AUTOMATION OF COMMON BUILDING ENERGY SIMULATION WORKFLOWS USING PYTHON

Clayton Miller[1], Christian Hersberger[1], and Marcus Jones[2]
[1]Architecture & Sustainable Building Technologies (SuAT),
Institute of Technology in Architecture (ITA), ETH Zürich
{miller.clayton, hersberger}@arch.ethz.ch
[2]EXERGY Studios s.r.o., Vienna University of Technology, marcus@exergystudios.com

ABSTRACT

A valuable skillset for building industry professionals is proficiency in high-level, scripting languages that can automate and perform many common repetitive or technically intensive tasks. This application-focused paper emphasizes the use of the Python programming language in various workflows common to the building performance modeling and simulation process. Python is an open, powerful, and easy-to-learn scripting language with an emphasis on programmer productivity. While the highlighted applications themselves are notably ordinary amongst building simulation practitioners, the novelty of this discussion is in the speed and usefulness of new Python libraries and data analysis techniques. Four short examples are illustrated: simulation input file templating, data exchange and interoperability, performance curve regression, and time-series output data postprocessing. An overview is presented of the growing current and planned Python libraries, extensions, and projects that are especially applicable and, in some cases, explicitly designed for the building industry.

INTRODUCTION

A majority of professionals in the building analysis industry are trained in the fields of mechanical or electrical engineering or architecture. The educational path for these disciplines has not traditionally included extensive training in computer science. A building simulator's toolkit is often limited to their simulation engine of choice, perhaps a graphical user interface (GUI) to create and manipulate input, and a spreadsheet program such as Microsoft Excel to interpret the raw output data into value. While these tools may be enough for the casual user, more powerful and flexible methods are needed for the dedicated simulation professional, especially those in research and development. Complimentary programs, such as the EnergyPlus Auxiliary Program library, are often created in order to automate certain specific processes, however, these solutions are static in their the ability to adapt to different users' custom needs (EnergyPlus, 2012). Much of the analysis in the field can be ad-hoc and more flexible and robust methods would enable a professional to understand and control how the data is being transformed.

Traditionally, if a building simulation expert were to learn a programming language, it would be generally a lower level, compiled language like Fortran or C. These languages are designed for application and system programming and a significant amount of effort and training is required to become proficient. The necessity of learning such languages has continued due to their execution speed in simulation engines and due to the fact that many legacy simulation programs are written in such languages. In the 1990's, many professionals also started utilizing spreadsheet-integrated scripting languages like Visual Basic for post-processing and rudimentary user interfaces for simplified tools.

Language-centric Solutions Using Python

The last twenty years of computer science has produced numerous high-level, scripting languages which were designed with a focus on human-readable syntax and ease of use. These languages may sacrifice memory or execution speed; however, the increase in coder productivity is remarkable, especially for entry-level programmers. Python, Ruby, and Lisp are examples of this genre. This paper outlines the use of the Python programming language in the building performance simulation and analysis context due to its popularity amongst computational scientists and the research community. The goal of this analysis is to outline the advantages of building simulation professionals in becoming proficient in a high-level scripting language and to provide guidance on the potential applications of this skill set.

Advantages of Python

Python is an *interpreted* and *dynamically typed* programming language that was first released by Guido van Rossum in 1990 and has since grown into one of the most popular programming languages with widespread use in research, commercial, and online environments.

An interpreted language, contrasting to a compiled language like C, does not create a machine executable code and is instead executed "on-the-fly". Interpreted languages often are better for fast code prototyping and platform independence for distribution of code. A dynamically typed language is one in which a variable can be reassigned to a different data type later in the code (such as a string being assigned to a variable name previously used as an integer). Furthermore, the variable type in Python does not need to be explicitly declared beforehand. Languages such as Fortran or C require explicit variable name declarations and and type assignments. It should be noted that although the advantages of interpreted and dynamically typed languages are still the subject of debate among computational scientists, it is the opinion of the authors that these features can lead to faster learning and implementation for a building simulation expert.

Python syntax is generally differentiated through its use of white-space, clarity, and expressiveness. It is a multi-paradigm language that can be used in object-oriented, functional, or imperative programming styles. Python furthermore has a number of general features which have contributed to its rapid growth (Oliphant, 2007):

- Numerous, well-developed library modules of scientific, statistics, engineering, and web-related extensions
- A large, enthusiastic Python community which provides responsive online support, documentation, and tutorials
- Strong emphasis on clear, human-readable, simple syntax designed to promote coder productivity and enjoyment
- Robust and easy extendability to users who want to create a domain-specific library module

Python's adaptability is apparent in its success as both a quick, impromptu scripting language for beginners and as a complete solution for large, internet companies; much of the systems that power companies like Dropbox, Google, and YouTube are written in Python. It is for the above reasons that the authors advocate the use of the Python language as an essential tool for building simulation professionals looking to increase robustness and productivity in their workflow.

Application to Building Performance Simulation

Building performance modeling and simulation is a growing scientific field focused in digitally reproducing the physical phenomena occurring in the built environment. As the field has progressed, software tools have become more complex and detailed and often studies will include hundreds or thousands of simulations. Many variations exist in the way simulation studies are conducted, however there is a general workflow structure that is common amongst most studies: a collection of characteristic 'meta' data about a building is compiled in an input format, this structured data is loaded into a simulation routine or engine, and the output results are extracted at the end of the process for analysis. This general workflow is illustrated in Table 1; it includes various referenced examples of studies that illustrate each general process category.

Furthermore, many simulation tools are derived from a research background within a small community, and lack the polish of commercial software. Given the complexity of many models and the number of assumptions that go into a simulation, the workflow of energy modeling and simulation can be described by Figure 1 as an iterative cycle where the model is continuously adjusted to correct errors and test assumptions.

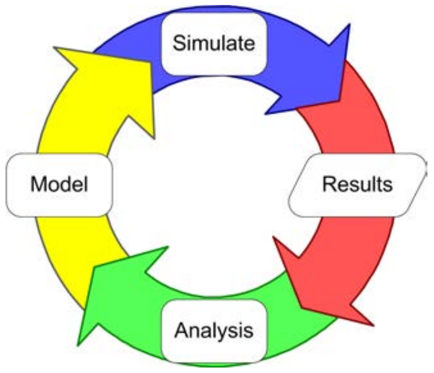


Figure 1: Wheel of Simulation (Ledinger and Jones, 2010)

The iterative nature of the simulation workflow can therefore further leverage any step which is automated using Python scripting, saving time each repetition.

Each of the categories from Table 1 will now be illustrated through four short characteristics examples. The first three topics focus on the pre-processing of input 'meta' of a building into the simulation: input file templating, data exchange, and performance curve development. The last example outlines the use of Python data analytics functions and libraries which empower fast simulation data output analysis and visualization.

INPUT FILE GENERATION

A fundamental component of the simulation workflow is the structured definition of the input characteristics of the building scenario to be examined through the calculation. Most major simulation engines have at least one graphical user interface in which the user will specify fields such as geometry, HVAC systems, weather data, and desired output variables.

The simulation needs of researchers and some professionals interested in parameterized building design optimization now also dictate that hundreds or even thousands of building simulation scenario inputs need to be created and organized. Although some GUI's are starting to include dedicated features related to parametric analysis, the setup of parametric studies is still very time-consuming and often cumbersome due to the limited possibilities of the macro languages used to calculate the values for the different parametrized simulation runs and the constraints in their algebraic and logic expressions. In the worst case scenario a professional would manually modify each input file. However, an alternative is to use the powerful text query and manipulation capabilities of the Python lan-

Table 1: General building simulation workflow and possible applications for utilizing high-level languages.

Preprocessing:	**Simulation Processing:**	**Postprocessing:**
Description of physical building properties, technical systems and boundary conditions	Series of transformational equations to simulate desired phenomena	Generally time- series output performance data
	Building Simulation Workflow →	
Examples:	*Examples:*	*Examples:*
Input file parametrization and templating for early stage design optimization (Schlueter and Thesseling, 2009)	Sensitivity analysis and simulation driver for model calibration (Bertagnolio and Andre, 2010)	Output data visualization (Raftery and Keane, 2011)
Creation of input files from other structured data sources exported from Building Information Models (BIM)	Object-oriented function libraries for encapsulating modeling characteristics such as fluid properties (Bell, 2013), (Bell et al., 2012)	Statistical analysis of output time series data (Reddy, 2011)
Performance curve generation for data-driven models used as input to whole building engines (Zhou et al., 2008)	Complex co-simulation coordination between different engines (Jones and Ledinger, 2010)	

guage to generate the parametrized simulation files. The value of using scripting in these scenarios is often proportional to the number of simulation runs that need to be completed. Apart from studies, where just numerical values of certain key parameters are altered, Python can also be used to simplify repetitive source code editing tasks; this will be illustrated in the next section on the example of HVAC system source code in EnergyPlus.

Input File Templatization

Source code for HVAC systems in EnergyPlus can become very large and cluttered. With all zone equipment, relevant thermal nodes, control algorithms, schedules and so forth it may easily consist of a few hundred lines of code per thermal zone. Whereas for small projects the source code can be generated with copy and paste and then adapted for each zone, this method will become nearly impossible for larger simulations with dozens or hundreds of thermal zones. As most of the code will be nearly identical for all zones it is hard to keep track of the changes and errors are very likely to occur.

Although EnergyPlus provides a macro language that allows automated source code generation, not everything can be handled with it: especially the linking of user-defined controls for the technical systems, and also ducts for air-based systems or pipes into water loops is difficult to automate with the engine's on-board means. Solutions that use EnergyPlus macros in combination with external scripts to generate the (HVAC system) source code for a set of predefined HVAC systems and combinations per zone exist (Raftery, 2012), but it is possible to go even further. First, Python scripts can be used to instantiate the templates for the equipment that is used per zone (ie: different types of air handling units, radiators, low-temperature floor heating, radiating ceiling panels, convectors). The required configurations and parameter values for the systems are taken from a spreadsheed or configuration text file such as gbXML. The next step is the automated linking of the hot and chilled water consumers into the corresponding water loops. Then ducts and air-based systems are linked as well as the source code snippets that control the technical systems. Figure 2 demonstrates an example workflow of the process. The use of Python for source code generation allows great flexibility and expandability: templates can be easily added for new zone equipment; the user only needs to define heating and cooling priorities for the equipment and specify how controls and air and water-based systems have to be linked into superior structures such as control sequences and air or water loops. It is the authors' experience that the usage of these types of Python scripts for simulation input file generation simplifies the process, speeds up repetitive code editing tasks and reduces error rates remarkably.

INPUT PARSING & INTEROPERABILITY

Source input files for energy simulation tools (energy model files) are often large and unwieldy. Many tools such as EnergyPlus, TRNSYS, Radiance, and ESP-r store the model in a plain text file. This approach has the benefit of being able to quickly edit the model of even parameterize the model, as discussed in the previous section. A model can contain a large of amount of information from different sources such as product catalogs, mechanical and architectural schedules, and other documents prepared by designers and contractors. A key task in energy modeling is converting this information into the syntax of the modeling and simulation tools. Using a scripting language such as Python allows users to avoid data entry errors and change a task that could consume hours of copy and pasting into one that takes milliseconds. As with all scripting projects, the modeler must consider the in-

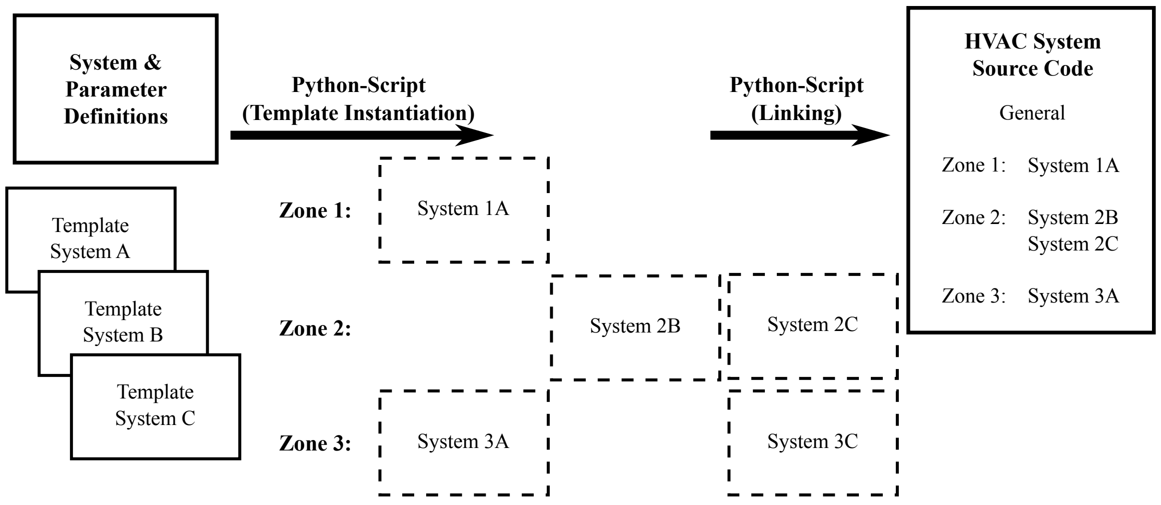


Figure 2: Input file templating workflow

vestment required to understand and create the script versus the time saved by using it in the future.

Application

Due to the history of simulation tools, the syntax of the modeling language is often customized to the specific tool. For example, Listing 1 is a pump as modeled using the TRNSYS 17 Simulation Studio. TRNSYS has a long history, and was originally programmed using Fortran punch cards. The main input file of a TRNSYS model is still called a "'deck file"', referring to a deck of punch cards.

The difficulty in parsing a language is related to the number of elements in the grammar, and the number of rules and exceptions required to define the language. Using this definition, we can see that the TRNSYS modeling language has rules such as "IF '*' character THEN (IF following '$' character THEN use in studio display, ELSE ignore this comment)" and "**IF** 'PARAMETERS N' string, **THEN** read in next 'N' lines as parameters", and many more rules.

*Listing 1: TRNSYS *.DCK*

```
UNIT 3 TYPE 3    Type3b
*$UNIT_NAME Type3b
*$MODEL .\Hydronics\Pumps\Type3b.tmf
*$POSITION 56 192
*$LAYER Water Loop #
PARAMETERS 5
50        ! 1 Max flow rate
4.19        ! 2 Fluid specific heat
60        ! 3 Maximum power
0.05        ! 4 Conversion coefficient
0.5        ! 5 Power coefficient
INPUTS 3
1,2        ! Inlet fluid temperature
1,3        !  Inlet mass flow rate
9,1         ! Type14h:Average value
*** INITIAL INPUT VALUES
20 40 1
```

In scripting, the processing of breaking down information is known as *parsing*. Parsing depends on breaking down the *language* of *strings* over a *grammar* and the languages *syntax*. A grammar defines the acceptable "words" of the language, and a syntax describes how the words are put together, just as in human languages such as English (Sipser, 2006). The TRNSYS grammar and syntax contains redundant information and is difficult to parse, compared to say the EnergyPlus modeling language as presented in listing 2, also for a pump. In EnergyPlus a model in the language is defined by a sequence of model attributes separated by a comma, terminated by a semicolon, and anything following an exclamation mark is ignored as a comment.

*Listing 2: EnergyPlus *.IDF*

```
Pump:ConstantSpeed,
  ChW Circ Pump, !- Name
  ChW Supply, !- Inlet
  ChW Pump, !- Outlet
  autosize, !- Rated Flow {m3/s}
  17.9, !- Head {Pa}
  autosize, !- Power Consumption {W}
  0.9, !- Motor Efficiency
  0.0, !- Fraction of Motor Inefficiencies
  INTERMITTENT; !- Pump Control Type
```

This format is less complex, however can it be done in a better way?

Extensible Markup Language (XML)

Extensible Markup Language (XML) is a programming language that is designed to be robust as a data storage language, which is easily as human and machine readable. Many readers will be familiar with HTML, hyper text markup language, which is similar to XML. XML is ideal for manipulating the type of data that is stored in building simulation software input files. XML, with an appropriate parser, can be used to convert between formats, verify files against a required format (validation), and perform complex searching and manipulation operations. Fortunately, there are many fast and well known parsing engines available, since XML forms such an integral part of contemporary data analysis. XML has significant advantages as a modeling language including:

1. **Validation** is well supported, and could be used to check for values in the correct range, units, connections, etc.
2. **Transformation** into different formats is possible using the XSLT Stylesheet Language pipeline.
3. **Analyzing** a model can be achieved elegantly using the XPATH query language.

These benefits also motivate the gbXML initiative (The Open Green Building XML Schema, 2012). A representation of the EnergyPlus pump model, parsed into XML, is presented in listing 3. Names have been shortened for display purposes.

*Listing 3: EnergyPlus *.XML*

```
<OBJECT>
  <CLASS>Pump:ConstantSpeed</CLASS>
  <ATTR desc="Name" units="">ChW Circ Pump</ATTR>
  <ATTR desc="Inlet" units="">ChW Supply</ATTR>
  <ATTR desc="Outlet" units="">ChW Pump</ATTR>
  <ATTR desc="Flow" units="m3/s">autosize</ATTR>
  <ATTR desc="Head" units="kPa">17.9</ATTR>
  <ATTR desc="Power" units="W"> autosize</ATTR>
  <ATTR desc="Efficiency" units="-">0.9</ATTR>
  <ATTR desc="Inefficiencies" units="-">0.0</ATTR>
  <ATTR desc="Control" units="">INTERMITTENT</ATTR>
</OBJECT>
```

In Python, there are many supported modules to add XML parsing, including a standard libary module that comes preinstalled as part of Python. In this paper, the authors employ the 'lxml' module (lxml Dev Team, 2013). Using the XPATH language in XML, sophisticated queries can be constructed. Although the XPATH query syntax also requires experience to use, it can be worth the effort if many more complex queries are required. In listing 4, a complex operation is carried out on an arbitrarily large IDF/XML file using the lxml module and Python. In this code example, all `Pump:ConstantSpeed` objects are selected which have `INTERMITTENT` control. The pump `Head` is changed to `9.5` kPa for all of these objects.

Listing 4: Parsing an EnergyPlus XML file

```
from lxml import etree
model = etree.parse('ePlus_model.xml')
s = "//CLASS[text()='Pump:ConstantSpeed']/.."
constantPumps = model.xpath(s)
for pump in constantPumps:
   s = "//ATTR[@desc='Control' and " /
       "text()='INTERMITTENT']"
   intermittent = pump.xpath(s)
   if intermittent:
      s = "//ATTR[@desc='Head']"
      pumpHead = intermittentPump[0].xpath(s)
      pumpHead[0].text = "9.5"
```

In this example, the query was broken down into multiple steps to illustrate the process. First the pumps are selected, then filtered for the `INTERMITTENT` control attribute, and finally updating the value. However, this query can be executed in one step. Another application could be selecting all exterior surfaces of a certain material facing due north, and updating the exterior solar reflectance to determine the effect of a thermal paint. The applications are limitless, and as the user gains familiarity with XML, utilities can be created to expedite the creation of these queries.
Using Python and XML, model data can be populated, translated, and analyzed efficiently and robustly.

PERFORMANCE CURVE REGRESSION

Most building simulation engines are considered 'grey-box' models in that they are driven by both physical, equation-based as well as statistical, data-driven calculation methods. The performance of equipment such as chillers, pumps, and unitary conditioning units is often characterized with a performance curve equation that is regresssed from manufacturer-provided performance data. This curve is an input object to the simulation and is an example of a data-driven component of the building simulation model. These coefficients of these curves are often provided directly by the manufacturer, however newer systems such as the variable refrigerant volume (VRV) system must be calculated by the simulation user. A guide for creating VRV performance curves for the EnergyPlus simulation engine is available and this section will describe implementation using Python and its available scientific libraries, Scipy and Numpy.

VRV Performance Curve Calculation

The VRV performance curve guidelines describe the workflow of extracting the 22 potentially necessary performance curves for simulating VRV systems in EnergyPlus (Raustad, 2012). A short, step-by-step example of regressing the VRV Cooling Capacity Ratio Modifier Performance Curve (CAPFT) is presented in this section.
The first step is to extract the appropriate manufacturer performance data for the desired system simulation. In this example we will use data from a Daikin VRV system which was obtained from a local Daikin distributor. The performance documentation is provided in a PDF format; the performance information must be copied manually or an automated PDF to CSV converter program may be used. A Python library known as PDFMiner is applicable to this task. The data necessary for the CAPFT regression are the instantaneous Cooling Capacity Ratio (CapRatio), Indoor Wet Bulb (IWB), Outdoor Dry Bulb (ODB) for each tested rating condition. The CAPFT function is designed to represent the change in capacity of the system as outdoor and indoor conditions vary:

$$\dot{Q}_{cool,available} = \dot{Q}_{cool,rated} \cdot CAPFT_{HP,cooling} \tag{1}$$

The CAPFT weighting is calculated by a biquadratic performance curve that utilizes zone cooling coil inlet wet-bulb temperature ($T_{wb,avg}$) and outdoor air dry-bulb entering the condenser (T_c):

$$\begin{aligned} CAPFT_{HP,cooling} &= a + b \cdot T_{wb,avg} + c \cdot (T_{wb,avg})^2 \\ &+ d \cdot T_c + e \cdot (T_c)^2 + f \cdot T_{wb,avg} \cdot T_c \end{aligned} \tag{2}$$

The performance curve coefficients (a to f) need to be regressed from the manufacturer's dataset.

In order to use Python to calculate coefficients, we use the Numpy and Scipy libraries. Numpy is an extension that adds support for multi-dimensional arrays and Scipy contains modules for optimization, linear algebra, integration, interpolation, and ordinary differential equation solvers in addition to many other mathematics and scientific functions (Jones et al., 2001) Given that IWB, ODB, and CapRatio are Python list objects containing the extracted manufacturer data, the coefficients can be regressed using the following Python code by first loading Numpy and Scipy libraries:

```
from scipy.optimize import leastsq
import numpy as np
```

Then Numpy arrays are formed with the raw data and an 'inital guess' list is created for input into the Least Square algorithm:

```
npIWB       = np.array(IWB)
npODB       = np.array(ODB)
npCapRatio = np.array(CapRatio)
p0=[0,0,0,0,0,0]
```

The Least Square regression function from the Scipy optimization library is called and the output is a Numpy array of the regressed coefficients (CAPFT):

```
CAPFT = leastsq(residualsTwoDimBiquadratic,
    p0,args=(npCapRatio,npIWB,npODB),
    full_output=1)
```

The residualsTwoDimBiquadratic functions used as an argument in the previous code is defined as:

```
def residualsTwoDimBiquadratic(p,I,A,B):
    return (I - TwoDimBiquadratic(A,B,p))

def TwoDimBiquadratic(A,B,p):
    return(p[0] + p[1]*A + p[2]*A**2
                + p[3]*B + p[4]*B**2
                + p[5]*A*B)
```

In this example, the output is a list of the fitted coefficients:

```
[-1.53742235e+00   1.98712839e-01
 -3.34991139e-03   1.77684674e-02
 -5.67708040e-05  -9.27353971e-04]
```

Inserting the calculated coefficients back into the biquadratic curve and plotting using the matplotlib library gives us a scatterplot representation of the capacity modifier curve as seen in Figure 3.

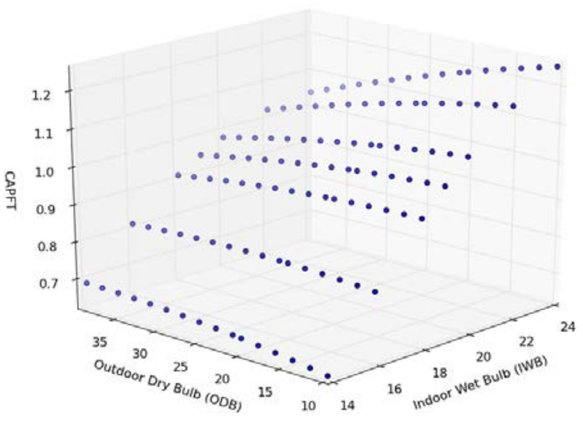


Figure 3: 3D Scatterplot of Capacity Modifier Function

POSTPROCESSING APPLICATIONS

The amount of output data generated from a single simulation engine can be as little as a single whole building metric to as much as a virtual sensor network of thousands of simulated measurements down to the seconds level. Processing and utilization of this data depends primarily on the simulation application; rating systems documentation may simply need a high-level snapshot of specific metrics while more intense research applications could be statistically comparing hundreds of simulated time-series datasets simultaneously. This section will exemplify some simple yet powerful data analytics and visualization functions available from the Python extension known as Pandas. Pandas is a library that provides data structures and analysis tools for data loading, preparation, and statistical modeling. It is especially useful for time-series data manipulation and analysis (McKinney, 2012).

An increasingly common simulation data processing application is comparison of simulated and measured data. The following example will focus on loading simulation output from the EnergyPlus simulation engine. The first step in this example is to take a typical CSV output file from an EnergyPlus simulation and load it into a Dataframe object using the Pandas library and a Python list (simcolumnlabels) of the output fields contained in the file:

```
import pandas as pd

simcolumnlabels=['timestamp','oadb',
'solarenergytransmitted','zonemeanairtemp',
'idealloadsheating','idealloadscooling',
'idealloadsoutlettemp','zonemechventilation']

simdata = pd.read_csv('energyplusoutput.csv',
    header=0,names=simcolumnlabels)
```

Pandas includes an integrated date and time parser which can recognize and convert most string type timestamps to a datetime type. However, the EnergyPlus output is in a form (ie: ' 08/01 24:00:00') which is not standard and must be manually converted. The main issue is in how Energyplus represents midnight as 24:00:00 as opposed to the standard 00:00:00. We can customize a timestamp extraction script to accomodate for this feature as well as to specify the year:

```
from datetime import datetime as dtm
from datetime import timedelta

timestampdict={}
for i,row in simdata.T.iteritems():
    t = str(2012) + row['timestamp']
    try:
      timestampdict[i] = dtm.strptime(t,
        '%Y %m/%d  %H:%M:%S')
    except ValueError:
      tcorr = t.replace(' 24', ' 23')
      timestampdict[i] = dtm.strptime(tcorr,
        '%Y %m/%d  %H:%M:%S')
      timestampdict[i] += timedelta(hours=1)

timestampseries = pd.Series(timestampdict)
```

The new timestamp is then added to the previously created dataframe as the index and the old string timestamp is dropped. At this point, a fully cleaned Pandas Dataframe object has been created which can be printed to visualize the attributes and structure of the dataset.

```
simdata.index = timestampseries
simdata = simdata.drop(['timestamp'],axis=1)
print simdata
```

The output from this script is:

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 44160 entries, 2012-08-01
    00:03:00 to 2012-11-01 00:00:00
Data columns:
oadb                     44160 non-null values
solarenergytransmitted44160 non-null values
zonemeanairtemp          44160 non-null values
idealloadsheating        44160 non-null values
idealloadscooling        44160 non-null values
idealloadsoutlettemp     44160 non-null values
zonemechventilation      44160 non-null values
dtypes: float64(7)
```

The dataset characterized by the Pandas dataframe object is now ready for various transformation functions based on the data analysis intent. First in terms of visualization, Pandas has integrated easy use of the matplotlib visualization library. For example, if a quick visualization of the transition period between heating and cooling for this simulation is desired, a new dataframe is created from a truncated section of the whole simulation and a subplot visualization is created.

```
transitionsimdata=simdata.truncate(before='
    09/15/2012', after='10/15/2012')
transitionsimdata.plot(subplots=True,figsize
    =(6,10))
```

The results of this script can be seen in Figure 4.

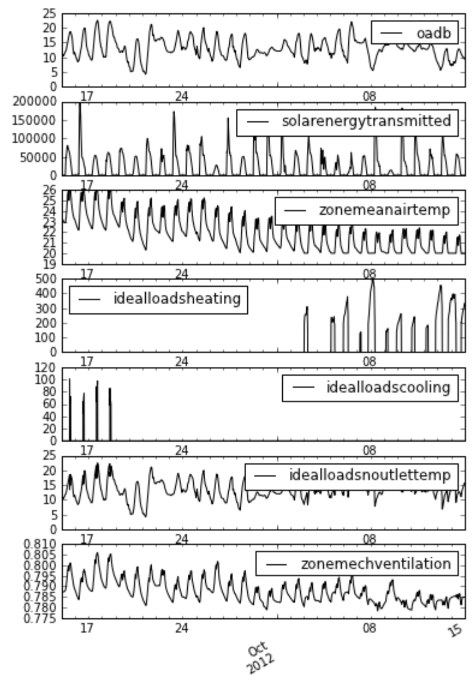
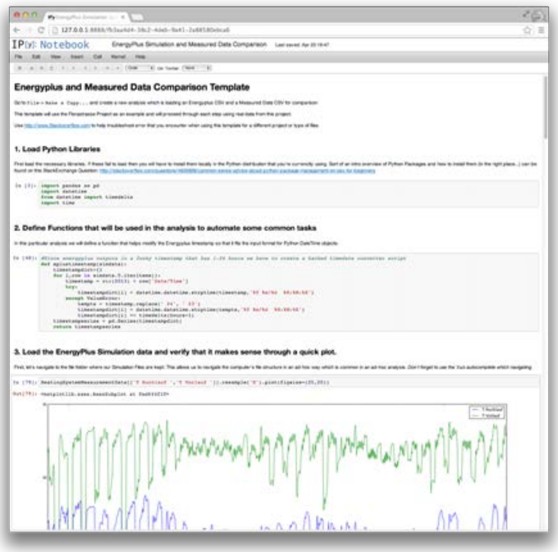


Figure 4: Time-series visualization of simulated data

A professional could then go on to load the measured data in a similar process by resampling the two datasets to the same frequency, using the Pandas merge functions, and visualizing the results. The IPython notebook (Pérez and Granger, 2007) is especially useful in then sharing the analysis with collaborators. The IPython notebook is a web-based editor that can create interactive analysis files that combine executable Python, R, or Octave scripts with text, figures, mathematical expressions, and even embedded videos. The notebooks are shared online with collaborators to easily communicate the information from a simulation study. Figure 5 shows an example screenshot of such an analysis.



IP[y]: Notebook EnergyPlus Simulation and Measured Data Comparison

Energyplus and Measured Data Comparison Template

[illegible]

1. Load Python Libraries

[illegible]

2. Define Functions that will be used in the analysis to automate some common tasks

[illegible]

3. Load the EnergyPlus Simulation data and verify that it makes sense through a quick plot.

[illegible]

Figure 5: IPython notebook screenshot

CONCLUSION

High level scripting languages are valuable to the efficient pre and post-processing workflows of building performance simulation professionals. This application paper overviews some common workflows that can be automated or simplified through the use of the Python. The goal of this analysis is to act as a catalyst for wider adoption of Python and similar languages by building simulation professionals.

Current and Future Python Building Simulation Projects

A small but growing number of buildings-focused Python libraries and projects are already available or in the planning stages. Table 2 illustrates the Python projects known to the authors and applicable to the building analysis domain.

Table 2: Building simulation Python projects

Project Name	Application
buildingspy	Modelica simulation driver and postprocessor library
CoolProp	Fluid and air properties library
PyBPS	Parametric run simulation manager
Pysolar	Solar irradation library
python-ifc	IFC file manipulation
Python-Powered Buildings	Python tutorials and code examples for building simulation
Revit Python Shell	Automation of Revit model
SimpleBuilding Beta	Simplified building simulation engine based on ISO 13790

Further Resources

More information on the building simulation-related Python projects and numerous code examples (including those presented in this paper) are made available online as IPython notebooks hosted at www.pythonpoweredbuilding.com.

ACKNOWLEDGEMENTS

The authors would like to acknowledge support from ETH Zurich and Technical University of Vienna.

REFERENCES

Bell, I. 2013. CoolProp: Open-Source, free fluids property database and Python library. http://coolprop.sourceforge.net/.

Bell, I. H., Lemort, V., Groll, E. A., Braun, J. E., King, G. B., and Horton, W. T. 2012. Liquid-flooded compression and expansion in scroll machines. Part I: Model development. *International Journal of Refrigeration*, 35(7):1878 – 1889.

Bertagnolio, S. and Andre, P. 2010. Development of an Evidence-Based Calibration Methodology Dedicated to Energy Audit of Office Buildings. Part 1: Methodology and Modeling. In *Proceedings of the 10th REHVA World Congress - Clima 2010, Anatalya, Turkey*.

EnergyPlus 2012. EnergyPlus Manual, Version 7.2: Auxiliary EnergyPlus Programs. *US Department of Energy*.

Jones, E., Oliphant, T., Peterson, P., et al. 2001. SciPy: Open source scientific tools for Python. http://www.scipy.org/.

Jones, M. and Ledinger, S. 2010. Pushing the limits of simulation complexity a building energy performance simulation of an exhibition centre in the UAE. In *Fourth National Conference of IBPSA–USA, New York City, New York*.

Ledinger, S. and Jones, M. 2010. Visualization of thermodynamics and energy performance for complex building system simulation. In *Proceedings of BauSIM 2010, Vienna Austria, 2010*.

lxml Dev Team 2013. lxml 3.10. http://www.lxml.de/.

McKinney, W. 2012. *Python for data analysis*. O'Reilly Media, Incorporated.

Oliphant, T. 2007. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20.

Pérez, F. and Granger, B. E. 2007. IPython: a System for Interactive Scientific Computing. *Comput. Sci. Eng.*, 9(3):21–29.

Raftery, P. 2012. Energyplus hvac generator. http://www.nuigalway.ie/iruse/hvacgenerator.html.

Raftery, P. and Keane, M. 2011. Visualizing patterns in building performance data. In *12th Conference of International Building Performance Simulation Association, Sydney*.

Raustad, R. 2012. Creating Performance Curves for Variable Refrigerant Flow Heat Pumps in EnergyPlus. Technical report, Florida Solar Energy Center.

Reddy, T. 2011. *Applied Data Analysis and Modeling for Energy Engineers and Scientists*. Springer.

Schlueter, A. and Thesseling, F. 2009. Building information model based energy/exergy performance assessment in early design stages. *Automation in Construction*, 18(2):153–163.

Sipser, M. 2006. *Introduction to the Theory of Computation*. Thomson.

The Open Green Building XML Schema 2012. Green Building XML. http://www.gbxml.org/.

Zhou, Y., Wu, J., Wang, R., Shiochi, S., and Li, Y. 2008. Simulation and experimental validation of the variable-refrigerant-volume (VRV) air-conditioning system in EnergyPlus. *Energy and Buildings*, 40(6):1041–1047.