# Advanced data analytics for system dynamics models using PySD

James Houghton, Michael Siegel

## Abstract

The emerging field of data science has potential for great synergy with system dynamics modeling: system dynamics providing a causal structure for the interpretation of increasingly available social data, and this data helping to anchor and improve system dynamics practice and extend its applicability.

PySD is a tool designed to facilitate the integration of system dynamics models and data science by bringing models created with traditional SD modeling platforms into the rapidly developing ecosystem of Python data science tools. This paper gives an overview of PySD's purpose and capabilities, describes its technical construction, documents its primary functionality, and illustrates several use cases with simple examples.

## Motivation: The (coming of) age of Big Data

The last few years have witnessed a massive growth in the collection of social and business data[1], and a corresponding boom of interest in learning about the behavior of social and business systems using this data. The field of 'data science' is developing a host of new techniques for dealing with and analyzing data, responding to an increase in demand for insights and the increased power of computing resources.

So far, however, these new techniques are largely confined to variants of statistical summary, categorization, and inference; and if causal models are used, they are generally static in nature, ignoring the dynamic complexity and feedback structures of the systems in question.[2] As the field of data science matures, there will be increasing demand for insights beyond those available through analysis unstructured by causal understanding. At that point data scientists may seek to add dynamic models of system structure to their toolbox.

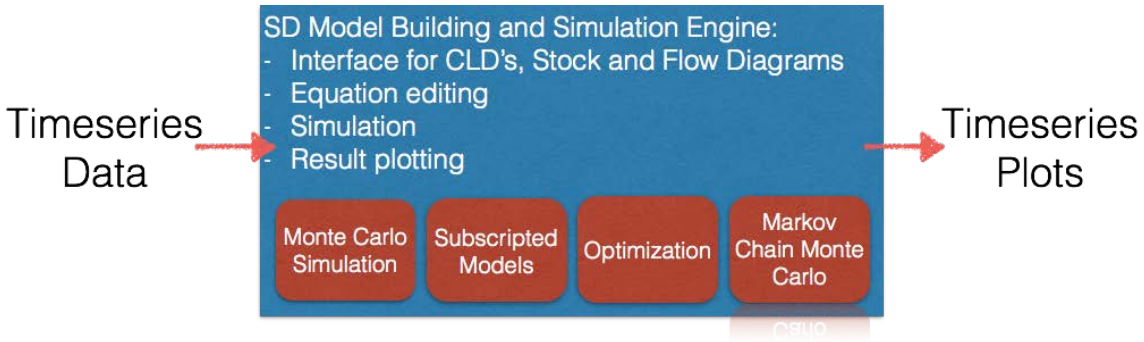
The field of system dynamics has always been interested in learning about social systems, and specializes in understanding dynamic complexity. There is likewise a long tradition of incorporating various forms of data into system dynamics models. While system dynamics practice has much to gain from the emergence of new volumes of social data, the community has yet to benefit fully from the data science revolution.[3,4]

There are a variety of reasons for this, the largest likely being that the two communities have yet to commingle to a great extent. A further, and ultimately more tractable reason is that the tools of system dynamics and the tools of data analytics are not tightly integrated, making joint method analysis unwieldy. There is a rich problem space that depends upon the ability of these fields to support one another, and so there is a need for tools that help the two methodologies work together. PySD is designed to meet this need.

## General approaches for integrating system dynamic models and data analytics

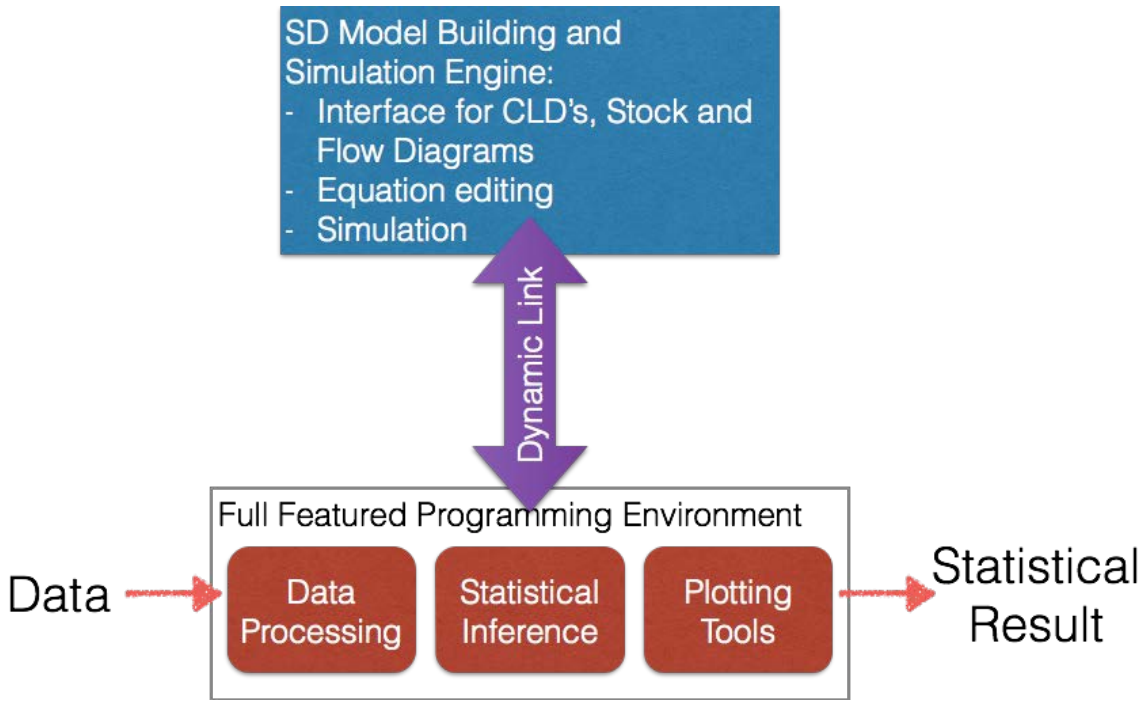
Before considering how system dynamics techniques can be used in data science applications, we should consider the variety of ways in which the system dynamics community has traditionally dealt with integration of data and models.

The first paradigm for using numerical data in support of modeling efforts is to import data into system dynamics modeling software. Algorithms for comparing models with data are built into the tool itself, and are usable through a graphical front-end interface as with model fitting in Vensim, or through a programming environment unique to the tool. When new techniques such as Markov chain Monte Carlo analysis become relevant to the system dynamics community, they are often brought into the SD tool.

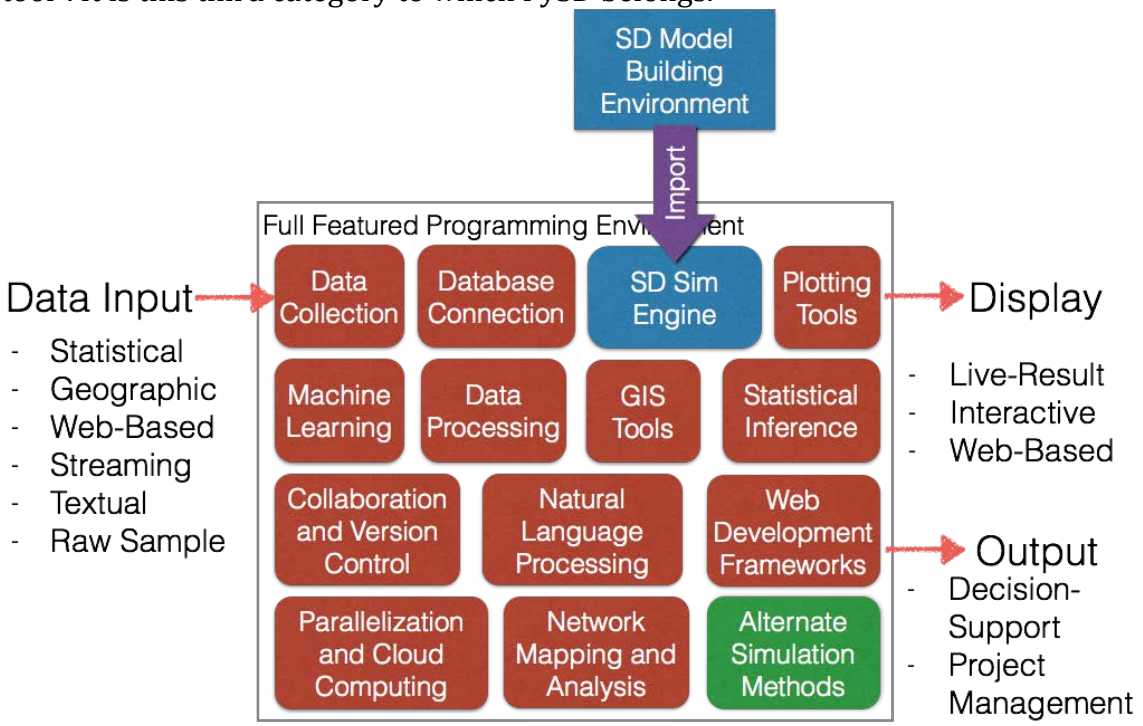



This approach appropriately caters to system dynamics modelers who want to take advantage of well-established data science techniques without needing to learn a programming language, and extends the functionality of system dynamics to the basics of integrated model analysis.

A second category of tools uses a standard system dynamics tool as a computation engine for analysis performed in a coding environment. This is the approach taken by the Exploratory Modeling Analysis (EMA) Workbench[5], or the Behavior Analysis and Testing Software (BATS)[6]. This first step towards bringing system dynamics to a more inclusive analysis environment enables many new types of model understanding, but imposes limits on the depth of interaction with models and the ability to scale simulation to support large analysis.




A third category of tools imports the models created by traditional tools to perform analyses independently of the original modeling tool. An example of this is SDM-Doc[7], a model documentation tool, or Abdel-Gawad et. al.'s eigenvector analysis tool[8]. It is this third category to which PySD belongs.




The central paradigm of PySD is that it is more efficient to bring the mature capabilities of system dynamics into an environment in use for active development

in data science, than to attempt to bring each new development in inference and machine learning into the system dynamics enclave.

PySD reads a model file – the product of a modeling program such as Vensim[9] or Stella/iThink[10] – and cross compiles it into Python, providing a simulation engine that can run these models natively in the Python environment. It is not a substitute for these tools, and cannot be used to replace a visual model construction environment.

### How this paper is structured

This paper makes the assumption that readers are more interested in the use of PySD than in its inner workings, and so after briefly discussing the structure and mechanics of the tool itself, we focus on demonstration and applications.

## Structure of the PySD module

PySD provides a set of translators that interpret a Vensim or XMILE[11] format model into a Python native class. The model components object represents the state of the system, and contains methods that compute auxiliary and flow variables based upon the current state.

The components object is wrapped within a Python class that provides methods for modifying and executing the model. These three pieces constitute the core functionality of the PySD module, and allow it to interact with the Python data analytics stack.

### Translation

The PySD module is capable of importing models from a Vensim model file (*.mdl) or an XMILE format xml file. Translation makes use of a Parsing Expression Grammar parser, using the third party Python library Parsimonious[12] to construct an abstract syntax tree based upon the full model file (in the case of Vensim) or individual expressions (in the case of XMILE).

The translators then crawl the tree, using a dictionary to translate Vensim or Xmile syntax into its appropriate Python equivalent. The use of a translation dictionary for all syntactic and programmatic components prevents execution of arbitrary code from unverified model files, and ensures that we only translate commands that PySD is equipped to handle. Any unsupported model functionality should therefore be discovered at import, instead of at runtime.

The use of a one-to-one dictionary in translation means that the breadth of functionality is inherently limited. In the case where no direct Python equivalent is available, PySD provides a library of functions such as pulse, step, etc. that are specific to dynamic model behavior.

In addition to translating individual commands between Vensim/XMILE and Python, PySD reworks component identifiers to be Python-safe by replacing spaces with underscores. The translator allows source identifiers to make use of alphanumeric characters, spaces, or the $ symbol.

### The model class

The translator constructs a Python class that represents the system dynamics model. The class maintains a dictionary representing the current values of each of the system stocks, and the current simulation time, making it a 'statefull' model in much the same way that the system itself has a specific state at any point in time.

The model class also contains a function for each of the model components, representing the essential model equations. The docstring for each function contains the model documentation and units as translated from the original model file. A query to any of the model functions will calculate and return its value according to the stored state of the system.

The model class maintains only a single state of the system in memory, meaning that all functions must obey the Markov property - that *the future state of the system can be calculated entirely based upon its current state*. In addition to simplifying integration, this requirement enables analyses that interact with the model at a step-by-step level. The downside to this design choice is that several components of Vensim or XMILE functionality – the most significant being the infinite order delay – are intentionally not supported. In many cases similar behavior can be approximated through other constructs.

Lastly, the model class provides a set of methods that are used to facilitate simulation. PySD uses the standard ordinary differential equations solver provided in the well-established Python library Scipy, which expects the state and its derivative to be represented as an ordered list. The model class provides the function `.d_dt()` that takes a state vector from the integrator and uses it to update the model state, and then calculates the derivative of each stock, returning them in a corresponding vector. A complementary function `.state_vector()` creates an ordered vector of states for use in initializing the integrator.

### The PySD class

The PySD class provides the machinery to get the model moving, supply it with data, or modify its parameters. In addition, this class is the primary way that users interact with the PySD module.

The basic function for executing a model is appropriately named `.run()`. This function passes the model into scipy's `odeint()` ordinary differential equations solver. The scipy integrator is itself utilizing the lsoda integrator from the Fortran library odepack[13], and so integration takes advantage of highly optimized low-level routines to improve speed. We use the model's timestep to set the maximum step

size for the integrator's adaptive solver to ensure that the integrator properly accounts for discontinuities.

The `.run()` function returns to the user a Pandas dataframe[14] representing the output of their simulation run. A variety of options allow the user to specify which components of the model they would like returned, and the timestamps at which they would like those measurements. Additional parameters make parameter changes to the model, modify its starting conditions, or specify how simulation results should be logged.

## Basic usage

This module is written for a user with basic familiarity of the Python programming language, and the most popular components of the Python data analytics ecosystem. This seems to be a reasonable prerequisite, as in-depth data analytics tends to require familiarity with general programming and database constructs. For those wishing to develop their knowledge of data analysis, or learn the specific tools present in the Python ecosystem, a resource list is available in the appendix.

### Installation

To install the PySD package from the Python package index into an established Python environment, use the pip[15] command:

```
pip install pysd
```

To install from the source, download the latest version from the project webpage: https://github.com/JamesPHoughton/pysd and in the subsequent directory use the Python command:

```
python setup.py install
```

### Dependencies

In addition to Python standard libraries, PySD builds on the core Python data analytics stack: Numpy[16], Scipy[17], Pandas[14], and Matplotlib[18]. In addition, it calls on Parsimonious[12] to handle translation.

Additional Python libraries that integrate well with PySD and bring additional data analytics capabilities to the analysis of SD models include PyMC[19], a library for performing Markov chain Monte Carlo analysis; Scikit-learn[20], a library for performing machine learning in Python; NetworkX[21], a library for constructing networks; and GeoPandas[22], a library for manipulating geographic data.

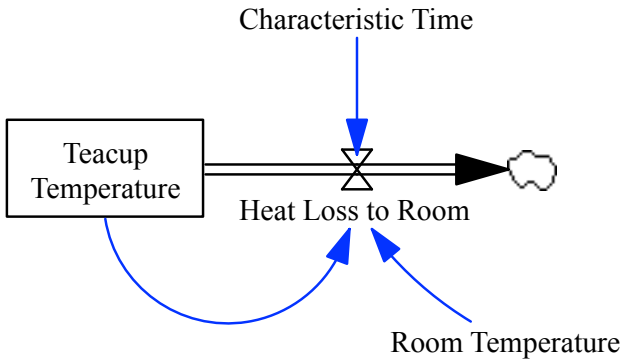
Additionally, the System Dynamics Translator utility developed by Robert Ward is useful for translating models from other system dynamics formats into the XMILE standard, to be read by PySD.

## Importing a model and getting started

To begin, we must first load the PySD module, and use it to import a supported model file:

```
import pysd
model = pysd.read_vensim('Teacup.mdl')
```

This code creates an instance of the PySD class loaded with an example model that we will use as the system dynamics equivalent of 'Hello World': a cup of tea cooling to room temperature, as seen in the figure to the right.




To view a synopsis of the model equations and documentation, print the model's components object:

```
print model.components
```

This will generate a listing of all the model elements, their documentation comments, units, equations, and initial values, where appropriate. Here is a sample from the teacup model:

```
Import of Teacup.mdl

characteristic_time
Units: minutes
Equation: 10

dteacup_temperature_dt
Units: degrees
Equation: -heat_loss_to_room()
Init: 180

final_time
the final time for the simulation.
Units: minute
Equation: 30

…
```

## Running the Model

The simplest way to simulate the model is to use the `.run()` command with no options:

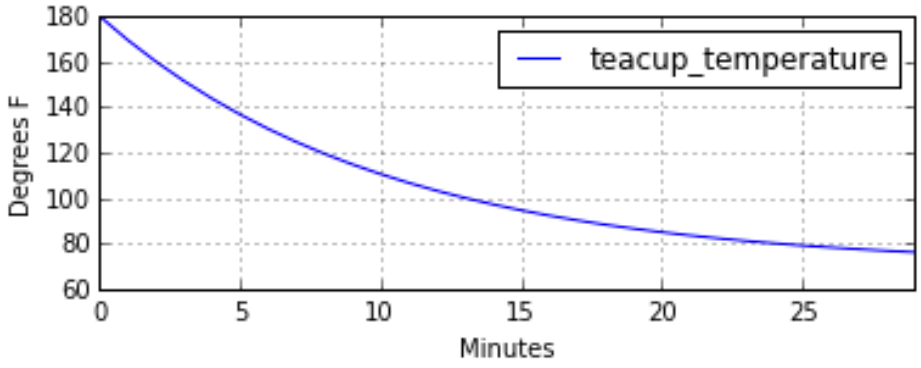
```
stocks = model.run()
```

This runs the model with the default parameters supplied by the model file, and returns a Pandas dataframe of the values of the stocks at every timestamp. In this case, the model has a single stock, and PySD returns a single data column:

| t | teacup_temperature |
|---|---|
| 0.000 | 180.000000 |
| 0.125 | 178.633556 |
| 0.250 | 177.284091 |
| 0.375 | 175.951387 |

Pandas gives us simple plotting capability, so we can see how the cup of tea behaves:

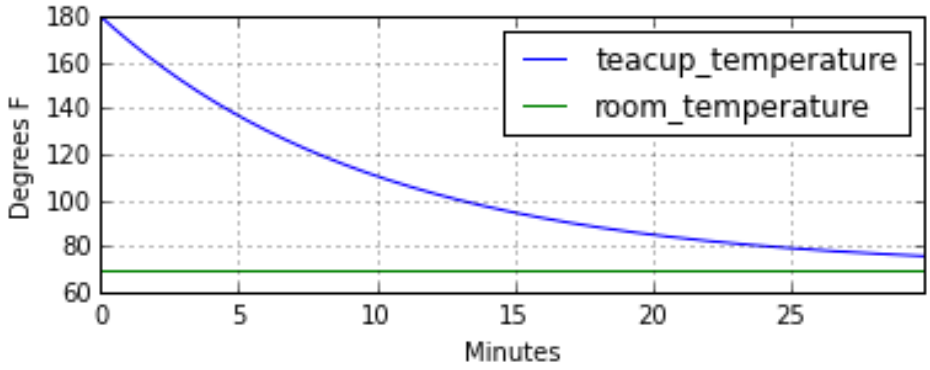
```
stocks.plot()
plt.ylabel('Degrees F')
plt.xlabel('Minutes')
```




## Outputting various run information

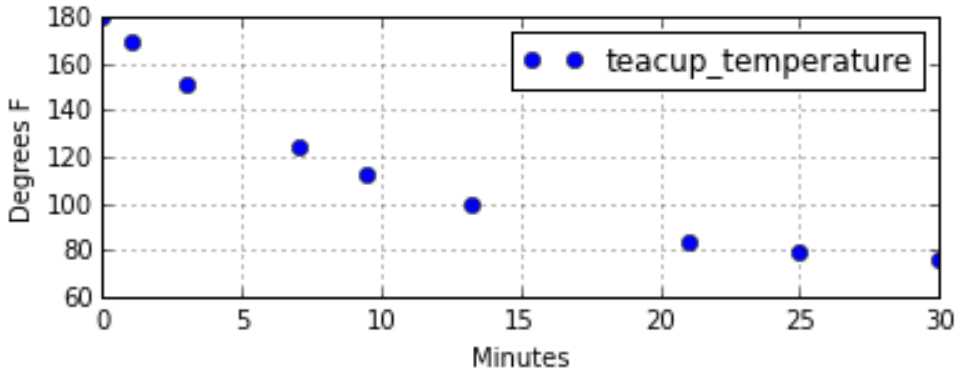
The `.run()` command has a few options that make it more useful. In many situations we want to access components of the model other than merely the stocks – we can specify which components of the model should be included in the returned dataframe by including them in a list that we pass to the `.run()` command, using the `return_columns` keyword argument.

```
values = model.run(return_columns=['teacup_temperature',
                                   'room_temperature'])
```




If the measured data that we are comparing with our model comes in at irregular timestamps, we may want to sample the model at timestamps to match. The `.run()` function gives us this ability with the `return_timestamps` keyword argument.

```
model.run(return_timestamps=[0,1,3,7,9.5,13.178,21,25,30])
```

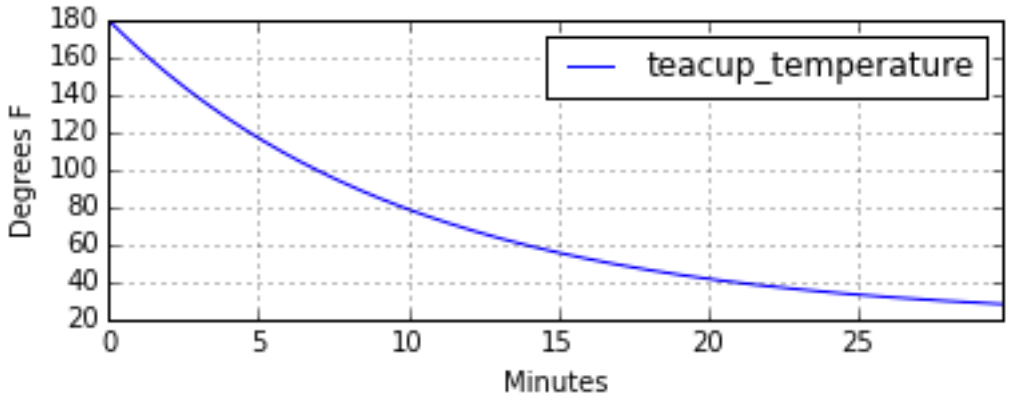



## Setting parameter values

In many cases, we want to modify the parameters of the model to investigate its behavior under different assumptions. There are several ways to do this in PySD, but the `.run()` function gives us a convenient method in the `params` keyword argument.

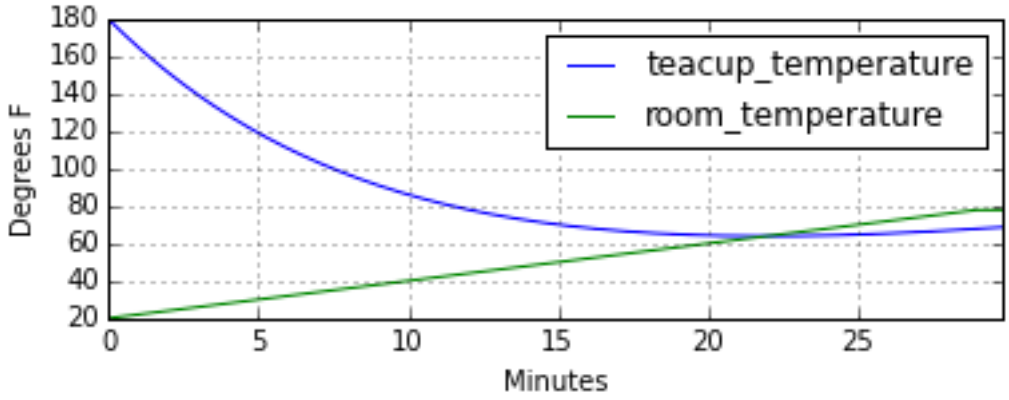
This argument expects a dictionary whose keys correspond to the components of the model. The associated values can either be a constant, or a Pandas series whose indices are timestamps and whose values are the values that the model component should take on at the corresponding time. For instance, in our model we can set the room temperature to a constant value:

```
model.run(params={'room_temperature':20})
```




Alternately, if we believe the room temperature is changing over the course of the simulation, we can give the run function a set of time-series values in the form of a Pandas series, and PySD will linearly interpolate between the given values in the course of its integration.

```
import pandas as pd
temp = pd.Series(index=range(30), data=range(20,80,2))
model.run(params={'room_temperature':temp})
```

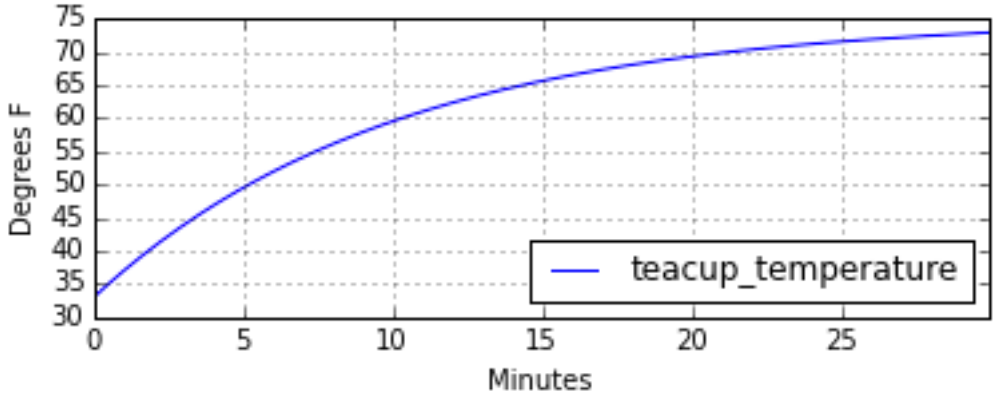



Note that once parameters are set by the run command, they are permanently changed within the model. We can also change model parameters without running the model, using PySD's `set_components(params={})` method, which takes the same `params` dictionary as the run function. We might choose to do this in situations where we'll be running the model many times, and only want to spend time setting the parameters once.

### Setting simulation initial conditions

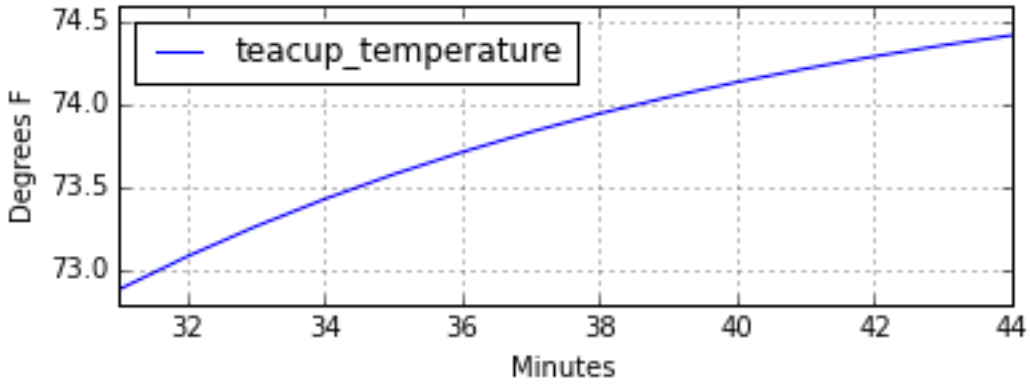
Finally, we can set the initial conditions of our model in several ways. We'll get into why this is helpful in the next section. So far, we've been using the default value for the `initial_condition` keyword argument, which is `'original'`. This value runs the model from the initial conditions that were specified originally by the model file. We can alternately specify a tuple containing the start time and a dictionary of values for the system's stocks. Here we start the model with the tea at just above freezing:

```
model.run(initial_condition=(0, {'teacup_temperature':33}))
```




Additionally we can run the model forward from its current position, by passing the `initial_condition` argument the keyword `'current'`. After having run the model from time zero to thirty, we can ask the model to continue running forward for another chunk of time:

```
model.run(initial_condition='current',
          return_timestamps=range(31,45))
```




The integration picks up at the last value returned in the previous run condition, and returns values at the requested timestamps.

### Querying current values

We can easily access the current value of a model component by calling its associated method in the components subclass. For instance, to find the temperature of the teacup, we simply call:

```
model.components.teacup_temperature()
```

### Collecting a history of returned values

The `.run()` function provides a flag named `collect` that instructs PySD to collect all output from a series of run commands into a record. This can be helpful when running the model forwards for a period of time, then returning control to the user, who will specify changes to the model, and continue the integration forwards.

The record is stored as a list of Pandas dataframes, one from each run. To access this record in its raw form, the user can access the `.record` attribute of the PySD class. It is usually more helpful to have a single dataframe which stitches together all of these pieces. We can access this via the `.get_record()` method.

## Advanced Usage

The power of PySD, and its motivation for existence, is its ability to tie in to other models and analysis packages in the Python environment. In this section we'll discuss how those connections happen.

### Replacing model components with more complex objects

In the last section we saw that a parameter could take on a single value, or a series of values over time, with PySD linearly interpolating between the supplied time-series values. Behind the scenes, PySD is translating that constant or time-series into a function that then goes on to replace the original component in the model. For instance, in the teacup example, the room temperature was originally a function defined through parsing the model file as something similar to:

```
def room_temperature():
    return 75
```

However, when we made the room temperature something that varied with time, PySD replaced this function with something like:

```
def room_temperature():
    return np.interp(t, series.index, series.values)
```

This drew on the internal state of the system, namely the time `t`, and the time-series data `series` that that we wanted to variable to represent. This process of substitution is available to the user, and we can replace functions ourselves, if we are careful.

Because PySD assumes that all components in a model are represented as functions taking no arguments, any component that we wish to modify must be replaced with a function taking no arguments. As the state of the system and all auxiliary or flow methods are public, our replacement function can call these methods as part of its internal structure.

In our teacup example, suppose we didn't know the functional form for calculating the heat lost to the room, but instead had a lot of data of teacup temperatures and heat flow rates. We could use a regression model (here a support vector regression from Scikit-Learn[20]) in place of the analytic function.

```
from sklearn.svm import SVR

regression = SVR()
regression.fit(X_training, Y_training)
```

Once the regression model is fit, we write a wrapper function for its predict method that accesses the input components of the model and formats the prediction for PySD.

```
def new_heatflow_function():
    """ Replaces the original flowrate equation
        with a regression model"""

    tea_temp = model.components.teacup_temperature()
    room_temp = model.components.room_temperature()
    return regression.predict([room_temp, tea_temp])[0]
```

We can substitute this function directly for the `heat_loss_to_room` model component.

```
model.components.heat_loss_to_room = new_heatflow_function
```

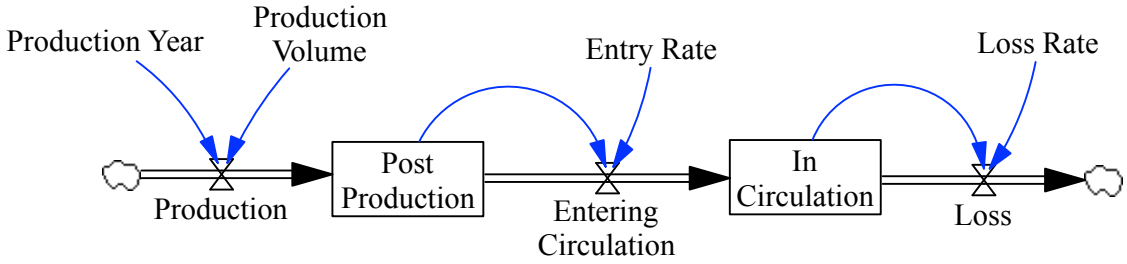
## Examples of PySD in action

In this section we'll cover some examples that use PySD to bring system dynamics models together with data. For detailed code for each of these examples, see the project homepage: https://github.com/JamesPHoughton/pysd

### Estimating the number of pennies in circulation using Markov Chain Monte Carlo, system dynamics, and a jar of pennies

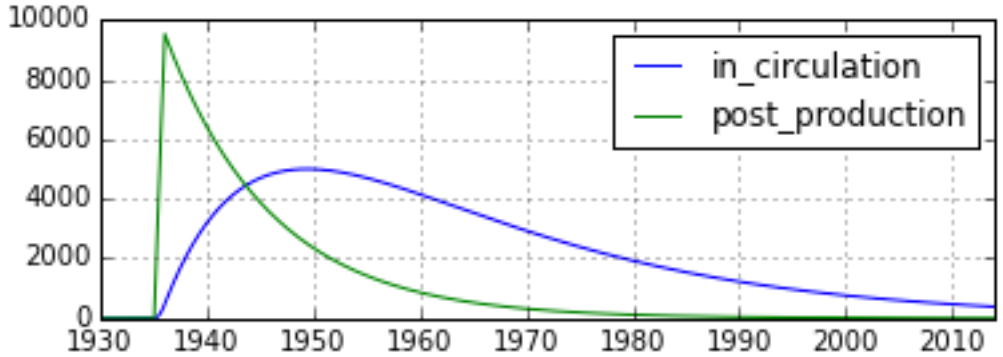
In this example we'll use a System Dynamics model to estimate the total number of pennies in circulation, based upon the number of pennies produced in a given year, and the number of pennies in a coin jar.

We start with a simple aging model for pennies, in essence a second order delay. Pennies are minted each year, and go into a stock of 'post production' pennies that are distributed to banks, etc. After this they go into general circulation, from which they are eventually lost.

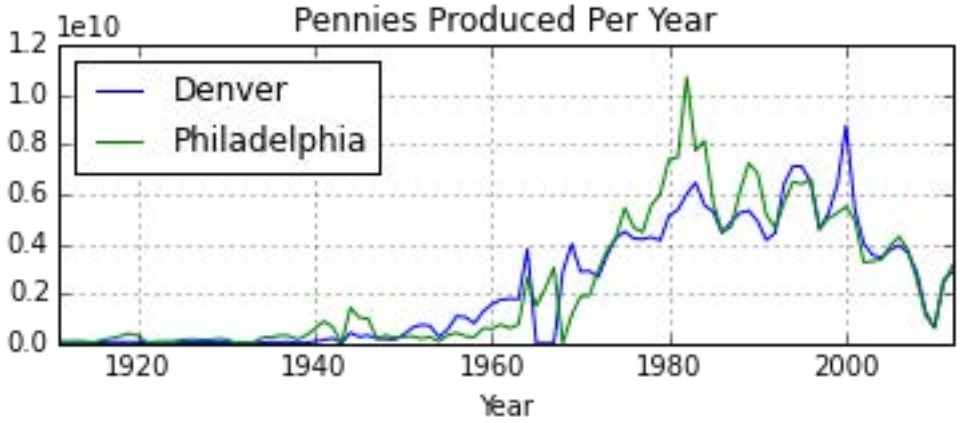



In this analysis we'll try and infer the parameters for entry into circulation and loss based upon the number of pennies produced in each year, and a random sample of pennies taken from circulation over a four-year period. An interesting component of this analysis is that it requires a whole suite of models (one for each model year) but these models don't interact with one another except through the sampling and statistical analysis we perform.

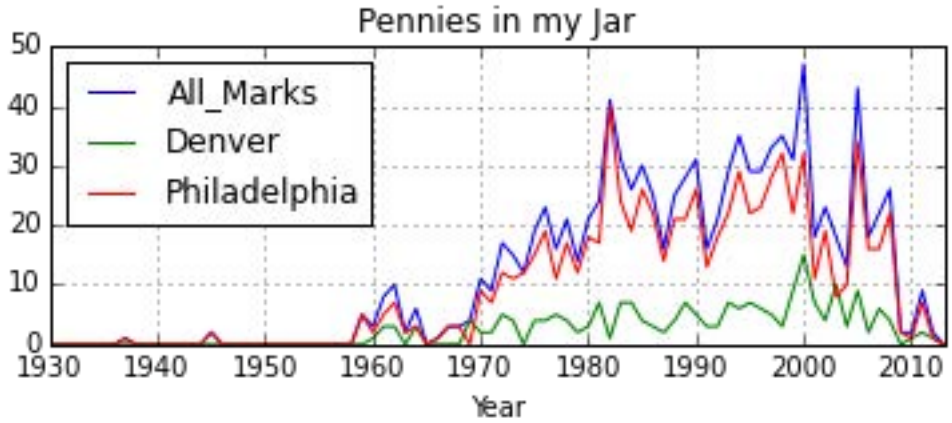
To get a sense for how the model behaves, we'll run it with arbitrary parameter values. We see that pennies enter the 'post production' stock quickly, and the 'in circulation' stock grows, peaks, and decays as the pennies get lost.



We'll start with some data about the number of coins produced in each year. We have production data for both the Denver and Philadelphia mints:




We'll also use 'data' (pennies) collected in a penny jar over the last few years:




For this analysis we'll focus just on the Philadelphia mint. We set up models for each year that pennies are produced, and initialize them with production data. We can store our models in a Pandas dataframe for convenience, along with the data. Here's a sample from the middle of the set:

| Year | model | Philadelphia Production | Philadelphia Samples |
|---|---|---|---|
| | | | |
| 1983 | Import of penny_jar.mdl | 77523.55000 | 24 |
| 1984 | Import of penny_jar.mdl | 81510.79000 | 19 |
| 1985 | Import of penny_jar.mdl | 56484.89887 | 26 |
| 1986 | Import of penny_jar.mdl | 44913.95493 | 22 |
| 1987 | Import of penny_jar.mdl | 46824.66931 | 14 |

With our models established, we are now ready to construct a Markov chain Monte Carlo simulation (MCMC). MCMC works by choosing an arbitrary value from a

distribution of input parameters, running the simulation, and asking what the likelihood of the data is given those parameters. It then decides whether to keep the selected parameters to display in an output distribution based upon this likelihood. In this demo, we'll use 'PyMC' to handle the MCMC algorithms. For an excellent primer on the use of this package, see Cam Davidson-Pilon's executable textbook 'Probabilistic Programming and Bayesian Methods for Hackers'[23].

We start then by setting up a 'prior' distribution for the loss rate and entry rate parameters that will be applied to each of the mint year models.

```
entry_rate = mc.Uniform('entry_rate', lower=0, upper=.99)

loss_rate = mc.Uniform('loss_rate', lower=0, upper=.3)
```

We'll ask our models for the population of coins from which the sample was drawn, and as this happened over a period of time, not all in the same time-step, we'll assume that there is equal likelihood that a sample was drawn (or a penny collected) any time during that window.

```
def get_population(model, entry_rate, loss_rate):

    in_circulation =
              model.run(params={'entry_rate':entry_rate,
                                'loss_rate':loss_rate},
                        return_columns=['in_circulation'],
                        return_timestamps=range(2011,2015))

    return in_circulation.mean()
```

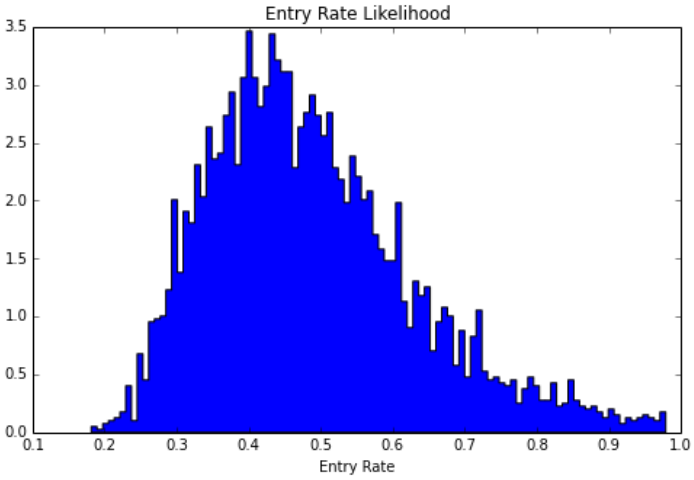
We then construct a function that returns to us the likelihood of the data given the distribution of pennies in circulation, as calculated by our model. PyMC expects this likelihood to be expressed as a log probability, to give resolution in the 'very small likelihood' regimes that our model will predict for our observations.

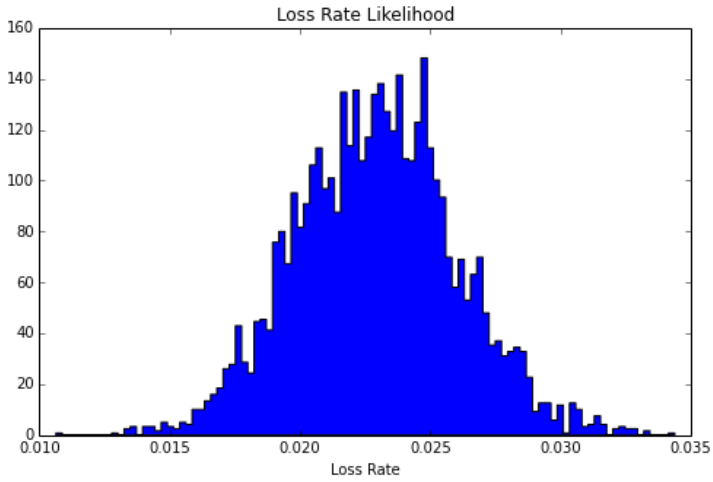
```
@mc.stochastic(trace=True, observed=True)

def circulation(entry_rate=entry_rate,
                loss_rate=loss_rate, value=1):

    mapfunc = lambda x: get_population(x, entry_rate,
                                       loss_rate)
    population = models['model'].apply(mapfunc)

    #transform to log probability and then normalize
    log_distribution = (np.log(population) -
                            np.log(population.sum()))

    #calculate the probability of the data
    log_prob = (models['Philadelphia Samples'] *
                            log_distribution).sum()

    return log_prob
```

We can now call on PyMC's algorithms to perform the MCMC calculation:

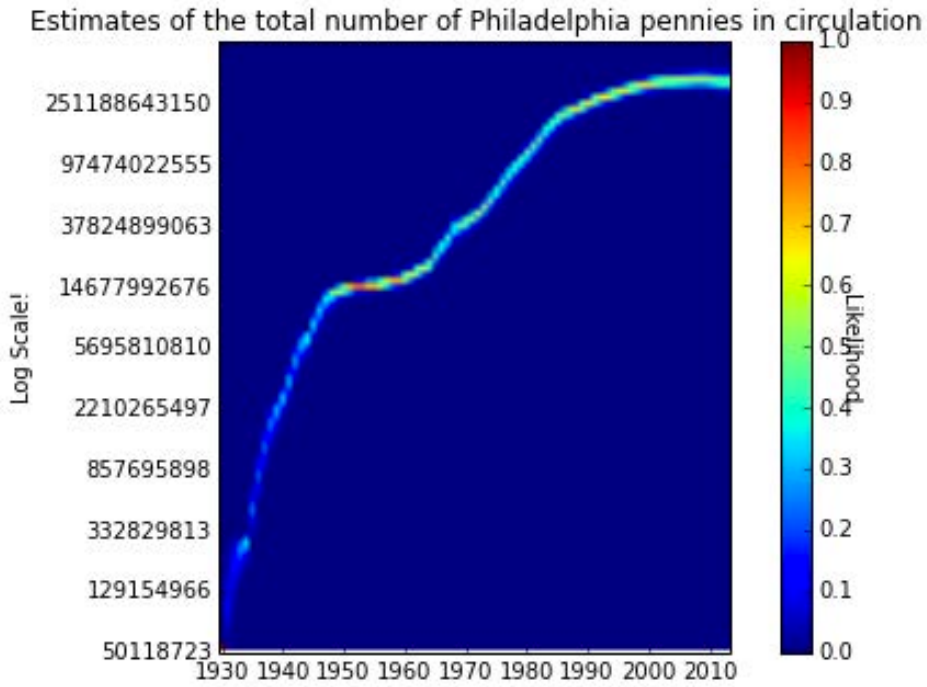
```
mcmodel = mc.Model([entry_rate, loss_rate, circulation])

mcmc = mc.MCMC(mcmodel)

mcmc.sample(20000)
```

Running the simulation, we get distributions for the `entry_rate` and `loss_rate` parameters:



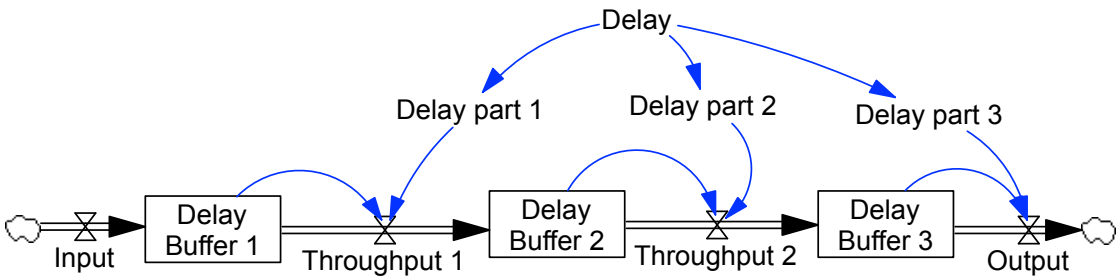

Together these let us (via regular Monte Carlo simulation) infer the number of pennies in circulation:




This predicts for 2015 about 250 billion pennies currently in circulation. It's impossible to know if our value is correct, but it is generally consistent with a US General Accounting Office estimate of 132 billion in 1996[24].

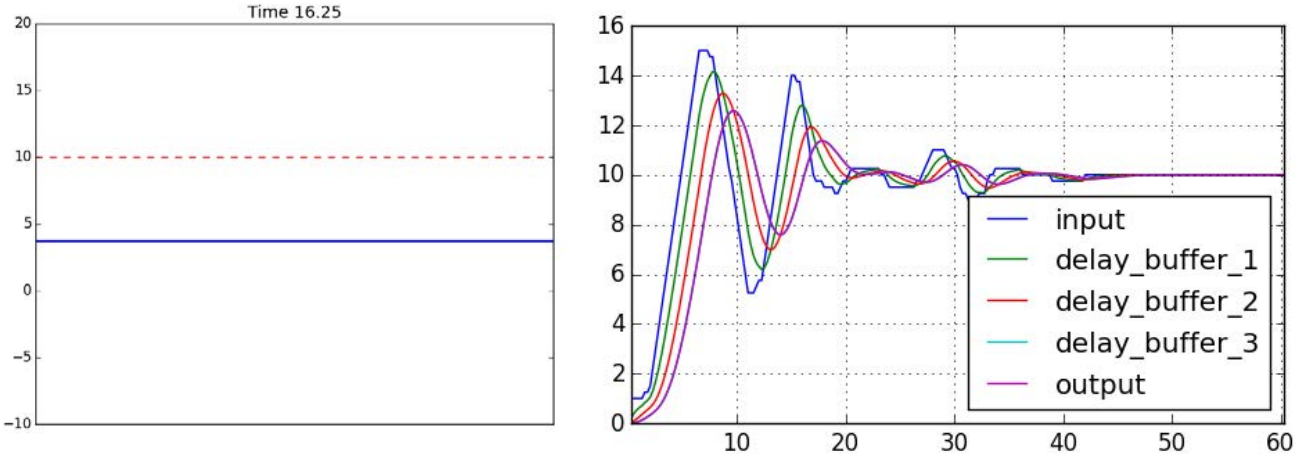
## Teaching intuition for oscillating systems with real-time simulation

In this example, we'll use PySD to join a model with data in real time. In this case, we're constructing a demo to teach students about how delays plus balancing feedback loops can yield oscillations. We begin with a simple third order delay, modeled (for pedagogical purposes) in all of its glorious detail:




We then show the student a graph such as the one on the left below, explaining that the blue line represents the 'Output' of this delay process, and their task is to press the up or down arrow keys to open or close the 'Input' valve, such that the blue 'Output' line comes to match the dashed red target. The student's input forms the last link in a goal-seeking feedback loop.

The invariable outcome is that the student overshoots and oscillates before coming to rest at the target. After the simulation, we plot their performance for them, showing the levels of each of the stocks (here on the same axis as the flows).




To make this happen, we interface PySD with the Matplotlib[18] animation module:

```
import pysd
from matplotlib import animation
```

We import the model, and set the delay parameter to our chosen value:

```
model = pysd.read_vensim('Third_Order_Delay.mdl')
model.set_components({'delay':5})
```

After setting up the plot, we instantiate a variable to serve as the state of the 'valve', and construct a function that will be executed whenever a key is pressed during the

simulation, and connect it to the plot window. This function increases or decreases the valve flowrate:

```
input_val = 1

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)
```

Now we need to connect this value to the value of the input in the model. We could do this by passing it into the `.run()` function as a static parameter at each step in the simulation, but it is simpler to just construct a single function (using Python inline function syntax `lambda`) that lets PySD check the value in each iteration.

```
model.components.input = lambda: input_val
```

Finally we have to set up the function that creates the animation. This function will be called at each frame in the animation. To render each frame of the animation, we must run the model forward by one time-step from its previous position, then plot the resulting output.

```
def animate(t):

    # run the model forward by one time-step

    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])
```

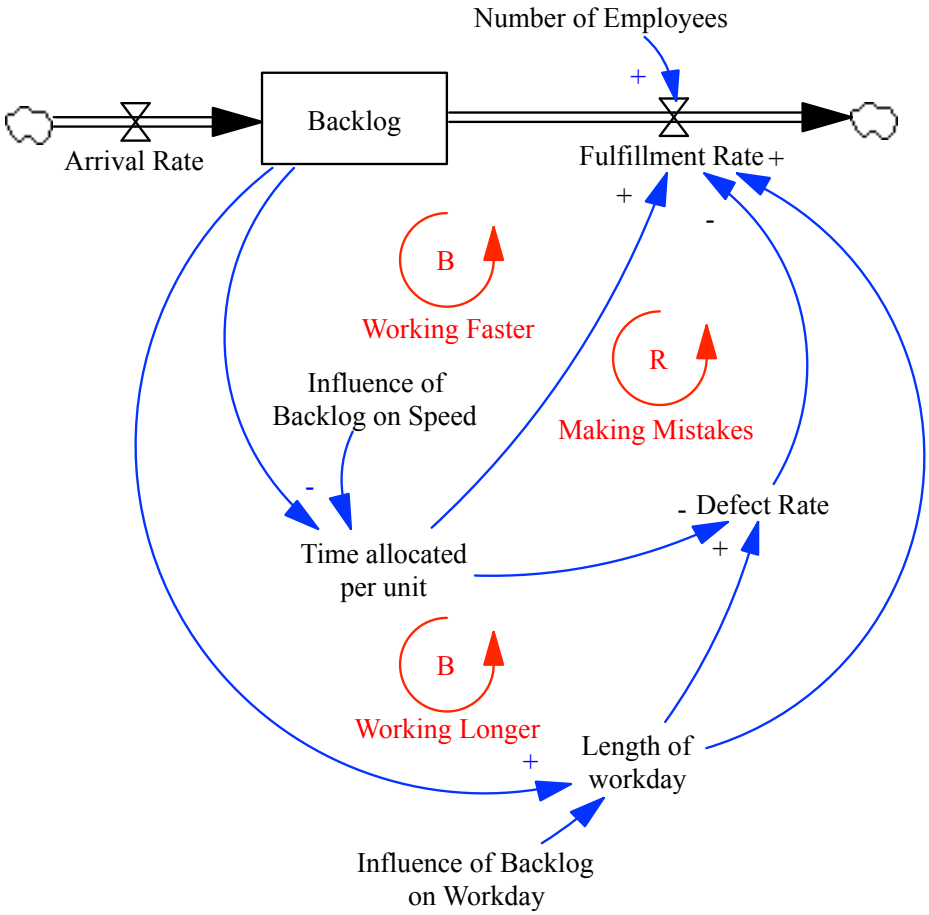
We then go on to call the animator with this function, which brings the graph to life and gives us the real-time interaction we are hoping for. We can then access the full history of the model using the `.get_record()` method, and create time-series plots.

```
record = model.get_record()
```

## Machine-learning regression in place of an equation

In the penny jar example, we executed a set of models and then compared their output with measurements. In the delay game example, we fed the model with live data from the user. This example will demonstrate the use of data and machine learning techniques to stand in for a structural equation in a model. We touched on this process earlier, in a superficial way, so lets get a little deeper into the data here.

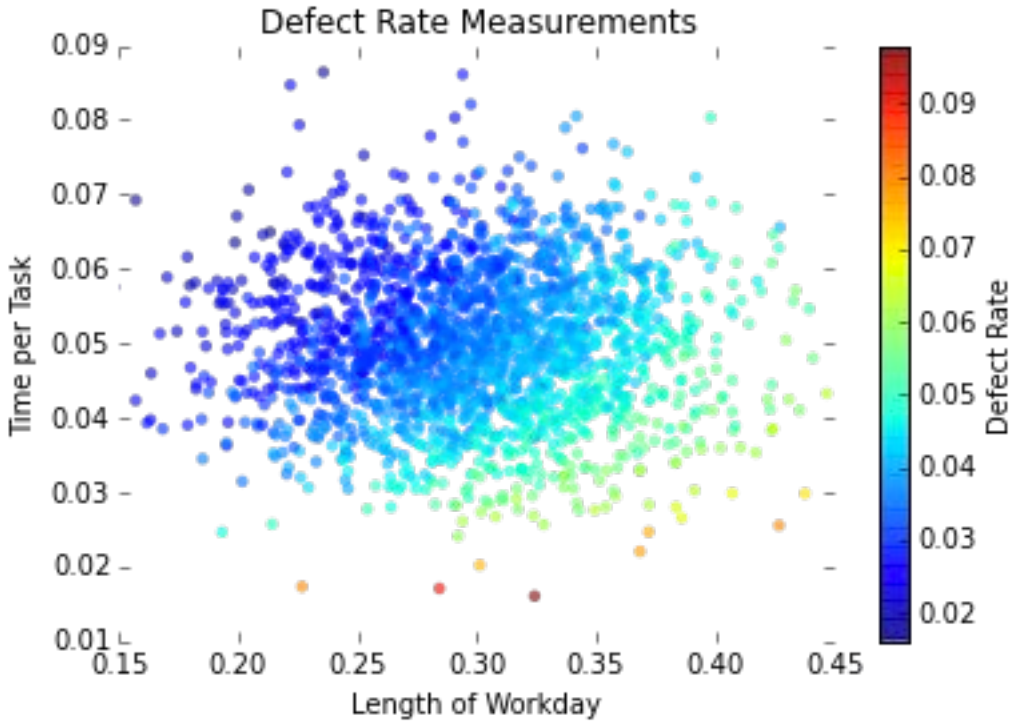
In this hypothetical case, we will look at a manufacturing firm that responds to a backlog by increasing the employee workday and pressuring employees to spend less time on each task. We note that the fraction of defective products produced by an employee during their shift depends upon both the time allocated per task and the length of the workday. We can build up a system dynamics model to investigate this feedback.




The firm collects data on workday, time per task, and defect rate for every worker for every shift:

| Workday | Time Per Task | Defect Rate |
|---|---|---|
| 0.168 | 0.039 | 0.022 |
| 0.271 | 0.033 | 0.040 |
| 0.400 | 0.042 | 0.024 |

In this case, we see that the defect rate is not a linear function of the two variables – there is some compounding effect that leads to an unknown and inseparable functional form for defect rate as a function of workday and time per unit.




To capture this in our model we can replace the equation for 'Defect Rate' with a support vector regression, from the Scikit-Learn module[20]. This ML regression utilizes the workday and time per unit as `Factors` and defect rate as the `Outcome`.

```
from sklearn.svm import SVR

regression = SVR()
regression.fit(Factors, Outcome)
```

We create a function to interface the regression with our system dynamics model, and then substitute it in for the original `defect_rate` equation.

```
def new_defect_function():
    """ Replaces the 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
```

As the next step in our modeling process, we could modify the influence that backlog has on speed and overtime to optimize a loss function based upon the cost of defective parts and of carrying a backlog.

## Future Development

The PySD tool is under development with the goal of creating (along with tools such as the EMA workbench and SDM-Doc) an integrated stack of system dynamics utilities that further the development of the field and its integration with other disciplines. This set of tools will need to attract a diverse user base and a committed group of developers who share responsibility for updating and maintenance.[1]

### Documentation and Changes

The project webpage on Github hosts updated documentation, and provides a mechanism for submitting bugs, feature requests, or other issues. Online documentation should be considered the definitive resource for use of this package, over this publication. The material presented in this document is being expanded into an online executable 'cookbook' of standard methods for integrating data and system dynamics models.

### Future direction

Future enhancements to PySD include the ability to use native Python datetime types to facilitate integration with time-series data; mechanisms for intelligently handling model units; structure for storing model parameter and run values in database backends, to facilitate model exploration; inclusion of arrays and subscripts; and support for popular macros.

### Integration with other tools

We hope to work with developers of other XMILE-stack system dynamics tools to include PySD as a standard component in a variety of different supporting tools. Components of PySD, such as the translators, are relatively modular and available for use in other projects.

### Availability and License

PySD is released under the MIT license[25], and is free to use, modify, and distribute according to that agreement. The software is available on Github at: https://github.com/JamesPHoughton/pysd, or through the Python Package Index at: https://pypi.python.org/pypi/pysd/. If you use PySD in support of published research, consider citing this paper to acknowledge that contribution.

---

[1] If you are interested in getting involved in development, contact James Houghton (Houghton@mit.edu).

## Bibliography


1. McAfee A, Brynjolfsson E. Big Data: The Management Revolution. *Harv Bus Rev*. 2012.

2. Mabry PL. Making sense of the data explosion: the promise of systems science. *Am J Prev Med*. 2011;40(5 Suppl 2):S159-S161. doi:10.1016/j.amepre.2011.02.001.

3. Pruyt E, Cunningham S. From data-poor to data-rich: system dynamics in the era of big data. *... Conf ...*. 2014. http://repository.tudelft.nl/view/ir/uuid:6ac45297-47e5-4ceb-99dc-3cc4f306a28c/. Accessed March 13, 2015.

4. Pruyt E, Kwakkel J. A bright future for system dynamics: From art to computational science and beyond. *... 30th Int Conf ...*. 2012. http://repository.tudelft.nl/view/ir/uuid:3f5b22cb-1552-4ef4-88bf-9f6d6deb14c3/. Accessed March 13, 2015.

5. Kwakkel J, Pruyt E. Exploratory Modeling and Analysis (EMA) Workbench. 2014. http://simulation.tbm.tudelft.nl/ema-workbench/contents.html.

6. Sücüllü C, Yücel G. Behavior Analysis and Testing Software (BATS). *ie.boun.edu.tr*. http://www.ie.boun.edu.tr/labs/sesdyn/projects/bats/files/Can_Sucullu_Gonenc_Yucel_BATS_Paper_ISDC_2014_Delft.pdf. Accessed March 2, 2015.

7. Martinez-Moyano I. Documentation for model transparency. *Syst Dyn Rev*. 2012. http://onlinelibrary.wiley.com/doi/10.1002/sdr.1471/full. Accessed March 2, 2015.

8. Abdel-Gawad A. Identifying dominant behavior patterns, links and loops: Automated eigenvalue analysis of system dynamics models. *Proc ...*. 2005. http://www.systemdynamics.org/conferences/2005/proceed/papers/ABDEL373.pdf. Accessed March 2, 2015.

9. Ventana Systems Inc. Vensim. http://vensim.com/.

10. isee Systems Inc. Stella/iThink. http://www.iseesystems.com/.

11. Diker VG, Allen RB. XMILE: towards an XML interchange language for system dynamics models. *Syst Dyn Rev*. 2005;21(4):351-359. doi:10.1002/sdr.321.

12. Rose E. Parsimonious. 2012. https://github.com/erikrose/parsimonious.

13. Hindmarsh A. ODEPACK, A Systematized Collection of ODE Solvers, RS Stepleman et al.(eds.), North-Holland, Amsterdam,(vol. 1 of), pp. 55-64. *IMACS Trans Sci Comput*. 1983. http://www.citeulike.org/group/2018/article/2644528. Accessed March 13, 2015.

14. PyData Development Team. Pandas: Python Data Analysis Library. 2012. http://pandas.pydata.org/. Accessed March 18, 2014.

15. The Pip Developers. Pip. 2008. http://pip.pypa.io.

16. Numpy Developers. NumPy Documentation. 2013. http://www.numpy.org/. Accessed March 13, 2014.

17. Jones E, Oliphant T, Peterson P. {SciPy}: Open source scientific tools for {Python}. 2014. http://www.citeulike.org/group/19049/article/13344001. Accessed February 18, 2015.

18. Hunter J, Dale D, Firing E, Droettboom M, The Matplotlib Development Team. Matplotlib: Python Plotting — Documentation. *2013*. http://matplotlib.org/. Accessed March 13, 2014.

19. Fonnesbeck CJ. PyMC User's Guide. *2012*. http://pymc-devs.github.io/pymc/. Accessed March 8, 2014.

20. Pedregosa F, Varoquaux G. Scikit-learn: Machine learning in Python. *J Mach …*. 2011. http://dl.acm.org/citation.cfm?id=2078195. Accessed March 2, 2015.

21. Hagberg A, Schult D, Swart P. Networkx. High productivity software for complex networks. 2013. https://networkx.github.io/. Accessed March 2, 2015.

22. GeoPandas developers. Geopandas. 2013.

23. Davidson-Pilon C. *Probabilistic Programming and Bayesian Methods for Hackers.*; 2013. http://camdavidsonpilon.github.io/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers/.

24. Gadsby JW. *FUTURE OF THE PENNY Options for Congressional Consideration*.; 1996.

25. The MIT License (MIT). *Open Source Initiat*. 1988. http://opensource.org/licenses/MIT. Accessed February 18, 2015.

26. Magoulas R, King J. *2013 Data Science Salary Survey: Tools, Trends, What Pays (and What Doesn't) for Data Professionals*. O'Reilly; 2014. http://www.oreilly.com/data/free/stratasurvey.csp.

27. McKinney W. *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*.; 2012. https://books.google.com/books?hl=en&lr=&id=JtJAkfzds4wC&oi=fnd&pg=PR3&dq=Python+for+Data+Analysis+Data+Wrangling+with+Pandas,+NumPy,+and+IPython&ots=pQsLxDUV4e&sig=HSj4Yodi5-tEePOHaqyvZVchG3w. Accessed February 23, 2015.

28. Rossant C. *IPython Interactive Computing and Visualization Cookbook*.; 2014. https://books.google.com/books?hl=en&lr=&id=yZafBAAAQBAJ&oi=fnd&pg=PT31&dq=IPython+Interactive+Computing+and+Visualization+Cookbook&ots=vVj6XRJ-JZ&sig=OfBGioP_WnkVCZ_N6Uq5sIwqAMc. Accessed March 13, 2015.

29. McKinney W. 10-minute tour of pandas. 2013. https://vimeo.com/59324550. Accessed March 13, 2015.

30. Russell M. *Mining the Social Web: Data Mining Facebook, Twitter, LinkedIn, Google+, GitHub, and More*.; 2013. https://books.google.com/books?hl=en&lr=&id=_VkrAQAAQBAJ&oi=fnd&pg=PR4&dq=Mining+the+social+web&ots=JqkwmFTxoK&sig=tnv-AHAc1vrwwgzZSCOVBxYZI68. Accessed March 13, 2015.

# Appendix

## Supported functions and their Python equivalents

As of this publication, PySD supports the following commands:

| Vensim | Python |
|---|---|
| lognormal | np.random.lognormal |
| modulo | np.mod |
| poisson | np.random.poisson |
| arcsin | np.arcsin |
| max | max |
| <= | <= |
| pulse | functions.pulse |
| < | < |
| step | functions.step |
| exprnd | np.random.exponential |
| integer | int |
| inf | np.inf |
| tan | np.tan |
| random uniform | np.random.rand |
| if then else | functions.if_then_else |
| cos | np.cos |
| random normal | functions.bounded_normal |
| min | min |
| ln | np.log |
| = | == |
| pulse train | functions.pulse_train |
| ramp | functions.ramp |
| sqrt | np.sqrt |
| arctan | np.arctan |
| abs | abs |
| >= | >= |
| exp | np.exp |
| arccos | np.arccos |

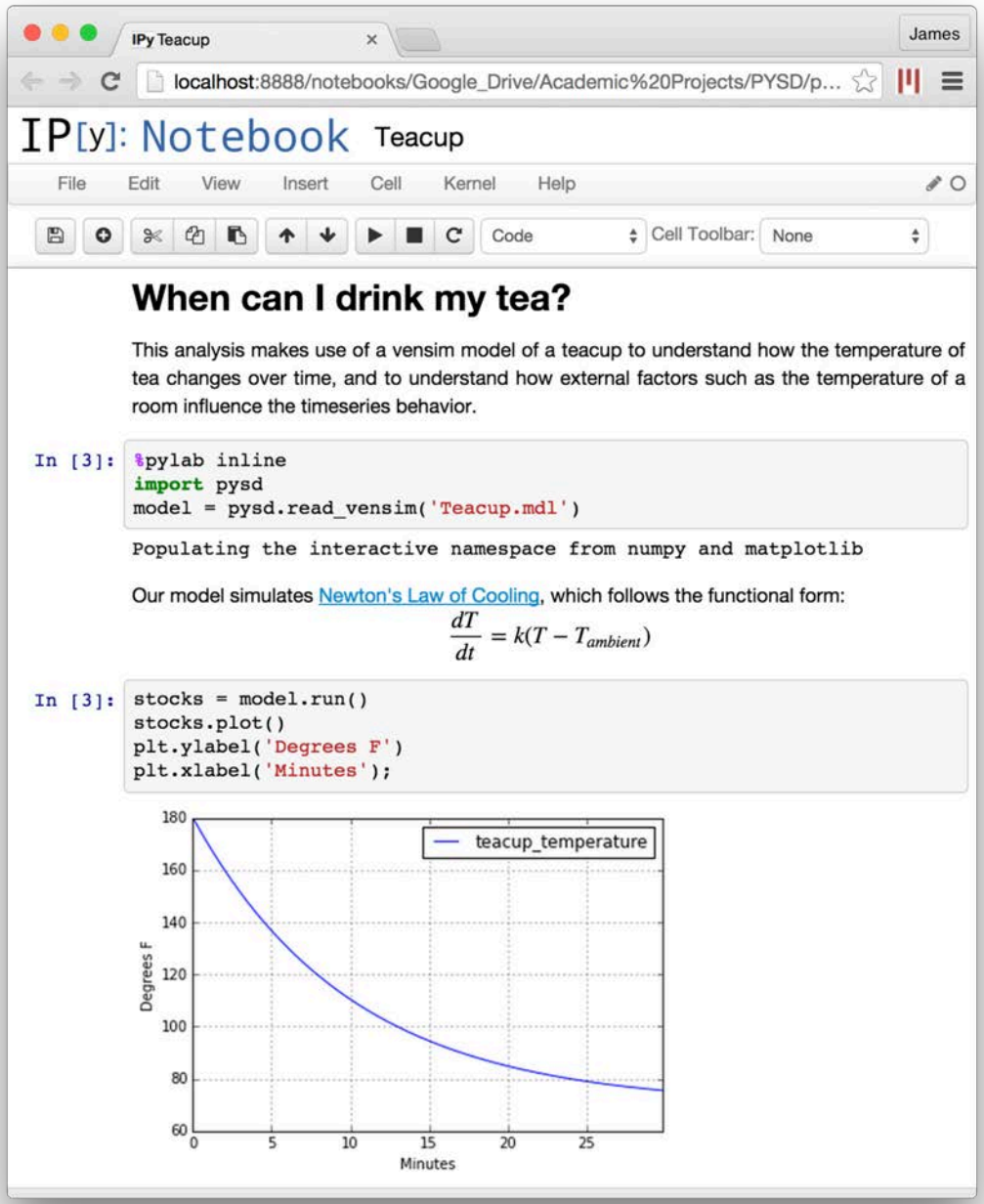
## Why Python?

There are a number of different environments used in data science, with more or less well-developed sets of supporting software, including Python, Matlab, R, Tableau, Java, Javascript, SAS/SPSS, and others[26]. The choice to develop this tool using Python represents the consideration of a number of factors, in approximate order of importance: open source accessibility, maturity of data science toolbox,

existing use within the system dynamics field, established user base in the data science community, flexibility for a variety of data collection and display tasks, modern programming constructs, and computational power.

## Working with the iPython Notebook

One of the better user interfaces for performing scientific analyses in Python and with PySD is the iPython Notebook. The Notebook provides a simple way to incorporate code, documentation, equations, references, and execution output in a document-like environment. This layout makes constructing and communicating analyses very straightforward. Notebooks can be shared using an online viewer available at [http://nbviewer.ipython.org/.](http://nbviewer.ipython.org/)



## How PySD cleans variable names

PySD allows variable names that start with a letter, and then contain some number of letters, numbers, dollar signs, and spaces. PySD converts all letters to lower case, and substitutes underscores for all spaces. Thus:

| Model Identifier | Sanitized Python |
|---|---|
| `Interest Rate` | `interest_rate` |
| `Return %` | **-not allowed-** |
| `Avgerage $s per Sale` | `Average_$s_per_sale` |

## Resource List

**Python Data Analytics Stack**

- Python for Data Analysis[27]: Wes McKinney
- Ipython Interactive Computing and Visualization Cookbook[28]: Cyrille Rossant
- Probabilistic Programming and Bayesian Methods for Hackers[23]: by Cam Davidson-Pilon
- 10-Minute tour of Pandas[29]: Wes McKinney
- Mining the Social Web[30]: by Matthew Russel

## Thanks

Special thanks to Stuart Madnick, Allen Moulton, David Keith, Hazhir Rahmandad, Eliot Rich, David Andersen, Luis Luna-Reyes, Robert Ward and David Weiss for their feedback and help with this project.