

BLD227462

# Connecting Third-Party Data and Automating Your Revit Designs in Dynamo

Adam McDonald  
Build Change

Patty Svenson  
Autodesk

Philippe Videau  
Autodesk

## Learning Objectives

- Learn how to recognize inefficiencies in manual workflows where data is being translated from one form to another (for instance, from 2D survey data to 3D Revit models)
- Learn how to recognize situations where you can capitalize on data structures in Dynamo to automate workflows
- Understand the use of custom nodes in Dynamo to parse and extract the data you need
- Learn how to use the parsed data to generate model elements in Revit
- Learn how to automate structural engineering analyses and the production of common Revit drawing (BONUS)

## Description

Ensuring that data stays at the center of the design-to-construction process, as well as connecting that information across architecture, engineering, and construction (AEC) disciplines, can lead to significant efficiency gains for the industry. In this class, we'll explore how to seamlessly integrate and transfer data across a building retrofit process—from on-site survey to structural retrofit design to project visualization and construction documentation. Learn how a team of Autodeskters and engineers from Build Change—a nonprofit organization that helps communities design and construct more-disaster-resistant buildings—used Dynamo to: connect survey data from a third-party application to Revit software; automate the creation of 3D as-builts, retrofit designs, and construction sheets; streamline structural engineering analysis; and provide project owners with an interactive visualization of the final design—and how these concepts and skills can be applied beyond the building retrofit process.

## **Speaker(s)**

### **Adam McDonald, New Frontier Technologies (Build Change)**

Adam a Registered Architect at New Frontier Technologies with 15 years experience combining work in the fields of architectural design, structural engineering, and construction management, in various international contexts. He holds an Master of Architecture degree from University of California Berkeley. Adam's work aims to improve the construction processes in communities that are prone to natural disasters, both from a planning and design perspective, as well as in building structures and performance.

### **Patty Svenson, Autodesk**

Based in San Francisco, Patty is a senior data scientist at Autodesk with the Advanced Analytics team in the Digital Platforms and Experience (DPE) organization. As a data scientist, Patty works on developing and maintaining machine learning models to predict retention for the early warning system, which is used by sales and marketing organizations to better target customer communication. She obtained her B.A. in math and econ from Dartmouth College in 2012, and her master's degree in analytics from Northwestern University in 2016.

### **Philippe Videau, Autodesk**

Philippe recently joined Autodesk's Forge team as a product manager, where he's helping to build out a workflow automation program for both the manufacturing and AEC spaces. Previously, much of his work involved communicating and connecting internal product and research teams as well as interfacing with manufacturing and construction customers to develop stories and content around how they were pushing the boundaries in their industries. He graduated in 2015 with a B.S. in Aerospace Engineering from UCLA.

## **Learn how to recognize inefficiencies in manual workflows where data is being translated from one form to another (for instance, from 2D survey data to 3D Revit models)**

### **Case Study**

In April 2018, The Autodesk Foundation, in collaboration with Team4Tech, sent a team of volunteers to the Build Change Colombia offices to develop a set of tools that could expedite our work in the Field. One Principal methodology of Build Change, in prevention of building collapse in future disasters, is to structurally retrofit houses that are poorly built. This requires full surveys of existing buildings.

In particular, one of the key Bottlenecks that Build Change had identified was the time it takes to record all existing conditions, including measurements, wall types, roof types, floor elevations, etc. for houses that are planned to be structurally retrofitted. One main challenge in the context of Colombia, and similarly in the Philippines and other areas, is the collection of field data when no two houses are the same. This requires generating a new, unique data set, and later a unique 3D model, for each and every house.

Prior to adopting Revit, our traditional method had been to go to the field and hand-sketch a plan of the house, record measurements on paper, and go back to the office to draw up the plans in AutoCAD. However, by digitizing our survey, we can not only feed our BIM workflow with standardized information readily processed, but we also have data that is transferrable between team members. This permits teams to stay in their locations, and send measurements from the field to the office, cutting travel time and expediting all parts of the process.

We found that there were simply no tools that could fully automate a custom generated floor plan for each house that was directly compatible with Revit. There were Reality Capture workflows that could scan the house, but still required manual tracing in Revit. So, the team was required to create a tool that was flexible enough to account for all of these varied housing characteristics, and had to incorporate data from a third-party application, manipulated to feed our Revit model.

Another constraint was that this solution needed to be low cost. Hardware purchases can often be a concern for governments or partners, and some of the more expensive scanning equipment is costly. Our solution was to use the mobile phones that we already have in our pockets, with a simple \$10 third-party app called MagicPlan. This can allow us to also crowdsource information, having partners of ours, or homeowners themselves, send us surveys of houses, enabling further scaling of the retrofitting program.

### **Project Goals**

Building upon the Revit Template that Build Change already had in place, we set out achieve 3 goals with the integration of Dynamo and a third-party application:

- Reduce amount of time required to collect data by implementing a digital survey, increasing efficiency and enabling Build Change to scale the retrofitting program.
- Increase the precision in the survey, reducing errors and missing data.
- Fully Automate the 3D modeling process, to reduce amount of time it then takes to generate drawings of an existing house to less than 2 hours by improving our current tools to transform survey data in to a Revit model.

## **Learn how to recognize situations where you can capitalize on data structures in Dynamo to automate workflows**

In general there are three things you need to look for when trying to decide if your workflow can be automated using Dynamo:

1. There are few manual inputs that are required in the workflow
2. The logic used in the input file is generally consistent and can be clearly documented
3. The logic used for data manipulation is generally consistent and can be clearly documented

The first point is self-explanatory – you cannot automate something if it requires a lot of manual touch points. For example, if the floorplan drawing were done on paper and not stored digitally, then the model creation cannot be fully automated. Even with the existence of handwriting recognition and machine learning, a computer will have considerable trouble digitizing a paper floorplan. Besides the obvious problems like the drawing is not to scale, people being inconsistent in where they write measurements, and handwriting recognition, it is extremely difficult to scan a drawing and automatically link all the relationships between elements of said drawing.

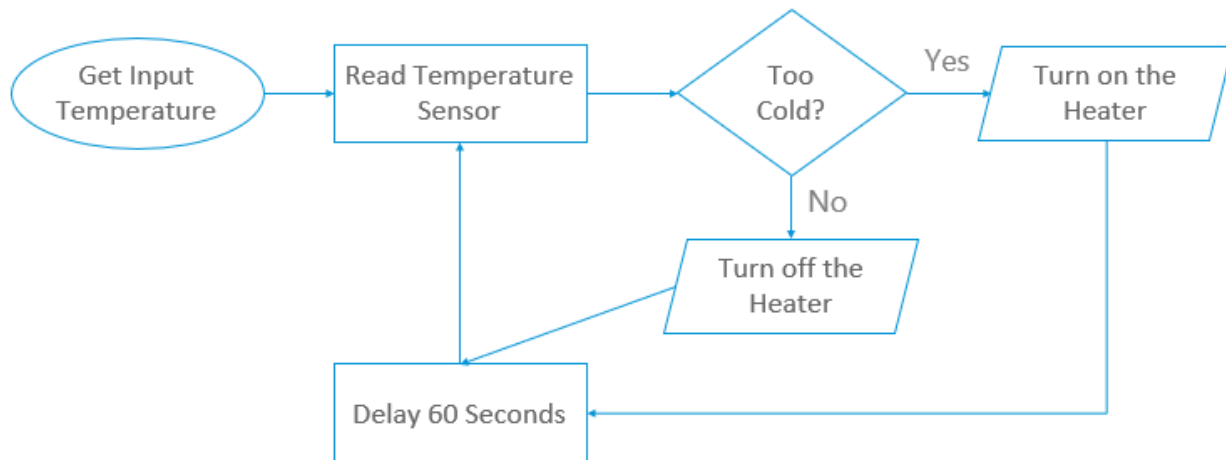
The second point is important because it tells you how your input data file is laid out. You cannot pull the data elements you want if you do not always know where they are, hence the consistency. Similarly, the third point is important because automation becomes more useful the more repetitive the workflow is. The consistency element for both the second and third point is important because automation is only useful if we do not spend more time manipulating the script than it saves us.

### **A few general thoughts on automation**

We automate when we identify an inefficiency in our process whose solution is rules based and systematically solved. For example, instead of getting up every time to turn on or off the heater when it got too hot or too cold, we invented the thermostat.

We can break down the thermostat into a programmatic framework:

A human triggers the process by giving the system a temperature input that he or she is comfortable with. The thermostat uses thermometers to detect the temperature in the house, and then embarks on a feedback loop.



*FEEDBACK PROCESS WORKFLOW OF A THERMOSTAT*

The process starts with the thermostat obtaining an input temperature

We read the temperature sensor installed in the house:

1. If the temperature of the house is greater than or equal to the input, turn off the heater to lower the heat and check back in 60 seconds
2. Else if the temperature of the house is less than the input, turn on the heater and check back in 60 seconds

The pseudo-code for this would look something like:

```

temperature_input = get_input()
current_temperature = get_temperature()
while thermostat_status is 'ON':
    if current_temperature < temperature_input
        then turn_on_heater() and wait 60 seconds
    else turn_off_heater() and wait 60 seconds
  
```

So long as we want the system to regulate temperature based on our inputs, automating the process makes the control system much more reliable and consistent than a manual system. An automated system is not plagued by laziness, so there are few delays from information transfer from point A to B; there are less misclicks since the data already exists digitally; and the system will be extremely consistent in its actions, because that is what it was programmed to do. Automation is great for streamlining workflows when there are a clear and predefined set of

actions governed by set rules. Conversely, automation can only be of very limited help in solving problems when you are unable to fully define the universe of rules governing the future actions that should be taken.

The data world follows these same principles. Leveraging data structures to shift data into a form that you can use is great when you have a problem that can be solved by a clear, predefined sequence of steps governed by a consistent and explicitly stated set of rules. For example, screening incoming inputs for anomalies is an automatable task - at the time of screening, you have a set of rules you can operate in to clearly judge whether the incoming inputs are good or bad. Optimizing the user experience of a website, however, is something that would be difficult to fully automate – you need a designer to be creative and come up with many iterations of the website to test. A machine can do some of the testing, but creativity is not something a machine can wholly replace.

### **Build Change: Automating the Revit model creation process**

Translating a 2D floorplan drawing into a 3D Revit model is a fantastic candidate for automation, simply because it's a direct mapping of information from one form to another. The rules for what should happen to the data do not change over time (start and end points for walls always indicate where a wall should be drawn). Earlier, we talked about why moving from paper to digitizing the initial floorplan drawing is important. We want to reiterate that this is critical because you can move digital information from one source to another with ease, but paper to digital almost always necessitates a manual process.

Now that we have this information stored digitally, we can take one of two approaches:

1. Have a person work off the drawing to create the 3D Revit model, or
2. Create a Dynamo script to use the data collected in the digitized 2D floorplan and do this automatically.

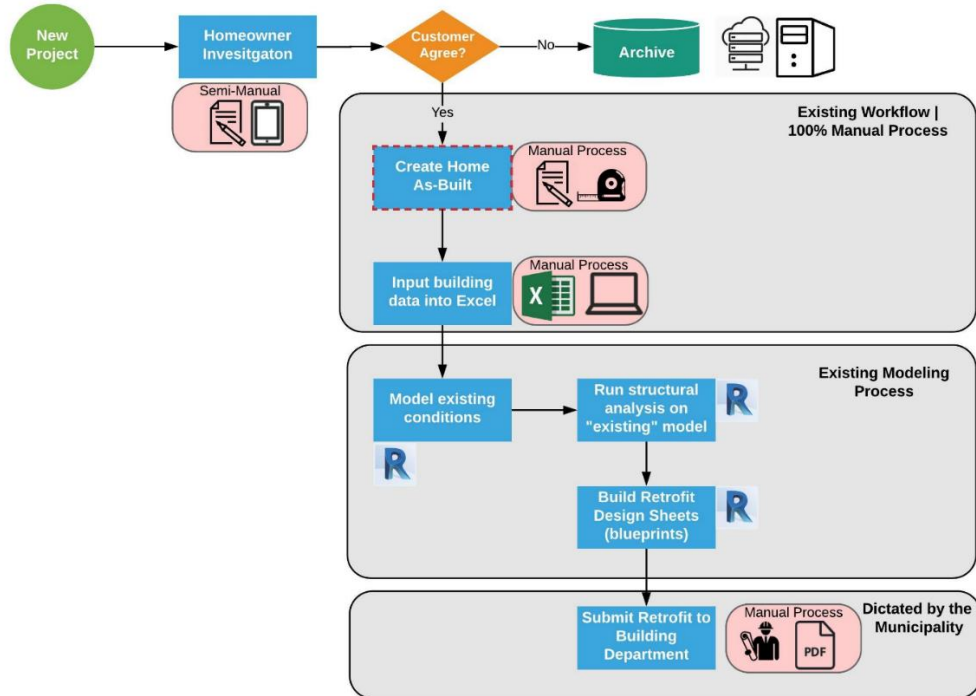
Having someone manually create the Revit model makes sense if there are only a few houses going through said workflow. It may not be worth the time and effort spent to figure out the rules behind when one action needs to be taken versus another if the volume going through the workflow is relatively low. However, if translating a 2D floorplan into a Revit model is a regular activity, then it is well worth putting in development time as automating the workflow could cut down the process from a few days to a couple of hours (where much of the time is spent in quality checking the generated drawing).

So how do we do all this?

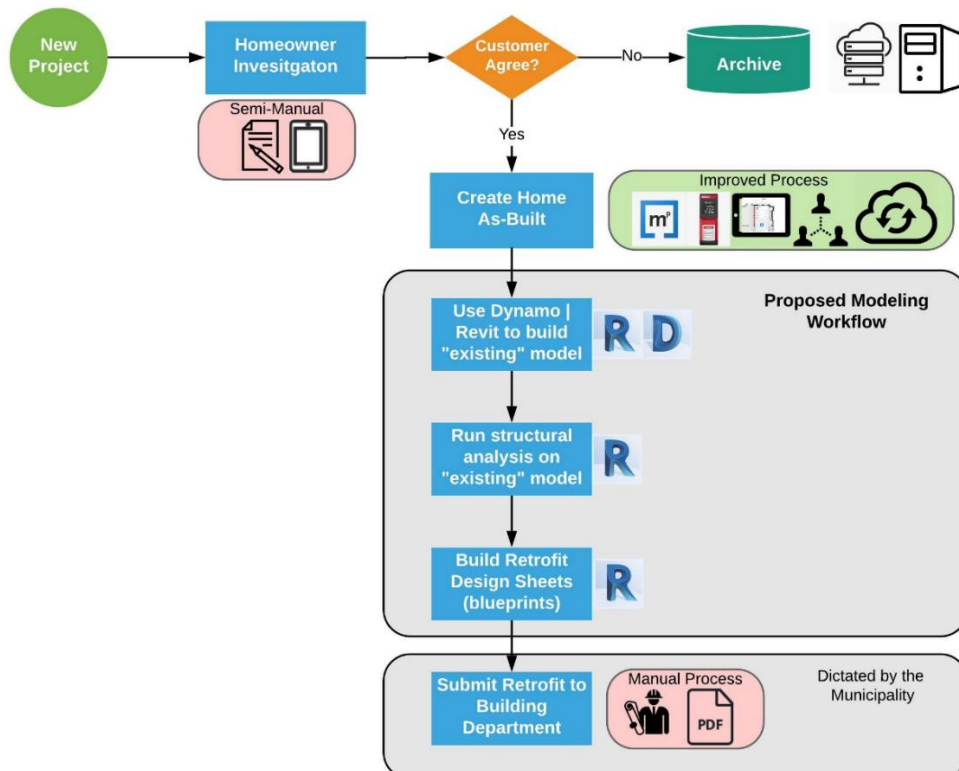
The general overview is that:

1. We export the digital 2D floorplan to a data file
2. We read this data file using Dynamo for Revit and load it using the IN
3. This data file is parsed (we take the string and chop it up into its constituent bits)
4. The data is then reorganized into a more user-friendly format and saved to a variable
5. The variable is then saved as the OUT and used in the Dynamo script

Our workflow in creating the as-built changed from a heavily manual workflow:



To a much-improved process that leverages the fact that the data exists digitally:





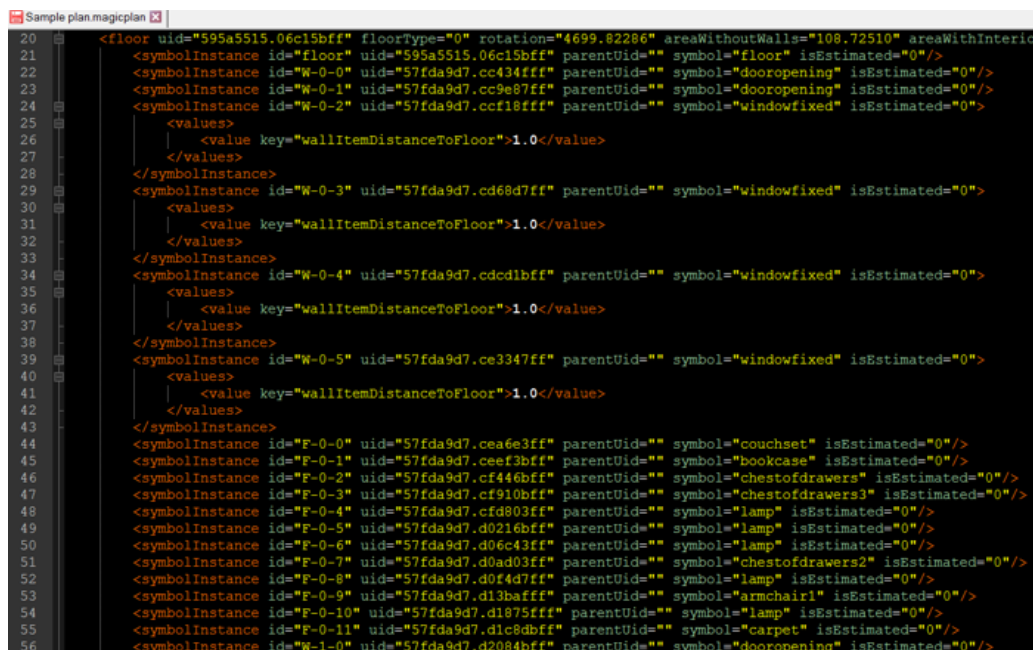
## Understand the use of custom nodes in Dynamo to parse and extract the data you need

Dynamo is compatible with an implementation of IronPython 2.7, which is compatible with Python 2.7. What this means is that you can use much of the functionality available in Python to write a script to create custom nodes. This frees you from being limited by just the nodes that currently exist in Dynamo and allows you to explore more effective ways of manipulating data, compacting your Dynamo scripts, and automating your workflows.

We can bring in data using an input file, process that file any way we need using a Python Script node, and output the data in whatever form we come up with. The flexibility we have in this data processing step allows us to organize the data coming in from different types of documents, from simple CSVs to more complicated XML files, and output exactly what we need. Doing this is not always necessary, especially for standard file formats, since Dynamo has nodes of its own available and optimized for certain standard tasks. However, once we deviate from a standard file format or standard tasks for which no node exists, making your own custom Python nodes is a good way to parse and extract the data you need.

For example, at Build Change, we created a series of custom Python nodes to extract just the specific information we wanted to use from an XML input. We could not use the standard XML parsing node because the file we were using was not a standard XML file.

Although it seems convoluted at first glance, the XML file from MagicPlan was laid out in a very useful manner for us.



```

20 <floor uid="595a5515.06c15bff" floorType="0" rotation="4699.82286" areaWithoutWalls="108.72510" areaWithInterio
21 <symbolInstance id="floor" uid="595a5515.06c15bff" parentUid="" symbol="floor" isEstimated="0"/>
22 <symbolInstance id="W-0-0" uid="57fda9d7.cc434fff" parentUid="" symbol="dooropening" isEstimated="0"/>
23 <symbolInstance id="W-0-1" uid="57fda9d7.cc9e87ff" parentUid="" symbol="dooropening" isEstimated="0"/>
24 <symbolInstance id="W-0-2" uid="57fda9d7.ccf18fff" parentUid="" symbol="windowfixed" isEstimated="0"/>
25 <values>
26 | <value key="WallItemDistanceToFloor">1.0</value>
27 </values>
28 </symbolInstance>
29 <symbolInstance id="W-0-3" uid="57fda9d7.cd68d7ff" parentUid="" symbol="windowfixed" isEstimated="0"/>
30 <values>
31 | <value key="WallItemDistanceToFloor">1.0</value>
32 </values>
33 </symbolInstance>
34 <symbolInstance id="W-0-4" uid="57fda9d7.cdcd1bfff" parentUid="" symbol="windowfixed" isEstimated="0"/>
35 <values>
36 | <value key="WallItemDistanceToFloor">1.0</value>
37 </values>
38 </symbolInstance>
39 <symbolInstance id="W-0-5" uid="57fda9d7.ce3347ff" parentUid="" symbol="windowfixed" isEstimated="0"/>
40 <values>
41 | <value key="WallItemDistanceToFloor">1.0</value>
42 </values>
43 </symbolInstance>
44 <symbolInstance id="F-0-0" uid="57fda9d7.cea6e3fff" parentUid="" symbol="couchset" isEstimated="0"/>
45 <symbolInstance id="F-0-1" uid="57fda9d7.ceef3bfff" parentUid="" symbol="bookcase" isEstimated="0"/>
46 <symbolInstance id="F-0-2" uid="57fda9d7.cf446bfff" parentUid="" symbol="chestofdrawers" isEstimated="0"/>
47 <symbolInstance id="F-0-3" uid="57fda9d7.cf910bfff" parentUid="" symbol="chestofdrawers3" isEstimated="0"/>
48 <symbolInstance id="F-0-4" uid="57fda9d7.cfd803fff" parentUid="" symbol="lamp" isEstimated="0"/>
49 <symbolInstance id="F-0-5" uid="57fda9d7.d0216bfff" parentUid="" symbol="lamp" isEstimated="0"/>
50 <symbolInstance id="F-0-6" uid="57fda9d7.d06c43fff" parentUid="" symbol="lamp" isEstimated="0"/>
51 <symbolInstance id="F-0-7" uid="57fda9d7.d0ad03fff" parentUid="" symbol="chestofdrawers2" isEstimated="0"/>
52 <symbolInstance id="F-0-8" uid="57fda9d7.d0f4d7fff" parentUid="" symbol="lamp" isEstimated="0"/>
53 <symbolInstance id="F-0-9" uid="57fda9d7.d13baffff" parentUid="" symbol="armchair1" isEstimated="0"/>
54 <symbolInstance id="F-0-10" uid="57fda9d7.d1875ffff" parentUid="" symbol="lamp" isEstimated="0"/>
55 <symbolInstance id="F-0-11" uid="57fda9d7.d1c8dbfff" parentUid="" symbol="carpet" isEstimated="0"/>
56 <symbolInstance id="W-1-0" uid="57fda9d7.d2084bfff" parentUid="" symbol="dooropening" isEstimated="0"/>

```

SAMPLE XML FILE FROM MAGICPLAN



Element	Description	Quantity
<plan>	The root element.	1
<values>	The attribute values of the plan.	0-1
<value>	The key and value of a specific attribute.	N
<floor>	The plan of a whole floor.	
<symbolInstance>	An instance of a symbol on the current floor along with its parameters. Symbol instances are referenced by furniture, door, and window elements.	N
<values>	The attribute values of the symbol instance.	0-1
<value>	The key and value of a specific attribute.	N
<floorRoom>	A room on the current floor.	N
<point>	A corner in the current room (the intersection of two walls).	N
<values>	The attribute values of the wall starting at the current point.	0-1
<value>	The key and value of a specific wall attribute.	N
<door>	A door located on a wall of the current room. Doors of two overlapping walls are represented once on each wall.	N
<window>	A window on a wall of the current room.	N
<furniture>	A piece of furniture in the current room.	N
<values>	The attribute values of the current room.	0-1
<value>	The key and value of a specific attribute.	N
<exploded>	An alternate representation of the current floor plan where walls are not grouped per room and each wall is represented only once.	1
<wall>	A wall on the current floor in its exploded representation.	N
<point>	The coordinates of one vertex (corner) of a broken line representing a wall in its exploded representation.	N
<door>	A door on the current floor. Each door is only represented once.	N
<window>	A window on the current floor. Each window is only represented once.	N
<furniture>	A piece of furniture on the current floor.	N

*THE LAYOUT OF A MAGICPLAN XML FILE FOLLOWS A PREDICTABLE HIERARCHY*

At the root of the XML file is a <plan> element, which contains attributes of the property as well as a list of <floor> elements representing the plans of the various floors.

Each <floor> element contains two complete and equivalent descriptions of the same floor plan: A list of <floorRoom> elements describing each room individually. Connecting walls and doors are described twice in this format.



for our purposes in creating custom nodes, you want Python 2.7, as Dynamo for Revit uses IronPython 2.7, which is compatible with Python 2.7. Anaconda also by default includes Spyder, a Python IDE, that you can use to develop on. You can download Anaconda here: <https://www.anaconda.com/download/#download>

2. Python is an excellent language for beginners, and there are many resources and tutorials that exist to help you learn how to code in Python. Some of them are found: <https://wiki.python.org/moin/BeginnersGuide/NonProgrammers>
3. There have been talks in prior AU sessions about using Python in Dynamo. This one in particular provides a good overview:
  - <http://au.autodesk.com/au-online/classes-on-demand/class-catalog/classes/year-2017/dynamo-studio/as124816-l#chapter=0>
4. The Autodesk Community also has various threads talking about learning Python. This one collates many useful threads:
  - <https://forum.dynamobim.com/t/python-nodes-basics/15872/2>

## Custom Nodes with Dynamo Python

We always start with code to tell the custom node to import modules that are useful to making the node work. Imports are useful because they bring in modules into Python. Rather than rewriting commonly used code, Python has common modules, which can be imported into your script. In the code that is available for download, we have two sets of import statements because Dynamo and a Python IDE require different imports. The sets have been labeled as such. By default, we assume you are going to use it in Dynamo, so the Python IDE imports are commented out with “#” preceding the statements. Remember: If you’re going swap to using the Python IDE version, please comment out the Dynamo version or your code will error.

```
###
#This section contains the important imports to make the code work
#Use this section if you are coding this in a Python IDE

import xml.etree.ElementTree as etree # Import ElementTree, which parses the XML
                                     PYTHON IDE VERSION: IMPORT SECTION OF PYTHON SCRIPT

###
#This section contains the important imports to make the code work
#Use this section if you are coding this in Dynamo

import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
import sys
sys.path.append("C:\Program Files (x86)\IronPython 2.7\Lib")
import xml.etree.ElementTree as etree # Import ElementTree, which parses the XML
                                     DYNAMO VERSION: IMPORT SECTION OF PYTHON SCRIPT
```

After importing the required modules (for example, below, we are using the etree module that we imported above), we use a File Path node to gather the filepath of the MagicPlan XML File

and pass it into our custom node so the node knows what file to read. You can pass multiple files IN, but you can only send one OUT object in Dynamo for Revit.

Once again, depending on if you're using a Python IDE vs Dynamo, the statements are slightly different. By default, we assume you are going to use it in Dynamo, so the Python IDE imports are commented out with "#" preceding the statements. Remember: If you're going swap to using the Python IDE version, please comment out the Dynamo version or your code will error.

```
###  
#Parse the XML from IN variable filepath  
## This is if you are coding this in a Python IDE  
inputFilePath = 'C:/Users/svensop/Documents/au - buildchange/code/' #File Address  
inputFile = 'Toms House-ver2.magicplan'  
tree = etree.parse(inputFilePath+inputFile) # Parse XML to Create Tree  
root = tree.getroot() # Get Root Structure
```

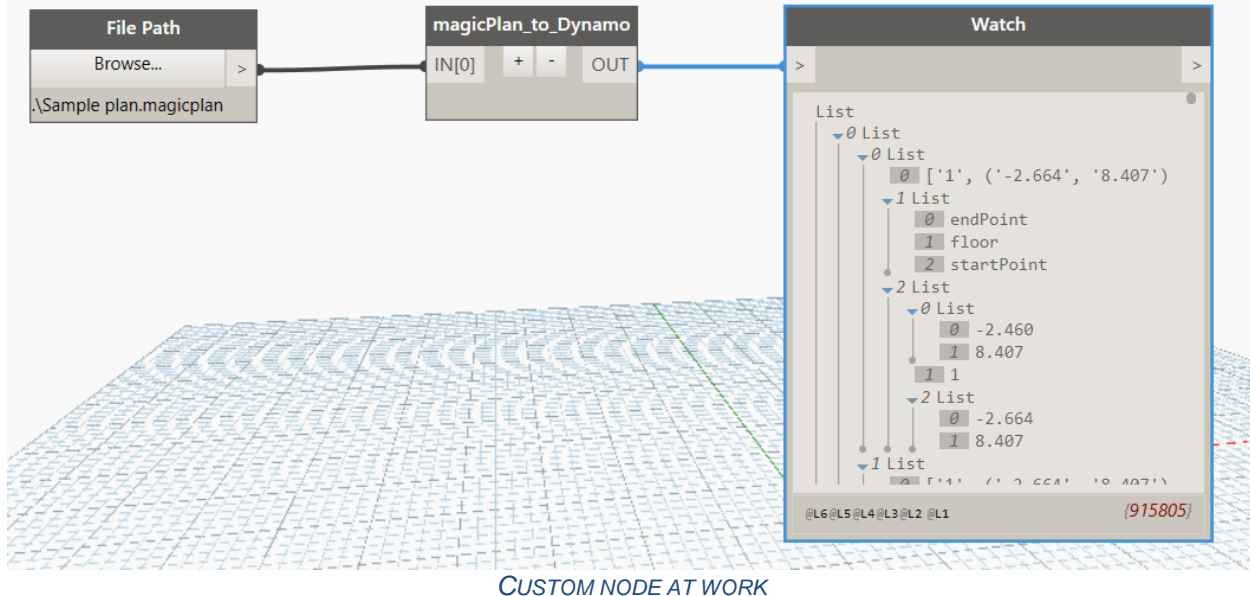
*PYTHON IDE VERSION: PASSING FILES IN AND CALLING ONE OF THE MODULES THAT WAS IMPORTED ABOVE*

For the Python IDE Version above, you will need to change the inputFilePath to wherever your file is located, and your inputFile to the name of the file.

```
###  
#Parse the XML from IN variable filepath  
#This is for handling the data input in Dynamo  
  
dataEnteringNode = IN[0]  
tree = etree.parse(dataEnteringNode) # Parse XML to Create Tree  
  
root = tree.getroot() # Get Root Structure
```

*DYNAMO VERSION: PASSING FILES IN AND CALLING ONE OF THE MODULES THAT WAS IMPORTED ABOVE*

The script then uses the parsed data file to traverse through the various pathways in the XML document and pulls the information we need in an organized pathway. So for example, if I wanted to pull the height of a window in the bedroom on the 2<sup>nd</sup> floor of a house, I would traverse the main node (the house) to the second floor, then search through the particular room with the window, and find the height of said window. It's this consistent and predictable organizational hierarchy that is integral to making this process work.



To collect the information, we use a python object called a dictionary. A dictionary is a Python object that is highly proficient at helping the user gather unordered information and organizing it so long as the data has a unique key associated with each object. Information is saved in a “key: value” pair. The value part of this “key: value” pair is not limited to just one string value and can be an embedded object, like another dictionary, a dictionary of dictionaries, or even a list object.

To give a quick example, let's say we have information on different objects in a room. In this example, let's suppose they are all located in a bedroom (called bedroom1). There are two windows and one door in the bedroom. Let's say we create a key to reference these specific objects by concatenating what room they are in and numbering the items.

Let's create a simple dictionary that lists the objects and their types:

objectDict1 - Dictionary (3 elements)

Key	Type	Size	Value
bedroom1_door1	str	1	door
bedroom1_window1	str	1	window
bedroom1_window2	str	1	window

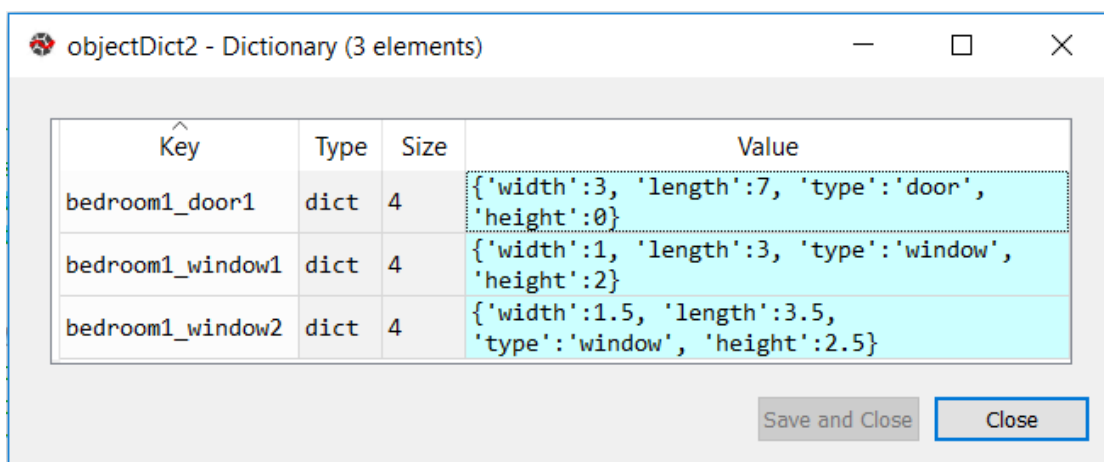
Save and Close Close

OBJECTDICT1

We create a key, which must be unique, for each object in the room, and for its value, we store what type of object it is.

We are also not limited to just storing a single string in the value field. As long as we have unique keys, we can organize and store the values associated with those keys in whatever manner we choose, including as dictionaries themselves.

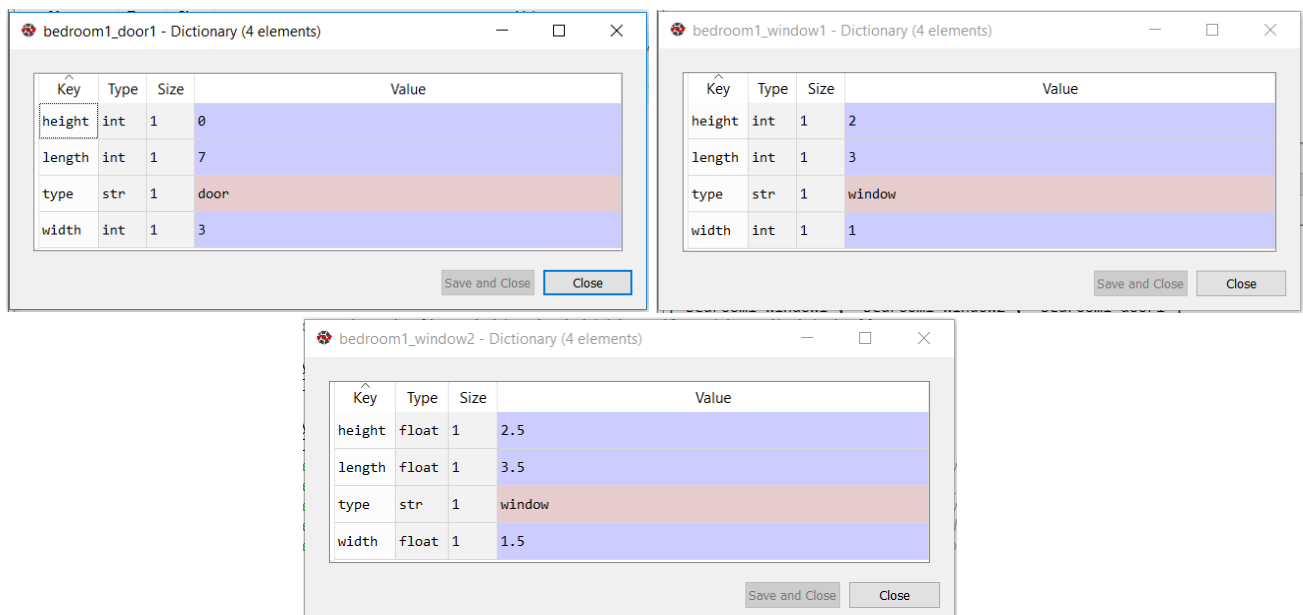
Let's say we have more attributes associated with each object than just the type of object it is. Let's say we also have width, length, and height attributes we want to store in the dictionary. To create the dictionary, we called it objectDict2 and entered the information:



Key	Type	Size	Value
bedroom1_door1	dict	4	{'width':3, 'length':7, 'type':'door', 'height':0}
bedroom1_window1	dict	4	{'width':1, 'length':3, 'type':'window', 'height':2}
bedroom1_window2	dict	4	{'width':1.5, 'length':3.5, 'type':'window', 'height':2.5}

VARIABLE EXPLORER VIEW OF OBJECTDICT2

We can zoom into what the value dictionary looks like for each row:



Key	Type	Size	Value
height	int	1	0
length	int	1	7
type	str	1	door
width	int	1	3

Key	Type	Size	Value
height	int	1	2
length	int	1	3
type	str	1	window
width	int	1	1

Key	Type	Size	Value
height	float	1	2.5
length	float	1	3.5
type	str	1	window
width	float	1	1.5

VARIABLE EXPLORER VIEW OF THE VALUE DICTIONARIES OF OBJECTDICT2



Here we see that for objectDict2, we have a top level dictionary, which contains information by each object in the bedroom. There are three objects, a door and two windows, so we have three rows. The key here is the objectID, and the value is a set of dictionary objects where the actual information pertaining to each object lives. For each object, we have 4 key:value pairs we want to store – the height, length, type, and width.

Getting information out of a dictionary is also relatively simple, and just requires a command:

```
In [5]: print(objectDict1.keys()) # This prints all keys in objectDict1
['bedroom1_window1', 'bedroom1_window2', 'bedroom1_door1']

In [6]: print(objectDict1.values()) # This prints all values in objectDict1
['window', 'window', 'door']

In [7]: print(objectDict2.keys()) # This prints all keys in objectDict2
['bedroom1_window1', 'bedroom1_window2', 'bedroom1_door1']

In [8]: print(objectDict2.values()) # This prints all values in objectDict2
[{'width': 1, 'length': 3, 'type': 'window', 'height': 2}, {'width': 1.5, 'length': 3.5, 'type': 'window', 'height': 2.5}, {'width': 3, 'length': 7, 'type': 'door', 'height': 0}]

In [9]: print(objectDict2['bedroom1_window1']) # This retrieves all values associated with
bedroom1_window1
{'width': 1, 'length': 3, 'type': 'window', 'height': 2}

In [10]: print(objectDict2['bedroom1_window1']['type']) # This retrieves what type of object
bedroom1_window1 is
window

In [11]: print(objectDict2['bedroom1_window1']['width']) # This retrieves the width of
bedroom1_window1
1

In [12]: print(objectDict2['bedroom1_window2']['length']) # This retrieves the length of
bedroom1_window2
3.5
```

We use the tree structure of the XML file and our knowledge of dictionaries to traverse the branches of the tree and collect the attributes we want into a well-organized dictionary. The above example is a much-simplified version of the data structure exported from MagicPlan, but the concepts are the same. In the end, we stored all the information that was collected into a dictionary object.

Once this information was collected, we then needed to further organize it into an output file that can be used in Dynamo as an OUT. When we coded this, we could not get Dynamo to use the dictionary object, so instead of outputting a dictionary, we outputted a list of lists instead by writing a function to translate all aspects of the dictionary of dictionaries to a list of lists:



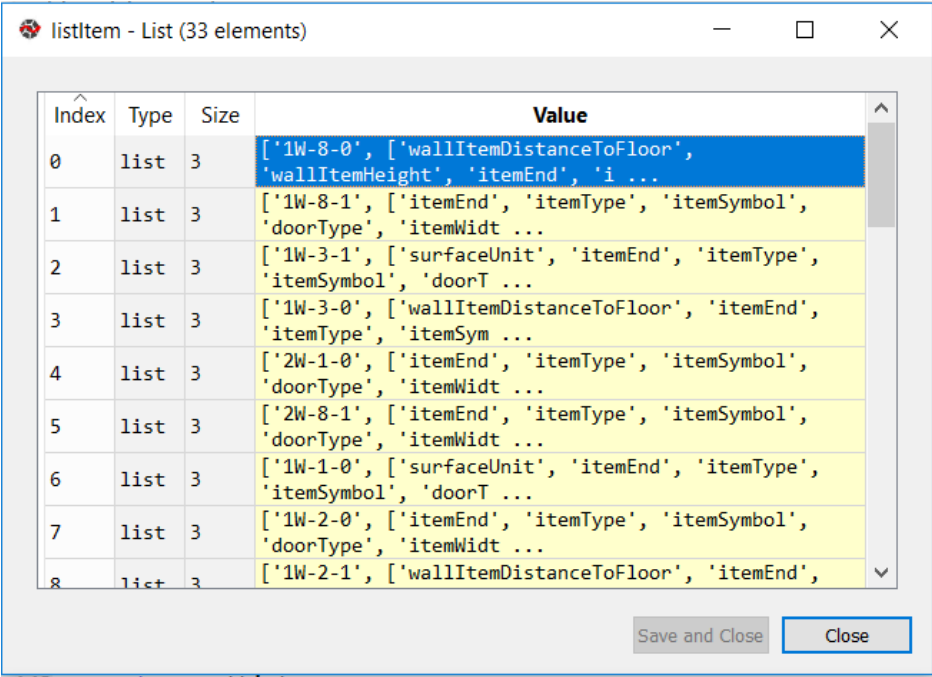
*# Translate dictionary of dictionaries to list of lists for Dynamo*

```
def dofDicts_to_lofLists(d):
    outList=[]
    for keyDict in d.keys():
        mainKey = keyDict
        mainVal = d[mainKey]
        keyList = []
        valueList = []
        for subKey in mainVal.keys():
            subValue = mainVal[subKey]
            keyList.append(subKey)
            valueList.append(subValue)
        outList.append([mainKey, keyList, valueList])
    return outList
```

*METHOD TO ORGANIZE OUTPUT FILE FROM DICTIONARY OF DICTIONARY TO A LIST OF LISTS*

All the scripts output a similar object, so we'll just go in depth into one example. Let us look at the node\_itemDict.py script. This is the custom node to grab the items (doors, windows, and furniture) and all their attributes. We will be using the sample file "Toms House-ver2.magicplan" for this example.

The final output is organized as a list of lists or lists, so the top level looks like:



Index	Type	Size	Value
0	list	3	['1W-8-0', ['wallItemDistanceToFloor', 'wallItemHeight', 'itemEnd', 'i ...
1	list	3	['1W-8-1', ['itemEnd', 'itemType', 'itemSymbol', 'doorType', 'itemWidth ...
2	list	3	['1W-3-1', ['surfaceUnit', 'itemEnd', 'itemType', 'itemSymbol', 'doorT ...
3	list	3	['1W-3-0', ['wallItemDistanceToFloor', 'itemEnd', 'itemType', 'itemSym ...
4	list	3	['2W-1-0', ['itemEnd', 'itemType', 'itemSymbol', 'doorType', 'itemWidth ...
5	list	3	['2W-8-1', ['itemEnd', 'itemType', 'itemSymbol', 'doorType', 'itemWidth ...
6	list	3	['1W-1-0', ['surfaceUnit', 'itemEnd', 'itemType', 'itemSymbol', 'doorT ...
7	list	3	['1W-2-0', ['itemEnd', 'itemType', 'itemSymbol', 'doorType', 'itemWidth ...
8	list	3	['1W-2-1', ['wallItemDistanceToFloor', 'itemEnd',

*LISTITEM DICTIONARY GENERATED BY PARSING TOMS HOUSE – VER2*

The first item of the list is associated with a window object with ID '1W-8-0'. When we drill down into this first item, we see that it is a list of 3 list elements. The first element is an itemID, which is unique to the item in question. The next element is a list of attribute names we have information for this particular item. The last element is a list of values associated with the list in index 1.

0 - List (3 elements)

Index	Type	Size	Value
0	str	1	1W-8-0
1	list	14	['wallItemDistanceToFloor', 'wallItemHeight', 'itemEnd', 'itemType', ' ...
2	list	14	['1.500000', '1.000000', ('1.969', '7.736'), 'window', 'windowfrench', ...

Save and Close Close

#### INFORMATION ABOUT ITEM 1W-8-0

So in other words, if you line up the two, they will form key, value pairs:

Index	Type	Size	Value
0	str	1	wallItemDistanceToFloor
1	str	1	wallItemHeight
2	str	1	itemEnd
3	str	1	itemType
4	str	1	itemSymbol
5	str	1	surfaceUnit
6	str	1	itemWidth
7	str	1	rollUnit
8	str	1	itemOrientation
9	str	1	windowType
10	str	1	distanceUnit
11	str	1	snappedHeight
12	str	1	id
13	str	1	itemStart

Save and Close Close

Index	Type	Size	Value
0	str	1	1.500000
1	str	1	1.000000
2	tuple	2	('1.969', '7.736')
3	str	1	window
4	str	1	windowfrench
5	str	1	m2
6	str	1	0.530
7	str	1	m
8	str	1	0
9	str	1	3
10	str	1	m
11	str	1	1.00000
12	str	1	8
13	tuple	2	('2.499', '7.736')

Save and Close Close

INDEX 1 AND 2 FORM THE KEY VALUE PAIRS OF ITEM 1W-8-0

We can see that item 1W-8-0 has an itemType of window, and a wallItemHeight of 1.

## Using Helper Functions

In Dynamo for Revit, this information may be hard to get at, so instead of using multiple nodes, we can instead, write more custom nodes in Python to make extracting this information easier.

*#This helper function takes an input of:*  
*#1. a list of lists of lists,*  
*#2. the index for the attribute name list (0, 1, 2,...etc.)*  
*#3. and the index for the value list (0, 1, 2, ...etc.)*  
*#4. a searchString for the attribute you want to grab*  
*#And it Returns a list of the values for the searchString attribute from the entire list*

```
def lookupIndexValFromString(inputList, attriIndex, valIndex, searchString):
    lookupIndex = []

    for l in inputList:
        search = l[attriIndex]
        index = search.index(searchString) if searchString in search else -1
        lookupIndex.append(index)

    resultIndex = []

    for i, l in enumerate(inputList):
        result = l[valIndex][lookupIndex[i]] if (lookupIndex[i] >= 0) else ''
        resultIndex.append(result)

    return resultIndex
```

*A HELPER FUNCTION DEVELOPED TO EXTRACT DATA MORE CLEANLY, RATHER THAN USING MANY STANDARD  
 NODES IN DYNAMO*

Based on the example organized above, say we wanted to grab all available information on wallItemDistanceToFloor for all items in our item list, we use the helper function in a custom Python node, bring in the output from the item dictionary custom node, and call:

```
lookupIndexValFromString(IN, 1, 2, 'wallItemDistanceToFloor')
```

Which returns:

```
['1.500000', '', '0.800000', '', '', '1.0', '0.500000', '1.500000', '2.000000', '2.000000', '1.0', '2.200000', '1.0', '0.800000', '1.0', '2.200000']
```

Which is a list of all 'wallItemDistanceToFloor' values for the items in our item dictionary.

## Learn how to use the parsed data to generate model elements in Revit

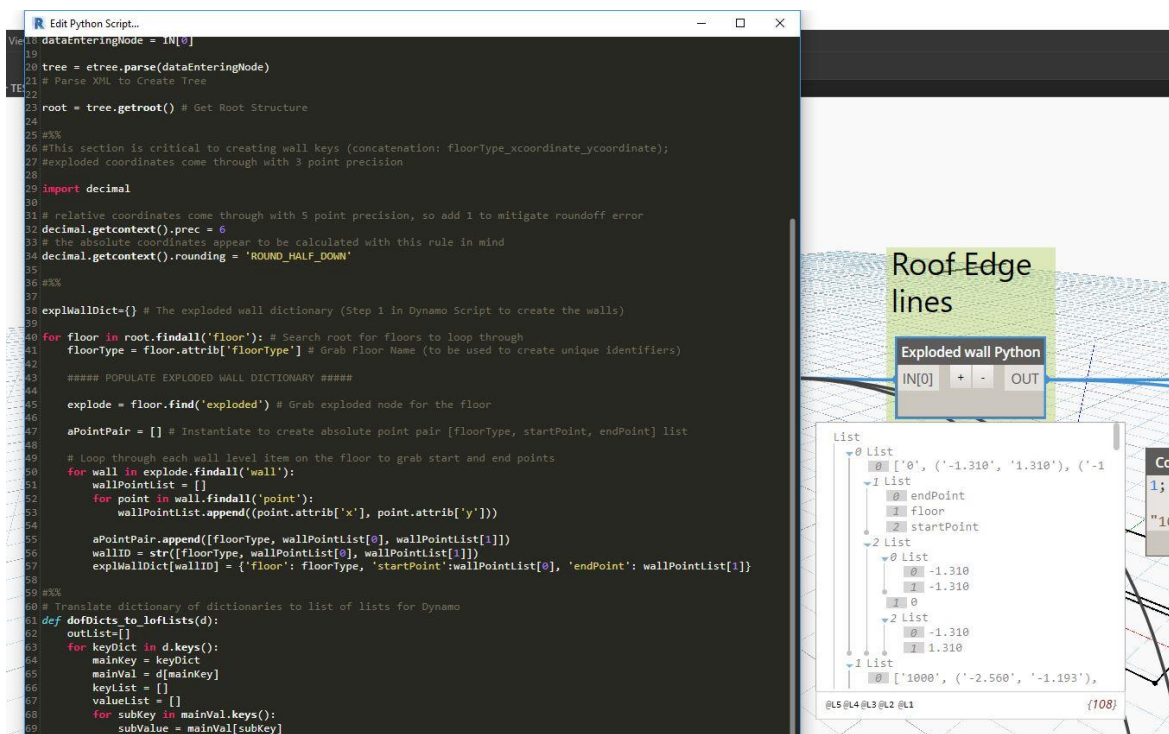
Ultimately, our goal in using Python to parse the data is to translate it into the lists that we are familiar with using in Dynamo. In our case, all MagicPlan data comes packaged in one large XML file. As discussed in the previous section, our first step is to break this file down and extract the individual pieces we will need to generate model geometry.

Revit model elements will require two principal categories of information in order to be inserted via dynamo. First, this will be its location geometry. For walls, beams, and other linear objects or system families, this will be represented as a line. For doors, windows and other insertable families this will be an insertion point.

Second, Revit will need all parameter values associated with those model elements. For instance, to generate a wall, it will need the Wall type, Height, Associated Level, and other parameters that we typically specify when modeling directly in Revit.

In looking closely at the XML data, we see that it is translated into a series of lines and points. Much in the same way that Revit represents the base parametric elements, MagicPlan represents a wall with a single centerline, and packages the start point and endpoint coordinates with that line. As mentioned previously, by working with the data's hierarchy in the survey itself, we were able to identify which lines would represent Interior walls, Exterior walls, Roof outlines, etc. These coordinate points are what we extract as a basis for our model generation.

Also, in MagicPlan, we have included information like wall types, heights, levels, etc. in our MagicPlan Survey, so that that data comes in the parsed XML dictionaries. This is recorded at the same time that we are creating the floorplan in the field.



Dynamo allows Python to be written directly into custom nodes, so we can build the same dictionary filtering techniques directly into the dynamo workflow itself, as shown in the image above. To bring the data into the workflow, we use a simple file path node to locate the XML file that we have saved from MagicPlan. Then, we can create a series of different Python script nodes that pull only the chunks of information that we will need for each part of the model's geometry. The challenge is to take that flat, 2D textual info and turn it into 3D model elements.

In the example below, we have broken that information down into three separate Python node. The first uses the wall geometries from the "Floor" level of MagicPlan. The second digs down one level in the hierarchy to the "Room" level, which contains slightly different, more specific information. The third digs one level further, to the "Element" level, allowing us to grab information on specific elements, such as an element that we created in the survey to give us data pertaining to the Roof Ridge.

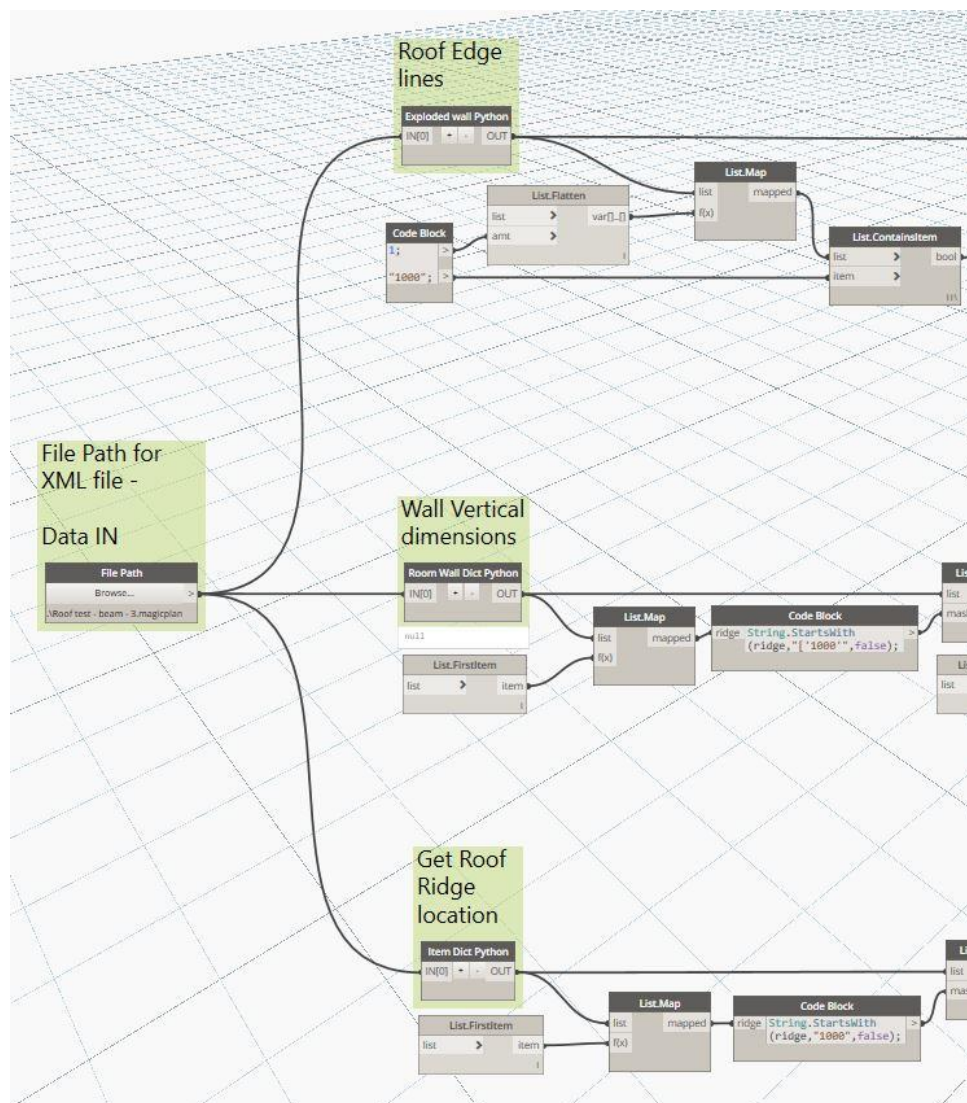


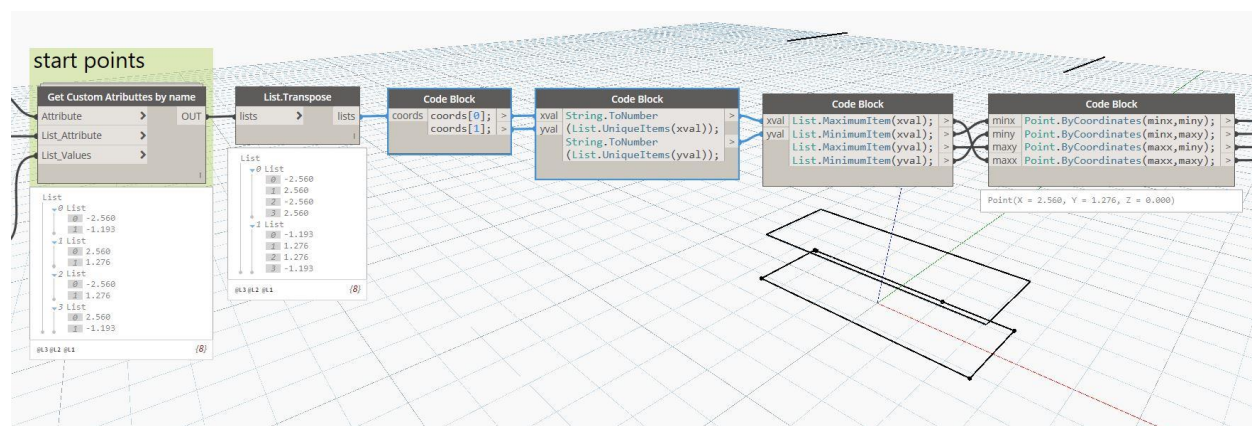
IMAGE 1: SEPARATING THE XML DATA VIA PYTHON NODES





Next, if we take a closer look at those lists, we see that they each contain 3 items: One is the identifier of the element, the second is the Attribute name, and the third being the Value of that attribute. Those lists contain the start point and endpoint of each side of the roof. We are interested in the Attributes and their Values, so we can discard the first item. The List.RestOfItems Node put into the list map will give us the second and third item of each list.

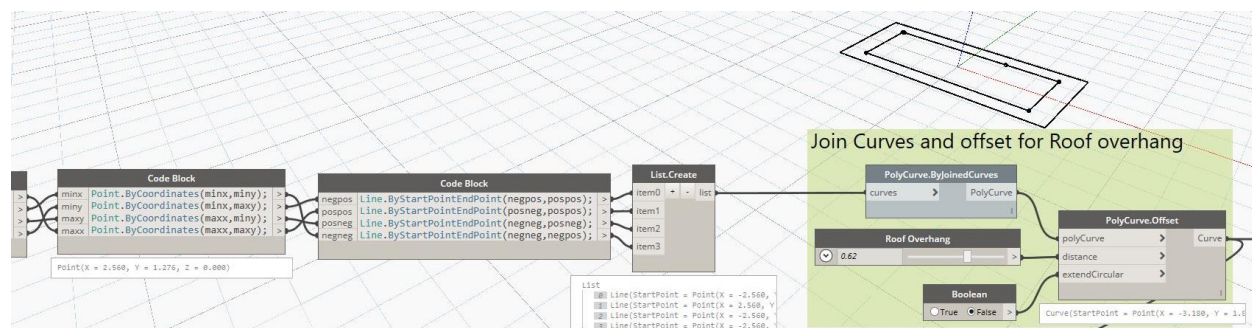
At the end of step 2, we have created a custom node called "Get Custom Attributes by Name", which allows us to Specify the name of the Attribute we want to see, and it returns the value associated with it. We want the "startPoint" of each line, so that we can identify the four corners of the roof (we can discard the endPoints since they overlap at each corner with the start points) This returns 4 separate lists of x,y coordinates.



STEP 3: USE COORDINATES TO CREATE POINTS, THEN LINES FROM THOSE POINTS

In step 3, we can use the List.Transpose node to give us two lists, one of all x-values and another of all y-values. Next, since we have imported this from XML data, and the Dynamo nodes will need a number data type, we have to convert the two lists into Numbers using the String.ToNumber node.

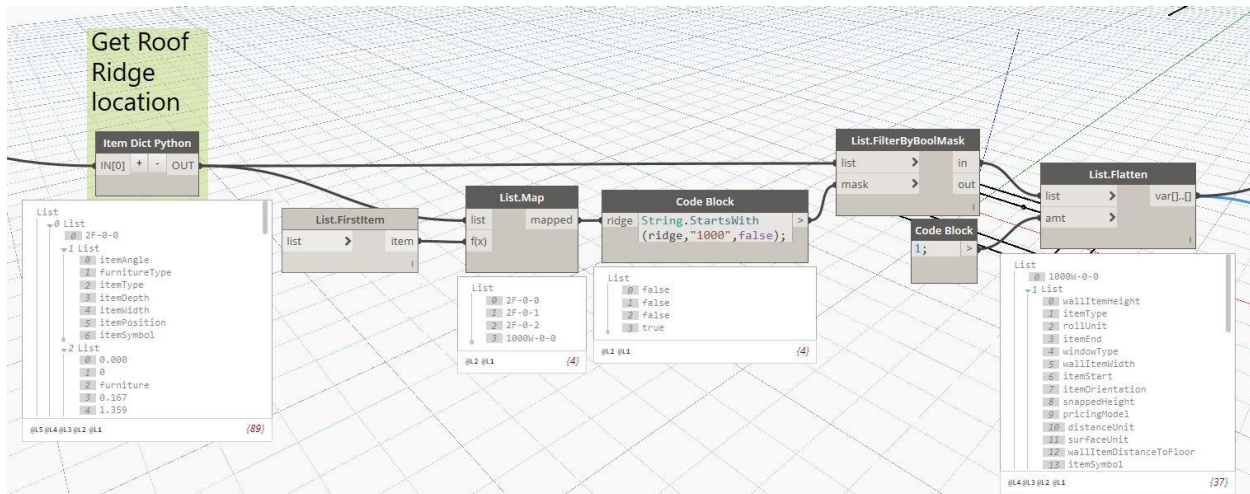
By Using the List.MaximumItem node, we can find the extreme x and y values. By plugging those values into different combinations of maximum and minimum x and y values, we arrive to the 4 corner points of our roof. After creating 4 lines using a similar combination of our roof, we can join those lines into a polycurve so that we can offset a specified distance for roof overhang.



STEP 4: OFFSET LINES TO GET FINAL OUTLINE OF ROOF, INCLUDING OVERHANG FROM ORIGINAL WALL LOCATION

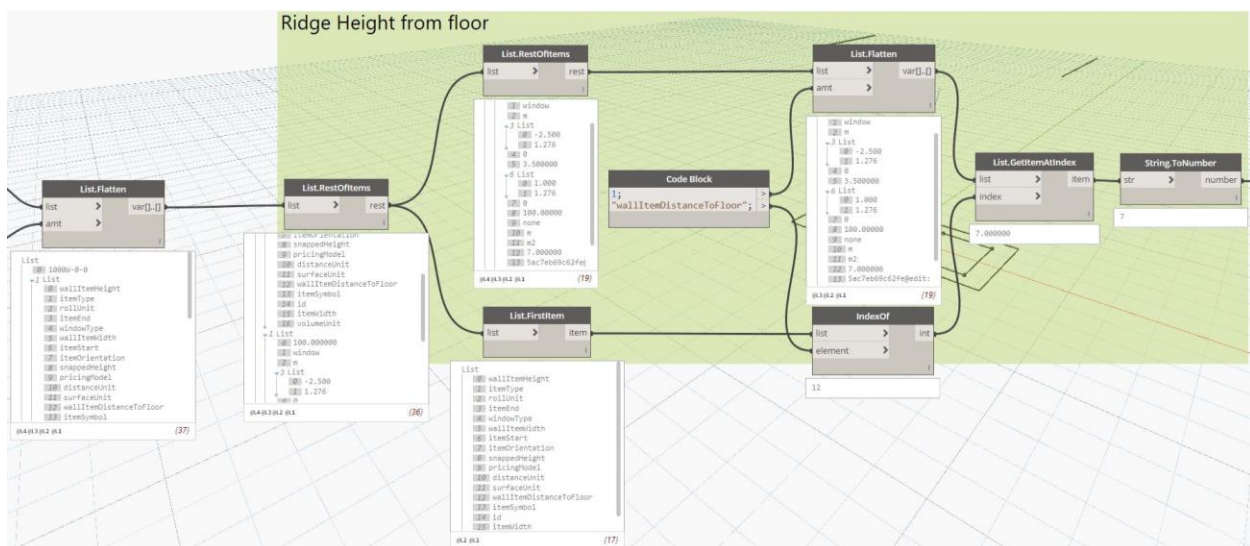


Now that we have the roof's outer edge defined, we still need the location and vertical information of the roof's ridge. In Magicplan, we had used a separate element to define that information, which comes in a different level of the hierarchy. Using the "Item Dictionary", or as we referred to it earlier, the "Element Level", we can pull that as separate information. Again, we need to filter the information to look for level 1000.



STEP 5: FILTER ELEMENT INFORMATION TO FIND ROOF RIDGE HEIGHT

Once we've isolated that list, we can isolate the single item that we are looking for, which is the Ridge height. In defining parameters ahead of time, in MagicPlan, we called this "wallItemDistanceToFloor". Similar to the list above, we have two items: 1. Attributes, and 2. Their Values. We know that the first list corresponds with the second, so we first need to look for the Index of the Attribute name in the first list. The IndexOf node will tell us that the item we want is at Index 12. With the List.GetItemAtIndex node, we find that item 12 is the distance of 7.00m.



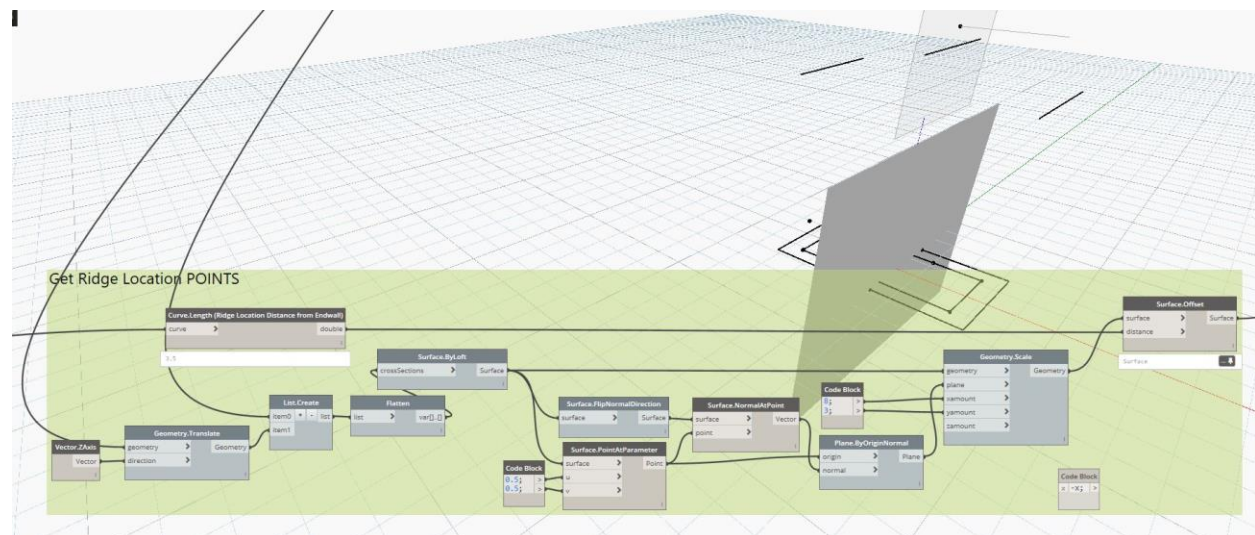
STEP 6: GET INDEX OF ROOF RIDGE HEIGHT TO FIND CORRESPONDING VALUE

Moving forward, there are a number of different strategies that could be used to build out the rest of the geometry. In our case, we were looking for something flexible, that could achieve a range of different roof geometries, instead of, for instance, always putting the ridge at the midpoint of the roof. We typically find that we're not dealing with perfect gabled roofs.

For that, we run through a series of nodes to manipulate the geometry that we've already created. In this example, we had extracted distance between the ridge and the roof's edge in the same way we extracted the information in the cases above. What we need is a way to copy one of the roof's edge line towards the center of the building to act as the ridgeline.

To achieve this, we had already pre-specified in our survey which line we would identify as the starting line for the distance to roof's ridge. We took that line, copied it up, and lofted a surface between the two. That gives us a basis for finding a normal of the surface that will always point to the center of the building, preventing copying our edge line in the wrong direction. (by using surface normals instead of specifying "translate in x-axis", we can accommodate a wider range of input types. Someday we might want to translate in a negative x-value, and by using normal direction, we will never have to manually specify that)

This allows us to create a plane that will be used to intersect our roof outline, and by using the Geometry.Intersect node, create two points. We can then take those two points and connect them to create our ridge location.



*STEP 7: CREATE SURFACE, TRANSLATE, AND FIND INTERSECTION POINTS TO DEFINE RIDGE LOCATION LINE*

Finally, we take our three lines and translate them to the proper location in the Z-axis, to represent the two low ends of the roof, and the ridge. That vertical information was extracted from the XML data in the center workflow shown in Image 1. These three lines will serve as the basis for the remaining roof framing divisions, and ultimately the centerlines of our roof framing members.

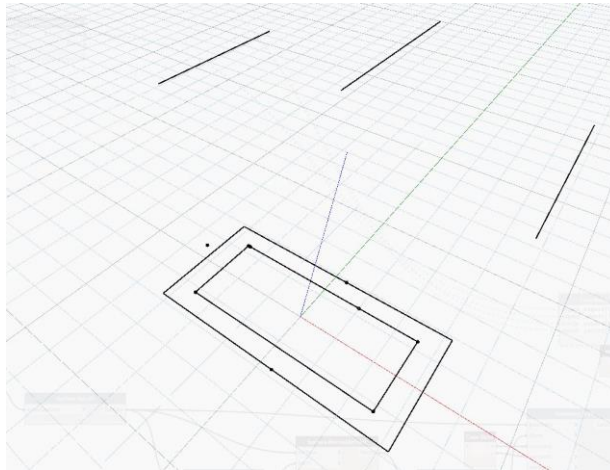
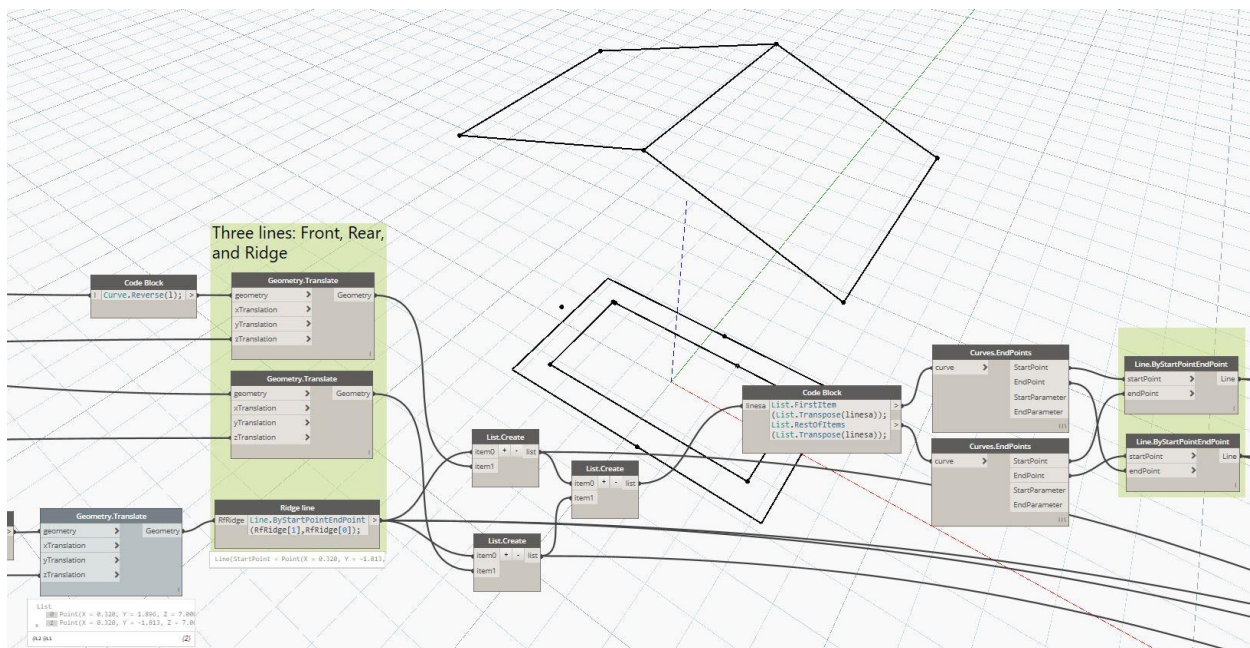


IMAGE 2: TWO LOW EDGES AND HIGH RIDGELINE FOR FINAL ROOF GEOMETRY

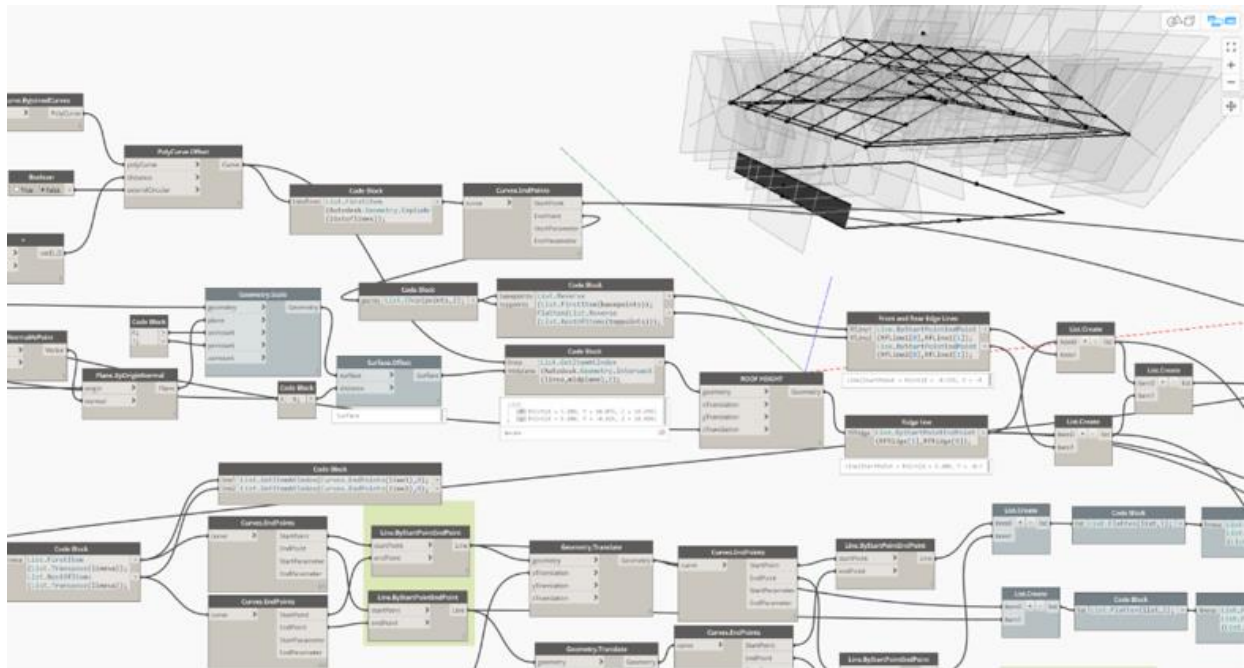
By finding the startpoints and endpoints of those 3 lines, we can connect them for a final 3D outline.



From there, the remaining roof framing depends on the type of construction. In our case, in Colombia and the Philippines, we typically find a simple system of rafters and purlins, with a sheet of corrugated metal on top.

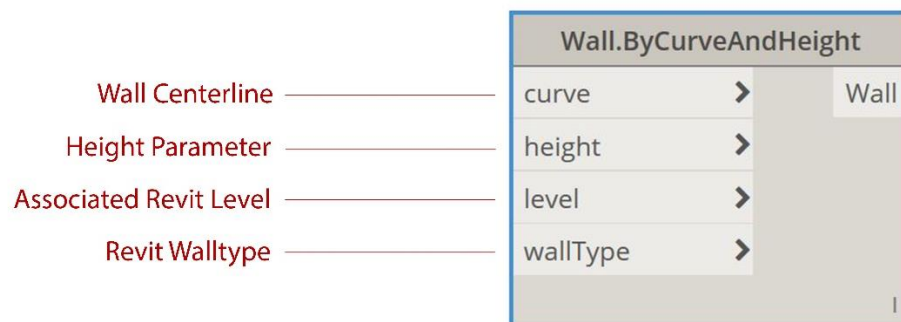
With that in mind, we run through a series of nodes that will find the lines and subdivide them a specified amount. The Curves,DivideCurve node (found in the Lunchbox Package) will allow us to create an even distribution of points along these outlines, and connect them using the Line.ByStartPointEndPoint node, finalizing the roof's framing locations.





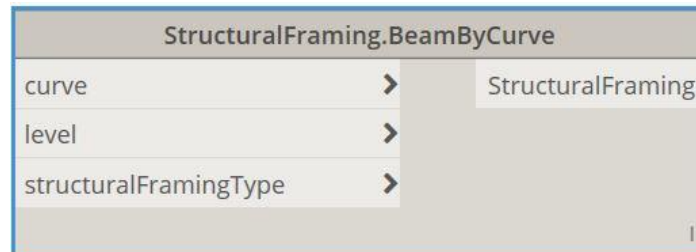
For final creation of model elements, we have to turn back to the parameters themselves. As previously mentioned, we are collecting 2 categories of information, as they pertain to Revit: 1. Location, and 2. All other parametric values.

By separating the initial data into separate lists, we can manipulate model elements individually, one by one. We can create parallel streams of Dynamo nodes that will produce different model categories, such as the centerlines of each wall, for example, and then plug that geometry into the Wall.ByCurveAndHeight node to insert the elements into the Revit model. In our case, the height parameter and Wall type are other pieces of information that we have included in our survey of the house, and is recorded at the same time that we are creating the floorplan in the field.



Going back to the Roof as another example, after subdividing the outlines of the roof, we are able to generate a series of centerlines that will act as the location line for structural framing members. We can plug these into the StructuralFraming.BeamByCurve node, which will

automatically place those items into the Revit model. We will simply need to ensure that we are using a structural framing Family that has already been loaded into our Revit Model template.

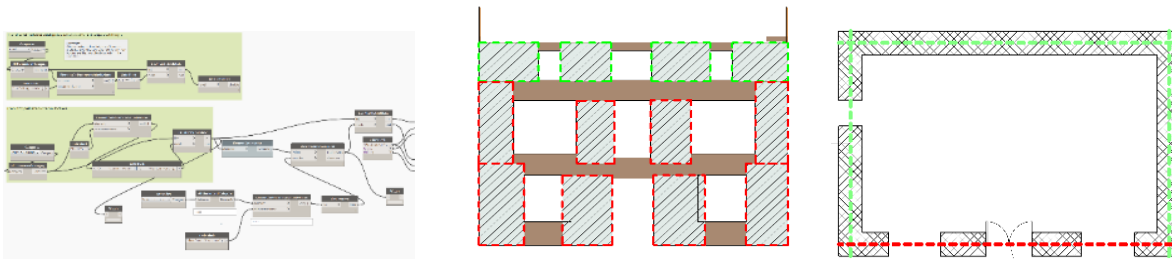


## Learn how to automate structural engineering analyses and the production of common Revit drawing

The final part of our automated process is to run structural check to ensure that our Retrofit design passes our structural Rule Checks. We have boiled our Structural design down to what we call a pre-engineered “type design”, and we check to ensure that our model complies with this.

Once the retrofit components are placed in the model, and the solution is finished, Dynamo automates the analysis of the model, and the retrofit components will turn either red or green depending on whether or not they pass the rules, as defined by our customizable type design. This warns the user that a change needs to be made. Once the retrofit design is adjusted to meet the criteria, the checks can be re-run, until all systems pass.

This is done by creating a number of shared parameters that have a yes/no value in the Revit Model itself. In the Visibility/Graphics window in Revit, we can create conditional color filters that turn red if the value of that parameter is no, or green if the value is yes. We then use dynamo to simply set the value of the parameter, by giving a “true” or “false” value to the Element.SetParameterByName node.



*Dynamo being used to automate the engineering rule checks*

## Example Check

1&2. Maximum / Minimum Wall Height Check:

Are all wall heights  $> 2.45\text{m}$  and  $< 2.75\text{m}$ ?

