eppy Documentation

Release 0.5.47

Santosh Philip

Oct 04, 2018

Contents

Contents:

CHAPTER 1

Installation

Type the following at your terminal to install **eppy**

```
pip install eppy
```

For additional functionality:

- Go to http://www.graphviz.org and install graphviz
- This is needed to draw loop diagrams.
- The rest of eppy will work without graphviz

Note: eppy runs on python2 and python 3.

1.1 Running eppy on other platforms

1.1.1 Eppy on Grasshopper (for use within Rhino)

Use the CPython plugin for Rhino-Grasshopper to run eppy in Grasshopper

CHAPTER 2

Eppy Tutorial

Authors: Santosh Philip, Leora Tanjuatco

Eppy is a scripting language for E+ idf files, and E+ output files. Eppy is written in the programming language Python. As a result it takes full advantage of the rich data structure and idioms that are avaliable in python. You can programmatically navigate, search, and modify E+ idf files using eppy. The power of using a scripting language allows you to do the following:

- Make a large number of changes in an idf file with a few lines of eppy code.
- Use conditions and filters when making changes to an idf file
- Make changes to multiple idf files.
- Read data from the output files of a E+ simulation run.
- Based to the results of a E+ simulation run, generate the input file for the next simulation run.

So what does this matter? Here are some of the things you can do with eppy:

- Change construction for all north facing walls.
- Change the glass type for all windows larger than 2 square meters.
- Change the number of people in all the interior zones.
- Change the lighting power in all south facing zones.
- Change the efficiency and fan power of all rooftop units.
- Find the energy use of all the models in a folder (or of models that were run after a certain date)
- If a model is using more energy than expected, keep increasing the R-value of the roof until you get to the expected energy use.

2.1 Quick Start

Here is a short IDF file that I'll be using as an example to start us off

```
VERSION,
    7.2;                     !- Version Identifier

SIMULATIONCONTROL,
    Yes,                     !- Do Zone Sizing Calculation
    Yes,                     !- Do System Sizing Calculation
    Yes,                     !- Do Plant Sizing Calculation
    No,                      !- Run Simulation for Sizing Periods
    Yes;                     !- Run Simulation for Weather File Run Periods

BUILDING,
    White House,             !- Name
    30.,                     !- North Axis {deg}
    City,                    !- Terrain
    0.04,                    !- Loads Convergence Tolerance Value
    0.4,                     !- Temperature Convergence Tolerance Value {deltaC}
    FullExterior,            !- Solar Distribution
    25,                      !- Maximum Number of Warmup Days
    6;                       !- Minimum Number of Warmup Days

SITE:LOCATION,
    CHICAGO_IL_USA TMY2-94846,  !- Name
    41.78,                   !- Latitude {deg}
    -87.75,                  !- Longitude {deg}
    -6.00,                   !- Time Zone {hr}
    190.00;                  !- Elevation {m}
```

To use eppy to look at this model, we have to run a little code first:

```
# you would normaly install eppy by doing
# python setup.py install
# or
# pip install eppy
# or
# easy_install eppy

# if you have not done so, uncomment the following three lines
import sys
# pathnameto_eppy = 'c:/eppy'
pathnameto_eppy = '../'
sys.path.append(pathnameto_eppy)

from eppy import modeleditor
from eppy.modeleditor import IDF
iddfile = "../eppy/resources/iddfiles/Energy+V7_2_0.idd"
fname1 = "../eppy/resources/idffiles/V_7_2/smallfile.idf"
```

```
IDF.setiddname(iddfile)
idf1 = IDF(fname1)
```

idf1 now holds all the data to your in you idf file.

Now that the behind-the-scenes work is done, we can print this file.

```
idf1.printidf()
```

```
VERSION,
    7.3;                      !- Version Identifier

SIMULATIONCONTROL,
    Yes,                      !- Do Zone Sizing Calculation
    Yes,                      !- Do System Sizing Calculation
    Yes,                      !- Do Plant Sizing Calculation
    No,                       !- Run Simulation for Sizing Periods
    Yes;                      !- Run Simulation for Weather File Run Periods

BUILDING,
    Empire State Building,    !- Name
    30.0,                     !- North Axis
    City,                     !- Terrain
    0.04,                     !- Loads Convergence Tolerance Value
    0.4,                      !- Temperature Convergence Tolerance Value
    FullExterior,             !- Solar Distribution
    25,                       !- Maximum Number of Warmup Days
    6;                        !- Minimum Number of Warmup Days

SITE:LOCATION,
    CHICAGO_IL_USA TMY2-94846,    !- Name
    41.78,                    !- Latitude
    -87.75,                   !- Longitude
    -6.0,                     !- Time Zone
    190.0;                    !- Elevation
```

Looks like the same file as before, except that all the comments are slightly different.

As you can see, this file has four objects:

- VERSION
- SIMULATIONCONTROL
- BUILDING
- SITE:LOCATION

So, let us look take a closer look at the BUILDING object. We can do this using this command:

```
print filename.idfobjects['OBJECTNAME']
```

```
print idf1.idfobjects['BUILDING']  # put the name of the object you'd like to look at␣
↪in brackets
```

```
[
BUILDING,
    Empire State Building,    !- Name
    30.0,                     !- North Axis
    City,                     !- Terrain
    0.04,                     !- Loads Convergence Tolerance Value
    0.4,                      !- Temperature Convergence Tolerance Value
    FullExterior,             !- Solar Distribution
    25,                       !- Maximum Number of Warmup Days
    6;                        !- Minimum Number of Warmup Days
]
```

We can also zoom in on the object and look just at its individual parts.

For example, let us look at the name of the building.

To do this, we have to do some more behind-the-scenes work, which we'll explain later.

```
building = idf1.idfobjects['BUILDING'][0]
```

Now we can do this:

```
print building.Name
```

```
Empire State Building
```

Now that we've isolated the building name, we can change it.

```
building.Name = "Empire State Building"
```

```
print building.Name
```

```
Empire State Building
```

Did this actually change the name in the model ? Let us print the entire model and see.

```
idf1.printidf()
```

```
VERSION,
    7.3;                      !- Version Identifier

SIMULATIONCONTROL,
    Yes,                      !- Do Zone Sizing Calculation
    Yes,                      !- Do System Sizing Calculation
    Yes,                      !- Do Plant Sizing Calculation
    No,                       !- Run Simulation for Sizing Periods
    Yes;                      !- Run Simulation for Weather File Run Periods

BUILDING,
    Empire State Building,    !- Name
    30.0,                     !- North Axis
    City,                     !- Terrain
    0.04,                     !- Loads Convergence Tolerance Value
    0.4,                      !- Temperature Convergence Tolerance Value
    FullExterior,             !- Solar Distribution
    25,                       !- Maximum Number of Warmup Days
    6;                        !- Minimum Number of Warmup Days

SITE:LOCATION,
    CHICAGO_IL_USA TMY2-94846,    !- Name
    41.78,                    !- Latitude
    -87.75,                   !- Longitude
    -6.0,                     !- Time Zone
    190.0;                    !- Elevation
```

Yes! It did. So now you have a taste of what eppy can do. Let's get started!

2.2 Modifying IDF Fields

That was just a quick example – we were showing off. Let's look a little closer.

As you might have guessed, changing an IDF field follows this structure:

```
object.fieldname = "New Field Name"
```

Plugging the object name (building), the field name (Name) and our new field name ("Empire State Building") into this command gave us this:

```
building.Name = "Empire State Building"
```

```
import eppy
# import eppy.ex_inits
# reload(eppy.ex_inits)
import eppy.ex_inits
```

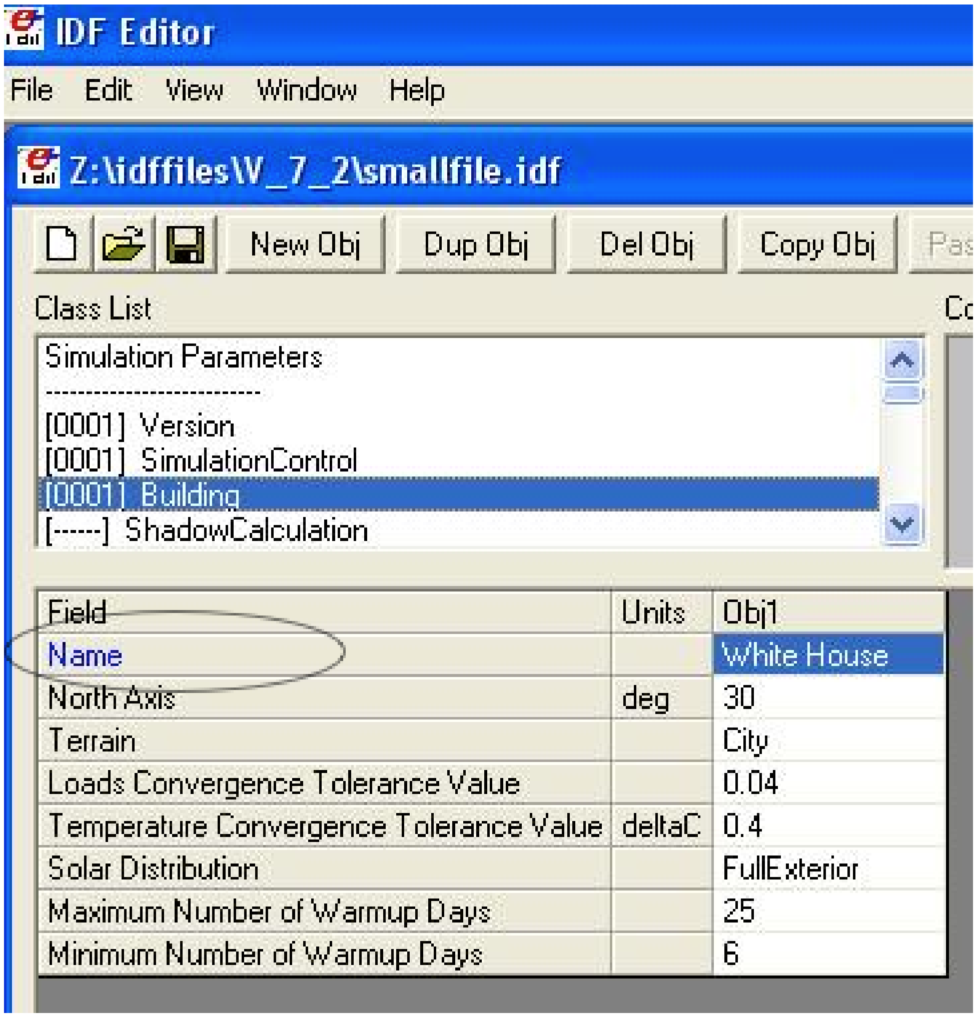
But how did we know that "Name" is one of the fields in the object "building"?

Are there other fields?

What are they called?

Let's take a look at the IDF editor:

```
from eppy import ex_inits #no need to know this code, it just shows the image below
for_images = ex_inits
for_images.display_png(for_images.idfeditor)
```



Simulation Parameters

[0001] Version
[0001] SimulationControl
[0001] Building
[------] ShadowCalculation

Field	Units	Obj1
Name		White House
North Axis	deg	30
Terrain		City
Loads Convergence Tolerance Value		0.04
Temperature Convergence Tolerance Value	deltaC	0.4
Solar Distribution		FullExterior
Maximum Number of Warmup Days		25
Minimum Number of Warmup Days		6

In the IDF Editor, the building object is selected.

We can see all the fields of the object "BUILDING".

They are:

- Name
- North Axis
- Terrain
- Loads Convergence Tolerance Value
- Temperature Convergence Tolerance Value
- Solar Distribution
- Maximum Number of Warmup Days
- Minimum Number of Warmup Days

Let us try to access the other fields.

```
print building.Terrain
```

```
City
```

How about the field “North Axis” ?

It is not a single word, but two words.

In a programming language, a variable has to be a single word without any spaces.

To solve this problem, put an underscore where there is a space.

So “North Axis” becomes “North_Axis”.

```
print building.North_Axis
```

```
30.0
```

Now we can do:

```
print building.Name
print building.North_Axis
print building.Terrain
print building.Loads_Convergence_Tolerance_Value
print building.Temperature_Convergence_Tolerance_Value
print building.Solar_Distribution
print building.Maximum_Number_of_Warmup_Days
print building.Minimum_Number_of_Warmup_Days
```

```
Empire State Building
30.0
City
0.04
0.4
FullExterior
25
6
```

Where else can we find the field names?

The IDF Editor saves the idf file with the field name commented next to field.

Eppy also does this.

Let us take a look at the “BUILDING” object in the text file that the IDF Editor saves

```
BUILDING,
    White House,             !- Name
    30.,                     !- North Axis {deg}
    City,                    !- Terrain
    0.04,                    !- Loads Convergence Tolerance Value
    0.4,                     !- Temperature Convergence Tolerance Value {deltaC}
    FullExterior,            !- Solar Distribution
    25,                      !- Maximum Number of Warmup Days
    6;                       !- Minimum Number of Warmup Days
```

This a good place to find the field names too.

It is easy to copy and paste from here. You can’t do that from the IDF Editor.

We know that in an E+ model, there will be only ONE “BUILDING” object. This will be the first and only item in the list “buildings”.

But E+ models are made up of objects such as “BUILDING”, “SITE:LOCATION”, “ZONE”, “PEOPLE”, “LIGHTS”. There can be a number of “ZONE” objects, a number of “PEOPLE” objects and a number of “LIGHTS” objects.

So how do you know if you’re looking at the first “ZONE” object or the second one? Or the tenth one? To answer this, we need to learn about how lists work in python.

2.3 Python lesson 1: lists

Eppy holds these objects in a python structure called list. Let us take a look at how lists work in python.

```
fruits = ["apple", "orange", "bannana"]
# fruits is a list with three items in it.
```

To get the first item in fruits we say:

```
fruits[0]
```

```
'apple'
```

Why “0” ?

Because, unlike us, python starts counting from zero in a list. So, to get the third item in the list we’d need to input 2, like this:

```
print fruits[2]
```

```
bannana
```

But calling the first fruit “fruit[0]” is rather cumbersome. Why don’t we call it firstfruit?

```
firstfruit = fruits[0]
print firstfruit
```

```
apple
```

We also can say

```
goodfruit = fruits[0]
redfruit = fruits[0]

print firstfruit
print goodfruit
print redfruit
print fruits[0]
```

```
apple
apple
apple
apple
```

As you see, we can call that item in the list whatever we want.

2.3.1 How many items in the list

To know how many items are in a list, we ask for the length of the list.

The function 'len' will do this for us.

```
print len(fruits)
```

```
3
```

There are 3 fruits in the list.

2.4 Saving an idf file

This is easy:

```
idf1.save()
```

If you'd like to do a "Save as. . . " use this:

```
idf1.saveas('something.idf')
```

2.5 Working with E+ objects

Let us open a small idf file that has only "CONSTRUCTION" and "MATERIAL" objects in it. You can go into "../idffiles/V_7_2/constructions.idf" and take a look at the file. We are not printing it here because it is too big.

So let us open it using the idfreader -

```
from eppy import modeleditor
from eppy.modeleditor import IDF

iddfile = "../eppy/resources/iddfiles/Energy+V7_2_0.idd"
try:
    IDF.setiddname(iddfile)
except modeleditor.IDDAlreadySetError as e:
    pass

fname1 = "../eppy/resources/idffiles/V_7_2/constructions.idf"
idf1 = IDF(fname1)
```

Let us print all the "MATERIAL" objects in this model.

```
materials = idf1.idfobjects["MATERIAL"]
print materials
```

```
[
Material,
    F08 Metal surface,          !- Name
    Smooth,                     !- Roughness
    0.0008,                     !- Thickness
    45.28,                      !- Conductivity
    7824.0,                     !- Density
```

(continues on next page)

(continued from previous page)

```
    500.0;                    !- Specific Heat
,
Material,
    I01 25mm insulation board,    !- Name
    MediumRough,              !- Roughness
    0.0254,                   !- Thickness
    0.03,                     !- Conductivity
    43.0,                     !- Density
    1210.0;                   !- Specific Heat
,
Material,
    I02 50mm insulation board,    !- Name
    MediumRough,              !- Roughness
    0.0508,                   !- Thickness
    0.03,                     !- Conductivity
    43.0,                     !- Density
    1210.0;                   !- Specific Heat
,
Material,
    G01a 19mm gypsum board,    !- Name
    MediumSmooth,             !- Roughness
    0.019,                    !- Thickness
    0.16,                     !- Conductivity
    800.0,                    !- Density
    1090.0;                   !- Specific Heat
,
Material,
    M11 100mm lightweight concrete,    !- Name
    MediumRough,              !- Roughness
    0.1016,                   !- Thickness
    0.53,                     !- Conductivity
    1280.0,                   !- Density
    840.0;                    !- Specific Heat
,
Material,
    F16 Acoustic tile,        !- Name
    MediumSmooth,             !- Roughness
    0.0191,                   !- Thickness
    0.06,                     !- Conductivity
    368.0,                    !- Density
    590.0;                    !- Specific Heat
,
Material,
    M01 100mm brick,          !- Name
    MediumRough,              !- Roughness
    0.1016,                   !- Thickness
    0.89,                     !- Conductivity
    1920.0,                   !- Density
    790.0;                    !- Specific Heat
,
Material,
    M15 200mm heavyweight concrete,    !- Name
    MediumRough,              !- Roughness
    0.2032,                   !- Thickness
    1.95,                     !- Conductivity
    2240.0,                   !- Density
    900.0;                    !- Specific Heat
```

(continues on next page)

(continued from previous page)

```
,
Material,
    M05 200mm concrete block,    !- Name
    MediumRough,               !- Roughness
    0.2032,                    !- Thickness
    1.11,                      !- Conductivity
    800.0,                     !- Density
    920.0;                     !- Specific Heat
,
Material,
    G05 25mm wood,             !- Name
    MediumSmooth,              !- Roughness
    0.0254,                    !- Thickness
    0.15,                      !- Conductivity
    608.0,                     !- Density
    1630.0;                    !- Specific Heat
]
```

As you can see, there are many material objects in this idf file.

The variable "materials" now contains a list of "MATERIAL" objects.

You already know a little about lists, so let us take a look at the items in this list.

```
firstmaterial = materials[0]
secondmaterial = materials[1]
```

```
print firstmaterial
```

```
Material,
    F08 Metal surface,         !- Name
    Smooth,                    !- Roughness
    0.0008,                    !- Thickness
    45.28,                     !- Conductivity
    7824.0,                    !- Density
    500.0;                     !- Specific Heat
```

Let us print secondmaterial

```
print secondmaterial
```

```
Material,
    I01 25mm insulation board,    !- Name
    MediumRough,               !- Roughness
    0.0254,                    !- Thickness
    0.03,                      !- Conductivity
    43.0,                      !- Density
    1210.0;                    !- Specific Heat
```

This is awesome!! Why?

To understand what you can do with your objects organized as lists, you'll have to learn a little more about lists.

2.6 Python lesson 2: more about lists

2.6.1 More ways to access items in a list

You should remember that you can access any item in a list by passing in its index.

The tricky part is that python starts counting at 0, so you need to input 0 in order to get the first item in a list.

Following the same logic, you need to input 3 in order to get the fourth item on the list. Like so:

```
bad_architects = ["Donald Trump", "Mick Jagger",
        "Steve Jobs", "Lady Gaga", "Santa Clause"]
print bad_architects[3]
```

```
Lady Gaga
```

But there's another way to access items in a list. If you input -1, it will return the last item. -2 will give you the second-to-last item, etc.

```
print bad_architects[-1]
print bad_architects[-2]
```

```
Santa Clause
Lady Gaga
```

2.6.2 Slicing a list

You can also get more than one item in a list:

bad_architects[first_slice:second_slice]

```
print bad_architects[1:3] # slices at 1 and 3
```

```
['Mick Jagger', 'Steve Jobs']
```

How do I make sense of this?

To understand this you need to see the list in the following manner:

```
[ "Donald Trump", "Mick Jagger", "Steve Jobs", "Lady Gaga", "Santa Clause" ]
 ^                ^              ^              ^            ^              ^
 0                1              2              3            4              5
-5               -4             -3             -2           -1
```

The slice operation bad_architects[1:3] slices right where the numbers are.

Does that make sense?

Let us try a few other slices:

```
print bad_architects[2:-1] # slices at 2 and -1
print bad_architects[-3:-1] # slices at -3 and -1
```

```
['Steve Jobs', 'Lady Gaga']
['Steve Jobs', 'Lady Gaga']
```

You can also slice in the following way:

```
print bad_architects[3:]
print bad_architects[:2]
print bad_architects[-3:]
print bad_architects[:-2]
```

```
['Lady Gaga', 'Santa Clause']
['Donald Trump', 'Mick Jagger']
['Steve Jobs', 'Lady Gaga', 'Santa Clause']
['Donald Trump', 'Mick Jagger', 'Steve Jobs']
```

I'll let you figure that out on your own.

2.6.3 Adding to a list

This is simple: the append function adds an item to the end of the list.

The following command will add 'something' to the end of the list called listname:

```
listname.append(something)
```

```
bad_architects.append("First-year students")
print bad_architects
```

```
['Donald Trump', 'Mick Jagger', 'Steve Jobs', 'Lady Gaga', 'Santa Clause', 'First-
↪year students']
```

2.6.4 Deleting from a list

There are two ways to do this, based on the information you have. If you have the value of the object, you'll want to use the remove function. It looks like this:

listname.remove(value)

An example:

```
bad_architects.remove("First-year students")
print bad_architects
```

```
['Donald Trump', 'Mick Jagger', 'Steve Jobs', 'Lady Gaga', 'Santa Clause']
```

What if you know the index of the item you want to remove?

What if you appended an item by mistake and just want to remove the last item in the list?

You should use the pop function. It looks like this:

listname.pop(index)

```
what_i_ate_today = ["coffee", "bacon", "eggs"]
print what_i_ate_today
```

```
['coffee', 'bacon', 'eggs']
```

```
what_i_ate_today.append("vegetables") # adds vegetables to the end of the list
# but I don't like vegetables
print what_i_ate_today
```

```
['coffee', 'bacon', 'eggs', 'vegetables']
```

```
# since I don't like vegetables
what_i_ate_today.pop(-1) # use index of -1, since vegetables is the last item in the␣
↪list
print what_i_ate_today
```

```
['coffee', 'bacon', 'eggs']
```

You can also remove the second item.

```
what_i_ate_today.pop(1)
```

```
'bacon'
```

Notice the 'bacon' in the line above.

pop actually 'pops' the value (the one you just removed from the list) back to you.

Let us pop the first item.

```
was_first_item = what_i_ate_today.pop(0)
print 'was_first_item =', was_first_item
print 'what_i_ate_today = ', what_i_ate_today
```

```
was_first_item = coffee
what_i_ate_today =  ['eggs']
```

what_i_ate_today is just 'eggs'?

That is not much of a breakfast!

Let us get back to eppy.

2.7 Continuing to work with E+ objects

Let us get those "MATERIAL" objects again

```
materials = idf1.idfobjects["MATERIAL"]
```

With our newfound knowledge of lists, we can do a lot of things.

Let us get the last material:

```
print materials[-1]
```

```
Material,
    G05 25mm wood,            !- Name
    MediumSmooth,             !- Roughness
    0.0254,                   !- Thickness
```

(continues on next page)

(continued from previous page)

```
    0.15,                     !- Conductivity
    608.0,                    !- Density
    1630.0;                   !- Specific Heat
```

How about the last two?

```
print materials[-2:]
```

```
[
Material,
    M05 200mm concrete block,    !- Name
    MediumRough,              !- Roughness
    0.2032,                   !- Thickness
    1.11,                     !- Conductivity
    800.0,                    !- Density
    920.0;                    !- Specific Heat
,
Material,
    G05 25mm wood,            !- Name
    MediumSmooth,             !- Roughness
    0.0254,                   !- Thickness
    0.15,                     !- Conductivity
    608.0,                    !- Density
    1630.0;                   !- Specific Heat
]
```

Pretty good.

2.7.1 Counting all the materials (or counting all objects)

How many materials are in this model ?

```
print len(materials)
```

```
10
```

2.7.2 Removing a material

Let us remove the last material in the list

```
was_last_material = materials.pop(-1)
```

```
print len(materials)
```

```
9
```

Success! We have only 9 materials now.

The last material used to be:

'G05 25mm wood'

```
print materials[-1]
```

```
Material,
    M05 200mm concrete block,    !- Name
    MediumRough,                !- Roughness
    0.2032,                     !- Thickness
    1.11,                       !- Conductivity
    800.0,                      !- Density
    920.0;                      !- Specific Heat
```

Now the last material in the list is:

'M15 200mm heavyweight concrete'

2.7.3 Adding a material to the list

We still have the old last material

```
print was_last_material
```

```
Material,
    G05 25mm wood,              !- Name
    MediumSmooth,               !- Roughness
    0.0254,                     !- Thickness
    0.15,                       !- Conductivity
    608.0,                      !- Density
    1630.0;                     !- Specific Heat
```

Let us add it back to the list

```
materials.append(was_last_material)
```

```
print len(materials)
```

```
10
```

Once again we have 10 materials and the last material is:

```
print materials[-1]
```

```
Material,
    G05 25mm wood,              !- Name
    MediumSmooth,               !- Roughness
    0.0254,                     !- Thickness
    0.15,                       !- Conductivity
    608.0,                      !- Density
    1630.0;                     !- Specific Heat
```

2.7.4 Add a new material to the model

So far we have been working only with materials that were already in the list.

What if we want to make new material?

Obviously we would use the function 'newidfobject'.

```
idf1.newidfobject("MATERIAL")
```

```
MATERIAL,
    ,                         !- Name
    ,                         !- Roughness
    ,                         !- Thickness
    ,                         !- Conductivity
    ,                         !- Density
    ,                         !- Specific Heat
    0.9,                      !- Thermal Absorptance
    0.7,                      !- Solar Absorptance
    0.7;                      !- Visible Absorptance
```

```
len(materials)
```

```
11
```

We have 11 items in the materials list.

Let us take a look at the last material in the list, where this fancy new material was added

```
print materials[-1]
```

```
MATERIAL,
    ,                         !- Name
    ,                         !- Roughness
    ,                         !- Thickness
    ,                         !- Conductivity
    ,                         !- Density
    ,                         !- Specific Heat
    0.9,                      !- Thermal Absorptance
    0.7,                      !- Solar Absorptance
    0.7;                      !- Visible Absorptance
```

Looks a little different from the other materials. It does have the name we gave it.

Why do some fields have values and others are blank ?

"addobject" puts in all the default values, and leaves the others blank. It is up to us to put values in the the new fields.

Let's do it now.

```
materials[-1].Name = 'Peanut Butter'
materials[-1].Roughness = 'MediumSmooth'
materials[-1].Thickness = 0.03
materials[-1].Conductivity = 0.16
materials[-1].Density = 600
materials[-1].Specific_Heat = 1500
```

```
print materials[-1]
```

```
MATERIAL,
    Peanut Butter,            !- Name
    MediumSmooth,             !- Roughness
    0.03,                     !- Thickness
```

(continues on next page)

(continued from previous page)

```
    0.16,                     !- Conductivity
    600,                      !- Density
    1500,                     !- Specific Heat
    0.9,                      !- Thermal Absorptance
    0.7,                      !- Solar Absorptance
    0.7;                      !- Visible Absorptance
```

2.7.5 Copy an existing material

```
Peanutbuttermaterial = materials[-1]
idf1.copyidfobject(Peanutbuttermaterial)
materials = idf1.idfobjects["MATERIAL"]
len(materials)
materials[-1]
```

```
MATERIAL,
    Peanut Butter,            !- Name
    MediumSmooth,             !- Roughness
    0.03,                     !- Thickness
    0.16,                     !- Conductivity
    600,                      !- Density
    1500,                     !- Specific Heat
    0.9,                      !- Thermal Absorptance
    0.7,                      !- Solar Absorptance
    0.7;                      !- Visible Absorptance
```

2.8 Python lesson 3: indentation and looping through lists

I'm tired of doing all this work, it's time to make python do some heavy lifting for us!

Python can go through each item in a list and perform an operation on any (or every) item in the list.

This is called looping through the list.

Here's how to tell python to step through each item in a list, and then do something to every single item.

We'll use a 'for' loop to do this.

```
for <variable> in <listname>:
    <do something>
```

A quick note about the second line. Notice that it's indented? There are 4 blank spaces before the code starts:

```
in python, indentations are used
to determine the grouping of statements
       some languages use symbols to mark
       where the function code starts and stops
       but python uses indentation to tell you this
                i'm using indentation to
                show the beginning and end of a sentence
       this is a very simple explanation
       of indentation in python
 if you'd like to know more, there is plenty of information
 about indentation in python on the web
```

It's elegant, but it means that the indentation of the code holds meaning.

So make sure to indent the second (and third and forth) lines of your loops!

Now let's make some fruit loops.

```
fruits = ["apple", "orange", "bannana"]
```

Given the syntax I gave you before I started rambling about indentation, we can easily print every item in the fruits list by using a 'for' loop.

```
for fruit in fruits:
   print fruit
```

```
apple
orange
bannana
```

That was easy! But it can get complicated pretty quickly. . .

Let's make it do something more complicated than just print the fruits.

Let's have python add some words to each fruit.

```
for fruit in fruits:
    print "I am a fruit said the", fruit
```

```
I am a fruit said the apple
I am a fruit said the orange
I am a fruit said the bannana
```

Now we'll try to confuse you:

```
rottenfruits = [] # makes a blank list called rottenfruits
for fruit in fruits: # steps through every item in fruits
    rottenfruit = "rotten " + fruit # changes each item to "rotten _____"
    rottenfruits.append(rottenfruit) # adds each changed item to the formerly empty␣
↪list
```

```
print rottenfruits
```

```
['rotten apple', 'rotten orange', 'rotten bannana']
```

```
# here's a shorter way of writing it
rottenfruits = ["rotten " + fruit for fruit in fruits]
```

Did you follow all that??

Just in case you didn't, let's review that last one:

```
["rotten " + fruit for fruit in fruits]
                   -------------------
                   This is the "for loop"
                   it steps through each fruit in fruits

["rotten " + fruit for fruit in fruits]
 -----------------
```

(continues on next page)

(continued from previous page)

```
 add "rotten " to the fruit at each step
 this is your "do something"

["rotten " + fruit for fruit in fruits]
---------------------------------------
give a new list that is a result of the "do something"
```

```
print rottenfruits
```

```
['rotten apple', 'rotten orange', 'rotten bannana']
```

2.8.1 Filtering in a loop

But what if you don't want to change *every* item in a list?

We can use an 'if' statement to operate on only some items in the list.

Indentation is also important in 'if' statements, as you'll see:

```
if <someconstraint>:
    <if the first line is true, do this>
<but if it's false, do this>
```

```
fruits = ["apple", "orange", "pear", "berry", "mango", "plum", "peach", "melon",
↪"bannana"]
```

```
for fruit in fruits:                  # steps through every fruit in fruits
    if len(fruit) > 5:                # checks to see if the length of the word is more␣
↪than 5
        print fruit                   # if true, print the fruit
                                      # if false, python goes back to the 'for' loop
                                         # and checks the next item in the list
```

```
orange
bannana
```

Let's say we want to pick only the fruits that start with the letter 'p'.

```
p_fruits = []                         # creates an empty list called p_fruits
for fruit in fruits:                  # steps through every fruit in fruits
    if fruit.startswith("p"):         # checks to see if the first letter is 'p', using␣
↪a built-in function
        p_fruits.append(fruit)        # if the first letter is 'p', the item is added to␣
↪p_fruits
                                      # if the first letter is not 'p', python goes back␣
↪to the 'for' loop
                                         # and checks the next item in the list
```

```
print p_fruits
```

```
['pear', 'plum', 'peach']
```

```
# here's a shorter way to write it
p_fruits = [fruit for fruit in fruits if fruit.startswith("p")]
```

```
[fruit for fruit in fruits if fruit.startswith("p")]
       -------------------
       for loop

[fruit for fruit in fruits if fruit.startswith("p")]
                           ------------------------
                           pick only some of the fruits

[fruit for fruit in fruits if fruit.startswith("p")]
 -----
 give me the variable fruit as it appears in the list, don't change it

[fruit for fruit in fruits if fruit.startswith("p")]
----------------------------------------------------
a fresh new list with those fruits
```

```
print p_fruits
```

```
['pear', 'plum', 'peach']
```

2.8.2 Counting through loops

This is not really needed, but it is nice to know. You can safely skip this.

Python's built-in function range() makes a list of numbers within a range that you specify.

This is useful because you can use these lists inside of loops.

```
range(4) # makes a list
```

```
[0, 1, 2, 3]
```

```
for i in range(4):
    print i
```

```
0
1
2
3
```

```
len(p_fruits)
```

```
3
```

```
for i in range(len(p_fruits)):
    print i
```

```
0
1
2
```

```
for i in range(len(p_fruits)):
    print p_fruits[i]
```

```
pear
plum
peach
```

```
for i in range(len(p_fruits)):
    print i,  p_fruits[i]
```

```
0 pear
1 plum
2 peach
```

```
for item_from_enumerate in enumerate(p_fruits):
    print item_from_enumerate
```

```
(0, 'pear')
(1, 'plum')
(2, 'peach')
```

```
for i, fruit in enumerate(p_fruits):
    print i, fruit
```

```
0 pear
1 plum
2 peach
```

2.9 Looping through E+ objects

If you have read the python explanation of loops, you are now masters of using loops.

Let us use the loops with E+ objects.

We'll continue to work with the materials list.

```
for material in materials:
    print material.Name
```

```
F08 Metal surface
I01 25mm insulation board
I02 50mm insulation board
G01a 19mm gypsum board
M11 100mm lightweight concrete
F16 Acoustic tile
M01 100mm brick
M15 200mm heavyweight concrete
M05 200mm concrete block
G05 25mm wood
Peanut Butter
Peanut Butter
```

```
[material.Name for material in materials]
```

```
[u'F08 Metal surface',
 u'I01 25mm insulation board',
 u'I02 50mm insulation board',
 u'G01a 19mm gypsum board',
 u'M11 100mm lightweight concrete',
 u'F16 Acoustic tile',
 u'M01 100mm brick',
 u'M15 200mm heavyweight concrete',
 u'M05 200mm concrete block',
 u'G05 25mm wood',
 'Peanut Butter',
 'Peanut Butter']
```

```
[material.Roughness for material in materials]
```

```
[u'Smooth',
 u'MediumRough',
 u'MediumRough',
 u'MediumSmooth',
 u'MediumRough',
 u'MediumSmooth',
 u'MediumRough',
 u'MediumRough',
 u'MediumRough',
 u'MediumSmooth',
 'MediumSmooth',
 'MediumSmooth']
```

```
[material.Thickness for material in materials]
```

```
[0.0008,
 0.0254,
 0.0508,
 0.019,
 0.1016,
 0.0191,
 0.1016,
 0.2032,
 0.2032,
 0.0254,
 0.03,
 0.03]
```

```
[material.Thickness for material in materials if material.Thickness > 0.1]
```

```
[0.1016, 0.1016, 0.2032, 0.2032]
```

```
[material.Name for material in materials if material.Thickness > 0.1]
```

```
[u'M11 100mm lightweight concrete',
 u'M01 100mm brick',
```

(continues on next page)

(continued from previous page)

```
 u'M15 200mm heavyweight concrete',
 u'M05 200mm concrete block']
```

```
thick_materials = [material for material in materials if material.Thickness > 0.1]
```

```
thick_materials
```

```
[
Material,
    M11 100mm lightweight concrete,    !- Name
    MediumRough,                 !- Roughness
    0.1016,                      !- Thickness
    0.53,                        !- Conductivity
    1280.0,                      !- Density
    840.0;                       !- Specific Heat
,

Material,
    M01 100mm brick,             !- Name
    MediumRough,                 !- Roughness
    0.1016,                      !- Thickness
    0.89,                        !- Conductivity
    1920.0,                      !- Density
    790.0;                       !- Specific Heat
,

Material,
    M15 200mm heavyweight concrete,    !- Name
    MediumRough,                 !- Roughness
    0.2032,                      !- Thickness
    1.95,                        !- Conductivity
    2240.0,                      !- Density
    900.0;                       !- Specific Heat
,

Material,
    M05 200mm concrete block,    !- Name
    MediumRough,                 !- Roughness
    0.2032,                      !- Thickness
    1.11,                        !- Conductivity
    800.0,                       !- Density
    920.0;                       !- Specific Heat
]
```

```
# change the names of the thick materials
for material in thick_materials:
    material.Name = "THICK " + material.Name
```

```
thick_materials
```

```
[
Material,
    THICK M11 100mm lightweight concrete,    !- Name
    MediumRough,                 !- Roughness
```

(continues on next page)

(continued from previous page)

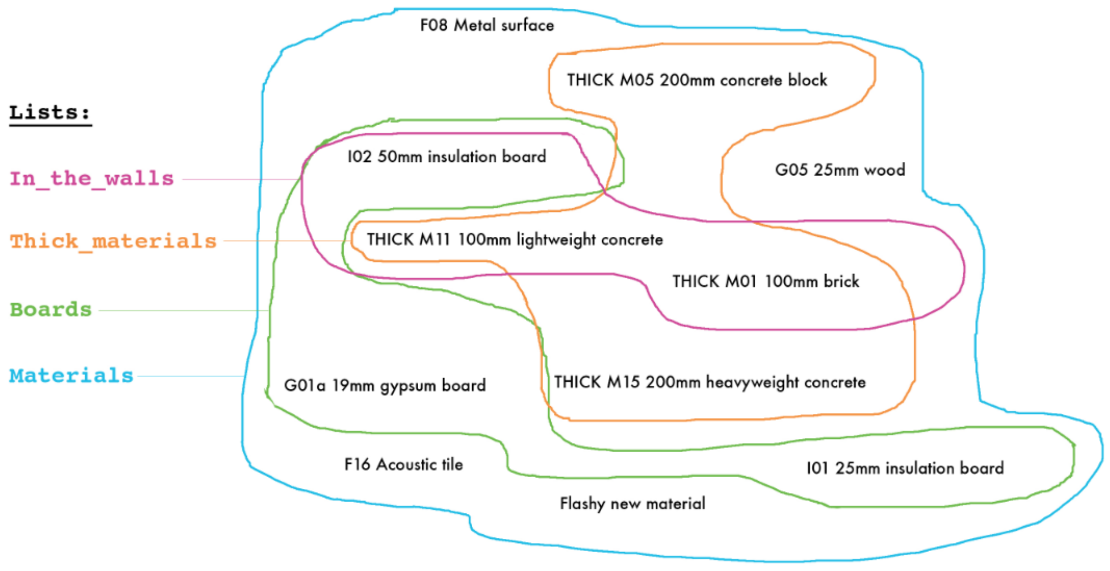
```
    0.1016,                   !- Thickness
    0.53,                     !- Conductivity
    1280.0,                   !- Density
    840.0;                    !- Specific Heat
,

Material,
    THICK M01 100mm brick,    !- Name
    MediumRough,              !- Roughness
    0.1016,                   !- Thickness
    0.89,                     !- Conductivity
    1920.0,                   !- Density
    790.0;                    !- Specific Heat
,

Material,
    THICK M15 200mm heavyweight concrete,    !- Name
    MediumRough,              !- Roughness
    0.2032,                   !- Thickness
    1.95,                     !- Conductivity
    2240.0,                   !- Density
    900.0;                    !- Specific Heat
,

Material,
    THICK M05 200mm concrete block,    !- Name
    MediumRough,              !- Roughness
    0.2032,                   !- Thickness
    1.11,                     !- Conductivity
    800.0,                    !- Density
    920.0;                    !- Specific Heat
]
```

So now we're working with two different lists: materials and thick_materials.

But even though the items can be separated into two lists, we're still working with the same items.

Here's a helpful illustration:

```
for_images.display_png(for_images.material_lists) # display the image below
```



```
# here's the same concept, demonstrated with code
# remember, we changed the names of the items in the list thick_materials
# these changes are visible when we print the materials list; the thick materials are␣
↪also in the materials list
[material.Name for material in materials]
```

```
[u'F08 Metal surface',
 u'I01 25mm insulation board',
 u'I02 50mm insulation board',
 u'G01a 19mm gypsum board',
 u'THICK M11 100mm lightweight concrete',
 u'F16 Acoustic tile',
 u'THICK M01 100mm brick',
 u'THICK M15 200mm heavyweight concrete',
 u'THICK M05 200mm concrete block',
 u'G05 25mm wood',
 'Peanut Butter',
 'Peanut Butter']
```

2.10 Geometry functions in eppy

Sometimes, we want information about the E+ object that is not in the fields. For example, it would be useful to know the areas and orientations of the surfaces. These attributes of the surfaces are not in the fields of surfaces, but surface objects *do* have fields that have the coordinates of the surface. The areas and orientations can be calculated from these coordinates.

Pyeplus has some functions that will do the calculations.

In the present version, pyeplus will calculate:

- surface azimuth
- surface tilt
- surface area

Let us explore these functions

```
# OLD CODE, SHOULD BE DELETED
# from idfreader import idfreader

# iddfile = "../iddfiles/Energy+V7_0_0_036.idd"
# fname = "../idffiles/V_7_0/5ZoneSupRetPlenRAB.idf"

# model, to_print, idd_info = idfreader(fname, iddfile)
# surfaces = model['BUILDINGSURFACE:DETAILED'] # all the surfaces
```

```
from eppy import modeleditor
from eppy.modeleditor import IDF

iddfile = "../eppy/resources/iddfiles/Energy+V7_2_0.idd"
try:
    IDF.setiddname(iddfile)
except modeleditor.IDDAlreadySetError as e:
    pass


fname1 = "../eppy/resources/idffiles/V_7_0/5ZoneSupRetPlenRAB.idf"
idf1 = IDF(fname1)
surfaces = idf1.idfobjects['BUILDINGSURFACE:DETAILED']
```

```
# Let us look at the first surface
asurface = surfaces[0]
print "surface azimuth =",  asurface.azimuth, "degrees"
print "surface tilt =", asurface.tilt, "degrees"
print "surface area =", asurface.area, "m2"
```

```
surface azimuth = 180.0 degrees
surface tilt = 90.0 degrees
surface area = 18.3 m2
```

```
# all the surface names
s_names = [surface.Name for surface in surfaces]
print s_names[:5] # print five of them
```

```
[u'WALL-1PF', u'WALL-1PR', u'WALL-1PB', u'WALL-1PL', u'TOP-1']
```

```
# surface names and azimuths
s_names_azm = [(sf.Name, sf.azimuth) for sf in surfaces]
print s_names_azm[:5] # print five of them
```

```
[(u'WALL-1PF', 180.0), (u'WALL-1PR', 90.0), (u'WALL-1PB', 0.0), (u'WALL-1PL', 270.0),␣
↪(u'TOP-1', 0.0)]
```

```
# or to do that in pretty printing
for name, azimuth in s_names_azm[:5]: # just five of them
    print name, azimuth
```

```
WALL-1PF 180.0
WALL-1PR 90.0
```

(continues on next page)

(continued from previous page)

```
WALL-1PB 0.0
WALL-1PL 270.0
TOP-1 0.0
```

```
# surface names and tilt
s_names_tilt = [(sf.Name, sf.tilt) for sf in surfaces]
for name, tilt in s_names_tilt[:5]: # just five of them
    print name, tilt
```

```
WALL-1PF 90.0
WALL-1PR 90.0
WALL-1PB 90.0
WALL-1PL 90.0
TOP-1 0.0
```

```
# surface names and areas
s_names_area = [(sf.Name, sf.area) for sf in surfaces]
for name, area in s_names_area[:5]: # just five of them
    print name, area, "m2"
```

```
WALL-1PF 18.3 m2
WALL-1PR 9.12 m2
WALL-1PB 18.3 m2
WALL-1PL 9.12 m2
TOP-1 463.6 m2
```

Let us try to isolate the exterior north facing walls and change their construnctions

```
# just vertical walls
vertical_walls = [sf for sf in surfaces if sf.tilt == 90.0]
print [sf.Name for sf in vertical_walls]
```

```
[u'WALL-1PF', u'WALL-1PR', u'WALL-1PB', u'WALL-1PL', u'FRONT-1', u'SB12', u'SB14', u
↪'SB15', u'RIGHT-1', u'SB21', u'SB23', u'BACK-1', u'SB32', u'SB35', u'LEFT-1', u'SB41
↪', u'SB43', u'SB45', u'SB51', u'SB54', u'WALL-1SF', u'WALL-1SR', u'WALL-1SB', u
↪'WALL-1SL']
```

```
# north facing walls
north_walls = [sf for sf in vertical_walls if sf.azimuth == 0.0]
print [sf.Name for sf in north_walls]
```

```
[u'WALL-1PB', u'SB15', u'BACK-1', u'WALL-1SB']
```

```
# north facing exterior walls
exterior_nwall = [sf for sf in north_walls if sf.Outside_Boundary_Condition ==
↪"Outdoors"]
print [sf.Name for sf in exterior_nwall]
```

```
[u'WALL-1PB', u'BACK-1', u'WALL-1SB']
```

```
# print out some more details of the north wall
north_wall_info = [(sf.Name, sf.azimuth, sf.Construction_Name) for sf in exterior_
↪nwall]
```

(continues on next page)

(continued from previous page)

```
for name, azimuth, construction in north_wall_info:
    print name, azimuth, construction
```

```
WALL-1PB 0.0 WALL-1
BACK-1 0.0 WALL-1
WALL-1SB 0.0 WALL-1
```

```
# change the construction in the exterior north walls
for wall in exterior_nwall:
    wall.Construction_Name = "NORTHERN-WALL" # make sure such a construction exists␣
↪in the model
```

```
# see the change
north_wall_info = [(sf.Name, sf.azimuth, sf.Construction_Name) for sf in exterior_
↪nwall]
for name, azimuth, construction in north_wall_info:
    print name, azimuth, construction
```

```
WALL-1PB 0.0 NORTHERN-WALL
BACK-1 0.0 NORTHERN-WALL
WALL-1SB 0.0 NORTHERN-WALL
```

```
# see this in all surfaces
for sf in surfaces:
    print sf.Name, sf.azimuth, sf.Construction_Name
```

```
WALL-1PF 180.0 WALL-1
WALL-1PR 90.0 WALL-1
WALL-1PB 0.0 NORTHERN-WALL
WALL-1PL 270.0 WALL-1
TOP-1 0.0 ROOF-1
C1-1P 0.0 CLNG-1
C2-1P 0.0 CLNG-1
C3-1P 0.0 CLNG-1
C4-1P 0.0 CLNG-1
C5-1P 180.0 CLNG-1
FRONT-1 180.0 WALL-1
C1-1 180.0 CLNG-1
F1-1 0.0 CLNG-1
SB12 45.0 INT-WALL-1
SB14 315.0 INT-WALL-1
SB15 0.0 INT-WALL-1
RIGHT-1 90.0 WALL-1
C2-1 0.0 CLNG-1
F2-1 0.0 CLNG-1
SB21 225.0 INT-WALL-1
SB23 315.784824603 INT-WALL-1
SB25 270.0 INT-WALL-1
BACK-1 0.0 NORTHERN-WALL
C3-1 0.0 CLNG-1
F3-1 0.0 CLNG-1
SB32 135.784824603 INT-WALL-1
SB34 224.215175397 INT-WALL-1
SB35 180.0 INT-WALL-1
```

(continues on next page)

(continued from previous page)

```
LEFT-1 270.0 WALL-1
C4-1 0.0 CLNG-1
F4-1 0.0 CLNG-1
SB41 135.0 INT-WALL-1
SB43 44.215175397 INT-WALL-1
SB45 90.0 INT-WALL-1
C5-1 0.0 CLNG-1
F5-1 0.0 CLNG-1
SB51 180.0 INT-WALL-1
SB52 90.0 INT-WALL-1
SB53 0.0 INT-WALL-1
SB54 270.0 INT-WALL-1
WALL-1SF 180.0 WALL-1
WALL-1SR 90.0 WALL-1
WALL-1SB 0.0 NORTHERN-WALL
WALL-1SL 270.0 WALL-1
BOTTOM-1 0.0 FLOOR-SLAB-1
F1-1S 0.0 CLNG-1
F2-1S 0.0 CLNG-1
F3-1S 0.0 CLNG-1
F4-1S 0.0 CLNG-1
F5-1S 0.0 CLNG-1
```

You can see the “NORTHERN-WALL” in the print out above.

This shows that very sophisticated modification can be made to the model rather quickly.

CHAPTER 3

HVAC Loops

3.1 Conceptual Introduction to HVAC Loops

Eppy builds threee kinds of loops for the energyplus idf file:

1. Plant Loops
2. Condensor Loops
3. Air Loops

All loops have two halves:

1. Supply side
2. Demand Side

The supply side provides the energy to the demand side that needs the energy. So the end-nodes on the supply side connect to the end-nodes on the demand side.

The loop is made up of branches connected to each other. A single branch can lead to multiple branches through a **splitter** component. Multiple branches can lead to a single branch through a **mixer** component.

Each branch is made up of components connected in series (in a line)

Eppy starts off by building the shape or topology of the loop by connecting the branches in the right order. The braches themselves have a single component in them, that is just a place holder. Usually it is a pipe component. In an air loop it would be a duct component.

The shape of the loop for the supply or demand side is quite simple.

It can be described in the following manner for the supply side

- The supply side starts single branch leads to a splitter
- The splitter leads to multiple branches
- these multiple branches come back and join in a mixer
- the mixer leads to a single branch that becomes end of the suppply side

For the demand side we have:

- The demand side starts single branch leads to a splitter
- The splitter leads to multiple branches
- these multiple branches come back and join in a mixer
- the mixer leads to a single branch that becomes end of the demand side

The two ends of the supply side connect to the two ends of the demand side.

Diagramtically the the two sides of the loop will look like this:

```
Supply Side:
------------
                -> branch1 ->
start_branch   --> branch2 --> end_branch
                -> branch3 ->
Demand Side:
------------

                  -> d_branch1 ->
d_start_branch   --> d_branch2 --> d_end_branch
                  -> d_branch3 ->
```

In eppy you could embody this is a list

```
supplyside = ['start_brandh',   [  'branch1',   'branch2',   'branch3'],   'end_branch
↪']
demandside = ['d_start_brandh', ['d_branch1', 'd_branch2', 'd_branch3'], 'd_end_branch
↪']
```

Eppy will build the build the shape/topology of the loop using the two lists above. Each branch will have a placeholder component, like a pipe or a duct:

```
branch1 = --duct--
```

Now we will have to replace the placeholder with the real components that make up the loop. For instance, branch1 should really have a pre-heat coil leading to a supply fan leading to a cooling coil leading to a heating coil:

```
new_branch = pre-heatcoil -> supplyfan -> coolingcoil -> heatingcoil
```

Eppy lets you build a new branch and you can replace branch1 with new_branch

In this manner we can build up the entire loop with the right components, once the initial toplogy is right

3.2 Building a Plant loops

Eppy can build up the topology of a plant loop using single pipes in a branch. Once we do that the simple branch in the loop we have built can be replaced with a more complex branch.

Let us try this out ans see how it works.

3.2.1 Building the topology of the loop

```
# you would normaly install eppy by doing
# python setup.py install
# or
# pip install eppy
# or
# easy_install eppy

# if you have not done so, uncomment the following three lines
import sys
# pathnameto_eppy = 'c:/eppy'
pathnameto_eppy = '../'
sys.path.append(pathnameto_eppy)
```

```
from eppy.modeleditor import IDF
from eppy import hvacbuilder

from StringIO import StringIO
iddfile = "../eppy/resources/iddfiles/Energy+V7_0_0_036.idd"
IDF.setiddname(iddfile)
```

```
# make the topology of the loop
idf = IDF(StringIO('')) # makes an empty idf file in memory with no file name
loopname = "p_loop"
sloop = ['sb0', ['sb1', 'sb2', 'sb3'], 'sb4'] # supply side of the loop
dloop = ['db0', ['db1', 'db2', 'db3'], 'db4'] # demand side of the loop
hvacbuilder.makeplantloop(idf, loopname, sloop, dloop)
idf.saveas("hhh1.idf")
```

We have made plant loop and saved it as hhh1.idf.
Now let us look at what the loop looks like.

3.2.2 Diagram of the loop

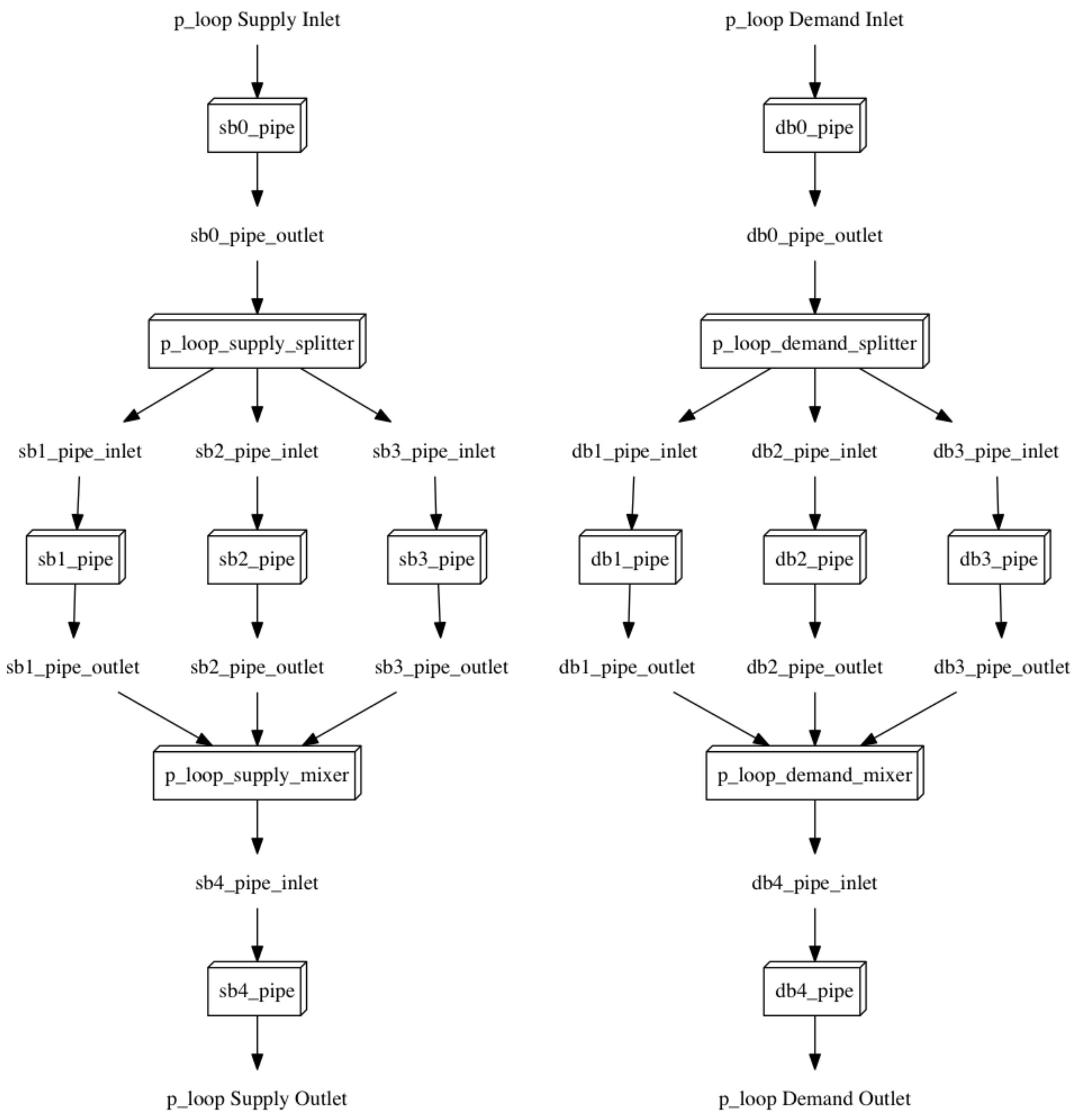
Let us use the script "eppy/useful_scripts/loopdiagrams.py" to draw this diagram

See Generating a Loop Diagram page for details on how to do this

Below is the diagram for this simple loop

Note: the supply and demnd sides are not connected in the diagram, but shown seperately for clarity

```
from eppy import ex_inits #no need to know this code, it just shows the image below
for_images = ex_inits
for_images.display_png(for_images.plantloop1) # display the image below
```



3.2.3 Modifying the topology of the loop

Let us make a new branch and replace the exisiting branch

The existing branch name is “sb0” and it contains a single pipe component sb0_pipe.

Let us replace it with a branch that has a chiller that is connected to a pipe which is turn connected to another pipe. So the connections in the new branch would look like “chiller-> pipe1->pipe2”

```
# make a new branch chiller->pipe1-> pipe2

# make a new pipe component
```

(continues on next page)

(continued from previous page)

```
pipe1 = idf.newidfobject("PIPE:ADIABATIC", 'np1')

# make a new chiller
chiller = idf.newidfobject("Chiller:Electric".upper(), 'Central_Chiller')

# make another pipe component
pipe2 = idf.newidfobject("PIPE:ADIABATIC", 'np2')

# get the loop we are trying to modify
loop = idf.getobject('PLANTLOOP', 'p_loop') # args are (key, name)
# get the branch we are trying to modify
branch = idf.getobject('BRANCH', 'sb0') # args are (key, name)
listofcomponents = [chiller, pipe1, pipe2] # the new components are connected in this␣
↪order
```

Now we are going to try to replace **branch** with the a branch made up of **listofcomponents**

- We will do this by calling the function replacebranch
- Calling replacebranch can throw an exception `WhichLoopError`
- In a moment, you will see why this exception is important

```
try:
    newbr = hvacbuilder.replacebranch(idf, loop, branch, listofcomponents, fluid=
↪'Water')
except hvacbuilder.WhichLoopError as e:
    print e
```

```
Where should this loop connect ?
CHILLER:ELECTRIC - Central_Chiller
[u'Chilled_Water_', u'Condenser_', u'Heat_Recovery_']
```

The above code throws the exception. It says that the idfobject `CHILLER:ELECTRIC - Central_Chiller` has three possible connections. The chiller has inlet outlet nodes for the following

- Chilled water
- Condenser
- Heat Recovery

eppy does not know which one to connect to, and it needs your help. We know that the chiller needs to be connected to the chilled water inlet and outlet. Simply copy `Chilled_Water_` from the text in the exception and paste as shown in the code below. (make sure you copy it exactly. eppy is a little nerdy about that)

```
# instead of passing chiller to the function, we pass a tuple (chiller, 'Chilled_
↪Water_').
# This lets eppy know where the connection should be made.
# The idfobject pipe does not have this ambiguity. So pipes do not need this extra␣
↪information
listofcomponents = [(chiller, 'Chilled_Water_'), pipe1, pipe2]

try:
    newbr = hvacbuilder.replacebranch(idf, loop, branch, listofcomponents, fluid=
↪'Water')
except Exception as e:
    print e
```

(continues on next page)

(continued from previous page)

```
else: # else will run only if the try suceeds
    print "no exception was thrown"

idf.saveas("hhh_new.idf")
```

```
no exception was thrown
```

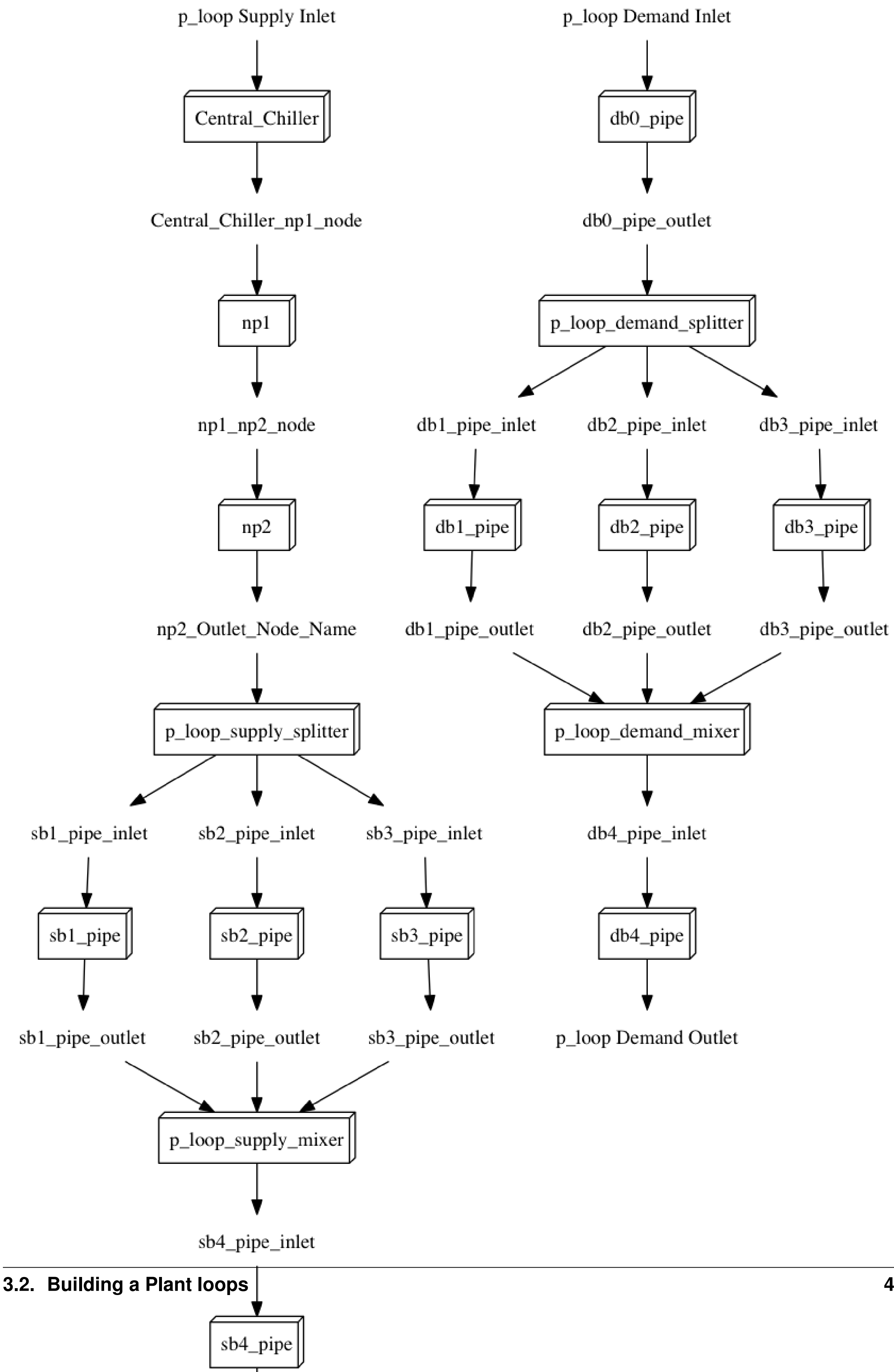
Tagential note: The "`try .. except .. else`" statement is useful here. If you have not run across it before, take a look at these two links

- http://shahriar.svbtle.com/the-possibly-forgotten-optional-else-in-python-try-statement
- https://docs.python.org/2/tutorial/errors.html

We have saved this as file "hhh_new.idf".
Let us draw the diagram of this file. (run this from eppy/eppy folder)

python ex_loopdiagram.py hhh_new.idf

```
from eppy import ex_inits #no need to know this code, it just shows the image below
for_images = ex_inits
for_images.display_png(for_images.plantloop2) # display the image below
```


p_loop Supply Inlet
Central_Chiller
Central_Chiller_np1_node
np1
np1_np2_node
np2
np2_Outlet_Node_Name
p_loop_supply_splitter
sb1_pipe_inlet
sb2_pipe_inlet
sb3_pipe_inlet
sb1_pipe
sb2_pipe
sb3_pipe
sb1_pipe_outlet
sb2_pipe_outlet
sb3_pipe_outlet
p_loop_supply_mixer
sb4_pipe_inlet
sb4_pipe
p_loop Demand Inlet
db0_pipe
db0_pipe_outlet
p_loop_demand_splitter
db1_pipe_inlet
db2_pipe_inlet
db3_pipe_inlet
db1_pipe
db2_pipe
db3_pipe
db1_pipe_outlet
db2_pipe_outlet
db3_pipe_outlet
p_loop_demand_mixer
db4_pipe_inlet
db4_pipe
p_loop Demand Outlet

This diagram shows the new components in the branch

Work flow with `WhichLoopError`

When you are writing scripts don't bother to use `try .. except` for `WhichLoopError`.

- Simply allow the exception to be raised.
- Use the information in the exception to update your code
- You may have to do this a couple of times in your script.
- In a sense you are letting eppy tell you how to update the script.

Question: I am writing an application using eppy, not just a script. The above workflow does not work for me

Response: Aha ! If that is the case, open an issue in github/eppy. We are lazy people. We don't write code unless it is needed :-)

3.2.4 Traversing the loop

It would be nice to move through the loop using functions "nextnode()" and "prevnode()"

Eppy indeed has such functions

Let us try to traverse the loop above.

```
# to traverse the loop we are going to call some functions ex_loopdiagrams.py,
# the program that draws the loop diagrams.
from eppy import ex_loopdiagram
fname = 'hhh_new.idf'
iddfile = '../eppy/resources/iddfiles/Energy+V8_0_0.idd'
edges = ex_loopdiagram.getedges(fname, iddfile)
# edges are the lines that draw the nodes in the loop.
# The term comes from graph theory in mathematics
```

The above code gets us the edges of the loop diagram. Once we have the edges, we can traverse through the diagram. Let us start with the "Central_Chiller" and work our way down.

```
from eppy import walk_hvac
firstnode = "Central_Chiller"
nextnodes = walk_hvac.nextnode(edges, firstnode)
print nextnodes
```

```
[u'np1']
```

```
nextnodes = walk_hvac.nextnode(edges, nextnodes[0])
print nextnodes
```

```
[u'np2']
```

```
nextnodes = walk_hvac.nextnode(edges, nextnodes[0])
print nextnodes
```

```
[u'p_loop_supply_splitter']
```

```
nextnodes = walk_hvac.nextnode(edges, nextnodes[0])
print nextnodes
```

```
[u'sb1_pipe', u'sb2_pipe', u'sb3_pipe']
```

This leads us to three components -> [‘sb1_pipe’, ‘sb2_pipe’, ‘sb3_pipe’]. Let us follow one of them

```
nextnodes = walk_hvac.nextnode(edges, nextnodes[0])
print nextnodes
```

```
[u'p_loop_supply_mixer']
```

```
nextnodes = walk_hvac.nextnode(edges, nextnodes[0])
print nextnodes
```

```
[u'sb4_pipe']
```

```
nextnodes = walk_hvac.nextnode(edges, nextnodes[0])
print nextnodes
```

```
[]
```

We have reached the end of this branch. There are no more components.

We can follow this in reverse using the function prevnode()

```
lastnode = 'sb4_pipe'
prevnodes = walk_hvac.prevnode(edges, lastnode)
print prevnodes
```

```
[u'p_loop_supply_mixer']
```

```
prevnodes = walk_hvac.prevnode(edges, prevnodes[0])
print prevnodes
```

```
[u'sb1_pipe', u'sb2_pipe', u'sb3_pipe']
```

```
prevnodes = walk_hvac.prevnode(edges, prevnodes[0])
print prevnodes
```

```
[u'p_loop_supply_splitter']
```

```
prevnodes = walk_hvac.prevnode(edges, prevnodes[0])
print prevnodes
```

```
[u'np2']
```

```
prevnodes = walk_hvac.prevnode(edges, prevnodes[0])
print prevnodes
```

```
[u'np1']
```

```
prevnodes = walk_hvac.prevnode(edges, prevnodes[0])
print prevnodes
```

```
[u'Central_Chiller']
```

```
prevnodes = walk_hvac.prevnode(edges, prevnodes[0])
print prevnodes
```

```
[]
```

All the way to where the loop ends

3.3 Building a Condensor loop

We build the condensor loop the same way we built the plant loop. Pipes are put as place holders for the components. Let us build a new idf file with just a condensor loop in it.

```
condensorloop_idf = IDF(StringIO(''))
loopname = "c_loop"
sloop = ['sb0', ['sb1', 'sb2', 'sb3'], 'sb4'] # supply side
dloop = ['db0', ['db1', 'db2', 'db3'], 'db4'] # demand side
theloop = hvacbuilder.makecondenserloop(condensorloop_idf, loopname, sloop, dloop)
condensorloop_idf.saveas("c_loop.idf")
```

Again, just as we did in the plant loop, we can change the components of the loop, by replacing the branchs and traverse the loop using the functions nextnode() and prevnode()

3.4 Building an Air Loop

Building an air loop is similar to the plant and condensor loop. The difference is that instead of pipes , we have ducts as placeholder components. The other difference is that we have zones on the demand side.

```
airloop_idf = IDF(StringIO(''))
loopname = "a_loop"
sloop = ['sb0', ['sb1', 'sb2', 'sb3'], 'sb4'] # supply side of the loop
dloop = ['zone1', 'zone2', 'zone3'] # zones on the demand side
hvacbuilder.makeairloop(airloop_idf, loopname, sloop, dloop)
airloop_idf.saveas("a_loop.idf")
```

Again, just as we did in the plant and condensor loop, we can change the components of the loop, by replacing the branchs and traverse the loop using the functions nextnode() and prevnode()

CHAPTER 4

Reading outputs from E+

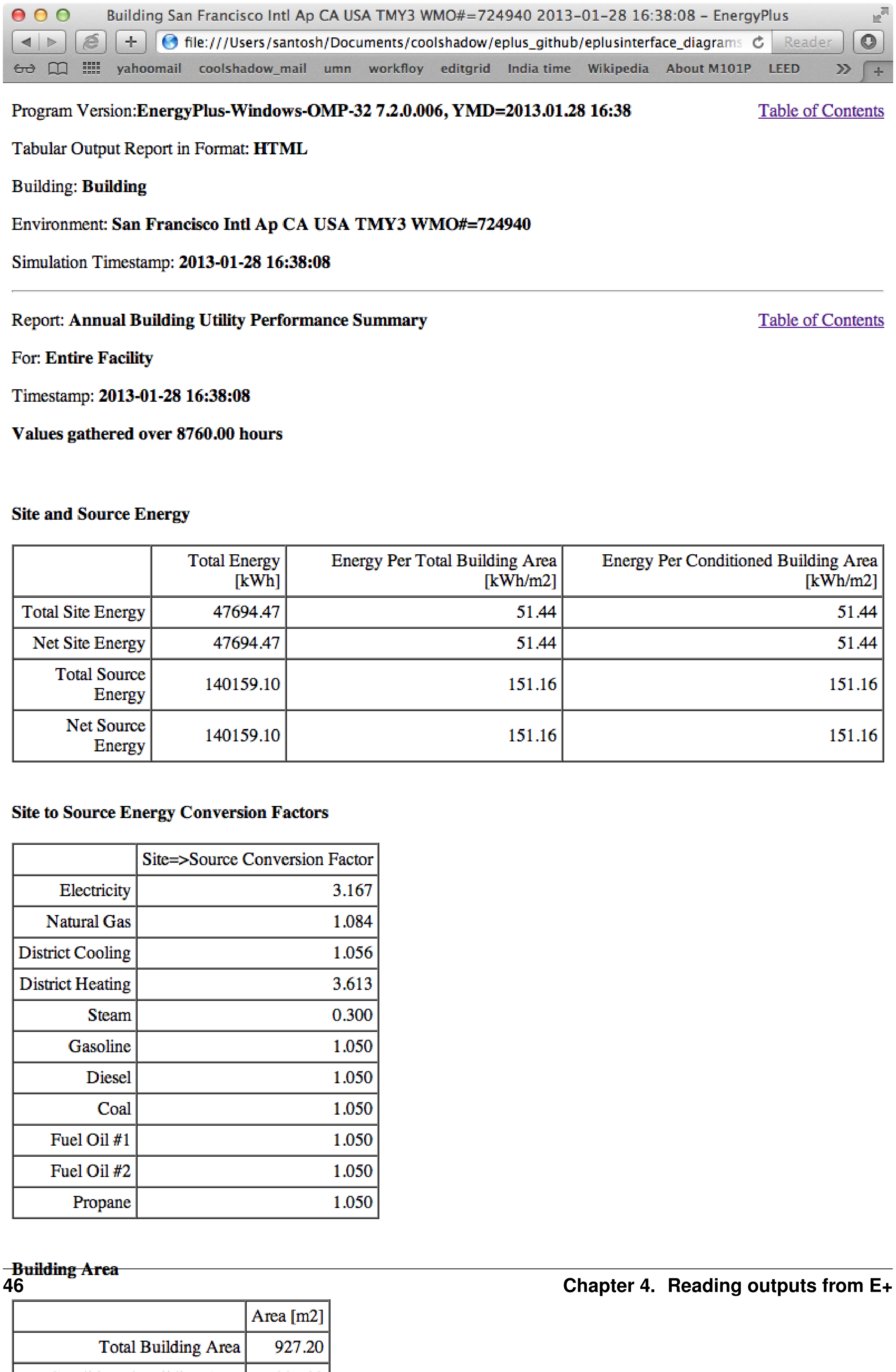
```
# some initial set up
# if you have not installed epp, and only downloaded it
# you will need the following lines
import sys
# pathnameto_eppy = 'c:/eppy'
pathnameto_eppy = '../'
sys.path.append(pathnameto_eppy)
```

4.1 Using titletable() to get at the tables

So far we have been making changes to the IDF input file. How about looking at the outputs.

Energyplus makes nice htmlout files that look like this.

```
from eppy import ex_inits #no need to know this code, it just shows the image below
for_images = ex_inits
for_images.display_png(for_images.html_snippet1) #display the image below
```


Building San Francisco Intl Ap CA USA TMY3 WMO#=724940 2013-01-28 16:38:08 - EnergyPlus
file:///Users/santosh/Documents/coolshadow/eplus_github/eplusinterface_diagrams
Reader
yahoomail
coolshadow_mail
umn
workfloy
editgrid
India time
Wikipedia
About M101P
LEED

Program Version:**EnergyPlus-Windows-OMP-32 7.2.0.006, YMD=2013.01.28 16:38** Table of Contents

Tabular Output Report in Format: **HTML**

Building: **Building**

Environment: **San Francisco Intl Ap CA USA TMY3 WMO#=724940**

Simulation Timestamp: **2013-01-28 16:38:08**

Report: **Annual Building Utility Performance Summary** Table of Contents

For: **Entire Facility**

Timestamp: **2013-01-28 16:38:08**

Values gathered over 8760.00 hours

Site and Source Energy

	Total Energy [kWh]	Energy Per Total Building Area [kWh/m2]	Energy Per Conditioned Building Area [kWh/m2]
Total Site Energy	47694.47	51.44	51.44
Net Site Energy	47694.47	51.44	51.44
Total Source Energy	140159.10	151.16	151.16
Net Source Energy	140159.10	151.16	151.16

Site to Source Energy Conversion Factors

	Site=>Source Conversion Factor
Electricity	3.167
Natural Gas	1.084
District Cooling	1.056
District Heating	3.613
Steam	0.300
Gasoline	1.050
Diesel	1.050
Coal	1.050
Fuel Oil #1	1.050
Fuel Oil #2	1.050
Propane	1.050

Building Area

	Area [m2]
Total Building Area	927.20

If you look at the clipping of the html file above, you see tables with data in them. Eppy has functions that let you access of these tables and get the data from any of it's cells.

Let us say you want to find the "Net Site Energy".

This is in table "Site and Source Energy".

The number you want is in the third row, second column and it's value is "47694.47"

Let us use eppy to extract this number

```
from eppy.results import readhtml # the eppy module with functions to read the html
fname = "../eppy/resources/outputfiles/V_7_2/5ZoneCAVtoVAVWarmestTempFlowTable_ABUPS.
↪html" # the html file you want to read
filehandle = open(fname, 'r').read() # get a file handle to the html file


htables = readhtml.titletable(filehandle) # reads the tables with their titles
```

If you open the python file readhtml.py and look at the function titletable, you can see the function documentation.

It says the following

> “”“return a list of [(title, table), …..] title = previous item with a <b> tag table = rows -> [[cell1, cell2, ..], [cell1, cell2, ..], ..]”“”

The documentation says that it returns a list. Let us take a look inside this list. Let us look at the first item in the list.

```
firstitem = htables[0]
print firstitem
```

```
(u'Site and Source Energy', [[u'', u'Total Energy [kWh]', u'Energy Per Total Building␣
↪Area [kWh/m2]', u'Energy Per Conditioned Building Area [kWh/m2]'], [u'Total Site␣
↪Energy', 47694.47, 51.44, 51.44], [u'Net Site Energy', 47694.47, 51.44, 51.44], [u
↪'Total Source Energy', 140159.1, 151.16, 151.16], [u'Net Source Energy', 140159.1,␣
↪151.16, 151.16]])
```

Ughh !!! that is ugly. Hard to see what it is.
Let us use a python module to print it pretty

```
import pprint
pp = pprint.PrettyPrinter()
pp.pprint(firstitem)
```

```
(u'Site and Source Energy',
 [[u'',
   u'Total Energy [kWh]',
   u'Energy Per Total Building Area [kWh/m2]',
   u'Energy Per Conditioned Building Area [kWh/m2]'],
  [u'Total Site Energy', 47694.47, 51.44, 51.44],
  [u'Net Site Energy', 47694.47, 51.44, 51.44],
  [u'Total Source Energy', 140159.1, 151.16, 151.16],
  [u'Net Source Energy', 140159.1, 151.16, 151.16]])
```

Nice. that is a little clearer

```
firstitem_title = firstitem[0]
pp.pprint(firstitem_title)
```

```
u'Site and Source Energy'
```

```
firstitem_table = firstitem[1]
pp.pprint(firstitem_table)
```

```
[[u'',
  u'Total Energy [kWh]',
  u'Energy Per Total Building Area [kWh/m2]',
  u'Energy Per Conditioned Building Area [kWh/m2]'],
 [u'Total Site Energy', 47694.47, 51.44, 51.44],
 [u'Net Site Energy', 47694.47, 51.44, 51.44],
 [u'Total Source Energy', 140159.1, 151.16, 151.16],
 [u'Net Source Energy', 140159.1, 151.16, 151.16]]
```

How do we get to value of “Net Site Energy”.
We know it is in the third row, second column of the table.

Easy.

```
thirdrow = firstitem_table[2] # we start counting with 0. So 0, 1, 2 is third row
print thirdrow
```

```
[u'Net Site Energy', 47694.47, 51.44, 51.44]
```

```
thirdrow_secondcolumn = thirdrow[1]
thirdrow_secondcolumn
```

```
47694.47
```

the text from the html table is in unicode.
That is why you see that weird ‘u’ letter.

Let us convert it to a floating point number

```
net_site_energy = float(thirdrow_secondcolumn)
net_site_energy
```

```
47694.47
```

Let us have a little fun with the tables.

Get the titles of all the tables

```
alltitles = [htable[0] for htable in htables]
alltitles
```

```
[u'Site and Source Energy',
 u'Site to Source Energy Conversion Factors',
 u'Building Area',
 u'End Uses',
 u'End Uses By Subcategory',
 u'Utility Use Per Conditioned Floor Area',
 u'Utility Use Per Total Floor Area',
 u'Electric Loads Satisfied',
 u'On-Site Thermal Sources',
 u'Water Source Summary',
 u'Comfort and Setpoint Not Met Summary',
 u'Comfort and Setpoint Not Met Summary']
```

Now let us grab the tables with the titles “Building Area” and “Site to Source Energy Conversion Factors”

twotables = [htable for htable in htables if htable[0] in [“Building Area”, “Site to Source Energy Conversion Factors”]] twotables

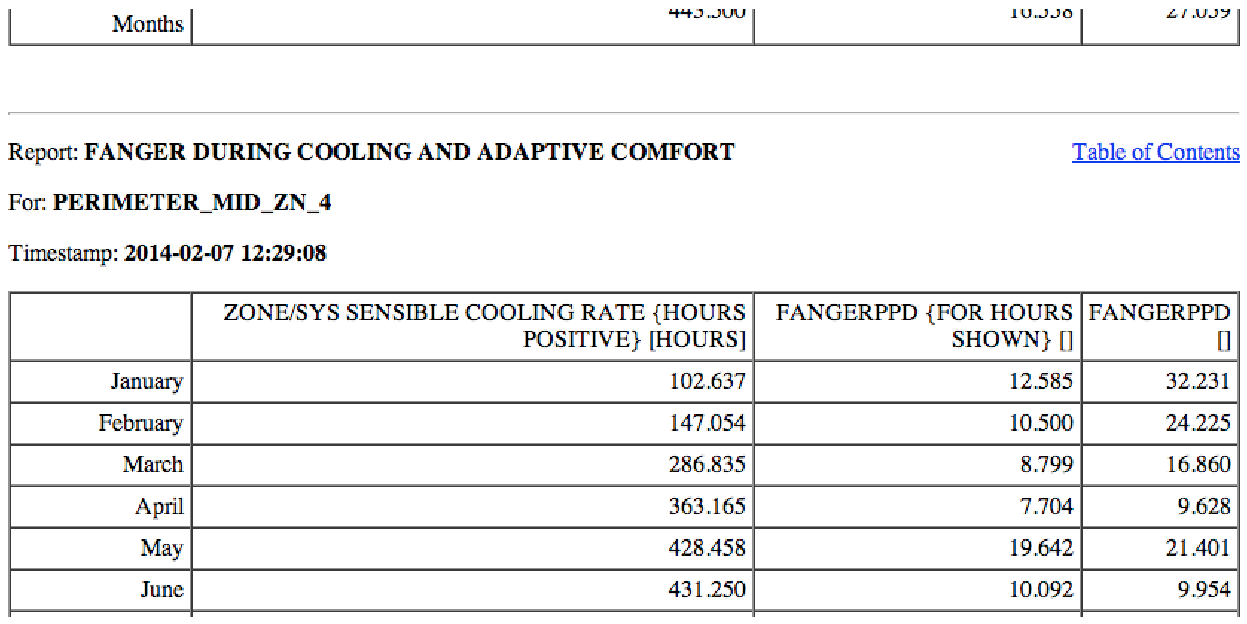
Let us leave readtables for now.

It gives us the basic functionality to read any of the tables in the html output file.

4.2 Using lines_table() to get at the tables

We have been using titletable() to get at the tables. There is a constraint using function titletable(). Titletable() assumes that there is a unique title (in HTML bold) just above the table. It is assumed that this title will adequetly describe the table. This is true in most cases and titletable() is perfectly good to use. Unfortuntely there are some tables that do not follow this rule. The snippet below shows one of them.

```
from eppy import ex_inits #no need to know this code, it just shows the image below
for_images = ex_inits
for_images.display_png(for_images.html_snippet2) # display the image below
```



Months

Report: **FANGER DURING COOLING AND ADAPTIVE COMFORT** Table of Contents

For: **PERIMETER_MID_ZN_4**

Timestamp: **2014-02-07 12:29:08**

	ZONE/SYS SENSIBLE COOLING RATE {HOURS POSITIVE} [HOURS]	FANGERPPD {FOR HOURS SHOWN} []	FANGERPPD []
January	102.637	12.585	32.231
February	147.054	10.500	24.225
March	286.835	8.799	16.860
April	363.165	7.704	9.628
May	428.458	19.642	21.401
June	431.250	10.092	9.954

Notice that the HTML snippet shows a table with three lines above it. The first two lines have information that describe the table. We need to look at both those lines to understand what the table contains. So we need a different function that will capture all those lines before the table. The funtion lines_table() described below will do this.

```
from eppy.results import readhtml # the eppy module with functions to read the html
fname = "../eppy/resources/outputfiles/V_8_1/ASHRAE30pct.PI.Final11_OfficeMedium_
↪STD2010_Chicago-baseTable.html" # the html file you want to read
filehandle = open(fname, 'r').read() # get a file handle to the html file


ltables = readhtml.lines_table(filehandle) # reads the tables with their titles
```

The html snippet shown above is the last table in HTML file we just opened. We have used lines_table() to read the tables into the variable ltables. We can get to the last table by ltable[-1]. Let us print it and see what we have.

```
import pprint
pp = pprint.PrettyPrinter()
pp.pprint(ltables[-1])
```

```
[[u'Table of Contents',
  u'Report: FANGER DURING COOLING AND ADAPTIVE COMFORT',
  u'For: PERIMETER_MID_ZN_4',
  u'Timestamp: 2014-02-07n    12:29:08'],
 [[u'',
   u'ZONE/SYS SENSIBLE COOLING RATE {HOURS POSITIVE} [HOURS]',
   u'FANGERPPD {FOR HOURS SHOWN} []',
   u'FANGERPPD []'],
  [u'January', 102.637, 12.585, 32.231],
  [u'February', 147.054, 10.5, 24.225],
  [u'March', 286.835, 8.799, 16.86],
  [u'April', 363.165, 7.704, 9.628],
  [u'May', 428.458, 19.642, 21.401],
  [u'June', 431.25, 10.092, 9.954],
  [u'July', 432.134, 8.835, 7.959],
  [u'August', 443.5, 9.743, 8.785],
  [u'September', 408.833, 15.91, 14.855],
  [u'October', 383.652, 6.919, 7.57],
  [u'November', 243.114, 8.567, 15.256],
  [u'December', 91.926, 14.298, 29.001],
  [u'xa0', u'xa0', u'xa0', u'xa0'],
  [u'Annual Sum or Average', 3762.56, 11.062, 16.458],
  [u'Minimum of Months', 91.926, 6.919, 7.57],
  [u'Maximum of Months', 443.5, 19.642, 32.231]]]
```

We can see that ltables has captured all the lines before the table. Let us make our code more explicit to see this

```
last_ltable = ltables[-1]
lines_before_table = last_ltable[0]
table_itself = last_ltable[-1]

pp.pprint(lines_before_table)
```

```
[u'Table of Contents',
 u'Report: FANGER DURING COOLING AND ADAPTIVE COMFORT',
 u'For: PERIMETER_MID_ZN_4',
 u'Timestamp: 2014-02-07n    12:29:08']
```

We found this table the easy way this time, because we knew it was the last one. How do we find it if we don't know where it is in the file ? Python comes to our rescue :-) Let assume that we want to find the table that has the following two lines before it.

- Report: FANGER DURING COOLING AND ADAPTIVE COMFORT
- For: PERIMETER_MID_ZN_4

```
line1 = 'Report: FANGER DURING COOLING AND ADAPTIVE COMFORT'
line2 = 'For: PERIMETER_MID_ZN_4'
#
# check if those two lines are before the table
line1 in lines_before_table and line2 in lines_before_table
```

```
True
```

```
# find all the tables where those two lines are before the table
[ltable for ltable in ltables
    if line1 in ltable[0] and line2 in ltable[0]]
```

```
[[[u'Table of Contents',
   u'Report: FANGER DURING COOLING AND ADAPTIVE COMFORT',
   u'For: PERIMETER_MID_ZN_4',
   u'Timestamp: 2014-02-07n    12:29:08'],
  [[u'',
    u'ZONE/SYS SENSIBLE COOLING RATE {HOURS POSITIVE} [HOURS]',
    u'FANGERPPD {FOR HOURS SHOWN} []',
    u'FANGERPPD []'],
   [u'January', 102.637, 12.585, 32.231],
   [u'February', 147.054, 10.5, 24.225],
   [u'March', 286.835, 8.799, 16.86],
   [u'April', 363.165, 7.704, 9.628],
   [u'May', 428.458, 19.642, 21.401],
   [u'June', 431.25, 10.092, 9.954],
   [u'July', 432.134, 8.835, 7.959],
   [u'August', 443.5, 9.743, 8.785],
   [u'September', 408.833, 15.91, 14.855],
   [u'October', 383.652, 6.919, 7.57],
   [u'November', 243.114, 8.567, 15.256],
   [u'December', 91.926, 14.298, 29.001],
   [u'xa0', u'xa0', u'xa0', u'xa0'],
   [u'Annual Sum or Average', 3762.56, 11.062, 16.458],
   [u'Minimum of Months', 91.926, 6.919, 7.57],
   [u'Maximum of Months', 443.5, 19.642, 32.231]]]]
```

That worked !

What if you want to find the words "FANGER" and "PERIMETER_MID_ZN_4" before the table. The following code will do it.

```
# sample code to illustrate what we are going to do
last_ltable = ltables[-1]
lines_before_table = last_ltable[0]
table_itself = last_ltable[-1]

# join lines_before_table into a paragraph of text
justtext = '\n'.join(lines_before_table)
print justtext
```

```
Table of Contents
Report: FANGER DURING COOLING AND ADAPTIVE COMFORT
For: PERIMETER_MID_ZN_4
Timestamp: 2014-02-07
    12:29:08
```

```
"FANGER" in justtext and "PERIMETER_MID_ZN_4" in justtext
```

```
True
```

```
# Let us combine the this trick to find the table
[ltable for ltable in ltables
    if "FANGER" in '\n'.join(ltable[0]) and "PERIMETER_MID_ZN_4" in '\n'.
↪join(ltable[0])]
```

```
[[[u'Table of Contents',
   u'Report: FANGER DURING COOLING AND ADAPTIVE COMFORT',
   u'For: PERIMETER_MID_ZN_4',
   u'Timestamp: 2014-02-07n    12:29:08'],
  [[u'',
    u'ZONE/SYS SENSIBLE COOLING RATE {HOURS POSITIVE} [HOURS]',
    u'FANGERPPD {FOR HOURS SHOWN} []',
    u'FANGERPPD []'],
   [u'January', 102.637, 12.585, 32.231],
   [u'February', 147.054, 10.5, 24.225],
   [u'March', 286.835, 8.799, 16.86],
   [u'April', 363.165, 7.704, 9.628],
   [u'May', 428.458, 19.642, 21.401],
   [u'June', 431.25, 10.092, 9.954],
   [u'July', 432.134, 8.835, 7.959],
   [u'August', 443.5, 9.743, 8.785],
   [u'September', 408.833, 15.91, 14.855],
   [u'October', 383.652, 6.919, 7.57],
   [u'November', 243.114, 8.567, 15.256],
   [u'December', 91.926, 14.298, 29.001],
   [u'xa0', u'xa0', u'xa0', u'xa0'],
   [u'Annual Sum or Average', 3762.56, 11.062, 16.458],
   [u'Minimum of Months', 91.926, 6.919, 7.57],
   [u'Maximum of Months', 443.5, 19.642, 32.231]]]]
```

4.3 Extracting data from the tables

The tables in the HTML page in general have text in the top header row. The first vertical row has text. The remaining cells have numbers. We can identify the numbers we need by looking at the labelin the top row and the label in the first column. Let us construct a simple example and explore this.

```
# ignore the following three lines. I am using them to construct the table below
from IPython.display import HTML
atablestring = '<TABLE cellpadding="4" style="border: 1px solid #000000; border-
↪collapse: collapse;" border="1">\n <TR>\n  <TD>&nbsp;</TD>\n  <TD>a b</TD>\n  <TD>b␣
↪c</TD>\n  <TD>c d</TD>\n </TR>\n <TR>\n  <TD>x y</TD>\n  <TD>1</TD>\n  <TD>2</TD>\n␣
↪  <TD>3</TD>\n </TR>\n <TR>\n  <TD>y z</TD>\n  <TD>4</TD>\n  <TD>5</TD>\n  <TD>6</TD>
↪\n </TR>\n <TR>\n  <TD>z z</TD>\n  <TD>7</TD>\n  <TD>8</TD>\n  <TD>9</TD>\n </TR>\n
↪</TABLE>'
```

(continues on next page)

(continued from previous page)

```
HTML(atablestring)
```

This table is actually in the follwoing form:

```
atable = [["",  "a b", "b c", "c d"],
     ["x y", 1,      2,      3 ],
     ["y z", 4,      5,      6 ],
     ["z z", 7,      8,      9 ],]
```

We can see the labels in the table. So we an look at row "x y" and column "c d". The value there is 3

right now we can get to it by saying atable[1][3]

```
print atable[1][3]
```

```
3
```

readhtml has some functions that will let us address the values by the labels. We use a structure from python called named tuples to do this. The only limitation is that the labels have to be letters or digits. Named tuples does not allow spaces in the labels. We could replace the space with an underscore ' _ '. So "a b" will become "a_b". So we can look for row "x_y" and column "c_d". Let us try this out.

```
from eppy.results import readhtml
h_table = readhtml.named_grid_h(atable)
```

```
print h_table.x_y.c_d
```

```
3
```

We can still get to the value by index

```
print h_table[0][2]
```

```
3
```

Note that we used atable[1][3], but here we used h_table[0][2]. That is because h_table does not count the rows and columns where the labels are.

We can also do the following:

```
print h_table.x_y[2]
# or
print h_table[0].c_d
```

```
3
3
```

Wow . . . that is pretty cool. What if we want to just check what the labels are ?

```
print h_table._fields
```

```
('x_y', 'y_z', 'z_z')
```

That gives us the horizontal lables. How about the vertical labels ?

```
h_table.x_y._fields
```

```
('a_b', 'b_c', 'c_d')
```

There you go !!!

How about if I want to use the labels differently ? Say I want to refer to the row first and then to the column. That woul be saying table.c_d.x_y. We can do that by using a different function

```
v_table = readhtml.named_grid_v(atable)
print v_table.c_d.x_y
```

```
3
```

And we can do the following

```
print v_table[2][0]
print v_table.c_d[0]
print v_table[2].x_y
```

```
3
3
3
```

Let us try to get the numbers in the first column and then get their sum

```
v_table.a_b
```

```
ntrow(x_y=1, y_z=4, z_z=7)
```

Look like we got the right column. But not in the right format. We really need a list of numbers

```
[cell for cell in v_table.a_b]
```

```
[1, 4, 7]
```

That looks like waht we wanted. Now let us get the sum

```
values_in_first_column = [cell for cell in v_table.a_b]
print values_in_first_column
print sum(values_in_first_column) # sum is a builtin function that will sum a list
```

```
[1, 4, 7]
12
```

To get the first row we use the variable h_table

```
values_in_first_row = [cell for cell in h_table.x_y]
print values_in_first_row
print sum(values_in_first_row)
```

```
[1, 2, 3]
6
```

CHAPTER 5

New functions

These are recently written functions that have not made it into the main documentation

5.1 Python Lesson: Errors and Exceptions

```
# you would normaly install eppy by doing
# python setup.py install
# or
# pip install eppy
# or
# easy_install eppy

# if you have not done so, uncomment the following three lines
import sys
# pathnameto_eppy = 'c:/eppy'
pathnameto_eppy = '../'
sys.path.append(pathnameto_eppy)
```

When things go wrong in your eppy script, you get "Errors and Exceptions".

To know more about how this works in python and eppy, take a look at Python: Errors and Exceptions

5.2 Setting IDD name

When you work with Energyplus you are working with **idf** files (files that have the extension *.idf). There is another file that is very important, called the **idd** file. This is the file that defines all the objects in Energyplus. Esch version of Energyplus has a different **idd** file.

So eppy needs to know which **idd** file to use. Only one **idd** file can be used in a script or program. This means that you cannot change the **idd** file once you have selected it. Of course you have to first select an **idd** file before eppy can work.

If you use eppy and break the above rules, eppy will raise an exception. So let us use eppy incorrectly and make eppy raise the exception, just see how that happens.

First let us try to open an **idf** file without setting an **idd** file.

```
from eppy import modeleditor
from eppy.modeleditor import IDF
fname1 = "../eppy/resources/idffiles/V_7_2/smallfile.idf"
```

Now let us open file fname1 without setting the **idd** file

```
try:
    idf1 = IDF(fname1)
except Exception, e:
    raise e
```

```
---------------------------------------------------------------------------

IDDNotSetError                            Traceback (most recent call last)

<ipython-input-7-44ad2b53d42c> in <module>()
      2     idf1 = IDF(fname1)
      3 except Exception, e:
----> 4     raise e
      5


IDDNotSetError: IDD file needed to read the idf file. Set it using IDF.
↪setiddname(iddfile)
```

OK. It does not let you do that and it raises an exception

So let us set the **idd** file and then open the idf file

```
iddfile = "../eppy/resources/iddfiles/Energy+V7_2_0.idd"
IDF.setiddname(iddfile)
idf1 = IDF(fname1)
```

That worked without raising an exception

Now let us try to change the **idd** file. Eppy should not let you do this and should raise an exception.

```
try:
    IDF.setiddname("anotheridd.idd")
except Exception, e:
    raise e
```

```
---------------------------------------------------------------------------

IDDAlreadySetError                        Traceback (most recent call last)

<ipython-input-9-52df819ac489> in <module>()
      2     IDF.setiddname("anotheridd.idd")
      3 except Exception, e:
----> 4     raise e
      5


IDDAlreadySetError: IDD file is set to: ../eppy/resources/iddfiles/Energy+V7_2_0.idd
```

Excellent!! It raised the exception we were expecting.

5.3 Check range for fields

The fields of idf objects often have a range of legal values. The following functions will let you discover what that range is and test if your value lies within that range

demonstrate two new functions:

- EpBunch.getrange(fieldname) # will return the ranges for that field
- EpBunch.checkrange(fieldname) # will throw an exception if the value is outside the range

```
from eppy import modeleditor
from eppy.modeleditor import IDF
iddfile = "../eppy/resources/iddfiles/Energy+V7_2_0.idd"
fname1 = "../eppy/resources/idffiles/V_7_2/smallfile.idf"
```

```
# IDF.setiddname(iddfile)# idd ws set further up in this page
idf1 = IDF(fname1)
```

```
building = idf1.idfobjects['building'.upper()][0]
print building
```

```
BUILDING,
    Empire State Building,    !- Name
    30.0,                     !- North Axis
    City,                     !- Terrain
    0.04,                     !- Loads Convergence Tolerance Value
    0.4,                      !- Temperature Convergence Tolerance Value
    FullExterior,             !- Solar Distribution
    25,                       !- Maximum Number of Warmup Days
    6;                        !- Minimum Number of Warmup Days
```

```
print building.getrange("Loads_Convergence_Tolerance_Value")
```

```
{u'maximum<': None, u'minimum': None, u'type': u'real', u'maximum': 0.5, u'minimum>':␣
↪0.0}
```

```
print building.checkrange("Loads_Convergence_Tolerance_Value")
```

```
0.04
```

Let us set these values outside the range and see what happens

```
building.Loads_Convergence_Tolerance_Value = 0.6
from eppy.bunch_subclass import RangeError
try:
    print building.checkrange("Loads_Convergence_Tolerance_Value")
except RangeError, e:
    raise e
```

```
---------------------------------------------------------------------------

RangeError                                Traceback (most recent call last)

<ipython-input-15-a824cb1ec673> in <module>()
      4     print building.checkrange("Loads_Convergence_Tolerance_Value")
      5 except RangeError, e:
----> 6     raise e
      7


RangeError: Value 0.6 is not less or equal to the 'maximum' of 0.5
```

So the Range Check works

5.4 Looping through all the fields in an idf object

We have seen how to check the range of field in the idf object. What if you want to do a *range check* on all the fields in an idf object ? To do this we will need a list of all the fields in the idf object. We can do this easily by the following line

```
print building.fieldnames
```

```
[u'key', u'Name', u'North_Axis', u'Terrain', u'Loads_Convergence_Tolerance_Value', u
↪'Temperature_Convergence_Tolerance_Value', u'Solar_Distribution', u'Maximum_Number_
↪of_Warmup_Days', u'Minimum_Number_of_Warmup_Days']
```

So let us use this

```
for fieldname in building.fieldnames:
    print "%s = %s" % (fieldname, building[fieldname])
```

```
key = BUILDING
Name = Empire State Building
North_Axis = 30.0
Terrain = City
Loads_Convergence_Tolerance_Value = 0.6
Temperature_Convergence_Tolerance_Value = 0.4
Solar_Distribution = FullExterior
Maximum_Number_of_Warmup_Days = 25
Minimum_Number_of_Warmup_Days = 6
```

Now let us test if the values are in the legal range. We know that "Loads_Convergence_Tolerance_Value" is out of range

```
from eppy.bunch_subclass import RangeError
for fieldname in building.fieldnames:
    try:
        building.checkrange(fieldname)
        print "%s = %s #-in range" % (fieldname, building[fieldname],)
    except RangeError as e:
        print "%s = %s #-****OUT OF RANGE****" % (fieldname, building[fieldname],)
```

```
key = BUILDING #-in range
Name = Empire State Building #-in range
```

```
North_Axis = 30.0 #-in range
Terrain = City #-in range
Loads_Convergence_Tolerance_Value = 0.6 #-**OUT OF RANGE**
Temperature_Convergence_Tolerance_Value = 0.4 #-in range
Solar_Distribution = FullExterior #-in range
Maximum_Number_of_Warmup_Days = 25 #-in range
Minimum_Number_of_Warmup_Days = 6 #-in range
```

You see, we caught the out of range value

5.5 Blank idf file

Until now in all our examples, we have been reading an idf file from disk:

- How do I create a blank new idf file
- give it a file name
- Save it to the disk

Here are the steps to do that

```python
# some initial steps
from eppy.modeleditor import IDF
iddfile = "../eppy/resources/iddfiles/Energy+V7_2_0.idd"
# IDF.setiddname(iddfile) # Has already been set

# - Let us first open a file from the disk
fname1 = "../eppy/resources/idffiles/V_7_2/smallfile.idf"
idf_fromfilename = IDF(fname1) # initialize the IDF object with the file name

idf_fromfilename.printidf()
```

```
VERSION,
    7.3;                        !- Version Identifier

SIMULATIONCONTROL,
    Yes,                        !- Do Zone Sizing Calculation
    Yes,                        !- Do System Sizing Calculation
    Yes,                        !- Do Plant Sizing Calculation
    No,                         !- Run Simulation for Sizing Periods
    Yes;                        !- Run Simulation for Weather File Run Periods

BUILDING,
    Empire State Building,      !- Name
    30.0,                       !- North Axis
    City,                       !- Terrain
    0.04,                       !- Loads Convergence Tolerance Value
    0.4,                        !- Temperature Convergence Tolerance Value
    FullExterior,               !- Solar Distribution
    25,                         !- Maximum Number of Warmup Days
    6;                          !- Minimum Number of Warmup Days

SITE:LOCATION,
    CHICAGO_IL_USA TMY2-94846,    !- Name
    41.78,                      !- Latitude
    -87.75,                     !- Longitude
```

(continues on next page)

(continued from previous page)

```
    -6.0,                     !- Time Zone
    190.0;                    !- Elevation
```

```
# - now let us open a file from the disk differently
fname1 = "../eppy/resources/idffiles/V_7_2/smallfile.idf"
fhandle = open(fname1, 'r') # open the file for reading and assign it a file handle
idf_fromfilehandle = IDF(fhandle) # initialize the IDF object with the file handle

idf_fromfilehandle.printidf()
```

```
VERSION,
    7.3;                      !- Version Identifier

SIMULATIONCONTROL,
    Yes,                      !- Do Zone Sizing Calculation
    Yes,                      !- Do System Sizing Calculation
    Yes,                      !- Do Plant Sizing Calculation
    No,                       !- Run Simulation for Sizing Periods
    Yes;                      !- Run Simulation for Weather File Run Periods

BUILDING,
    Empire State Building,    !- Name
    30.0,                     !- North Axis
    City,                     !- Terrain
    0.04,                     !- Loads Convergence Tolerance Value
    0.4,                      !- Temperature Convergence Tolerance Value
    FullExterior,             !- Solar Distribution
    25,                       !- Maximum Number of Warmup Days
    6;                        !- Minimum Number of Warmup Days

SITE:LOCATION,
    CHICAGO_IL_USA TMY2-94846,    !- Name
    41.78,                    !- Latitude
    -87.75,                   !- Longitude
    -6.0,                     !- Time Zone
    190.0;                    !- Elevation
```

```
# So IDF object can be initialized with either a file name or a file handle

# - How do I create a blank new idf file
idftxt = "" # empty string
from StringIO import StringIO
fhandle = StringIO(idftxt) # we can make a file handle of a string
idf_emptyfile = IDF(fhandle) # initialize the IDF object with the file handle

idf_emptyfile.printidf()
```

It did not print anything. Why should it. It was empty.

What if we give it a string that was not blank

```
# - The string does not have to be blank
idftxt = "VERSION, 7.3;" # Not an emplty string. has just the version number
fhandle = StringIO(idftxt) # we can make a file handle of a string
idf_notemptyfile = IDF(fhandle) # initialize the IDF object with the file handle

```

(continues on next page)

(continued from previous page)

```
idf_notemptyfile.printidf()
```

```
VERSION,
    7.3;                      !- Version Identifier
```

Aha !

Now let us give it a file name

```
# - give it a file name
idf_notemptyfile.idfname = "notemptyfile.idf"
# - Save it to the disk
idf_notemptyfile.save()
```

Let us confirm that the file was saved to disk

```
txt = open("notemptyfile.idf", 'r').read()# read the file from the disk
print txt
```

```
!- Darwin Line endings

VERSION,
    7.3;                      !- Version Identifier
```

Yup ! that file was saved. Let us delete it since we were just playing

```
import os
os.remove("notemptyfile.idf")
```

5.6 Deleting, copying/adding and making new idfobjects

5.6.1 Making a new idf object

Let us start with a blank idf file and make some new "MATERIAL" objects in it

```
# making a blank idf object
blankstr = ""
from StringIO import StringIO
idf = IDF(StringIO(blankstr))
```

To make and add a new idfobject object, we use the function IDF.newidfobject(). We want to make an object of type "MATERIAL"

```
newobject = idf.newidfobject("material".upper()) # the key for the object type has to␣
↪be in upper case
                                     # .upper() makes it upper case
```

```
print newobject
```

```
MATERIAL,
    ,                         !- Name
    ,                         !- Roughness
```

(continues on next page)

(continued from previous page)

```
    ,                        !- Thickness
    ,                        !- Conductivity
    ,                        !- Density
    ,                        !- Specific Heat
    0.9,                     !- Thermal Absorptance
    0.7,                     !- Solar Absorptance
    0.7;                     !- Visible Absorptance
```

Let us give this a name, say “Shiny new material object”

```
newobject.Name = "Shiny new material object"
print newobject
```

```
MATERIAL,
    Shiny new material object,    !- Name
    ,                        !- Roughness
    ,                        !- Thickness
    ,                        !- Conductivity
    ,                        !- Density
    ,                        !- Specific Heat
    0.9,                     !- Thermal Absorptance
    0.7,                     !- Solar Absorptance
    0.7;                     !- Visible Absorptance
```

```
anothermaterial = idf.newidfobject("material".upper())
anothermaterial.Name = "Lousy material"
thirdmaterial = idf.newidfobject("material".upper())
thirdmaterial.Name = "third material"
print thirdmaterial
```

```
MATERIAL,
    third material,          !- Name
    ,                        !- Roughness
    ,                        !- Thickness
    ,                        !- Conductivity
    ,                        !- Density
    ,                        !- Specific Heat
    0.9,                     !- Thermal Absorptance
    0.7,                     !- Solar Absorptance
    0.7;                     !- Visible Absorptance
```

Let us look at all the “MATERIAL” objects

```
print idf.idfobjects["MATERIAL"]
```

```
[
MATERIAL,
    Shiny new material object,    !- Name
    ,                        !- Roughness
    ,                        !- Thickness
    ,                        !- Conductivity
    ,                        !- Density
    ,                        !- Specific Heat
    0.9,                     !- Thermal Absorptance
    0.7,                     !- Solar Absorptance
```

(continues on next page)

(continued from previous page)

```
    0.7;                      !- Visible Absorptance
,
MATERIAL,
    Lousy material,           !- Name
    ,                         !- Roughness
    ,                         !- Thickness
    ,                         !- Conductivity
    ,                         !- Density
    ,                         !- Specific Heat
    0.9,                      !- Thermal Absorptance
    0.7,                      !- Solar Absorptance
    0.7;                      !- Visible Absorptance
,
MATERIAL,
    third material,           !- Name
    ,                         !- Roughness
    ,                         !- Thickness
    ,                         !- Conductivity
    ,                         !- Density
    ,                         !- Specific Heat
    0.9,                      !- Thermal Absorptance
    0.7,                      !- Solar Absorptance
    0.7;                      !- Visible Absorptance
]
```

As we can see there are three MATERIAL idfobjects. They are:

1. Shiny new material object
2. Lousy material
3. third material

5.6.2 Deleting an idf object

Let us remove 2. Lousy material. It is the second material in the list. So let us remove the second material

```
idf.popidfobject('MATERIAL', 1) # first material is '0', second is '1'
```

```
MATERIAL,
    Lousy material,           !- Name
    ,                         !- Roughness
    ,                         !- Thickness
    ,                         !- Conductivity
    ,                         !- Density
    ,                         !- Specific Heat
    0.9,                      !- Thermal Absorptance
    0.7,                      !- Solar Absorptance
    0.7;                      !- Visible Absorptance
```

```
print idf.idfobjects['MATERIAL']
```

```
[
MATERIAL,
    Shiny new material object,    !- Name
```

(continues on next page)

(continued from previous page)

```
    ,                           !- Roughness
    ,                           !- Thickness
    ,                           !- Conductivity
    ,                           !- Density
    ,                           !- Specific Heat
    0.9,                        !- Thermal Absorptance
    0.7,                        !- Solar Absorptance
    0.7;                        !- Visible Absorptance
,
MATERIAL,
    third material,             !- Name
    ,                           !- Roughness
    ,                           !- Thickness
    ,                           !- Conductivity
    ,                           !- Density
    ,                           !- Specific Heat
    0.9,                        !- Thermal Absorptance
    0.7,                        !- Solar Absorptance
    0.7;                        !- Visible Absorptance
]
```

You can see that the second material is gone ! Now let us remove the first material, but do it using a different function

```
firstmaterial = idf.idfobjects['MATERIAL'][-1]
```

```
idf.removeidfobject(firstmaterial)
```

```
print idf.idfobjects['MATERIAL']
```

```
[
MATERIAL,
    Shiny new material object,    !- Name
    ,                           !- Roughness
    ,                           !- Thickness
    ,                           !- Conductivity
    ,                           !- Density
    ,                           !- Specific Heat
    0.9,                        !- Thermal Absorptance
    0.7,                        !- Solar Absorptance
    0.7;                        !- Visible Absorptance
]
```

So we have two ways of deleting an idf object:

1. popidfobject -> give it the idf key: "MATERIAL", and the index number
2. removeidfobject -> give it the idf object to be deleted

5.6.3 Copying/Adding an idf object

Having deleted two "MATERIAL" objects, we have only one left. Let us make a copy of this object and add it to our idf file

```
onlymaterial = idf.idfobjects["MATERIAL"][0]
```

```
idf.copyidfobject(onlymaterial)
```

```
print idf.idfobjects["MATERIAL"]
```

```
[
MATERIAL,
    Shiny new material object,    !- Name
    ,                            !- Roughness
    ,                            !- Thickness
    ,                            !- Conductivity
    ,                            !- Density
    ,                            !- Specific Heat
    0.9,                         !- Thermal Absorptance
    0.7,                         !- Solar Absorptance
    0.7;                         !- Visible Absorptance
,
MATERIAL,
    Shiny new material object,    !- Name
    ,                            !- Roughness
    ,                            !- Thickness
    ,                            !- Conductivity
    ,                            !- Density
    ,                            !- Specific Heat
    0.9,                         !- Thermal Absorptance
    0.7,                         !- Solar Absorptance
    0.7;                         !- Visible Absorptance
]
```

So now we have a copy of the material. You can use this method to copy idf objects from other idf files too.

5.7 Making an idf object with named arguments

What if we wanted to make an idf object with values for it's fields? We can do that too.

```
gypboard = idf.newidfobject('MATERIAL', Name="G01a 19mm gypsum board",
                            Roughness="MediumSmooth",
                            Thickness=0.019,
                            Conductivity=0.16,
                            Density=800,
                            Specific_Heat=1090)
```

```
print gypboard
```

```
MATERIAL,
    G01a 19mm gypsum board,    !- Name
    MediumSmooth,                !- Roughness
    0.019,                       !- Thickness
    0.16,                        !- Conductivity
    800,                         !- Density
    1090,                        !- Specific Heat
    0.9,                         !- Thermal Absorptance
    0.7,                         !- Solar Absorptance
    0.7;                         !- Visible Absorptance
```

newidfobject() also fills in the default values like "Thermal Absorptance", "Solar Absorptance", etc.

```
print idf.idfobjects["MATERIAL"]
```

```
[
MATERIAL,
    Shiny new material object,    !- Name
    ,                            !- Roughness
    ,                            !- Thickness
    ,                            !- Conductivity
    ,                            !- Density
    ,                            !- Specific Heat
    0.9,                         !- Thermal Absorptance
    0.7,                         !- Solar Absorptance
    0.7;                         !- Visible Absorptance
,
MATERIAL,
    Shiny new material object,    !- Name
    ,                            !- Roughness
    ,                            !- Thickness
    ,                            !- Conductivity
    ,                            !- Density
    ,                            !- Specific Heat
    0.9,                         !- Thermal Absorptance
    0.7,                         !- Solar Absorptance
    0.7;                         !- Visible Absorptance
,
MATERIAL,
    G01a 19mm gypsum board,    !- Name
    MediumSmooth,                !- Roughness
    0.019,                       !- Thickness
    0.16,                        !- Conductivity
    800,                         !- Density
    1090,                        !- Specific Heat
    0.9,                         !- Thermal Absorptance
    0.7,                         !- Solar Absorptance
    0.7;                         !- Visible Absorptance
]
```

5.8 Renaming an idf object

It is easy to rename an idf object. If we want to rename the gypboard object that we created above, we simply say:

gypboard.Name = "a new name".

But this could create a problem. What if this gypboard is part of a "CONSTRUCTION" object. The construction object will refer to the gypboard by name. If we change the name of the gypboard, we should change it in the construction object.

But there may be many constructions objects using the gypboard. Now we will have to change it in all those construction objects. Sounds painfull.

Let us try this with an example:

```
interiorwall = idf.newidfobject("CONSTRUCTION", Name="Interior Wall",
                 Outside_Layer="G01a 19mm gypsum board",
```

(continues on next page)

(continued from previous page)

```
                 Layer_2="Shiny new material object",
                 Layer_3="G01a 19mm gypsum board")
print interiorwall
```

```
CONSTRUCTION,
    Interior Wall,            !- Name
    G01a 19mm gypsum board,    !- Outside Layer
    Shiny new material object,    !- Layer 2
    G01a 19mm gypsum board;    !- Layer 3
```

to rename gypboard and have that name change in all the places we call modeleditor.rename(idf, key, oldname, newname)

```
modeleditor.rename(idf, "MATERIAL", "G01a 19mm gypsum board", "peanut butter")
```

```
MATERIAL,
    peanut butter,            !- Name
    MediumSmooth,             !- Roughness
    0.019,                    !- Thickness
    0.16,                     !- Conductivity
    800,                      !- Density
    1090,                     !- Specific Heat
    0.9,                      !- Thermal Absorptance
    0.7,                      !- Solar Absorptance
    0.7;                      !- Visible Absorptance
```

```
print interiorwall
```

```
CONSTRUCTION,
    Interior Wall,            !- Name
    peanut butter,            !- Outside Layer
    Shiny new material object,    !- Layer 2
    peanut butter;            !- Layer 3
```

Now we have "peanut butter" everywhere. At least where we need it. Let us look at the entir idf file, just to be sure

```
idf.printidf()
```

```
MATERIAL,
    Shiny new material object,    !- Name
    ,                         !- Roughness
    ,                         !- Thickness
    ,                         !- Conductivity
    ,                         !- Density
    ,                         !- Specific Heat
    0.9,                      !- Thermal Absorptance
    0.7,                      !- Solar Absorptance
    0.7;                      !- Visible Absorptance

MATERIAL,
    Shiny new material object,    !- Name
    ,                         !- Roughness
    ,                         !- Thickness
    ,                         !- Conductivity
```

(continues on next page)

(continued from previous page)

```
    ,                          !- Density
    ,                          !- Specific Heat
    0.9,                       !- Thermal Absorptance
    0.7,                       !- Solar Absorptance
    0.7;                       !- Visible Absorptance

MATERIAL,
    peanut butter,             !- Name
    MediumSmooth,              !- Roughness
    0.019,                     !- Thickness
    0.16,                      !- Conductivity
    800,                       !- Density
    1090,                      !- Specific Heat
    0.9,                       !- Thermal Absorptance
    0.7,                       !- Solar Absorptance
    0.7;                       !- Visible Absorptance

CONSTRUCTION,
    Interior Wall,             !- Name
    peanut butter,             !- Outside Layer
    Shiny new material object,    !- Layer 2
    peanut butter;             !- Layer 3
```

5.8.1 Turn off default values

Can I turn off the defautl values. Yes you can:

```
defaultmaterial = idf.newidfobject("MATERIAL",
                                    Name='with default')
print defaultmaterial
nodefaultmaterial = idf.newidfobject("MATERIAL",
                                    Name='Without default',
                                 defaultvalues=False)
print nodefaultmaterial
```

```
MATERIAL,
    with default,              !- Name
    ,                          !- Roughness
    ,                          !- Thickness
    ,                          !- Conductivity
    ,                          !- Density
    ,                          !- Specific Heat
    0.9,                       !- Thermal Absorptance
    0.7,                       !- Solar Absorptance
    0.7;                       !- Visible Absorptance


MATERIAL,
    Without default;           !- Name
```

- But why would you want to turn it off.
- Well . . . . sometimes you have to
- Try it with the object `DAYLIGHTING:CONTROLS`, and you will see the need for `defaultvalues=False`

Of course, internally EnergyPlus will still use the default values it it is left blank. If just won't turn up in the IDF file.

5.9 Zone area and volume

The idf file has zones with surfaces and windows. It is easy to get the attributes of the surfaces and windows as we have seen in the tutorial. Let us review this once more:

```
from eppy import modeleditor
from eppy.modeleditor import IDF
iddfile = "../eppy/resources/iddfiles/Energy+V7_2_0.idd"
fname1 = "../eppy/resources/idffiles/V_7_2/box.idf"
# IDF.setiddname(iddfile)
```

```
idf = IDF(fname1)
```

```
surfaces = idf.idfobjects["BuildingSurface:Detailed".upper()]
surface = surfaces[0]
print "area = %s" % (surface.area, )
print "tilt = %s" % (surface.tilt, )
print "azimuth = %s" % (surface.azimuth, )
```

```
area = 30.0
tilt = 180.0
azimuth = 0.0
```

Can we do the same for zones ?

Not yet .. not yet. Not in this version on eppy

But we can still get the area and volume of the zone

```
zones = idf.idfobjects["ZONE"]
zone = zones[0]
area = modeleditor.zonearea(idf, zone.Name)
volume = modeleditor.zonevolume(idf, zone.Name)
print "zone area = %s" % (area, )
print "zone volume = %s" % (volume, )
```

```
zone area = 30.0
zone volume = 90.0
```

Not as slick, but still pretty easy

Some notes on the zone area calculation:

- area is calculated by summing up all the areas of the floor surfaces
- if there are no floors, then the sum of ceilings and roof is taken as zone area
- if there are no floors, ceilings or roof, we are out of luck. The function returns 0

5.10 Using JSON to update idf

we are going to update `idf1` using json. First let us print the `idf1` before changing it, so we can see what has changed once we make an update

```
idf1.printidf()
```

```
VERSION,
    7.3;                      !- Version Identifier

SIMULATIONCONTROL,
    Yes,                      !- Do Zone Sizing Calculation
    Yes,                      !- Do System Sizing Calculation
    Yes,                      !- Do Plant Sizing Calculation
    No,                       !- Run Simulation for Sizing Periods
    Yes;                      !- Run Simulation for Weather File Run Periods

BUILDING,
    Empire State Building,    !- Name
    30.0,                     !- North Axis
    City,                     !- Terrain
    0.6,                      !- Loads Convergence Tolerance Value
    0.4,                      !- Temperature Convergence Tolerance Value
    FullExterior,             !- Solar Distribution
    25,                       !- Maximum Number of Warmup Days
    6;                        !- Minimum Number of Warmup Days

SITE:LOCATION,
    CHICAGO_IL_USA TMY2-94846,    !- Name
    41.78,                    !- Latitude
    -87.75,                   !- Longitude
    -6.0,                     !- Time Zone
    190.0;                    !- Elevation
```

```
import eppy.json_functions as json_functions
json_str = {"idf.VERSION..Version_Identifier":8.5,
            "idf.SIMULATIONCONTROL..Do_Zone_Sizing_Calculation": "No",
            "idf.SIMULATIONCONTROL..Do_System_Sizing_Calculation": "No",
            "idf.SIMULATIONCONTROL..Do_Plant_Sizing_Calculation": "No",
            "idf.BUILDING.Empire State Building.North_Axis": 52,
            "idf.BUILDING.Empire State Building.Terrain": "Rural",
            }
json_functions.updateidf(idf1, json_str)
```

```
idf1.printidf()
```

```
VERSION,
    8.5;                      !- Version Identifier

SIMULATIONCONTROL,
    No,                       !- Do Zone Sizing Calculation
    No,                       !- Do System Sizing Calculation
    No,                       !- Do Plant Sizing Calculation
    No,                       !- Run Simulation for Sizing Periods
    Yes;                      !- Run Simulation for Weather File Run Periods

BUILDING,
    Empire State Building,    !- Name
    52,                       !- North Axis
    Rural,                    !- Terrain
    0.6,                      !- Loads Convergence Tolerance Value
```

(continues on next page)

(continued from previous page)

```
    0.4,                     !- Temperature Convergence Tolerance Value
    FullExterior,            !- Solar Distribution
    25,                      !- Maximum Number of Warmup Days
    6;                       !- Minimum Number of Warmup Days

SITE:LOCATION,
    CHICAGO_IL_USA TMY2-94846,    !- Name
    41.78,                   !- Latitude
    -87.75,                  !- Longitude
    -6.0,                    !- Time Zone
    190.0;                   !- Elevation
```

Compare the first printidf() and the second printidf().

The syntax of the json string is described below:

```
idf.BUILDING.Empire State Building.Terrain": "Rural"

The key fields are seperated by dots. Let us walk through each field:

idf -> make a change to the idf. (in the future there may be changes that are not␣
↪related to idf)
BUILDING -> the key for object to be changed
Empire State Building -> The name of the object. In other word - the value of the␣
↪field `Name`
Terrain -> the field to be changed

"Rural" -> the new value of the field

If the object does not have a `Name` field, you leave a blank between the two dots␣
↪and the first object will be changed.
This is done for the version number change.

"idf.VERSION..Version_Identifier":8.5
```

You can also create a new object using JSON, using the same syntax. Take a look at this:

```
json_str = {"idf.BUILDING.Taj.Terrain": "Rural",}
json_functions.updateidf(idf1, json_str)
idf1.idfobjects['building'.upper()]
# of course, you are creating an invalid E+ file. But we are just playing here.
```

```
[
BUILDING,
    Empire State Building,   !- Name
    52,                      !- North Axis
    Rural,                   !- Terrain
    0.6,                     !- Loads Convergence Tolerance Value
    0.4,                     !- Temperature Convergence Tolerance Value
    FullExterior,            !- Solar Distribution
    25,                      !- Maximum Number of Warmup Days
    6;                       !- Minimum Number of Warmup Days
,
BUILDING,
    Taj,                     !- Name
    0.0,                     !- North Axis
    Rural,                   !- Terrain
```

(continues on next page)

(continued from previous page)

```
    0.04,                     !- Loads Convergence Tolerance Value
    0.4,                      !- Temperature Convergence Tolerance Value
    FullExterior,             !- Solar Distribution
    25,                       !- Maximum Number of Warmup Days
    6;                        !- Minimum Number of Warmup Days
]
```

What if you object name had a dot `.` in it? Will the json_function get confused?

If the name has a dot in it, there are two ways of doing this.

```
# first way
json_str = {"idf.BUILDING.Taj.with.dot.Terrain": "Rural",}
json_functions.updateidf(idf1, json_str)
# second way (put the name in single quotes)
json_str = {"idf.BUILDING.'Another.Taj.with.dot'.Terrain": "Rural",}
json_functions.updateidf(idf1, json_str)
```

```
idf1.idfobjects['building'.upper()]
```

```
[
BUILDING,
    Empire State Building,    !- Name
    52,                       !- North Axis
    Rural,                    !- Terrain
    0.6,                      !- Loads Convergence Tolerance Value
    0.4,                      !- Temperature Convergence Tolerance Value
    FullExterior,             !- Solar Distribution
    25,                       !- Maximum Number of Warmup Days
    6;                        !- Minimum Number of Warmup Days
,
BUILDING,
    Taj,                      !- Name
    0.0,                      !- North Axis
    Rural,                    !- Terrain
    0.04,                     !- Loads Convergence Tolerance Value
    0.4,                      !- Temperature Convergence Tolerance Value
    FullExterior,             !- Solar Distribution
    25,                       !- Maximum Number of Warmup Days
    6;                        !- Minimum Number of Warmup Days
,
BUILDING,
    Taj.with.dot,             !- Name
    0.0,                      !- North Axis
    Rural,                    !- Terrain
    0.04,                     !- Loads Convergence Tolerance Value
    0.4,                      !- Temperature Convergence Tolerance Value
    FullExterior,             !- Solar Distribution
    25,                       !- Maximum Number of Warmup Days
    6;                        !- Minimum Number of Warmup Days
,
BUILDING,
    Another.Taj.with.dot,     !- Name
    0.0,                      !- North Axis
    Rural,                    !- Terrain
    0.04,                     !- Loads Convergence Tolerance Value
```

(continues on next page)

(continued from previous page)

```
    0.4,                      !- Temperature Convergence Tolerance Value
    FullExterior,             !- Solar Distribution
    25,                       !- Maximum Number of Warmup Days
    6;                        !- Minimum Number of Warmup Days
]
```

Note When you us the json update function:

- The json function expects the `Name` field to have a value.
- If you try to update an object with a blank `Name` field, the results may be unexpected (undefined ? :-). So don't do this.
- If the object has no `Name` field (some don't), changes are made to the first object in the list. Which should be fine, since usually there is only one item in the list
- In any case, if the object does not exist, it is created with the default values

5.10.1 Use Case for JSON update

If you have an eppy running on a remote server somewhere on the internet, you can change an idf file by sending it a JSON over the internet. This is very useful if you ever need it. If you don't need it, you shouldn't care :-)

5.11 Open a file quickly¶

It is rather cumbersome to open an IDF file in eppy. From the tutorial, the steps look like this:

```
from eppy import modeleditor
from eppy.modeleditor import IDF

iddfile = "../eppy/resources/iddfiles/Energy+V7_2_0.idd"
fname = "../eppy/resources/idffiles/V_7_2/smallfile.idf"
IDF.setiddname(iddfile)
idf = IDF(fname)
```

- You have to find the IDD file on your hard disk.
- Then set the IDD using setiddname(iddfile).
- Now you can open the IDF file

Why can't you just open the IDF file without jumping thru all those hoops. Why do you have to find the IDD file. What is the point of having a computer, if it does not do the grunt work for you.

The function easyopen will do the grunt work for you. It will automatically read the version number from the IDF file, locate the correct IDD file and set it in eppy and then open your file. It works like this:

```
from eppy.easyopen import easyopen

fname = './eppy/resources/idffiles/V8_8/smallfile.idf'
idf = easyopen(fname)
```

For this to work,

- the IDF file should have the VERSION object. You may not have this if you are just working on a file snippet.
- you need to have the version of EnergyPlus installed that matches your IDF version.

- Energyplus should be installed in the default location.

If easyopen does not work, use the long winded steps shown in the tutorial. That is guaranteed to work

Note: easyopen() will also take a epw argument looking like:

```
idf = easyopen(fname, epw=path2weatherfile)
```

If you pass the epw argument:

```
idf.run()
```

will work. To know more about idf.run() see here.

5.12 Other miscellaneous functions¶

5.12.1 Fan power in Watts, BHP and fan cfm¶

We normally think of fan power in terms of Brake Horsepower (BHP), Watts. Also when working with IP units it is useful to think of fan flow volume in terms of cubic feet per minute (cfm).

Energyplus does not have fields for those values. With eppy we have functions that will calculate the values

- fan power in BHP
- fan power in Watts
- fan flow in CFM

It will work for the following objects:

- FAN:CONSTANTVOLUME
- FAN:VARIABLEVOLUME
- FAN:ONOFF
- FAN:ZONEEXHAUST
- FANPERFORMANCE:NIGHTVENTILATION

The sample code would look like this:

```
thefans = idf.idfobjects['Fan:VariableVolume'.upper()]
thefan = thefans[0]
bhp = thefan.fanpower_bhp
watts = thefan.fanpower_watts
cfm = thefan.fan_maxcfm
```

Note: This code was hacked together quickly. Needs peer review in ../eppy/fanpower.py

CHAPTER 6

Running EnergyPlus from Eppy

It would be great if we could run EnergyPlus directly from our IDF wouldn't it?

Well here's how we can.

```
# you would normaly install eppy by doing
# python setup.py install
# or
# pip install eppy
# or
# easy_install eppy

# if you have not done so, uncomment the following three lines
import sys
# pathnameto_eppy = 'c:/eppy'
pathnameto_eppy = '../'
sys.path.append(pathnameto_eppy)
```

```
from eppy.modeleditor import IDF

iddfile = "/Applications/EnergyPlus-8-3-0/Energy+.idd"
IDF.setiddname(iddfile)
```

```
idfname = "/Applications/EnergyPlus-8-3-0/ExampleFiles/BasicsFiles/Exercise1A.idf"
epwfile = "/Applications/EnergyPlus-8-3-0/WeatherData/USA_IL_Chicago-OHare.Intl.AP.
↪725300_TMY3.epw"

idf = IDF(idfname, epwfile)
idf.run()
```

if you are in a terminal, you will see something like this:

```
Processing Data Dictionary
Processing Input File
Initializing Simulation
```

(continues on next page)

(continued from previous page)

```
Reporting Surfaces
Beginning Primary Simulation
Initializing New Environment Parameters
Warming up {1}
Warming up {2}
Warming up {3}
Warming up {4}
Warming up {5}
Warming up {6}
Starting Simulation at 07/21 for CHICAGO_IL_USA COOLING .4% CONDITIONS DB=>MWB
Initializing New Environment Parameters
Warming up {1}
Warming up {2}
Warming up {3}
Warming up {4}
Warming up {5}
Warming up {6}
Starting Simulation at 01/21 for CHICAGO_IL_USA HEATING 99.6% CONDITIONS
Writing final SQL reports
EnergyPlus Run Time=00hr 00min  0.24sec
```

It's as simple as that to run using the EnergyPlus defaults, but all the EnergyPlus command line interface options are also supported.

To get a description of the options available, as well as the defaults you can call the Python built-in help function on the IDF.run method and it will print a full description of the options to the console.

```
help(idf.run)
```

```
Help on method run in module eppy.modeleditor:

run(self, **kwargs) method of eppy.modeleditor.IDF instance
    This method wraps the following method:

    run(idf=None, weather=None, output_directory=u'', annual=False, design_day=False, 
↪idd=None, epmacro=False, expandobjects=False, readvars=False, output_prefix=None, 
↪output_suffix=None, version=False, verbose=u'v', ep_version=None)
        Wrapper around the EnergyPlus command line interface.

        Parameters
        ----------
        idf : str
            Full or relative path to the IDF file to be run, or an IDF object.

        weather : str
            Full or relative path to the weather file.

        output_directory : str, optional
            Full or relative path to an output directory (default: 'run_outputs)

        annual : bool, optional
            If True then force annual simulation (default: False)

        design_day : bool, optional
            Force design-day-only simulation (default: False)
```

(continues on next page)

(continued from previous page)

```
        idd : str, optional
            Input data dictionary (default: Energy+.idd in EnergyPlus directory)

        epmacro : str, optional
            Run EPMacro prior to simulation (default: False).

        expandobjects : bool, optional
            Run ExpandObjects prior to simulation (default: False)

        readvars : bool, optional
            Run ReadVarsESO after simulation (default: False)

        output_prefix : str, optional
            Prefix for output file names (default: eplus)

        output_suffix : str, optional
            Suffix style for output file names (default: L)
                L: Legacy (e.g., eplustbl.csv)
                C: Capital (e.g., eplusTable.csv)
                D: Dash (e.g., eplus-table.csv)

        version : bool, optional
            Display version information (default: False)

        verbose: str
            Set verbosity of runtime messages (default: v)
                v: verbose
                q: quiet

        ep_version: str
            EnergyPlus version, used to find install directory. Required if run() is
            called with an IDF file path rather than an IDF object.

        Returns
        -------
        str : status

        Raises
        ------
        CalledProcessError

        AttributeError
            If no ep_version parameter is passed when calling with an IDF file path
            rather than an IDF object.
```

Note 1: idf.run() works for E+ version >= 8.3 *Note 2:* idf.run(readvars=True) has been tested only for E+ version >= 8.9. It may work with earlier versions

6.1 Running in parallel processes

One of the great things about Eppy is that it allows you to set up a lot of jobs really easily. However, it can be slow running a lot of EnergyPlus simulations, so it's pretty important that we can make the most of the processing power you have available by running on multiple CPUs.

Again this is as simple as you'd hope it would be.

You first need to create your jobs as a list of lists in the form:

```
[[<IDF object>, <dict of command line parameters>], ...]
```

The example here just creates 4 identical jobs apart from the output_directory the results are saved in, but you would obviously want to make each job different.

Then run the jobs on the required number of CPUs using runIDFs…

… and your results will all be in the output_directory you specified.

CHAPTER 7

Useful Scripts

7.1 Location of the scripts

Here are some scripts that you may find useful. They are in the folder "./eppy/useful_scripts"

And now for some housekeeping before we start off

```
import os
os.chdir("../eppy/useful_scripts")
# changes directory, so we are where the scripts are located
```

```
# you would normaly install eppy by doing
# python setup.py install
# or
# pip install eppy
# or
# easy_install eppy

# if you have not done so, the following three lines are needed
import sys
# pathnameto_eppy = 'c:/eppy'
pathnameto_eppy = '../../'
sys.path.append(pathnameto_eppy)
```

If you look in the folder "./eppy/useful_scripts", you fill find the following scripts

The scripts are:

```
- eppy_version.py
- idfdiff.py
- loopdiagram.py
- eppyreadtest_folder.py
- eppyreadtest_file.py
```

7.2 eppy_version.py

Many scripts will print out some help information, if you use the –help option. Let us try that

```
%%bash
# ignore the line above. It simply lets me run a command line from ipython notebook
python eppy_version.py --help
```

```
usage: eppy_version.py [-h]

I print the current version of eppy. Being polite, I also say hello !

optional arguments:
  -h, --help  show this help message and exit
```

That was useful !

Now let us try running the program

```
%%bash
# ignore the line above. It simply lets me run a command line from ipython notebook
python eppy_version.py
```

```
Hello! I am  eppy version 0.4.6.4a
```

7.3 Redirecting output to a file

Most scripts will print the output to a terminal. Sometimes we want to send the output to a file, so that we can save it for posterity. We can do that py using ">" with the filename after that. For eppy_version.py, it will look like this:

python eppy_version.py > save_output.txt

Some of the following scripts will generate csv or html outputs. We can direct the output to a file with .html extension and open it in a browser

7.4 Compare two idf files - idfdiff.py

This script will compare two idf files. The results will be displayed printed in "csv" format or in "html" format.

You would run the script from the command line. This would be the terminal on Mac or unix, and the dos prompt on windows. Let us look at the help for this script, by typing:

```
%%bash
# ignore the line above. It simply lets me run a command line from ipython notebook
python idfdiff.py -h
```

```
usage: idfdiff.py [-h] (--csv | --html) idd file1 file2

Do a diff between two idf files. Prints the diff in csv or html file format.
You can redirect the output to a file and open the file using as a spreadsheet
or by using a browser

```

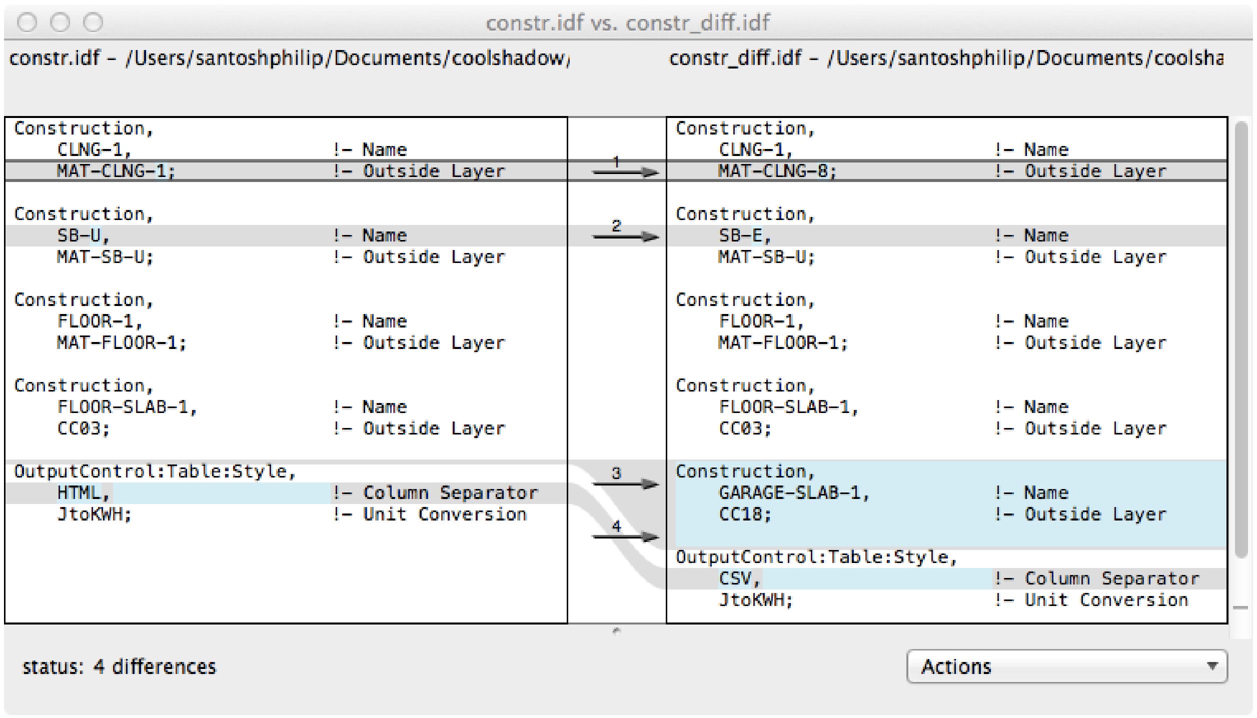
(continues on next page)

(continued from previous page)

```
positional arguments:
  idd          location of idd file = ./somewhere/eplusv8-0-1.idd
  file1        location of first with idf files = ./somewhere/f1.idf
  file2        location of second with idf files = ./somewhere/f2.idf

optional arguments:
  -h, --help  show this help message and exit
  --csv
  --html
```

Now let us try this with two “idf” files that are slightly different. If we open them in a file comparing software, it would look like this:

```
from eppy.useful_scripts import doc_images #no need to know this code, it just shows␣
↪the image below
for_images = doc_images
for_images.display_png(for_images.filemerge) # display the image below
```



```
constr.idf vs. constr_diff.idf
constr.idf - /Users/santoshphilip/Documents/coolshadow/     constr_diff.idf - /Users/santoshphilip/Documents/coolsha

Construction,                                         Construction,
    CLNG-1,                  !- Name                      CLNG-1,                  !- Name
    MAT-CLNG-1;              !- Outside Layer     1       MAT-CLNG-8;              !- Outside Layer

Construction,                                         Construction,
    SB-U,                    !- Name              2       SB-E,                    !- Name
    MAT-SB-U;                !- Outside Layer             MAT-SB-U;                !- Outside Layer

Construction,                                         Construction,
    FLOOR-1,                 !- Name                      FLOOR-1,                 !- Name
    MAT-FLOOR-1;             !- Outside Layer             MAT-FLOOR-1;             !- Outside Layer

Construction,                                         Construction,
    FLOOR-SLAB-1,            !- Name                      FLOOR-SLAB-1,            !- Name
    CC03;                    !- Outside Layer             CC03;                    !- Outside Layer

OutputControl:Table:Style,                        3   Construction,
    HTML,                    !- Column Separator          GARAGE-SLAB-1,           !- Name
    JtoKWH;                  !- Unit Conversion   4       CC18;                    !- Outside Layer

                                                      OutputControl:Table:Style,
                                                          CSV,                     !- Column Separator
                                                          JtoKWH;                  !- Unit Conversion

status: 4 differences                                                    Actions
```

There are 4 differences between the files. Let us see what idfdiff.py does with the two files. We will use the –html option to print out the diff in html format.

```
%%bash
# python idfdiff.py idd file1 file2
python idfdiff.py --html ../resources/iddfiles/Energy+V7_2_0.idd ../resources/
↪idffiles/V_7_2/constructions.idf ../resources/idffiles/V_7_2/constructions_diff.idf
```

```
<html><p>file1 = ../resources/idffiles/V_7_2/constructions.idf</p><p>file2 = ../
↪resources/idffiles/V_7_2/constructions_diff.idf</p><table border="1"><tr><th>Object␣
↪Key</th><th> Object Name</th><th> Field Name</th><th> file1</th><th> file2</th></tr>
↪<tr><td>MATERIAL</td><td>F08 Metal surface</td><td></td><td>is here</td><td>not here
↪</td></tr><tr><td>MATERIAL</td><td>F08 Metal surface haha</td><td></td><td>not here
↪</td><td>is here</td></tr><tr><td>MATERIAL</td><td>G05 25mm wood</td><td>
↪Conductivity</td><td>0.15</td><td>0.155</td></tr><tr><td>CONSTRUCTION</td><td>
↪Exterior Door</td><td>Outside Layer</td><td>F08 Metal surface</td><td>F08 Metal␣
↪surface haha</td></tr></table></html>
```

(continues on next page)

(continued from previous page)

reprinting the output again for clarity:

<html><p>file1 = ../resources/idffiles/V_7_2/constructions.idf</p><p>file2 = ../resources/idffiles/V_7_2/constructions_diff.idf</p><table border="1"><tr><th>Object Key</th><th> Object Name</th><th> Field Name</th><th> file1</th><th> file2</th></tr><tr><td>MATERIAL</td><td>F08 Metal surface</td><td></td><td>not here</td><td>is here</td></tr><tr><td>MATERIAL</td><td>F08 Metal surface haha</td><td></td><td>is here</td><td>not here</td></tr><tr><td>MATERIAL</td><td>G05 25mm wood</td><td>Conductivity</td><td>0.15</td><td>0.155</td></tr><tr><td>CONSTRUCTION</td><td>Exterior Door</td><td>Outside Layer</td><td>F08 Metal surface</td><td>F08 Metal surface haha</td></tr></table></html>

It does look like html :-). We need to redirect this output to a file and then open the file in a browser to see what it looks like. Displayed below is the html file

```
from eppy.useful_scripts import doc_images #no need to know this code, it just shows␣
↪the image below
from IPython.display import HTML
h = HTML(open(doc_images.idfdiff_path, 'r').read())
h
```

Pretty straight forward. Scroll up and look at the origin text files, and see how idfdiff.py understands the difference

Now let us try the same thin in csv format

```
%%bash
# python idfdiff.py idd file1 file2
python idfdiff.py --csv ../resources/iddfiles/Energy+V7_2_0.idd ../resources/idffiles/
↪V_7_2/constr.idf ../resources/idffiles/V_7_2/constr_diff.idf
```

```
file1 = ../resources/idffiles/V_7_2/constr.idf
file2 = ../resources/idffiles/V_7_2/constr_diff.idf

Object Key, Object Name, Field Name, file1, file2
CONSTRUCTION,CLNG-1,Outside Layer,MAT-CLNG-1,MAT-CLNG-8
CONSTRUCTION,GARAGE-SLAB-1,,is here,not here
CONSTRUCTION,SB-E,,is here,not here
CONSTRUCTION,SB-U,,not here,is here
OUTPUTCONTROL:TABLE:STYLE, ,Column Separator,HTML,CSV
```

We see the same output, but now in csv format. You can redirect it to a ".csv" file and open it up as a spreadsheet

7.5 loopdiagram.py

7.5.1 Diagrams of HVAC loops

This script will draw all the loops in an idf file. It is a bit of a hack. So it will work on most files, but sometimes it will not :-(. But it is pretty useful when it works.

If it does not work, send us the idf file and we'll try to fix the code

Make sure grapphviz is installed for this script to work

Again, we'll have to run the script from the terminal. Let us look at the help for this script

```
%%bash
# ignore the line above. It simply lets me run a command line from ipython notebook
python loopdiagram.py --help
```

```
usage: loopdiagram.py [-h] idd file

draw all the  loops in the idf file
There are two output files saved in the same location as the idf file:
- idf_file_location/idf_filename.dot
- idf_file_location/idf_filename.png

positional arguments:
  idd         location of idd file = ./somewhere/eplusv8-0-1.idd
  file        location of idf file = ./somewhere/f1.idf

optional arguments:
  -h, --help  show this help message and exit
```

Pretty straightforward. Simply open png file and you will see the loop diagram. (ignore the dot file for now. it will be documented later)

So let us try this out with and simple example file. We have a very simple plant loop in "../resources/idffiles/V_7_2/plantloop.idf"

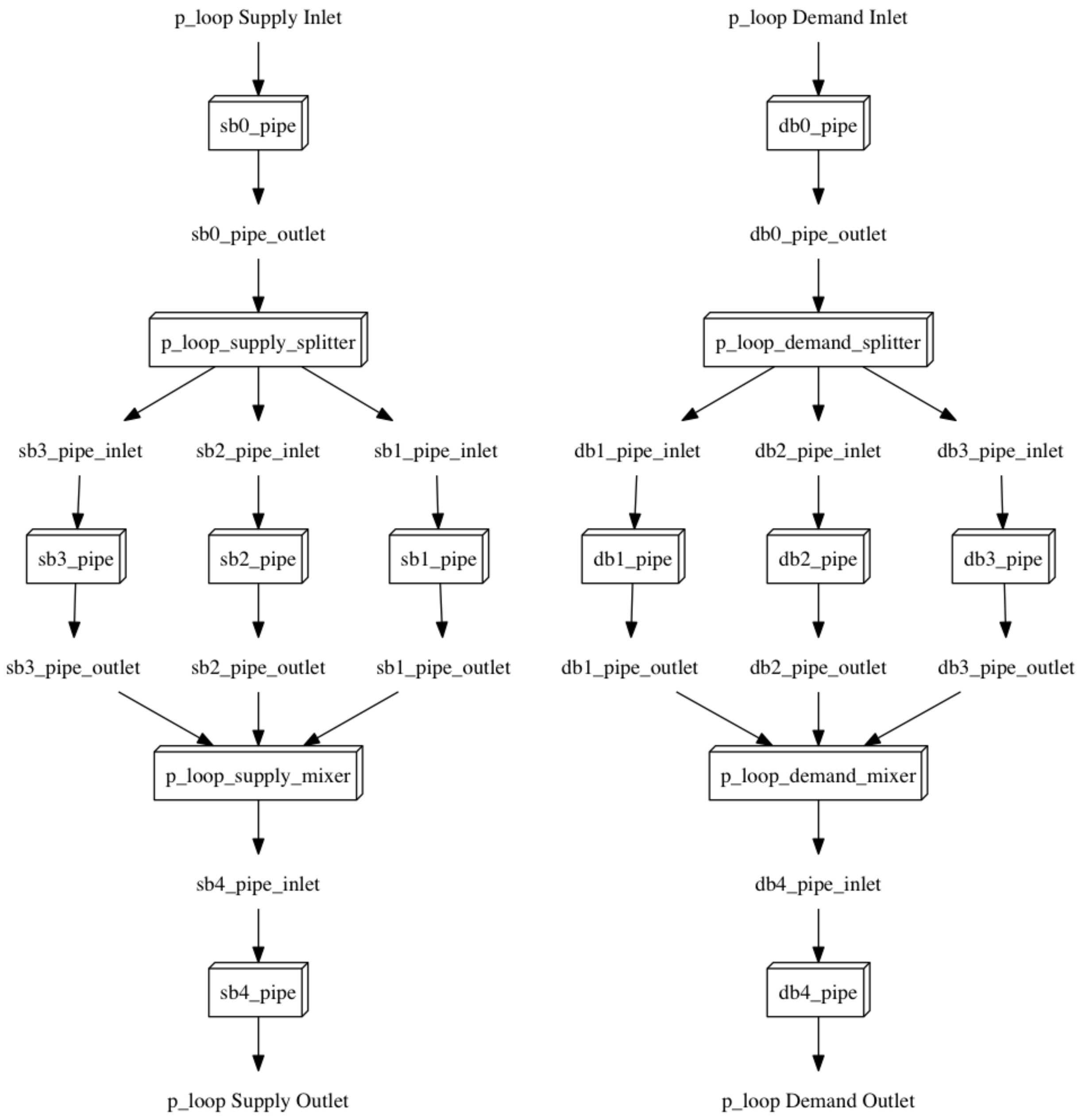
```
%%bash
# ignore the line above. It simply lets me run a command line from ipython notebook
python loopdiagram.py ../resources/iddfiles/Energy+V7_2_0.idd ../resources/idffiles/V_
↪7_2/plantloop.idf
```

```
constructing the loops
cleaning edges
making the diagram
saved file: ../resources/idffiles/V_7_2/plantloop.dot
saved file: ../resources/idffiles/V_7_2/plantloop.png
```

The script prints out it's progress. On larger files, this might take a few seconds. If we open this file, it will look like the diagram below

Note: the supply and demnd sides are not connected in the diagram, but shown seperately for clarity

```
from eppy.useful_scripts import doc_images #no need to know this code, it just shows␣
↪the image below
for_images = doc_images
for_images.display_png(for_images.plantloop) # display the image below
```



That diagram is not a real system. Does this script really work ?

Try it yourself. Draw the daigram for "../resources/idffiles/V_7_2/5ZoneCAVtoVAVWarmestTempFlow.idf"

7.5.2 Names in loopdiagrams

- Designbuilder is an energyplus editor autogenerates object names like "MyHouse:SAPZone1"
- Note the ":" in the name.
- Unfortunatley ":" is a reserved character when making a loop diagrams. (eppy uses pydot and grapphviz which has this constraint)
- to work around this, loopdiagram will replace all ":" with a "__"

- So the names in the diagram will not match the names in your file, but you can make out what is going on

7.6 eppyreadtest_folder.py

Not yet documented

7.7 eppyreadtest_file.py

Not yet documented

CHAPTER 8

Eppy Functions

This Document is a work in progress

The most commonly used eppy functions are gathered here.

IDF functions:

- idf = IDF(fname) # fname or fhandle
- idf.printidf
- idf1.idfobjects[‘BUILDING’]
- idf.save
- idf.saveas
- idf.newidfobject
- idf.copyidfobject
- idf.newidfobject
- idf.removeidfobject
- idf.popidfobject
- idf.copyidfobject

idfobjects function:

- building.Name
- surface azimuth
- surface tilt
- surface area
- building.getrange(“Loads_Convergence_Tolerance_Value”)
- building.checkrange(“Loads_Convergence_Tolerance_Value”)
- building.fieldnames

Other Functions:

- area = modeleditor.zonearea(idf, zone.Name)
- volume = modeleditor.zonevolume(idf, zone.Name)
- json_functions.updateidf(idf1, json_str)
- idf_helpers.getidfobjectlist
- idf_helpers.copyidfintoidf

CHAPTER 9

Eppy Recipies

This Document is a work in progress

Common ways in which eppy can be used

CHAPTER 10

Changes

10.1 release r0.5.48

10.1.1 2018-10-03

- using cookiecuter <https://github.com/audreyr/cookiecutter-pypackage> in eppy
- **fixed bug in idf.run()**
  - the bug resulted in the working directory changing if the run was done with an invalid idf

10.2 release r0.5.47

10.2.1 2018-09-25

- fixed bug in useful_scripts/idfdiff.py

10.2.2 2018-04-23

- idf.newidfobject() has a parameter defaultvlaues=True or False. This can be toggled to set or not set the default values in the IDF file

10.2.3 2018-03-24

- fixed a bug, where some idfobject fields stayed as strings even though they were supposed to be numbers

10.2.4 2018-03-21

- new function easyopen(idffile) will automatically set the IDD file and open the IDF file. This has been documented in ./docs/source/newfunctions.rst

10.2.5 2017-12-11

- Added documentation in the installation section on how to run eppy in grasshopper
- added functions to get fan power in watts, bhp and fan flow in cfm for any fan object. This has been documented in ./docs/source/newfunctions.rst

10.3 release r0.5.46

10.3.1 2017-12-10

- documentation is now at http://eppy.readthedocs.io/en/latest/

10.4 release r0.5.45

10.4.1 2017-10-01

- fixed a bug in the setup.py (It was not installing some required folders)
- updated documentation to include how to run Energyplus from eppy
- **format of the table file was changed in E+ 8.7.**
    - readhtml is updated to be able to read the new format (it still reads the older versions)

10.5 release r0.5.44

10.5.1 2017-05-23

- **IDF.run() works with E+ version >= 8.3**
    - This will run the idf file
    - documentation updated to reflect this
- **Some changes made to support eppy working on grasshopper**
    - more work needs to be done on this

10.6 release r0.5.43

10.6.1 2017-02-09

fixed the bug in the setup file

10.7 release r0.5.42

10.8 2016-12-31

bugfix for idfobjects with no fieldnames. Such fields are named A!, A2, A3/ N1, N2, N3 taken from the IDD file

There is a bug in the setup.py in this version

10.8.1 2016-11-02

It is now possible to run E+ from eppy

10.9 release r0.5.41

10.9.1 2016-09-14

bugfix in loopdiagram.py. Some cleanup by removing extra copies of loopdiagram.py

10.10 release r0.5.40

10.10.1 2016-09-06

This is a release for python2 and python3. pip install will automatically install the correct version.

10.11 release r0.5.31

10.11.1 2016-09-04

bugfix so that json_functions can have idf objects with names that have dots in them

10.12 release r0.5.3

10.12.1 2016-07-21

tab completion of fileds (of idfobjects) works in ipython and ipython notebook

10.12.2 2016-07-09

added:

- construction.rfactor and material.rfactor
- construction.uvalue and material.uvalue
- construction.heatcapacity and material.heatcapacity

- the above functions do not work in all cases yet. But are still usefull

added:

- zone.zonesurfaces -> return all surfaces of the zone
- surface.subsurfaces -> will return all the subsurfaces (windows, doors etc.) that belong to the surface

added two functions that scan through the entire idf file:

- EpBunch.getreferingobjs(args)
- EpBunch.get_referenced_object(args)
- they make it possible for an idf object to scan through it's idf file and find other idf objects that are related to it (thru object-list and reference)

10.12.3 2016-05-31

refactored code for class IDF and class EpBunch fixed a bug in modeleditor.newidfobject

10.13 release r0.5.2

10.13.1 2016-05-27

added ability to update idf files thru JSON messages.

10.13.2 2016-04-02

Replaced library bunch with munch

10.14 release r0.5.1

10.14.1 2016-02-07

- bug fix -> read files that have mixed line endings. Both DOS and Unix line endings

10.15 release r0.5

10.15.1 2015-07-12

- python3 version of eppy is in ./p3/eppy
- eppy license has transitioned from GPLv3 to MIT license
- made some bugfixes to hvacbuilder.py

10.15.2 2015-05-30

- bugfix in ./eppy/Air:useful_scripts/idfdiff.py
- **added in ./eppy/Air:useful_scripts/idfdiff_missing.py**
    - this displays only the missing objects in either file

10.15.3 2015-05-27

- **idf.saveas(newname) changes the idf.idfname to newname**
    - so the next idf.save() will save to newname
- to retain the original idf.idfname use idf.savecopy(copyname)

10.15.4 2015-05-26

updated the following: - idf.save(lineendings='default') - idf.saveas(fname, lineendings='default')

- **optional argument lineendings**
    - if lineendings='default', uses the line endings of the platform
    - if lineendings='windows', forces windows line endings
    - if lineendings='unix', forces unix line endings

10.16 release r0.464a

10.16.1 2015-01-13

r0.464a released on 2015-01-13. This in alpha release of this version. There may be minor updates after review from users.

10.16.2 2015-01-06

- Developer documentation has been completed
- Added a stubs folder with scripts that can be used as templates

10.16.3 2014-10-21

- fixed a bug in script eppy/useful_scripts/loopdiagram.py

10.16.4 2014-09-01

- added a script eppy/useful_scripts/loopdiagram.py:

```
python loopdiagram.py --help

usage: loopdiagram.py [-h] idd file

draw all the  loops in the idf file
There are two output files saved in the same location as the idf file:
- idf_file_location/idf_filename.dot
- idf_file_location/idf_filename.png

positional arguments:
  idd         location of idd file = ./somewhere/eplusv8-0-1.idd
  file        location of idf file = ./somewhere/f1.idf

optional arguments:
  -h, --help  show this help message and exit
```

- fixed a bug in hvacbuilder.makeplantloop and hvacbuilder.makecondenserloop

10.17 release r0.463

10.17.1 2014-08-21

- added eppy/useful_scripts/eppy_version.py
- updated documentation to match

10.18 release r0.462

10.18.1 2014-08-19

- **added a script that can compare two idf files. It is documented in "Useful Scripts". The script is in**
    - eppy/usefull_scripts/idfdiff.py
- **added two scripts that test if eppy works when new versions of energyplus are released. Documentation for this is not yet**
    - eppy/usefull_scripts/eppyreadtest_file.py
    - eppy/usefull_scripts/eppyreadtest_folder.py
- fixed a bug where eppy would not read backslashes in a path name. Some idf objects have fields that are path names. On dos/windows machines these path names have backslashes

CHAPTER 11

Developer Documentation for eppy

The documentation you have read so far is written for *Users of eppy*. Users of eppy will simply install eppy and use it. They may be writing a lot of code using eppy, but they will not be changing any code within eppy itself.

This section of the documentation is for people who want to contribute to the development of eppy. In contrast to *eppy users* people who contribute code to eppy will be changing and adding code to eppy. To change code within eppy, developers will need to have a good sense of how eppy is written and structured. This section hopes to give clarity to such issues. It is assumed that most developers of eppy will not be professional programmers and may come from a rage of programming abilities. So this documentation is slanted to such an audience.

If you look at the table of contents below, You will notice sections on the *history* and *philosophy* of eppy. Eppy was built off an earlier program, and a lot of the data structure and program design, will become clear when you see the history behind eppy. The philosophy of eppy alludes to the Unix philosophy of *Do one thing, and do it well*.

11.1 Topics

11.1.1 Philosophy of Eppy

Author Santosh Philip

The *Philosophy of eppy* alludes to the Unix Philosophy. The philosophy of eppy takes inspiration from the *Unix Philosophy*, that speaks of doing one thing and doing it well. Unix has the tradition of having small tools that do a single thing. A number of tools can be stitched together to do a more complex task. Of course, since eppy is written in Python, it adheres to the "Zen of Python"

The rules of Unix are shown in the appendix at the bottom of this page.(You may find that eppy breaks the rules at times)

Purpose of Eppy

The purpose of eppy is to be a *scripting language* to Energyplus. So what does that mean ? Let us start by describing what it is not.

What eppy is **not**:

- Eppy is **not** a User Interface to Energyplus.
- Eppy is **not** a HVAC builder for Energyplus
- Eppy is **not** a Geometry builder for Energyplus
- Eppy does **not** make batch runs for Energyplus
- Eppy does **not** optimize Energyplus models

Now you may ask, "What use is eppy, if it cannot do any of these things ?"

Aha ! Eppy is a scripting language. That means you can write scripts that can help to do any of those things listed above. And many more that are not listed above :-)

A scripting language has to be *simple* and *expressive*. It has to be simple so that it is easy to understand and use, and should ideally have a negligible learning curve. Now, it can be simple, but may be simplistic and you may be able to do only simple things in it. To go beyond the simple, the language has to be expressive. Most natural languages (like English or Malayalam) are expressive and you can write poetry in them. Of course, some languages are more expressive than others, which is why Malayalam is more poetic than English … just kidding here :-)

In the world of programming languages, python is a very expressive language. If you doubt this, you should take a look at the *Zen of Python* to see the philosophy behind python. Eppy attempts to piggyback on the expressiveness of python. Developers who are working on eppy should use this as a guiding principle, especially if you are working deep within the innards of eppy. Some of this is tricky coding, because you are overriding the functionality within python itself. Depending on your coding ability and what you want to learn, you should pick an area you want to work in

Question: You claim that Eppy is **not** a HVAC builder for Energyplus. I see a module called hvacbulider.py in eppy.

Answer: Yup ! you are right. I cheated. Hvacbulider should not be in eppy. We are at an early stage of the evolution of eppy and it made sense to keep it within eppy for now. In the future, it will be made into a standalone library that will use eppy for it's functionality.

The power of the idd file

Due to historic reasons, the developers of Energyplus were tasked to write the simulation engine without an user interface. This meant they had to write Energyplus in such a way that others could write user interfaces to it. This constraint forced them to make two excellent design decisions:

- All models in Energyplus are text files called *idf* files. Text files are human readable and are platform agnostic.
- There is a special text file called an *idd* file, that describes all the objects that can be in an Energyplus model. Again this file is human readable and it is very well documented with comments. If you intend to do any development in Energyplus or eppy, it is well worth taking a look at this file. It is usually called *Energy+.idd* and is found in the main Energyplus folder
- When newer versions of Energyplus are released, an updated idd file is released with it. The simulation engine within Energyplus uses the idd file to parse the objects in the model. If an interface software is able to read and understand the idd file, it can effectively understand any idf file.
- Eppy is written so that it can read and understand idd files, including idd files of future releases.In principle, eppy should be able to work with any future version of Energyplus without any changes. At the date this documentation was written, eppy can read all idf files, from version 1 to version 8.

Idd file and eppy

Both the idd file and idf file are text files. If you look at these files, you will see that there is a clear structure that is visible to you, the reader. Eppy reads these text files and pushes it into classic python data structures, like lists and dictionaries. Once this information is in lists and dicts, it becomes much easier to walk through the data structure and to add, delete and change the information.

At it's heart, eppy is simply a scripting language that puts the idd and idf file into a python data structure. Then it uses the power of python to edit the idf file.

Appendix: Unix Philosophy

From http://www.catb.org/esr/writings/taoup/html/ch01s06.html:

- Rule of Modularity: Write simple parts connected by clean interfaces.
- Rule of Clarity: Clarity is better than cleverness.
- Rule of Composition: Design programs to be connected to other programs.
- Rule of Separation: Separate policy from mechanism; separate interfaces from engines.
- Rule of Simplicity: Design for simplicity; add complexity only where you must.
- Rule of Parsimony: Write a big program only when it is clear by demonstration that nothing else will do.
- Rule of Transparency: Design for visibility to make inspection and debugging easier.
- Rule of Robustness: Robustness is the child of transparency and simplicity.
- Rule of Representation: Fold knowledge into data so program logic can be stupid and robust.
- Rule of Least Surprise: In interface design, always do the least surprising thing.
- Rule of Silence: When a program has nothing surprising to say, it should say nothing.
- Rule of Repair: When you must fail, fail noisily and as soon as possible.
- Rule of Economy: Programmer time is expensive; conserve it in preference to machine time.
- Rule of Generation: Avoid hand-hacking; write programs to write programs when you can.
- Rule of Optimization: Prototype before polishing. Get it working before you optimize it.
- Rule of Diversity: Distrust all claims for "one true way".
- Rule of Extensibility: Design for the future, because it will be here sooner than you think.

11.1.2 History of Eppy

Author Santosh Philip (santosh_philip at yahoo.com)

EPlusInterface

EPlusInterface is a text based interface to Energyplus. EPlusInterface is the direct ancestor to Eppy. The data structure of EPlusInterface was simple and quite robust. EPlusInterface also had good functions to read the idd file and the idf files. In principle the idd file reader of EPlusInterface was written so that it could read any version of the idd file. Eppy directly uses the file readers from EPlusInterface

11.1.3 The Future of eppy

The future of eppy lies in the use of eppy to build a more comprehensive tool kit to work with Energyplus models. In a sense eppy should not grow much at all. Eppy should become more effective as tool kit at a very granular level.

Longterm TODOs for eppy

Some of the possibilities for the future growth of eppy are:

- A geometry builder for Energyplus. The open source 3-D software Blender has a python scripting language. One possibility would be to use Blender as a geometry interface for eppy and Energyplus
- A HVAC Builder for Energyplus.
- A HVAC diagram tool for Energyplus. Eppy has a reasonbly good diagram tool for HVAC network. Right now it shows a purely static image of the network. An enhancement of this would be to ability to edit the network through the diagram
- A User interface for Energyplus. This may be redundant, since the present IDF editor is quite good.
- A web based user interface Energyplus. The path of least resistance to do this would be to a python web framework like django, flask, pyramid or bottle. Bottle is a very light weight web framework and could be used to make small special purpose interfaces for Energyplus.

The eppy project will take a lead on some of these projects. Working on these projects will feed back information into eppy, on how to improve the functionality of eppy

11.1.4 Underlying Data Structure of eppy

As described in previous sections, eppy was built on EplusInterface

Let us open a small **idf** file to explore the data structure

```
# you would normaly install eppy by doing
# python setup.py install
# or
# pip install eppy
# or
# easy_install eppy

# if you have not done so, uncomment the following three lines
import sys
# pathnameto_eppy = 'c:/eppy'
pathnameto_eppy = '../../../'
sys.path.append(pathnameto_eppy)
```

```
from eppy import modeleditor
from eppy.modeleditor import IDF
iddfile = "../../../eppy/resources/iddfiles/Energy+V7_2_0.idd"
fname1 = "../../../eppy/resources/idffiles/V_7_2/dev1.idf"

IDF.setiddname(iddfile)
idf1 = IDF(fname1)
idf1.printidf()
```

```
VERSION,
    7.3;                      !- Version Identifier

SIMULATIONCONTROL,
    Yes,                      !- Do Zone Sizing Calculation
    Yes,                      !- Do System Sizing Calculation
    Yes,                      !- Do Plant Sizing Calculation
    No,                       !- Run Simulation for Sizing Periods
    Yes;                      !- Run Simulation for Weather File Run Periods

BUILDING,
    Empire State Building,    !- Name
    30.0,                     !- North Axis
    City,                     !- Terrain
    0.04,                     !- Loads Convergence Tolerance Value
    0.4,                      !- Temperature Convergence Tolerance Value
    FullExterior,             !- Solar Distribution
    25,                       !- Maximum Number of Warmup Days
    6;                        !- Minimum Number of Warmup Days

SITE:LOCATION,
    CHICAGO_IL_USA TMY2-94846,    !- Name
    41.78,                    !- Latitude
    -87.75,                   !- Longitude
    -6.0,                     !- Time Zone
    190.0;                    !- Elevation

MATERIAL:AIRGAP,
    F04 Wall air space resistance,    !- Name
    0.15;                     !- Thermal Resistance

MATERIAL:AIRGAP,
    F05 Ceiling air space resistance,    !- Name
    0.18;                     !- Thermal Resistance
```

Original Data Structure in EPlusInterface

The original data structure in EPlusInterface was stupidly simple and robust. In fact attributes **stupidly simple** and **robust** seem to go together. Eppy evolved in such a way that this data structure is still retained. The rest of eppy is simply syntactic sugar for this data structure.

from: https://www.princeton.edu/~achaney/tmve/wiki100k/docs/Syntactic_sugar.html *"Syntactic sugar is a computer science term that refers to syntax within a programming language that is designed to make things easier to read or to express, while alternative ways of expressing them exist. Syntactic sugar"*

Let us take a look at this data structure. If we open an idf file with eppy we can explore the original data structure that comes from EPlusInterface.

Note The variable names are not very intuitive at this level. I did not know what I was doing when I wrote this code and now we are stuck with it

There are three varaibles that hold all the data we need. They are:

- `idf1.model.dtls`
- `idf1.model.dt`
- `idf1.idd_info`

```
dtls = idf1.model.dtls # names of all the idf objects
dt = idf1.model.dt # the idf model
idd_info = idf1.idd_info # all the idd data
```

idf1.model.dtls - Overview

```
dtls = idf1.model.dtls # names of all the idf objects
print type(dtls)
```

```
<type 'list'>
```

```
# dtls is a list
print dtls[:10] # print the first ten items
```

```
['LEAD INPUT', 'SIMULATION DATA', 'VERSION', 'SIMULATIONCONTROL', 'BUILDING',
↪'SHADOWCALCULATION', 'SURFACECONVECTIONALGORITHM:INSIDE',
↪'SURFACECONVECTIONALGORITHM:OUTSIDE', 'HEATBALANCEALGORITHM',
↪'HEATBALANCESETTINGS:CONDUCTIONFINITEDIFFERENCE']
```

```
print len(dtls) # print the numer of items in the list
```

```
683
```

Couple of points to note about `dtls`:

- **dtls** is a list of all the names of the Energyplus objects.
- This list is extracted from the the **idd** file
- the list is in the same order as the objects in the **idd** file

idf1.model.dt - Overview

```
dt = idf1.model.dt # the idf model
print type(dt)
```

```
<type 'dict'>
```

```
# print 10 of the keys
print dt.keys()[:10]
```

```
['ZONEHVAC:OUTDOORAIRUNIT', 'TABLE:TWOINDEPENDENTVARIABLES',
↪'ENERGYMANAGEMENTSYSTEM:INTERNALVARIABLE', 'AVAILABILITYMANAGER:NIGHTCYCLE',
↪'GROUNDHEATTRANSFER:SLAB:BLDGPROPS', 'GENERATOR:MICROTURBINE',
↪'SHADING:BUILDING:DETAILED', 'EVAPORATIVECOOLER:INDIRECT:RESEARCHSPECIAL',
↪'ZONEHVAC:PACKAGEDTERMINALAIRCONDITIONER', 'CONSTRUCTION:WINDOWDATAFILE']
```

```
# dt is a dict
number_of_keys = len(dt.keys())
print number_of_keys
```

```
683
```

- The keys of **dt** are names of the objects (note that they are in capitals)
- Items in a python dict are unordered. So the keys may be in any order
- **dtls** will give us these names in the same order as they are in the idd file.
- so use **dtls** if you want the keys in an order

We'll look at **dt** in further detail later

idf1.idd_info - Overview

```
idd_info = idf1.idd_info # all the idd data
print type(idd_info)
```

```
<type 'list'>
```

```
print len(idd_info) # number of items in the list
```

```
683
```

```
# print the first three items
idd_info[:3]
```

```
[[{}],
 [{}],
 [{'format': ['singleLine'], 'unique-object': ['']},
  {'default': ['7.0'],
   'field': ['Version Identifier'],
   'required-field': ['']}]]
```

```
# print the first three items in seperate lines
for i, item in enumerate(idd_info[:3]):
    print "%s. %s" % (i, item)
```

```
0. [{}]
1. [{}]
2. [{'unique-object': [''], 'format': ['singleLine']}, {'default': ['7.0'], 'field': [
↪'Version Identifier'], 'required-field': ['']}]
```

That does not make much sense. Below is the first 3 items from the idd file

```
Lead Input;

Simulation Data;

\group Simulation Parameters

Version,
      \unique-object
      \format singleLine
  A1 ; \field Version Identifier
```

(continues on next page)

(continued from previous page)

```
      \required-field
      \default 7.0
```

- If you compare the text file with the sturcture of idd_info, you can start to see the similarities
- Note that the idd_info does not have the object name.
- This was an unfortunate design decision that we are stuck with now :-(.
- We need to jump through some hoops to get to an item in idd_info

```
# the object "VERSION" is the third item in idd_info
# to get to "VERSION" we need to find it's location in the list
# we use "dtls" to do this
location_of_version = dtls.index("version".upper())
print location_of_version
```

```
2
```

```
# print idd_info of "VERSION"
idd_info[location_of_version]
```

```
[{'format': ['singleLine'], 'unique-object': ['']},
 {'default': ['7.0'], 'field': ['Version Identifier'], 'required-field': ['']}]
```

NOTE:

- the idd file is very large and uses a lot of memory when pulled into idd_info
- only one copy of idd_info is kept when eppy is running.
- This is the reason, eppy throws an exception when you try to set the idd file when it has already been set

idf1.model.dt - in detail

Let us look at a specific object, say **MATERIAL:AIRGAP** in idf1.model.dt

```
dt = idf1.model.dt
```

```
airgaps = dt['MATERIAL:AIRGAP'.upper()]
print type(airgaps)
```

```
<type 'list'>
```

```
airgaps
```

```
[['MATERIAL:AIRGAP', 'F04 Wall air space resistance', 0.15],
 ['MATERIAL:AIRGAP', 'F05 Ceiling air space resistance', 0.18]]
```

A snippet of the **idf** text file shows this

```
MATERIAL:AIRGAP,
    F04 Wall air space resistance,    !- Name
    0.15;                     !- Thermal Resistance
```

(continues on next page)

(continued from previous page)

```

MATERIAL:AIRGAP,
    F05 Ceiling air space resistance,    !- Name
    0.18;                        !- Thermal Resistance
```

Notice the following things about idf1.model.dt:

- The idf model is held within a dict.
- the keys in the dict are names of the IDF objects in caps, such as BUILDING, VERSION, CONSTRUCTION, MATERIAL:AIRGAP etc.
- The values in the dict are lists
- the list contains lists. This means that **airgaps** can contain more than one airgap.
- So airgaps = [airgap1, airgap2, . . .].
- where, airgaps1 = [Type_of_Object, field1, field2, field3, ...]
- In airgaps1, all types have been converted. Note that "Thermal Resistance" is a float and not a string

What about an Energyplus object that does not exist in the idf file ?

```
roofs = dt['ROOF']
print roofs
```

```
[]
```

You get an empty list, meaning there are no roof items within roofs

idf1.idd_info - in detail

Let us find the idd_info for airgaps

```
location_of_airgaps = dtls.index("material:airgap".upper())
print location_of_airgaps
```

```
50
```

```
idd_airgaps = idd_info[location_of_airgaps]
idd_airgaps
```

```
[{'memo': ['Air Space in Opaque Construction'], 'min-fields': ['2']},
 {'field': ['Name'],
  'reference': ['MaterialName'],
  'required-field': [''],
  'type': ['alpha']},
 {'field': ['Thermal Resistance'],
  'minimum>': ['0'],
  'type': ['real'],
  'units': ['m2-K/W']}]
```

Compare to text in idd file:

```
Material:AirGap,
       \min-fields 2
       \memo Air Space in Opaque Construction
  A1 , \field Name
       \required-field
       \type alpha
       \reference MaterialName
  N1 ; \field Thermal Resistance
       \units m2-K/W
       \type real
       \minimum> 0
```

- idd_airgaps gives details about each field
- the last field N1 says that *type = real*
- This tells us that the text value coming from the the test file has to be converted to a float

Syntactic Sugar

from: https://www.princeton.edu/~achaney/tmve/wiki100k/docs/Syntactic_sugar.html *"Syntactic sugar is a computer science term that refers to syntax within a programming language that is designed to make things easier to read or to express, while alternative ways of expressing them exist"*

Wikipedia article on syntactic sugar

All the rest of the code in eppy is simply syntactic sugar over the data structure in model.dtls, model.dt and idd_info

Of course, the above statement is a gross exageration, but it gives you a basis for understanding the code that comes later. At the end of the day, any further code is simply a means for changing the data within model.dt. And you need to access the data within model.dtls and idd_info to do so.

Bunch

Bunch is a great library that subclasses dict. You can see it at:

- https://pypi.python.org/pypi/bunch/1.0.1
- https://github.com/dsc/bunch

Let us first take a look at a dict

```
adict = {'a':1, 'b':2, 'c':3}
adict
```

```
{'a': 1, 'b': 2, 'c': 3}
```

```
# one would access the values in this dict by:
print adict
print adict['a']
print adict['b']
print adict['c']
```

```
{'a': 1, 'c': 3, 'b': 2}
1
2
3
```

Bunch allows us to do this with a lot less typing

```
from bunch import Bunch
bunchdict = Bunch(adict)
print bunchdict
print bunchdict.a
print bunchdict.b
print bunchdict.c
```

```
Bunch(a=1, b=2, c=3)
1
2
3
```

Let us take a look at variable **airgaps** from the previous section.

```
airgaps
```

```
[['MATERIAL:AIRGAP', 'F04 Wall air space resistance', 0.15],
 ['MATERIAL:AIRGAP', 'F05 Ceiling air space resistance', 0.18]]
```

```
airgap1, airgap2 = airgaps[0], airgaps[1]
```

```
airgap1
```

```
['MATERIAL:AIRGAP', 'F04 Wall air space resistance', 0.15]
```

We are going to subclass bunch so that we can do the following to **airgap1** from the previous section:

- airgap1.Name
- airgap1.Thermal_Resistance

to remind you, the text file we are reading looks like this:

```
MATERIAL:AIRGAP,
    F04 Wall air space resistance,    !- Name
    0.15;                             !- Thermal Resistance
```

- We are using the field names that come from the idd file
- A space and other illegal (illegal for python) characters are replaced by an underscore

It is a little tricky tring to use bunch with airgap, because:

- airgap is a list
- but bunch works on dicts

So we do it in the following way:

- we make a new Bunch from the **airgap** list.
- The Bunch is made by by doing airgap1 = Bunch({“Name” : “F04 Wall air space resistance”, “Thermal_Resistance” : 0.15})

- This will allow us to use the dot notation we see in bunch
- Of course if we make changes in this Bunch, the **airgap** list does not change
- Ideally we would like to see the changes reflected in the **airgap** list
- We subclass Bunch as EpBunch. EpBunch is designed so that changes in EpBunch will make changes to the **airgap** list

Note: Some simplifications were made in the explanations above. So take it with a pinch of salt :-)

EpBunch

The code of EpBunch is in eppy/bunch_subclass.py. If you look at the code you will see The subclassing happening in the following manner:

- Bunch -> EpBunch1 -> EpBunch2 -> ….. -> EpBunch5 , where "Bunch -> EpBunch" means "EpBunch subclassed from Bunch"
- then EpBunch = EpBunch5

Question: Are you demented ? Why don't you just subclass Bunch -> EpBunch ?

Answer: One can get demented trying to subclass from dict. This is pretty tricky coding and testing-debugging is difficult, since we are overriding built-in functions of dict. When you make mistakes there, the subclassed dict just stops working, or does very strange things. So I built it in a carefull and incremental way, fully testing before subclassing again. Each subclass implements some functionality and the next one implements more.

EpBunch is described in more detail in the next section

11.1.5 EpBunch

Author Santosh Philip.

EpBunch is at the heart of what makes eppy easy to use. Specifically Epbunch is what allows us to use the syntax `building.Name` and `building.North_Axis`. Some advanced coding had to be done to make this happen. Coding that would be easy for professional programmers, but not for us ordinary folk :-(

Most of us who are going to be coding eppy are not professional programmers. I was completely out of my depth when I did this coding. I had the code reviewed by programmers who do this for a living (at python meetups in the Bay Area). In their opinion, I was not doing anything fundamentally wrong.

Below is a fairly long explanation, to ease you into the code. Read through the whole thing without trying to understand every detail, just getting a birds eye veiw of the explanation. Then read it again, you will start to grok some of the details. All the code here is working code, so you can experiment with it.

Magic Methods (Dunders) of Python

To understand how EpBunch or Bunch is coded, one has to have an understanding of the magic methods of Python. (For a background on magic methods, take a look at http://www.rafekettler.com/magicmethods.html) Let us dive straight into this with some examples

```
adict = dict(a=10, b=20) # create a dictionary
print adict
print adict['a']
print adict['b']
```

```
{'a': 10, 'b': 20}
10
20
```

What happens when we say d['a'] ?

This is where the magic methods come in. Magic methods are methods that work behind the scenes and do some magic. So when we say d['a'], The dict is calling the method `__getitem__('a')`.

Magic methods have a *double underscore* "__", called **dunder** methods for short

Let us override that method and see what happens.

```
class Funnydict(dict): # we are subclassing dict here
    def __getitem__(self, key):
        value = super(Funnydict, self).__getitem__(key)
        return "key = %s, value = %s" % (key, value)

funny = Funnydict(dict(a=10, b=20))
print funny
```

```
{'a': 10, 'b': 20}
```

The print worked as expected. Now let us try to print the values

```
print funny['a']
print funny['b']
```

```
key = a, value = 10
key = b, value = 20
```

Now that worked very differently from a dict

So it is true, funny['a'] does call `__getitem__()` that we just wrote

Let us go back to the variable **adict**

```
# to jog our memory
print adict
```

```
{'a': 10, 'b': 20}
```

```
# this should not work
print adict.a
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)

<ipython-input-5-8aa7211fcb66> in <module>()
      1 # this should not work
----> 2 print adict.a


AttributeError: 'dict' object has no attribute 'a'
```

What method gets called when we say **adict.a** ?

The magic method here is `__getattr__()` and `__setattr__()`. Shall we override them and see if we can get the dot notation to work ?

```
class Like_bunch(dict):
    def __getattr__(self, name):
        return self[name]
    def __setattr__(self, name, value):
        self[name] = value

lbunch = Like_bunch(dict(a=10, b=20))
print lbunch
```

```
{'a': 10, 'b': 20}
```

Works like a dict so far. How about **lbunch.a** ?

```
print lbunch.a
print lbunch.b
```

```
10
20
```

Yipeee !!! I works

How about `lbunch.nota = 100`

```
lbunch.anot = 100
print lbunch.anot
```

```
100
```

All good here. But don't trust the code above too much. It was simply done as a demonstration of **dunder** methods and is not fully tested.

Eppy uses the bunch library to do something similar. You can read more about the bunch library in the previous section.

Open an IDF file

Once again let us open a small idf file to test.

```
# you would normaly install eppy by doing
# python setup.py install
# or
# pip install eppy
# or
# easy_install eppy

# if you have not done so, uncomment the following three lines
import sys
# pathnameto_eppy = 'c:/eppy'
pathnameto_eppy = '../../../'
sys.path.append(pathnameto_eppy)
```

```
from eppy import modeleditor
from eppy.modeleditor import IDF
iddfile = "../../../eppy/resources/iddfiles/Energy+V7_2_0.idd"
fname1 = "../../../eppy/resources/idffiles/V_7_2/dev1.idf"

IDF.setiddname(iddfile)
idf1 = IDF(fname1)
idf1.printidf()
```

```
VERSION,
    7.3;                        !- Version Identifier

SIMULATIONCONTROL,
    Yes,                        !- Do Zone Sizing Calculation
    Yes,                        !- Do System Sizing Calculation
    Yes,                        !- Do Plant Sizing Calculation
    No,                         !- Run Simulation for Sizing Periods
    Yes;                        !- Run Simulation for Weather File Run Periods

BUILDING,
    Empire State Building,      !- Name
    30.0,                       !- North Axis
    City,                       !- Terrain
    0.04,                       !- Loads Convergence Tolerance Value
    0.4,                        !- Temperature Convergence Tolerance Value
    FullExterior,               !- Solar Distribution
    25,                         !- Maximum Number of Warmup Days
    6;                          !- Minimum Number of Warmup Days

SITE:LOCATION,
    CHICAGO_IL_USA TMY2-94846,      !- Name
    41.78,                      !- Latitude
    -87.75,                     !- Longitude
    -6.0,                       !- Time Zone
    190.0;                      !- Elevation

MATERIAL:AIRGAP,
    F04 Wall air space resistance,      !- Name
    0.15;                       !- Thermal Resistance

MATERIAL:AIRGAP,
    F05 Ceiling air space resistance,      !- Name
    0.18;                       !- Thermal Resistance
```

```
dtls = idf1.model.dtls
dt = idf1.model.dt
idd_info = idf1.idd_info
```

```
dt['MATERIAL:AIRGAP']
```

```
[['MATERIAL:AIRGAP', 'F04 Wall air space resistance', 0.15],
 ['MATERIAL:AIRGAP', 'F05 Ceiling air space resistance', 0.18]]
```

```
obj_i = dtls.index('MATERIAL:AIRGAP')
obj_idd = idd_info[obj_i]
obj_idd
```

```
[{'memo': ['Air Space in Opaque Construction'], 'min-fields': ['2']},
 {'field': ['Name'],
  'reference': ['MaterialName'],
  'required-field': [''],
  'type': ['alpha']},
 {'field': ['Thermal Resistance'],
  'minimum>': ['0'],
  'type': ['real'],
  'units': ['m2-K/W']}]
```

For the rest of this section let us look at only one airgap object

```
airgap = dt['MATERIAL:AIRGAP'][0]
airgap
```

```
['MATERIAL:AIRGAP', 'F04 Wall air space resistance', 0.15]
```

Subclassing of Bunch

Let us review our knowledge of bunch

```
from bunch import Bunch
adict = {'a':1, 'b':2, 'c':3}
bunchdict = Bunch(adict)
print bunchdict
print bunchdict.a
print bunchdict.b
print bunchdict.c
```

```
Bunch(a=1, b=2, c=3)
1
2
3
```

Bunch lets us use dot notation on the keys of a dictionary. We need to find a way of making `airgap.Name` work. This is not straightforward because, airgap is **list** and Bunch works on **dicts**. It would be easy if airgap was in the form `{'Name' : 'F04 Wall air space resistance', 'Thermal Resistance' : 0.15}`.

The rest of this section is a simplified version of how EpBunch works.

```
class EpBunch(Bunch):
    def __init__(self, obj, objls, objidd, *args, **kwargs):
        super(EpBunch, self).__init__(*args, **kwargs)
        self.obj = obj
        self.objls = objls
        self.objidd = objidd
```

The above code shows how EpBunch is initialized. Three variables are passed to EpBunch to initialize it. They are `obj, objls, objidd`.

```
obj = airgap
objls = ['key', 'Name', 'Thermal_Resistance'] # a function extracts this from idf1.
↪idd_info
objidd = obj_idd
#
```

(continues on next page)

(continued from previous page)

```
print obj
print objls
# let us ignore objidd for now
```

```
['MATERIAL:AIRGAP', 'F04 Wall air space resistance', 0.15]
['key', 'Name', 'Thermal_Resistance']
```

Now we override `__setattr__()` and `__getattr__()` in the following way

```
class EpBunch(Bunch):
    def __init__(self, obj, objls, objidd, *args, **kwargs):
        super(EpBunch, self).__init__(*args, **kwargs)
        self.obj = obj
        self.objls = objls
        self.objidd = objidd

    def __getattr__(self, name):
        if name in ('obj', 'objls', 'objidd'):
            return super(EpBunch, self).__getattr__(name)
        i = self.objls.index(name)
        return self.obj[i]

    def __setattr__(self, name, value):
        if name in ('obj', 'objls', 'objidd'):
            super(EpBunch, self).__setattr__(name, value)
            return None
        i = self.objls.index(name)
        self.obj[i] = value
```

```
# Let us create a EpBunch object
bunch_airgap = EpBunch(obj, objls, objidd)
# Use this table to see how __setattr__ and __getattr__ work in EpBunch

obj   = ['MATERIAL:AIRGAP', 'F04 Wall air space resistance', 0.15                 ]
objls = ['key',             'Name',                          'Thermal_Resistance']
i     =   0                  1                                2
```

```
print bunch_airgap.Name
print bunch_airgap.Thermal_Resistance
```

```
F04 Wall air space resistance
0.15
```

```
print bunch_airgap.obj
```

```
['MATERIAL:AIRGAP', 'F04 Wall air space resistance', 0.15]
```

Let us change some values using the dot notation

```
bunch_airgap.Name = 'Argon in gap'
```

```
print bunch_airgap.Name
```

```
Argon in gap
```

```
print bunch_airgap.obj
```

```
['MATERIAL:AIRGAP', 'Argon in gap', 0.15]
```

Using the dot notation the value is changed in the list

Let us make sure it actually has done that.

```
idf1.model.dt['MATERIAL:AIRGAP'][0]
```

```
['MATERIAL:AIRGAP', 'Argon in gap', 0.15]
```

`EpBunch` acts as a wrapper around `idf1.model.dt['MATERIAL:AIRGAP'][0]`

In other words `EpBunch` is just **Syntactic Sugar** for `idf1.model.dt['MATERIAL:AIRGAP'][0]`

Variables and Names in Python

At this point your reaction may, "I don't see how all those values in `idf1.model.dt` changed". If such question arises in your mind, you need to read the following:

- Other languages have 'variables'
- Python has 'names'
- Also see Facts and myths about Python names and values

This is especially important if you are experienced in other languages, and you expect the behavior to be a little different. Actually follow and read those links in any case.

Continuing with EpBunch

EpBunch_1

The code for EpBunch in the earlier section will work, but has been simplified for clarity. In file `bunch_subclass.py` take a look at the class **EpBunch_1** . This class does the first override of `__setattr__` and `__getattr__`. You will see that the code is a little more involved, dealing with edge conditions and catching exceptions.

EpBunch_1 also defines `__repr__`. This lets you print EpBunch in a human readable format. Further research indicates that `__str__` should have been used to do this, not `__repr__` :-(

EpBunch_2

`EpBunch_2` is subclassed from `EpBunch_1`.

It overrides `__setattr__` and `__getattr__` to add a small functionality that has not been documented or used. The idea was to give the ability to shorten field names with alias. So `building.Maximum_Number_of_Warmup_Days` could be made into `building.warmupdays`.

I seemed like a good idea when I wrote it. Ignore it for now, although it may make a comeback :-)

EpBunch_3

`EpBunch_3` is subclassed from `EpBunch_2`.

EpBunch_3 adds the ability to add functions to EpBunch objects. This would allow the object to make calculations using data within the object. So `BuildingSurface:Detailed` object has all the geometry data of the object. The function 'area' will let us calculate the are of the object even though area is not a field in `BuildingSurface:Detailed`.

So you can call `idf1.idfobjects["BuildingSurface:Detailed"][0].area` and get the area of the surface.

At the moment, the functions can use only data within the object for it's calculation. We need to extend this functionality so that calculations can be done using data outside the object. This would be useful in calculating the volume of a Zone. Such a calculation would need data from the surfaces that the aone refers to.

EpBunch_4

`EpBunch_4` is subclassed from `EpBunch_3`.

`EpBunch_4` overrides `_setitem__` and `__getitem__`. Right now `airgap.Name` works. This update allows `airgap["Name"]` to work correctly too

EpBunch_5

`EpBunch_5` is subclassed from `EpBunch_4`.

`EpBunch_5` adds functions that allows you to call functions `getrange` and `checkrange` for a field

Finally EpBunch

`EpBunch = EpBunch_5`

Finally `EpBunch_5` is named as EpBunch. So the rest of the code uses EpBunch and in effect it uses `Epbunch_5`

11.1.6 Idf_MSequence - Syntactic Sugar work

Underlying Data structure of again

Let us open a small idf file and look at the underlying data structure.

```
# assume we have open an IDF file called idf
# let us add three construction objects to it
idf.newidfobject('construction'.upper(), Name='C1')
idf.newidfobject('construction'.upper(), Name='C2')
idf.newidfobject('construction'.upper(), Name='C3')
constructions = idf.idfobjects['construction'.upper()]
print constructions
```

```
[
CONSTRUCTION,
    C1;                       !- Name
,
```

(continues on next page)

(continued from previous page)

```
CONSTRUCTION,
    C2;                      !- Name
,
CONSTRUCTION,
    C3;                      !- Name
]
```

We know that constructions us just syntactic sugar around the underlying data structure. Let us call the underlying data structure *real_constructions*

```
# set real_constructions
real_constructions = = idf.model.dt['construction'.upper()]
print real_constructions
```

```
[['CONSTRUCTION', 'C1'], ['CONSTRUCTION', 'C2'], ['CONSTRUCTION', 'C3']]
```

```
real_constructions -> the underlying data structure
constructions -> syntactic sugar for real_constructions
```

So any changes made in constructions should reflected in constructions. Let us test this out.

```
constructions[0].Name = 'New C1'
print constructions
```

```
[
CONSTRUCTION,
    New C1;                  !- Name
,
CONSTRUCTION,
    C2;                      !- Name
,
CONSTRUCTION,
    C3;                      !- Name
]
```

```
print real_constructions
```

```
[['CONSTRUCTION', 'New C1'], ['CONSTRUCTION', 'C2'], ['CONSTRUCTION', 'C3']]
```

Even though we made the change only in *constructions*, we can see the changes in both *constructions* and *real_constructions*. `Ep_Bunch` takes care of this for us.

```
print 'type for constructions', type(constructions)
```

```
type for constructions <type 'list'>
```

since constructions is a list, we can do all the list operations on it. Let us try some of them:

```
constructions.pop(0)
```

```
CONSTRUCTION,
    New C1;                  !- Name
```

```
print constructions
```

```
[
CONSTRUCTION,
    C2;                       !- Name
,
CONSTRUCTION,
    C3;                       !- Name
]
```

That makes sense. We poped the first item in the list and now we have only two items.

Is this change reflected in real_constructions ?

```
print real_constructions
```

```
[['CONSTRUCTION', 'New C1'], ['CONSTRUCTION', 'C2'], ['CONSTRUCTION', 'C3']]
```

Dammit !! Why not ?

We still have 3 items in real_constructions and 2 items in constructions

```
print 'type for constructions', type(constructions)
print 'id of constructions', id(constructions)
print 'type for real_constructions', type(constructions)
print 'id of real_constructions', id(real_constructions)
```

```
type for constructions <type 'list'>
id of constructions 4576898440
type for real_constructions <type 'list'>
id of real_constructions 4535436208
```

- Both `constructions` and `real_constructions` are lists.
- But looking at their ids, it is clear that they are two different lists.
- poping an item in one list will not pop it in the other list :-(
- In `constructions[0].Name = "New C1"` we see changes to an item within `constructions` is reflected within `real_constructions`
- `EpBunch` takes care of that connection
- We are having problems with the list functions.
- we see that pop() does not work for us
- similarly the results of append(), insert(), sort() and reverse() in `constructions` will not be reflected in `real_constructions`

This is how it works in eppy version 0.5

We need to fix this. Now we describe how this problem was fixed.

`constructions` should be a list-like wrapper around `real_constructions`. Python has an excellent data structure called `collections.MutableSequence` that works perfectly for this. Alex Martelli has a great discussion of this in this stackoverflow thread Overriding append method after inheriting from a Python List

- So we make a class `eppy.idf_msequence.Idf_MSequence` that inherits form `collections.MutableSequence`

- `constructions` is now an instance of `eppy.idf_msequence.Idf_MSequence`
- reading the above stackoverflow thread and the code wihtin `eppy.idf_msequence.Idf_MSequence` should show you how it works
- version of eppy higher than 0.5 will use `eppy.idf_msequence.Idf_MSequence`

Let us take a look at how it works (in all versions of eppy newer than 0.5):

```
# using eppy version greater than 0.5
import sys
# pathnameto_eppy = 'c:/eppy'
pathnameto_eppy = '../../../'
sys.path.append(pathnameto_eppy)
from eppy import modeleditor
from eppy.modeleditor import IDF
iddfile = "../../../eppy/resources/iddfiles/Energy+V7_2_0.idd"
fname1 = "../../../eppy/resources/idffiles/V_7_2/smallfile.idf"
IDF.setiddname(iddfile)
idf = IDF(fname1)

idf.newidfobject('construction'.upper(), Name='C1')
idf.newidfobject('construction'.upper(), Name='C2')
idf.newidfobject('construction'.upper(), Name='C3')
constructions = idf.idfobjects['construction'.upper()]
```

```
print constructions
```

```
[
CONSTRUCTION,
    C1;                       !- Name
,
CONSTRUCTION,
    C2;                       !- Name
,
CONSTRUCTION,
    C3;                       !- Name
]
```

```
real_constructions = idf.model.dt['construction'.upper()]
print real_constructions
```

```
[['CONSTRUCTION', 'C1'], ['CONSTRUCTION', 'C2'], ['CONSTRUCTION', 'C3']]
```

Shall we test `pop(0)` here ?

```
constructions.pop(0)
```

```
CONSTRUCTION,
    C1;                       !- Name
```

```
print constructions
```

```
[
CONSTRUCTION,
    C2;                       !- Name
```

(continues on next page)

(continued from previous page)

```
,
CONSTRUCTION,
    C3;                       !- Name
]
```

```
print real_constructions
```

```
[['CONSTRUCTION', 'C2'], ['CONSTRUCTION', 'C3']]
```

Awesome !!! both `constructions` and `real_constructions` have the same number of items

```
print type(constructions)
print type(real_constructions)
```

```
<class 'eppy.idf_msequence.Idf_MSequence'>
<type 'list'>
```

what kind of sorcery is this. How did that work. How does `Idf.Msequence` do this magic ? Let us look at that link in stackoverflow. The question raised in stackovverflow is:

I want to create a list that can only accept certain types. As such, I'm trying to inherit from a list in Python, and overriding the append() method like so: and there is a sample code after this.

Alex Martelli responds:

Not the best approach! Python lists have so many mutating methods that you'd have to be overriding a bunch (and would probably forget some).

Rather, wrap a list, inherit from collections.MutableSequence, and add your checks at the very few "choke point" methods on which MutableSequence relies to implement all others. Alex's code follows after this point. In `eppy.idf_msequence` I have included Alex's code.

Stop here and read through the stackoverflow link

Well . . . you don't really have to. It does go off on some tangents unrelated to what we do in eppy.

The strategy in `eppy.idf_msequence.Idf_MSequence` is to have two lists, list1 and list2. To play with this I made a simple class `TwoLists`. Here `TwoLists` acts just like a list. Any operation list operation on `TwoLists` will result in a similar operation on both list1 and list2. `TwoLists` is not used in eppy, I simply use it to flesh out how `MutableSequence` can be used. I am going to play with `TwoLists` here to show you how cool it is :-)

```
from eppy.idf_msequence import TwoLists
twolists = TwoLists()
print twolists
```

```
list1 = [], list2 = []
```

```
twolists.append(5)
print twolists
```

```
list1 = [5], list2 = ['r_5']
```

```
twolists.append(dict(a=15))
print twolists
```

```
list1 = [5, {'a': 15}], list2 = ['r_5', "r_{'a': 15}"]
```

```
twolists.insert(1, 42)
print twolists
```

```
list1 = [5, 42, {'a': 15}], list2 = ['r_5', 'r_42', "r_{'a': 15}"]
```

```
twolists.pop(-1)
```

```
{'a': 15}
```

```
print twolists
```

```
list1 = [5, 42], list2 = ['r_5', 'r_42']
```

Isn't that neat !! `Idf_MSequence` works in a similar way. Out of sheer laziness I am going to let you figure it out on your own. (ignore `Idf_MSequence_old`, since that went in the wrong direction)

11.1.7 IDF in modeleditor

The previous section talks about EpBunch, which deals with a single object from Energyplus. Here we put all the pieces together so that we have the entire **IDF** file

Class IDF

IDF0 is the first class that was written. As the code was refined, it was further refined to by subclassing to IDF1, IDF2, IDF3. Finally the following was set as IDF = IDF3

Class IDF0

Some important methods in IDF0

IDF0.setiddname

This method has a decorator `@classmethod`. This decorator makes the method a class method. From a stackoverflow comment, I found a brief description of when this should be used.

```
"Class methods are essential when you are doing set-up or computation that
precedes the creation of an actual instance, because until the instance exists
you obviously cannot use the instance as the dispatch point for your method
calls"
```

Having said that, I am outside my comfort zone on trying to explain this in any depth. I will simply explain what I am doing with this here. Below is a brief explanation intent.

- the idd file is a very large file. There is a large overhead in opening more than one idd file.
- A design decision was made to use only one idd file to be used in a script.
- This means that you cannot open two idf files that are of different version (meaning they will use different idd files)

- You can open any number of idf file as long as they are of the same version (meaing, the use the same idd file)

The class method allows us to achieve the objective:

- The class method 'setiddname', allows us to set the name of the idd file, before creating an instance of IDF. It is set by the statement `IDF.setiddname(iddfilename)`
- There are other class methods that make sure that this idd name cannot be changed.
- An instance of the class IDF is created using the statement `idf = IDF(idffilename)`. This can be done multiple times, creating multiple instances and they will all use the same idd file

IDF0.`__init__`

IDF is initialized by passing it the idf file name. I would look like this:

> idf1 = IDF(filename) # filename can be a file name, file handle or an StringIO

- Once the class is initialized, it will read the idf file.
- If this the first time the class is inititalized, the idd file set by `setiddname()` will be read too.
- the idd file is read only once and then same copy is used.

IDF0.read

The read function is called when the class IDF is initialized. The read function calls routines from the original EPlusInterface program to do the actual read. The read get the values for the following variables:

- idfobjects
- model.dt
- model.dtls
- idd_info

The functions within EPlusInterface are not documented here. I wrote them in 2004 and I have no idea how I did them. they have been working really well with some minor updates. I don't intent to poke at that code yet.

Other IDF methods

The other functions in IDF0, IDF1, IDF2 and IDF3 not too complicated. It should be possible to understand them by reading the code.

Sometime in the future, these functions will be documented later in more detail

11.1.8 Unit Testing Eppy

Pytest for eppy

Pytest framework is used to unit test eppy

This section is yet to be documented

11.1.9 Documenting eppy

Sphinx is used to document eppy. Sphinx uses restructured text (*.rst) to build up the documentation. Some of the documentation is done directly in restructured text files. In other cases ipython notebooks are used to generate the restructured text files.

This documentation is incomplete.

When completed it will list the steps needed to let you to add to the documentation and then generate the html files

Restructured text

Ipython notebook

Sphinx

CHAPTER 12

LICENSE

CHAPTER 13

Indices and tables

- genindex
- modindex
- search