
Factors = []
Outcome = []
numpoints = 2000
for workday, time_per_task in zip(np.random.normal(loc=.3, scale=.05,
   size=numpoints), np.random.normal(loc=.05, scale=.01, size=numpoints)):
   Factors.append([workday, time_per_task])
   Outcome.append( 0*workday**2/(time_per_task**2) + 1/time_per_task**1.5 +
                  1000*workday**1.5)

data = pd.DataFrame(Factors, columns=['Workday', 'Time per Task'])
data['Defect Rate'] = Outcome
data['Defect Rate']//= data['Defect Rate'].max()*10
```
```python
data['Defect Rate'] += np.random.normal(scale=.003, size=len(data['Defect Rate']))
data.head()
data.to_csv('Manufacturing_Defects_Synthetic_Data.csv')
```

## Ebola Data Loader

In this notebook, we’ll format data from The World Health Organization for future analysis

```python
%pylab inline
import pandas as pd
import re

Populating the interactive namespace from numpy and matplotlib

```

```
#read in the raw data
rawdata = pd.read_csv('Ebola_in_SL_Raw_WHO_Data.csv')
rawdata.iloc[1]
```

<table>
<thead>
<tr>
<th>COUNTRY (CODE)</th>
<th>SLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNTRY (DISPLAY)</td>
<td>Sierra Leone</td>
</tr>
<tr>
<td>COUNTRY (URL)</td>
<td>NaN</td>
</tr>
<tr>
<td>EBOLA_MEASURE (CODE)</td>
<td>CASES</td>
</tr>
<tr>
<td>EBOLA_MEASURE (DISPLAY)</td>
<td>Number of cases</td>
</tr>
<tr>
<td>EBOLA_MEASURE (URL)</td>
<td>NaN</td>
</tr>
<tr>
<td>CASE_DEFINITION (CODE)</td>
<td>CONFIRMED</td>
</tr>
<tr>
<td>CASE_DEFINITION (DISPLAY)</td>
<td>Confirmed</td>
</tr>
<tr>
<td>CASE_DEFINITION (URL)</td>
<td>NaN</td>
</tr>
<tr>
<td>EBOLA_DATA_SOURCE (CODE)</td>
<td>PATIENTDB</td>
</tr>
<tr>
<td>EBOLA_DATA_SOURCE (DISPLAY)</td>
<td>Patient database</td>
</tr>
<tr>
<td>EBOLA_DATA_SOURCE (URL)</td>
<td>NaN</td>
</tr>
<tr>
<td>EPI_WEEK (CODE)</td>
<td>2015-W07</td>
</tr>
<tr>
<td>EPI_WEEK (DISPLAY)</td>
<td>09 to 15 February 2015</td>
</tr>
<tr>
<td>EPI_WEEK (URL)</td>
<td>NaN</td>
</tr>
<tr>
<td>INDICATOR_TYPE (CODE)</td>
<td>SITREP_NEW</td>
</tr>
<tr>
<td>INDICATOR_TYPE (DISPLAY)</td>
<td>New</td>
</tr>
<tr>
<td>INDICATOR_TYPE (URL)</td>
<td>NaN</td>
</tr>
<tr>
<td>DATAPACKAGEID (CODE)</td>
<td>2015-04-22</td>
</tr>
<tr>
<td>DATAPACKAGEID (DISPLAY)</td>
<td>Data package 2015-04-22</td>
</tr>
<tr>
<td>DATAPACKAGEID (URL)</td>
<td>NaN</td>
</tr>
<tr>
<td>Display Value</td>
<td>92</td>
</tr>
<tr>
<td>Numeric</td>
<td>92</td>
</tr>
<tr>
<td>Low</td>
<td>NaN</td>
</tr>
<tr>
<td>High</td>
<td>NaN</td>
</tr>
<tr>
<td>Comments</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Name: 1, dtype: object

```

```python
#parse the dates column
import dateutil
def parsedate(week_string):
    end_date_str = re.split(' to ', week_string)[1]
    return (dateutil.parser.parse(end_date_str))
```
rawdata['End Date'] = rawdata['EPI_WEEK (DISPLAY)'].apply(parsedate)
rawdata.head()

data = rawdata[rawdata['EBOLA_DATA_SOURCE (CODE)']=='PATIENTDB']
data = data[['End Date','Numeric']]  
data.sort(columns='End Date', inplace=True)     
data.dropna(inplace=True)            
data['Timedelta'] = data['End Date']-data['End Date'].iloc[0]       
data['Weeks'] = data['Timedelta'].apply(lambda a: pd.tslib.Timedelta(a).days/7)  
data.set_index('Weeks', inplace=True) 
data.columns=['New Reported Cases']     
data['Cumulative Cases'] = data['New Reported Cases'].cumsum()

data.plot()

<matplotlib.axes._subplots.AxesSubplot at 0x10bb9a990>

data.to_csv('Ebola_in_SL_Data.csv')

**Parse gmail .mbox file**

This notebook takes an archive of gmail messages and from them extracts a value for the time between when a conversation involving the account holder begins, and when they first reply, or join the conversation.

To download an .mbox file from gmail, containing an archive of your email messages, visit https://takeout.google.com/settings/takeout. (This is valid as of Feb 2016 - however this may not always be the case!)

A lot of the message headers are described here: http://www.ietf.org/rfc/rfc2076.txt
The python standard library has a module for dealing with this type of file, which we’ll draw on extensively, along with pandas.

```python
%pylab inline
import mailbox
import pandas as pd

Populating the interactive namespace from numpy and matplotlib

mb = mailbox.mbox('All mail Including Spam and Trash.mbox')

Step 1: extract relevant information from mbox file

In this analysis we will want to group messages according to the conversation they are a part of. Gmail makes this easy for us as one of the message attributes, X-GM-THRID, is a unique thread identifier, assigned by google as it compiles messages together into conversation threads.

We’ll also want the field X-Gmail-Labels, which will tell us if the message is categorized as Sent, so that we can distinguish it from messages that were received by the account holder. Lastly, we want the date and time that the message was sent: Date.

We don’t need any information about the content of the message, or even who it was to or from.

```python
keys = ['Date', 'X-Gmail-Labels', 'X-GM-THRID']
message_list = []

for message in mb.itervalues():
    dmessage = dict(message.items())
    message_list.append({key:dmessage[key] if key in dmessage.keys() else '' for key in keys})

print len(message_list), 'messages'
print '**'*50

print message_list[:3]
```

messages = pd.DataFrame(message_list)
messages.index = messages['Date'].apply(lambda x: pd.to_datetime(x, errors='coerce'))
messages.drop(['Date'], axis=1, inplace=True)
print messages.shape
print messages.head()
```

127005 messages
************************************************************************************************

```python
[("Date": 'Mon, 07 Sep 2015 23:45:16 +0000',
 'X-GM-THRID': '1511700066770539101',
 'X-Gmail-Labels': 'Important'),
 ('Date': 'Wed, 9 Sep 2015 08:56:18 -0400',
 'X-GM-THRID': '1511700066770539101',
 'X-Gmail-Labels': 'Important'),
 ('Date': 'Wed, 9 Sep 2015 09:53:19 +0000',
 'X-GM-THRID': '1511828916942046097',
 'X-Gmail-Labels': 'Important')]
```

(127005, 2)
Step 2: group the messages by thread

We'll begin by grouping the messages into their various threads.

```python
conversation_list = []
threads = messages.groupby(by='X-GM-THRID')
print len(threads), 'threads total'

counts = threads.aggregate('count')['X-Gmail-Labels'].value_counts()
counts.plot(logy=True, linewidth=0, marker='.', alpha=.5)
plt.ylabel('Number of Threads')
plt.xlabel('Length of Thread')
```

32966 threads total

Step 3: find relevant timestamps

To simplify our analysis, we will assume that we are interested in the time between when a thread starts and when
the account holder joins the conversation by sending their first message in the thread. To find these times, we sort the
thread on its index, and take the first message time as the start of the conversation.

Then we filter to just the messages labeled *Sent*, and (as they are still sorted) take the first of these.

```python
for name, group in threads:
    if len(group) > 1:
        if 'Sent' in group['X-Gmail-Labels'].values:
            group.sort_index(inplace=True)
            tstart = group.index[0]
            tjoin = group[group['X-Gmail-Labels']=='Sent'].index[0]
```
Step 4: Calculate the time delta

We now can subtract the one from the other, and convert the result into units of days.

```python
delta = conversations['tjoin']-conversations['tstart']
days = 1.* delta.dt.total_seconds() / 3600 / 24
```

```
     0   0.000000
     1   0.196354
     2   0.000000
     3   0.000000
     4   0.000000
          dtype: float64
```

Step 5: Filter out threads started by the account holder

When the time difference between the first message in the thread and the first message sent by the account holder is zero, then we assume that the account holder initiated the thread. As we are interested in seeing how the account holder deals with emails they receive, we filter these out.

```python
days = days[days>0]
days = days.reset_index()[0]
days.head()
```

```
      0   0.196354
      1   0.076539
      2   0.352731
      3   0.321343
      4   0.720741
Name: 0, dtype: float64
```

Step 6: Save to csv

```python
days.to_csv('days_to_join_conversation.csv')
```
import pandas as pd

towns = pd.read_csv('towns.csv', header=None, names=['Town', 'Type', 'County', 'Governance', 'Population 2010', 'Year Established'])
towns.head()

print ', '.join([n.replace(' ', '_').replace('-', '_') for n in towns['Town'].values])


', '.join([n.replace(',', '') for n in towns['Population 2010'].values])

3.9. Data Used in this Cookbook
Chapters to be Written

This cookbook is a work in progress. In the future, I’d like to develop examples in the following major areas. If you’d like to suggest additional topics, or have a model and/or dataset that would make for a good case study, feel free to shoot me an email (james.p.houghton@gmail.com) or post on the Github issue tracker.

This is a notional listing of desired chapters, with existing chapters

1. Model Fitting to data
   (a) optimizing in the phase space
   (b) Frequency Domain
   (c) Fitting to multiple datasets
   (d) Fitting with unobserved states/ unobserved stocks

2. Monte Carlo Analysis
   (a) Sensitivity Analysis
   (b) Propagation of Uncertainties

3. Markov Chain Monte Carlo
   (a) Estimating parameter distributions for a single model
   (b) Reversible jump MCMC for model selection

4. Working with geographic data
5. Patch Models
6. Linking models together
7. Surrogating model functions with data
   (a) Nearest neighbors methods
8. Plotting and displaying simulation output
   (a) Static Visualization
   (b) Interactive Plots
9. Driving a model with external data
10. Loading and processing data in real time
11. Kalman filtering on realtime data
12. Multiple Models
13. Clustering model outputs
   1. based upon timeseries features
   2. based upon derivative features
   3. based upon phase-plane features
   4. based upon estimated(fit) parameters
14. Bootstrapping/cross validations
   1. Multiple datasets
   2. Partitioned timeseries
15. Statistical screening
   1. Screening for sensitivity analysis
16. Hidden markov models and dynamic models
17. Machine Learning predictors and dynamic models
18. Testing policy robustness
19. Extreme conditions testing - unit testing on models.
20. Pysd and Exploratory Modeling Analysis Workbench

End Notes

For instruction

The notebooks in this cookbook can be used as slide decks. To do so, install the rise package.
Extra Resources

The PySD Documentation can answer most questions about how PySD itself should be used, with basic examples and function reference.
CHAPTER 5

Indices and tables

- genindex
- modindex
- search