

BLD221667

Creating Automated Documentation with Dynamo

Drew Jarvis

SolidCAD

Learning Objectives

- Learn how to automate the creation of View and Sheets
- Learn how to create 2D schematics from 3D model geometry and associated data
- Learn how to take your ideas for automation and put them into effect using Dynamo
- Get more comfortable using Python in Dynamo

Description

Dynamo can help you save time and money by streamlining the laborious tasks involved in documentation set creation. During this session, we will look at how you can create custom graphical column schedules, electrical fire-alarm schematics, and custom area schedules—all from model geometry. We will also investigate view and sheet creation graphs that will ensure that you and your company can concentrate on creating the best design without having to sweat it on the documentation of that design.

Speaker

An innovative and motivated professional with over 18 years' experience in CAD Management, design, and training. Extensive experience in the theoretical and practical implementation of BIM both from a strategic and tactical level. With a strong passion for the AEC industry and a solid business sense and entrepreneurial background I am always looking to see how emerging technologies can help companies remain competitive while advancing their practices.

Overview

This handout goes into detail of the graphs covered in the presentation, please attend or watch a recording of the presentation to get an overview of the graphs and their uses prior to diving into the detailed explanation of their structure in this document. You can also download the graphs to investigate them hand on.

Note, all graphs in this handout are created using version 2.0.1 of Dynamo

Contents

Creating Automated Documentation with Dynamo	1
Learning Objectives	1
Description	1
Speaker	1
Overview	2
Creating Views and Sheets	3
Creating the Discipline and Drawing Name's	3
Constructing the Discipline and Building Number Combination	4
Combine the Discipline/Building Number with the Levels	5
Create the Views.....	6
The Titleblocks.....	6
Create the Sheets	7
Fire Alarm Schematic	8
PART 1 - Resetting the values	8
PART 2 - Assigning the values to X & Y parameters.....	9
Assigning by Panel	9
Assigning by Circuit and Level	10
PART 3 - Creating the Drafting View Single Line Elements.....	10
Graphical Column Schedule	14
Create the Level Annotations	14
Create the Wall and Column Annotations.....	17
Area Schedule	19
Gather the model elements	19
Grouping the elements.....	20
Extraction of a parameter value and Storage in a Dictionary	20
Chopping the lists	21
Exporting to Excel	23
Python	25
Reading other peoples code is a great way to learn	25

Creating Views and Sheets

Revit project files consist of many views and sheets, these views and sheets require specific naming conventions, which leads very handily to dynamo automation. I have created many automations for the creation of View and Sheets in the past, these may include setting View Templates, View Types, Names and Numbers along with Scope Boxes and more.

As always, please read this handout in conjunction with the actual Dynamo Graph and the follow along video posted with the class on the Autodesk University 2018 class site.

In this example we are going to use the following Naming convention:

AA011-010 – Cover Sheet

Where

AA = Discipline/Sub Discipline

01 = Building Number

1 = Level

010 = Unique Number

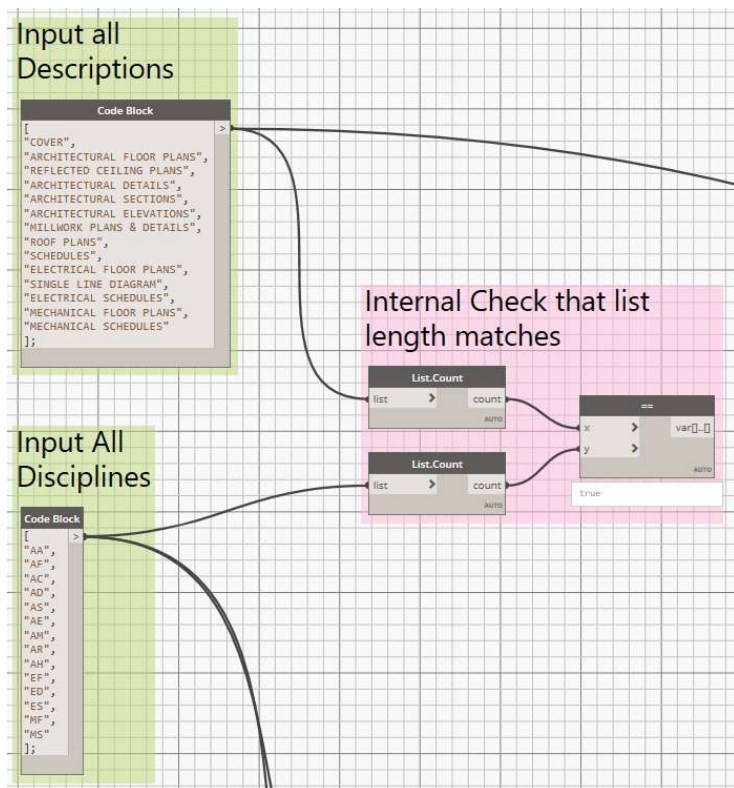
Cover Sheet = Drawing Name

Creating the Discipline and Drawing Name's

We are going to create one view per discipline per level per building in the example graph, so we need to define the names of the views along with the disciplines. This one is very simple, we are simply going to create a list of the values required using code blocks, I have put a very simple equals Node that verifies that the number of Disciplines is equal to the number of Drawing Names

This provides a value of true or false, true indicates that the number of values in each list are equal, if false appears then something needs to be checked.

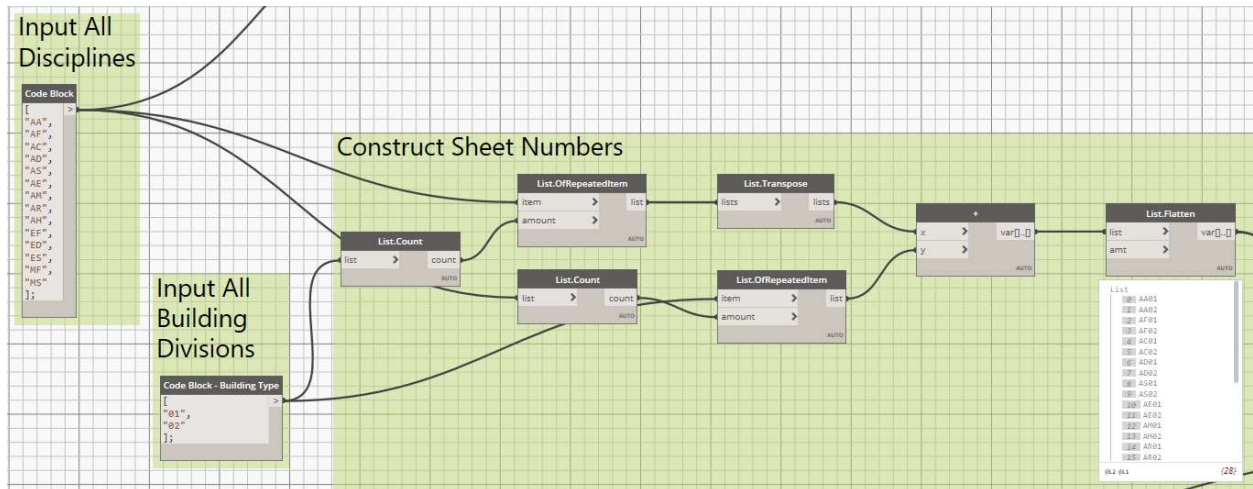
List creation with code blocks require that you put each value between a comma, with square brackets surrounding the list of elements for example:



["a","b","c"]

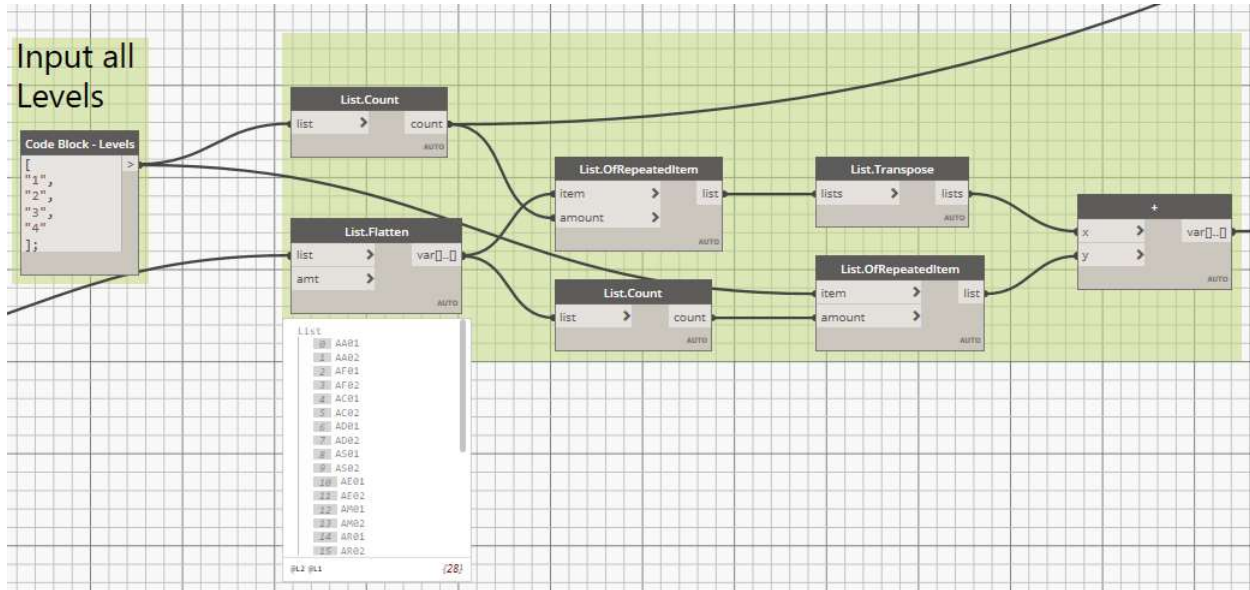
[1,2,3]

Constructing the Discipline and Building Number Combination

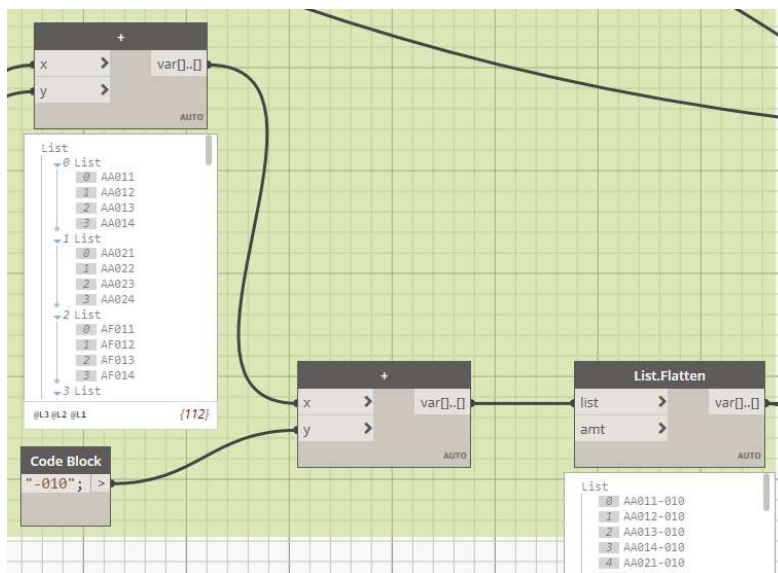


The name of the sheet starts with the Discipline, followed by the Building Number, so if we have AA and AB for discipline and we have 2 buildings, 01 and 02 then we end up with 4 combinations, AA01, AA02, AB01 & AB02. The easiest way to do this is to count each of the lists then repeat the list by the count of the other list. This does result in lists that are not aligned with each other, therefore a Transpose Node comes in handy to align them. The addition Node is then used to combine them, which is like a string concat but just a little bit simpler. Finally, a Flatten Node is used to get a single list rather than a list of sub lists.

Combine the Discipline/Building Number with the Levels



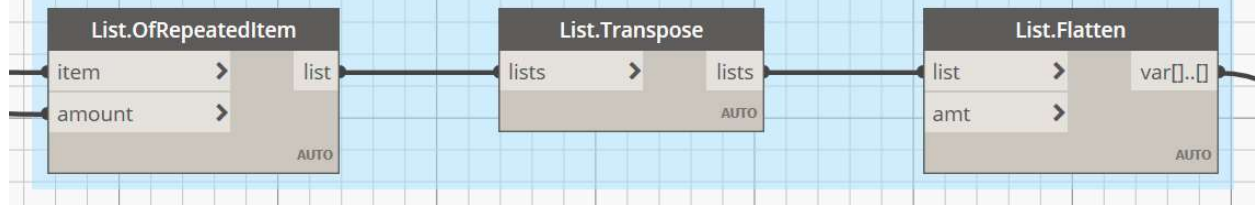
This process is very similar to the above, we are getting the count for the 2 lists and then repeating the lists by the length of the other list.



Next up we add the 010 to each value we have constructed. This value of 010 is basically a placeholder and if we make additional views for that subdiscipline on that level of that building then we can add more values like 011 or 012.

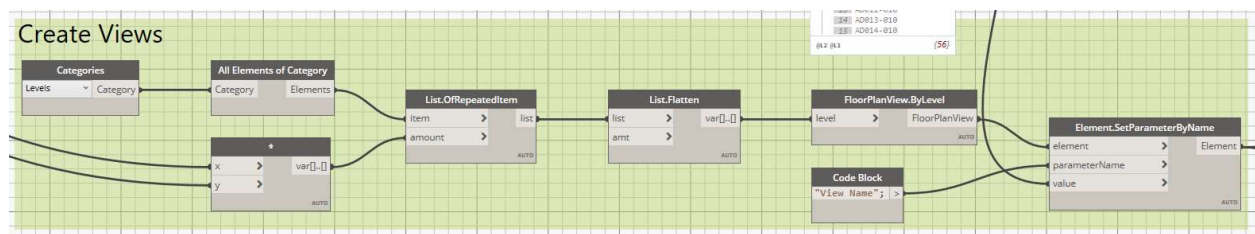
Before we create the Sheets, we need to create a list of the Drawing Names, we are not going to include the Level in the drawing name, although it would be easy to do (just trying to keep the example graph somewhat simple)

Construct list of Sheet Names



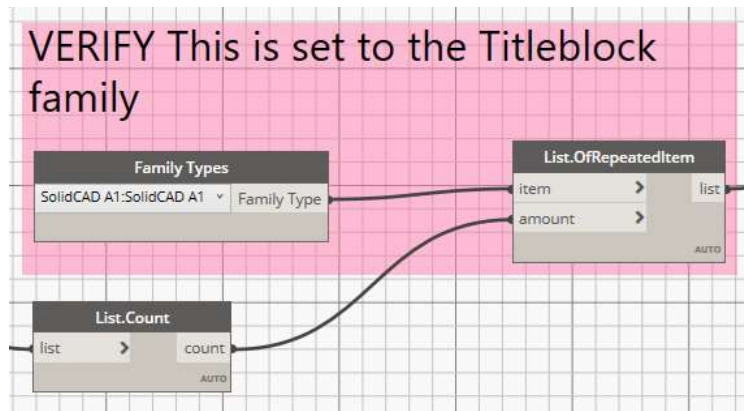
Create the Views

We will use the Sheet Numbers as the Names of the Views, then place each view on each sheet. To create the View we need to provide a Level, so as we need multiple views per level to correspond with the multiple Disciplines and Building Numbers we simply find the Levels with a All Elements of Category Node which we then repeat by the number of Discipline and Building Numbers combinations



The Titleblocks

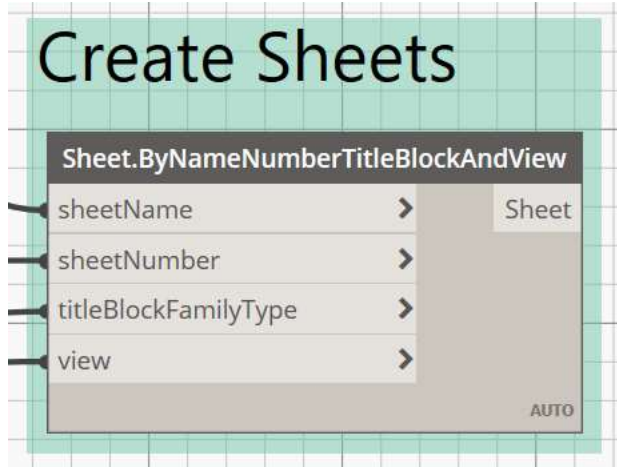
We are going to use the same titleblock on every sheet, so we just need a count of the sheets and an instance of the TitleBlock which we can find from the Family Types Node.



Create the Sheets

There are a couple of nodes for creating sheets in Dynamo, for this example I have decided to use the one where you specify the Name, Number, Titleblock and view to be placed on the Sheet. I have run the example with the views being the same name as the sheet for simplicity, this may or may not be possible for you but you will just have to create a list of views to be placed.

A work around if you want a Sheet with no views is to create a blank legend and have this placed on the Sheets, for whatever reason this results in the sheet being created with no view being placed.



Fire Alarm Schematic

The fire alarm schematic graph intends to take a model and identify the elements that need to be represented in a schematic as 2D elements. The process I have used involved storing the position of the 2D element on the schematic on the 3D elements. The location is defined in an additive way, ie it takes the current value on the 3D element and add's to it, this allows me to add values based on the Level, Panel and Circuit. The downside of this is that you need to start with clean values of zero, otherwise it all goes to pot!

Therefore, I have had to split it up into 3 segments, one to reset the values on the model elements, the next to calculate the values on the model elements and the final one to use these values on the model elements to insert the detail components into the drafting view.

As always, please read this handout in conjunction with the actual Dynamo Graph and the follow along video posted with the class on the Autodesk University 2018 class site.

Revit has a complete model of the Systems within the building, however these are the actual model representation of the elements which are very useful for Floor Plan documents, but the single line documents are a large part of any drawing set and these are largely ignored by Revit.

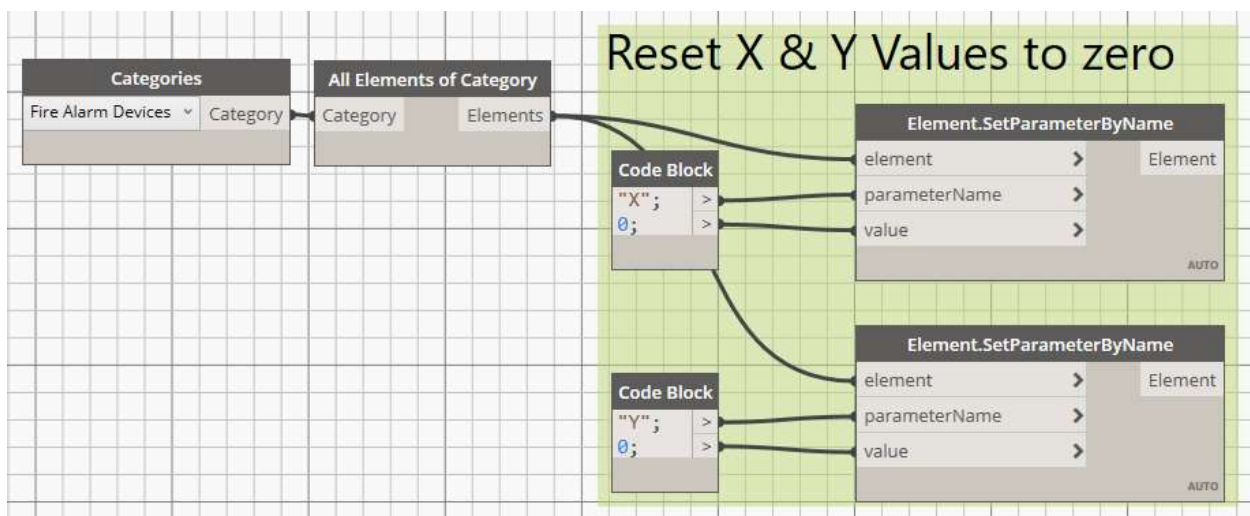
However, the logical connections of these elements are stored within the model, all we need to do is provide some framework for getting them into a view.

A drafting view is a great way to represent these objects, however the placement of these in a manual method is error prone and time consuming, so what about using Dynamo to automate this production? Even if we can get it to 90% that's got to be better than manually producing these things right?

This one is going to be a big one...

PART 1 - Resetting the values

First of all, we need to reset the values on the model elements, this is because in the 2nd stage we are going to be updating the values using a lot of addition, so if the values are not zero to start then we are in trouble.



The above simply finds all Fire Alarm Devices and uses SetParameterByName to set the parameters named X & Y to zero.

PART 2 - Assigning the values to X & Y parameters

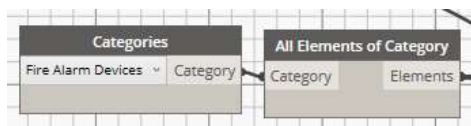
This is the bulk of the workflow. Here we are going to find the Fire Alarm Device elements in the project, then sort them by the Panel they are attached to and the Circuit on that Panel along with the Level they are on. We can then assign X & Y values to the Fire Alarm devices that will be the insertion points for the Detail Components within the Drafting view.

In the code we will Group the Levels vertically at intervals of 100mm, group the Panels Horizontally at intervals of 500mm, group the Circuits Vertically at intervals of 20mm and space the individual elements within each circuit at intervals of 20mm.

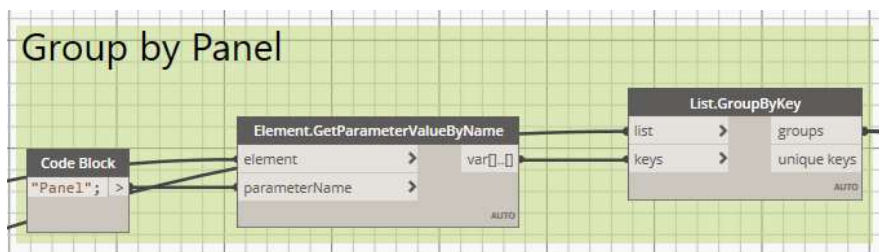
By assigning these values in a cumulative way we can ensure that the Detail Components will be placed in their own areas of the Drafting View, however as we are adding values to parameters in multiple places in the graph we have to ensure that we start at a value of zero for all X & Y parameters (Part 1).

Assigning by Panel

We start by assembling all of the Fire Alarm Devices in the model so that we can start to assign values to the X & Y parameters using a Categories and All Elements of Category combo



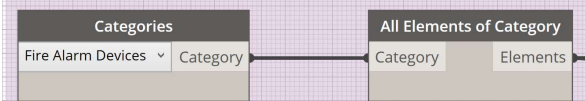
Next we Group the elements by their Panel parameter, this is a built in parameter for Fire Alarm Devices, we use a List.GroupByKey to ensure that all of the elements on each panel are group together.



Now we can setup the values to assign to the X value, essentially for each panel we want to increment the value by 500, so we need to know how many elements are on each panel and then create a list of values of the correct length

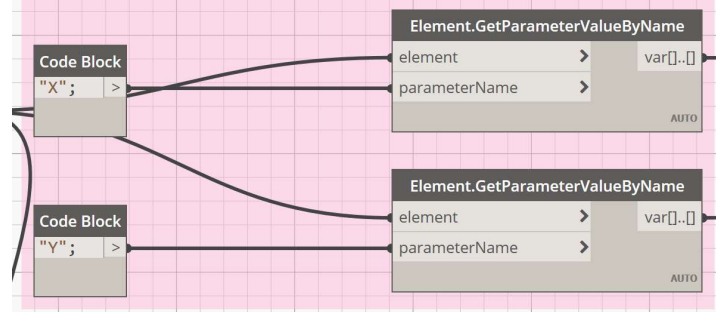
First, we collect all of the Fire alarm devices with a Categories and an All Elements of Category Node (a personal favorite combination)

Select All the Fire Alarm Devices



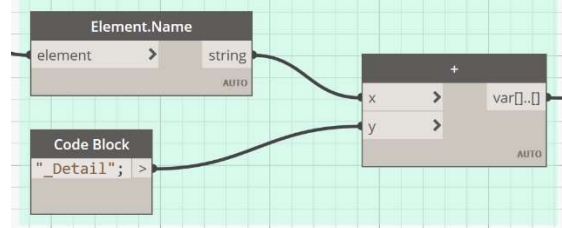
Next we read the custom parameters X & Y using the GetParameterValueByName Node

Get the assigned X & Y values



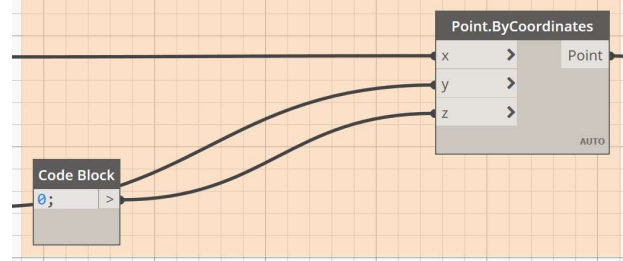
At the same time, we are getting the name of the Detail Component family to be placed by extracting the model family name and adding "_Detail" to it... now obviously this only works if you have detail components with the same naming convention as the model families (with the extension of _Detail) we use a simple + Node for this.

Add "_Detail" to the Family Type Name



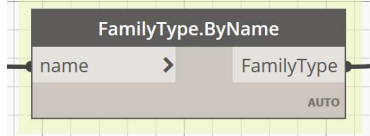
Now we take the X & Y from above and combine it with a Z value of zero as it's a drafting view.

Set the Insertion Coordinates



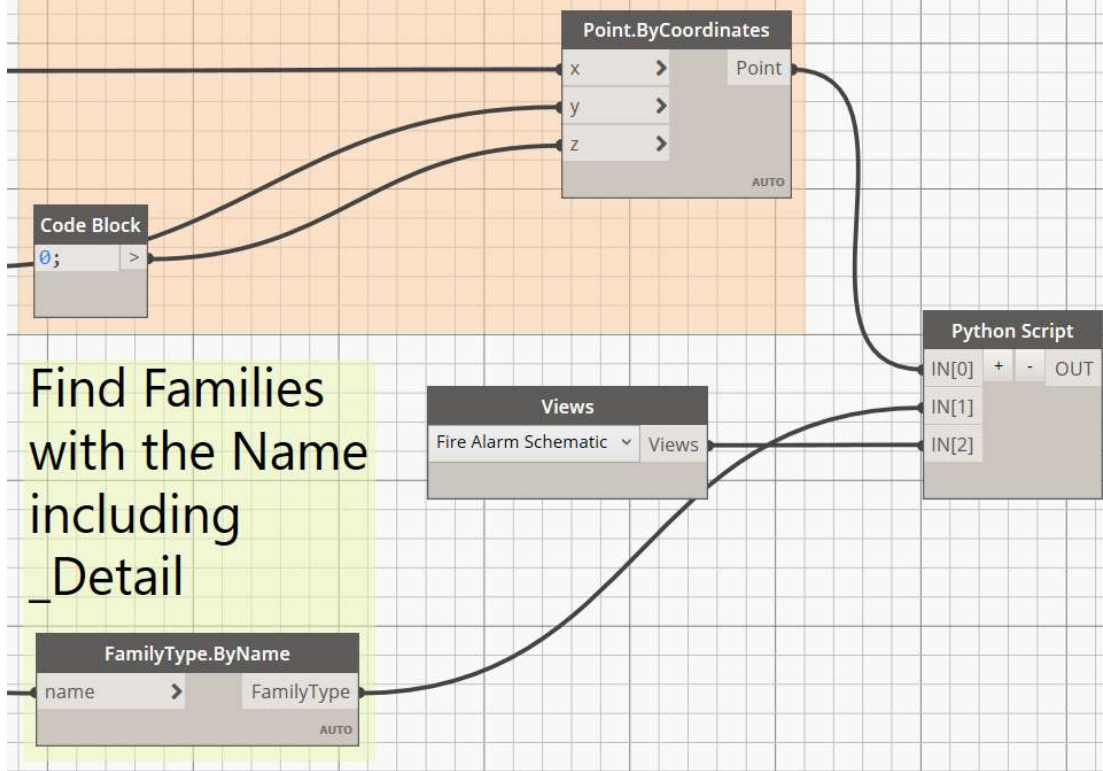
We find the actual Detail Component Families from the Names we created

Find Families
with the Name
including
_Detail



Finally, we bring this all together and use some Python Script to place the Detail Components onto the Fire Alarm Schematic Drafting View that needs to exist in the model...

Set the Insertion Coordinates



Here is the Python code in beautiful red, white, black and yellow (and a little violet)


```
Python Script
1 import clr
2 clr.AddReference('RevitAPI')
3 from Autodesk.Revit.DB import *
4
5 clr.AddReference("RevitNodes")
6 import Revit
7 clr.ImportExtensions(Revit.Elements)
8 clr.ImportExtensions(Revit.GeometryConversion)
9
10 clr.AddReference("RevitServices")
11 import RevitServices
12 from RevitServices.Persistence import DocumentManager
13 from RevitServices.Transactions import TransactionManager
14
15 doc = DocumentManager.Instance.CurrentDBDocument
16 points = UnwrapElement(IN[0])
17 famtype = UnwrapElement(IN[1])
18 lvl = UnwrapElement(IN[2])
19 elementlist = list()
20 counter = 0
21
22 TransactionManager.Instance.EnsureInTransaction(doc)
23 # make sure family symbol is active
24 for dc in famtype:
25     if dc.IsActive == False:
26         dc.Activate()
27         doc.Regenerate()
28 for point in points:
29     newobj = doc.Create.NewFamilyInstance(point.ToXYZ(), famtype
30     [counter], lvl)
31     elementlist.append(newobj.ToDSType(False))
32     counter = counter + 1
33 TransactionManager.Instance.TransactionTaskDone()
34 OUT = elementlist
```

Run Save Changes Revert

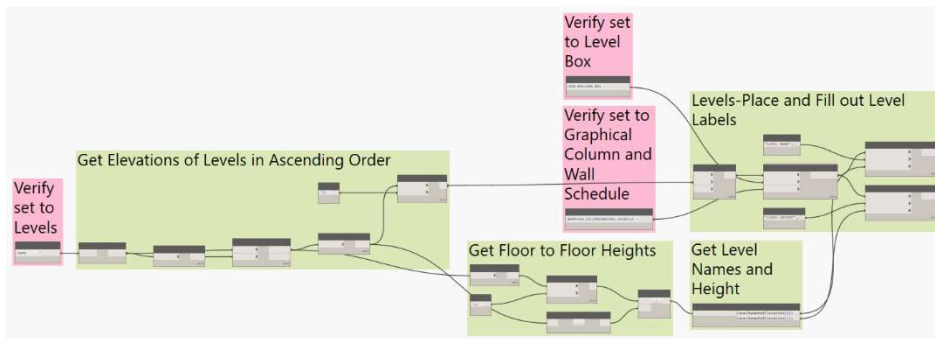
Graphical Column Schedule

This graph came about from the limitations of the Graphical Column Schedule built into Revit, I had a requirement that structural walls be included in the graphical schedule and it's often the case that people are looking for more control over the representation of their data than is available in the OOTB Graphical Column Schedule.

As always, please read this handout in conjunction with the actual Dynamo Graph and the follow along video posted with the class on the Autodesk University 2018 class site.

Create the Level Annotations

The annotation to the right shows the Levels in the project including the Name and the Height, these are constructed from Detail Items Named LEVEL BOX

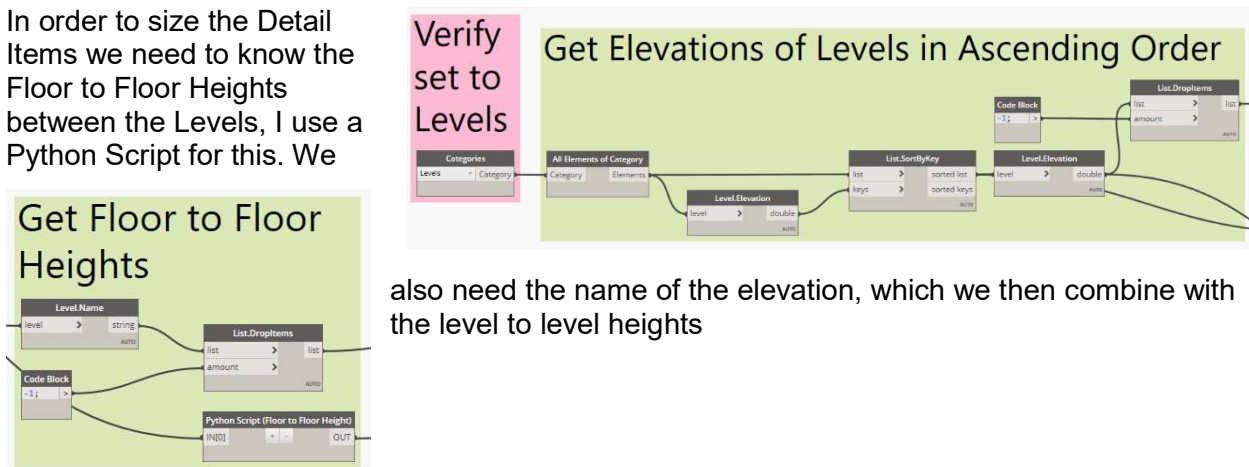


Level 3	3700
Level 2	1500
Ground	3300
P1	2800

To place these Detail Items, first we need to get the Levels from the project and sort them in ascending order of their elevation (not their name). We start by finding all of the levels by using Companies and All Elements of Category, next we can use a sort by key using the elevation values of the levels.

Next we get the elevation values from the sorted levels and drop the last item (we don't need to show the top level as nothing will exist above this level).

In order to size the Detail Items we need to know the Floor to Floor Heights between the Levels, I use a Python Script for this. We



also need the name of the elevation, which we then combine with the level to level heights

The Python takes the elevation of each level and the level above and finds the difference between them, this is the floor to floor height.

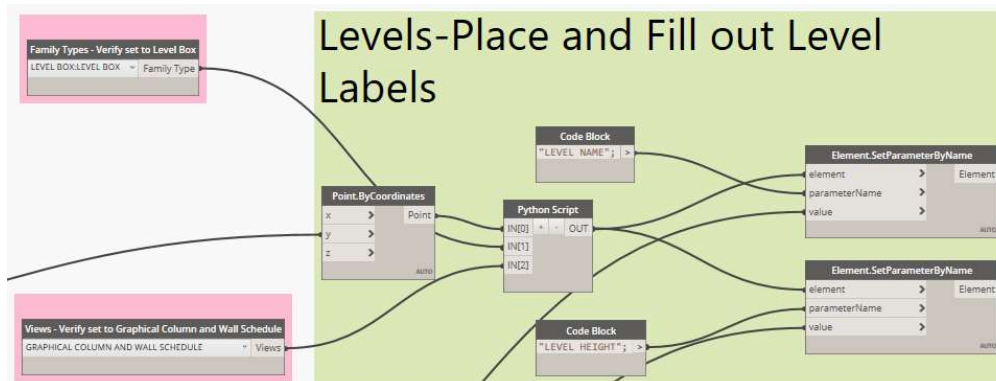


```

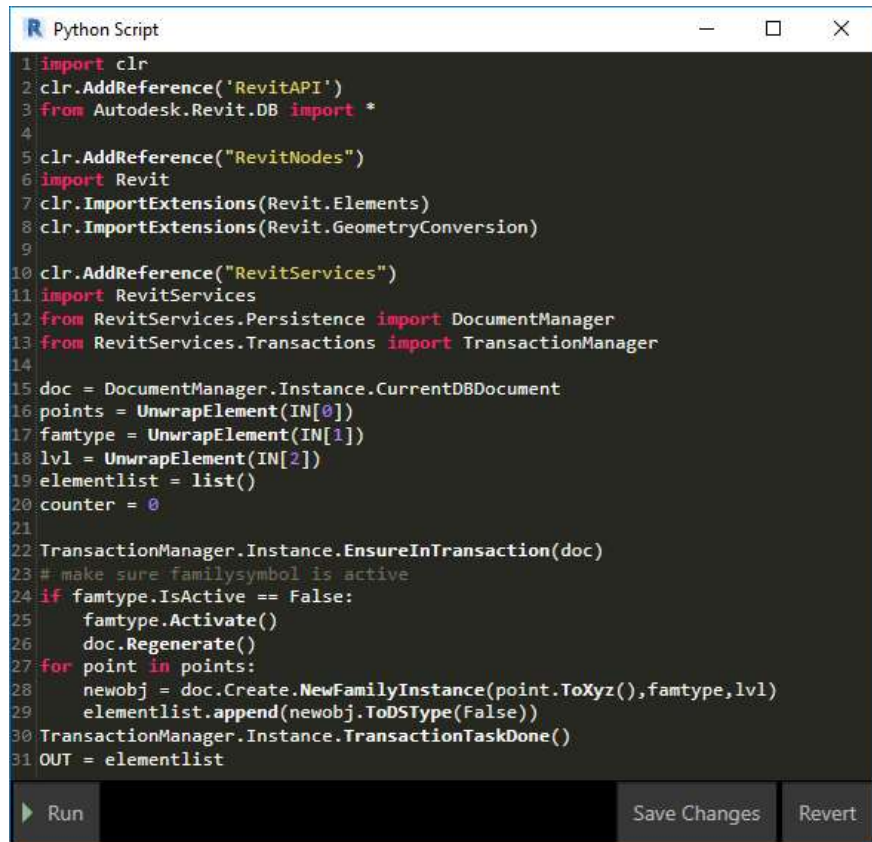
1 import clr
2 clr.AddReference('ProtoGeometry')
3 from Autodesk.DesignScript.Geometry import *
4 #The inputs to this node will be stored as a list in the IN variables.
5 list = IN[0]
6
7 Heights = []
8
9
10 for i in range(len(list)-1):
11     Base = list[i]
12     Top = list[i+1]
13     Heights.append(Top-Base)
14
15 #Assign your output to the OUT variable.
16 OUT = Heights
  
```

Run Save Changes Revert

Finally, we need to place the Detail Items, note that we reference the Family Type and View from the Family Types and Views comboBox Nodes, this is problematic in real application as the data stored in the comboBox is the index of the comboBox, so if you open this in a different project or a modified project the list of items in the comboBox may have changed. To elaborate, lets say there are 50 families and the 32nd one is Named DoorSingle and this is the one you select in the comboBox, now if you add 5 families starting with the letter B to the project then reopen the graph your comboBox will no longer be referencing your family correctly, it will be referencing the family 5 items before it in the alphabetic list. We can get around this by using a string and searching for the family, but this has issues too as if the family has been renamed or removed then this will cause an error.



We use some Python to actually place the Detail Families, then a couple of SetParameterByName to set the LEVEL NAME and LEVEL HEIGHT family parameters on the Detail Items to ensure the Detail Items have the correct size and information.

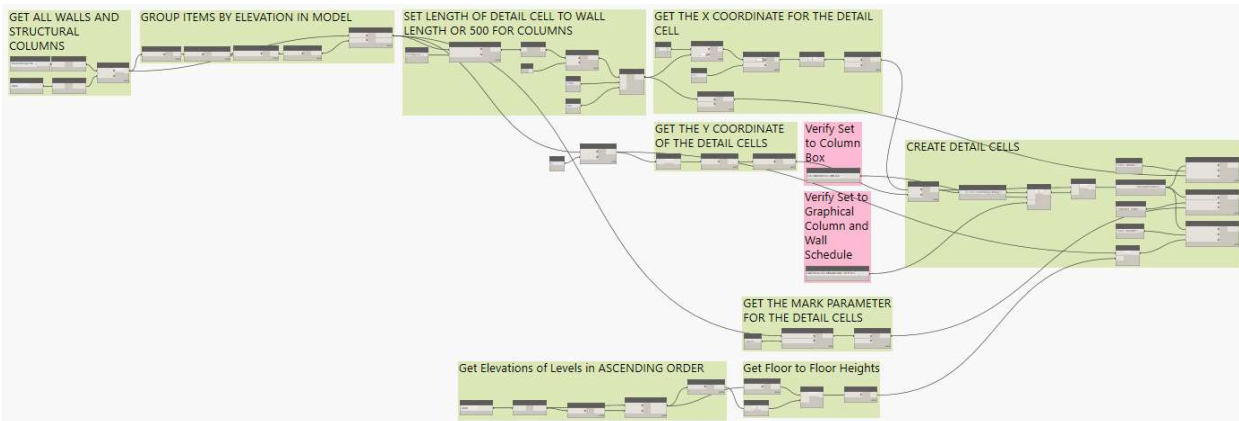


```
1 import clr
2 clr.AddReference('RevitAPI')
3 from Autodesk.Revit.DB import *
4
5 clr.AddReference("RevitNodes")
6 import Revit
7 clr.ImportExtensions(Revit.Elements)
8 clr.ImportExtensions(Revit.GeometryConversion)
9
10 clr.AddReference("RevitServices")
11 import RevitServices
12 from RevitServices.Persistence import DocumentManager
13 from RevitServices.Transactions import TransactionManager
14
15 doc = DocumentManager.Instance.CurrentDBDocument
16 points = UnwrapElement(IN[0])
17 famtype = UnwrapElement(IN[1])
18 lvl = UnwrapElement(IN[2])
19 elementlist = list()
20 counter = 0
21
22 TransactionManager.Instance.EnsureInTransaction(doc)
23 # make sure familysymbol is active
24 if famtype.IsActive == False:
25     famtype.Activate()
26     doc.Regenerate()
27 for point in points:
28     newobj = doc.Create.NewFamilyInstance(point.ToXYZ(), famtype, lvl)
29     elementlist.append(newobj.ToDSType(False))
30 TransactionManager.Instance.TransactionTaskDone()
31 OUT = elementlist
```

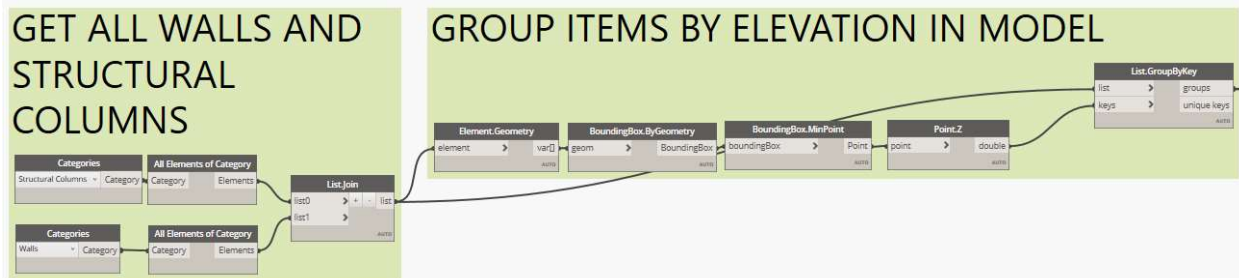
Run Save Changes Revert

Create the Wall and Column Annotations

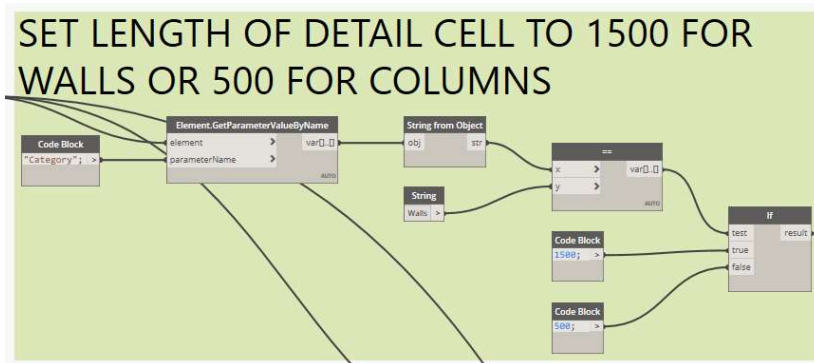
The idea here is to get all of the Walls and Columns in the model and then use the location and dimension of them to setup the Detail Items that will be inserted



We start by getting all of the Walls and Structural Columns using Categories and All Elements of Category, we then use the geometry of these elements to find the minimum point (Z axis value) of each one to sort them in order from bottom to top of the project.

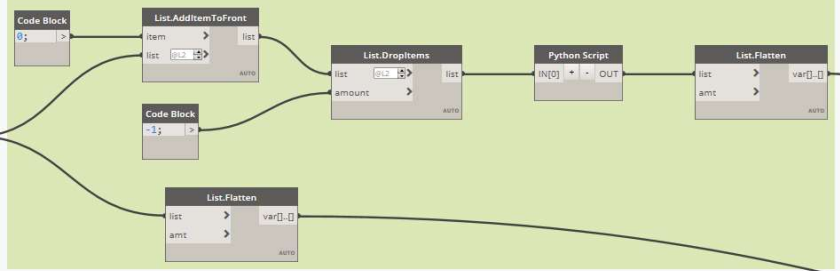


We define the length of the annotations in the Graphical Schedule using default values of 1500mm for Walls and 500mm for Columns, we do this by sorting the Category of the elements and using an IF Node.

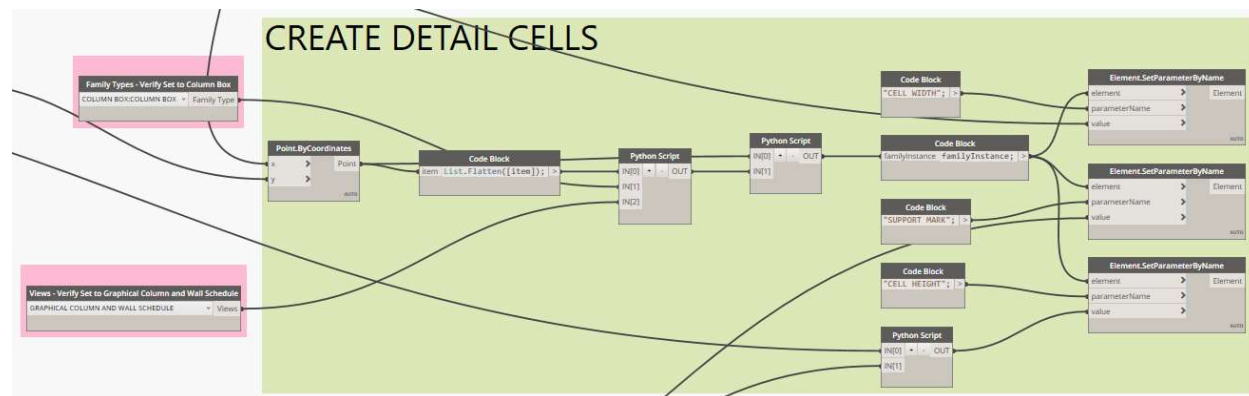


Now we have a list of the widths (500 and 1500 values) we need to use these to generate a list of insertion points in the X axis, we add a zero to the list (as a starting point) then modify the list to be an incremental sum value using a Python Script, so if we have widths of 500, 1500, 500, 500 then we would end up with values of 0, 500, 2000, 2500, 3000 as the insertion points in the X axis.

GET THE X COORDINATE FOR THE DETAIL CELL



Finally, we bring together all of the values to insert the Families similar to how we inserted the Level Detail Items above.



Area Schedule

The idea behind this graph is to show that data does not need to be exclusively shown as one element per row in a Revit schedule. One of the things that new users of Revit have trouble with is understanding that a Revit schedule is not the same as an Excel spreadsheet. While there are many powerful tools within Revit schedules, you are always limited by the fact that a row is equal to an element, or a row is equal to a group of elements if you tell it to bring them together via Grouping/Sorting.

As always, please read this handout in conjunction with the actual Dynamo Graph and the follow along video posted with the class on the Autodesk University 2018 class site.

In the example I have here we are looking to display information in Excel, grouped in a way that Revit can not do, we will have Area objects with parameters for Department → Sub Department → and Role, we will show them in the following way:

Department

Sub Department

Role

Role

Department

Sub Department

Role

Role

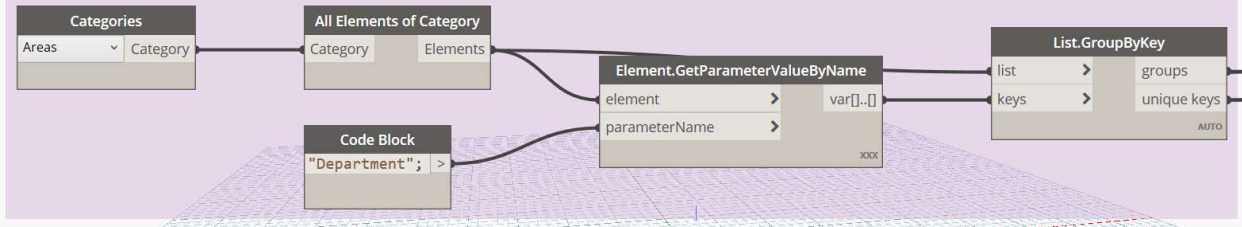
We will be playing with some data extraction and sorting that was a little bit harder than I thought it would be before I started so there is a pretty good chance that I have not done this efficiently. One of the great things about Dynamo and Programming in general is that you may create a solution for the same thing 2 months apart which solve the same issue but the solution is totally different. I like to think that I get better at these things as time goes on, but maybe every time is just a bit of a craps shoot (Note I spelt that one as 2 words 😊)

The test Revit project file that comes with this graph is setup with the Project Parameters for Department, Sub Department and Role, Role is essentially the end user of the office, and then Sub Department and Department are the Departments they belong to in the company.

Gather the model elements

We start by getting all the area elements in the model and grouping them by the Department parameter.

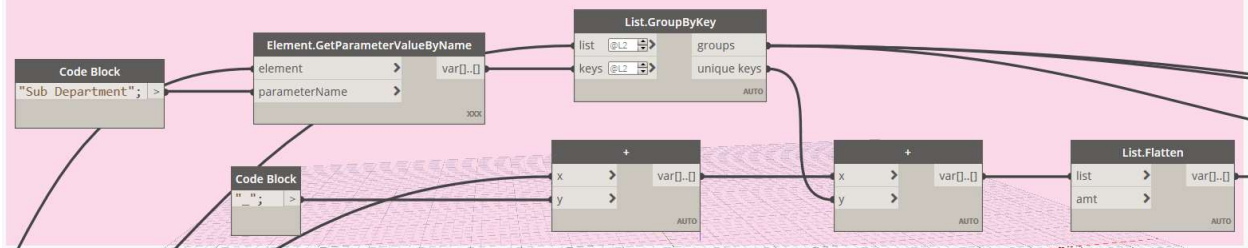
Get all Areas and Group by Department Parameter Value



Grouping the elements

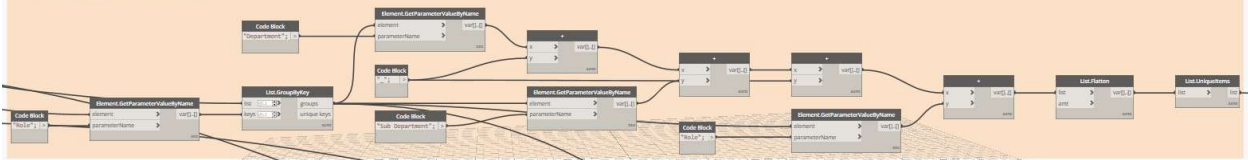
Now we get a list of all the Department + Sub Department Names along with the elements sorted and grouped by the Department + Sub Department.

Construct all of the Sub Department Names



Now we get a list of all the Department + Sub Department + Roles Names along with the elements sorted and grouped by the Department + Sub Department + Roles,

Construct all of the Role Names

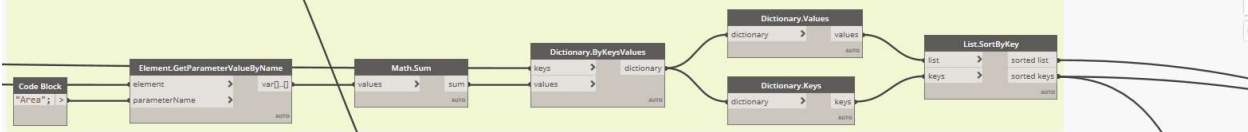


Extraction of a parameter value and Storage in a Dictionary

We then find the Area along with the Name for each Area element and store these values within Dictionarys that we create. Dictionary's are interesting elements in Python, they enable you to store information as Key Value pairs. You can store pretty much anything in the dictionary, but it is easiest to think of it as a single callable value that has other data assiated with it. An example would be a Room number as the Key and when you reference this Key it retures data about the room like number, area, department etc.

My reasoning for using Dictionaries here is that I want to store the Area for the 4 Departments as well as the Area for the 14 Sub Departments and finally the Area for the 20 Roles so that they are easily callable later.

Get Area of Each Department & Sort Alphabetically



Get Area of Each Sub Department & Sort Alphabetically



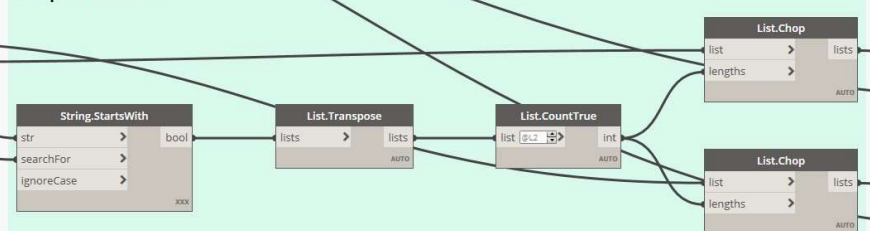
Get Area of Each Role & Sort Alphabetically



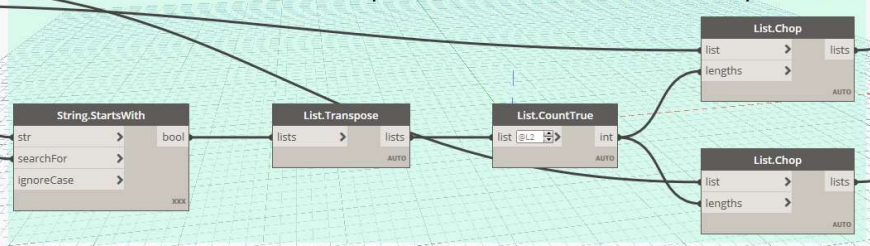
Chopping the lists

I now have my Sub Departments stored but I want to see them further split into Department + Sub Department values, I know how many of each Sub Department elements I have in each Department so I can take this count value and use it to chop the main list resulting in sub lists that have the values for each Dub Department within the Department. The same process is then applied for the Roles within the Sub Departments.

Sub divide the Sub Departments into separate lists based on Department



Sub divide the Roles into separate lists based on Sub Department



I found that the most complicated part of this was to organize the data into a hierarchy of Department, Sub Department, Role...

Department

Sub Department

Role

Role

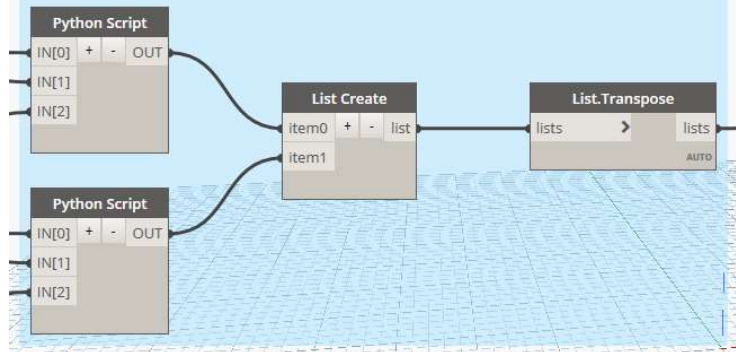
Department

Sub Department

Role

Role

Manipulate the Data for Excel



Honestly if anyone has a better way with OOTB Dynamo Nodes please let me know (drew.jarvis@solidcad.ca)... I went for Python in the end as this allows me to control lists and loops much easier.

Here is an example of the Python Script (the top node in the image above):

```

Python Script
1 # The inputs to this node will be stored as a list in the IN variables.
2 Department = IN[0]
3 SubDepartment = IN[1]
4 Role = IN[2]
5 Names = []
6 i = 0
7 j = 0
8
9 # Place your code below this line
10 for department in Department:
11     Names.append("")
12     Names.append(department)
13     for subdepartment in SubDepartment[i]:
14         Names.append(subdepartment + " (Total)")
15         for role in Role[j]:
16             Names.append(role)
17         j = j+1
18     i = i+1
19
20 # Assign your output to the OUT variable.
21 OUT = Names
  
```

As you can see it is taking in the Department, Sub Department and the Roles and simply creating a long list of them in the hierarchy that I need. Note that with the for loops they have a colon character at the end of the line and the next line is indented (with a Tab).

Exporting to Excel

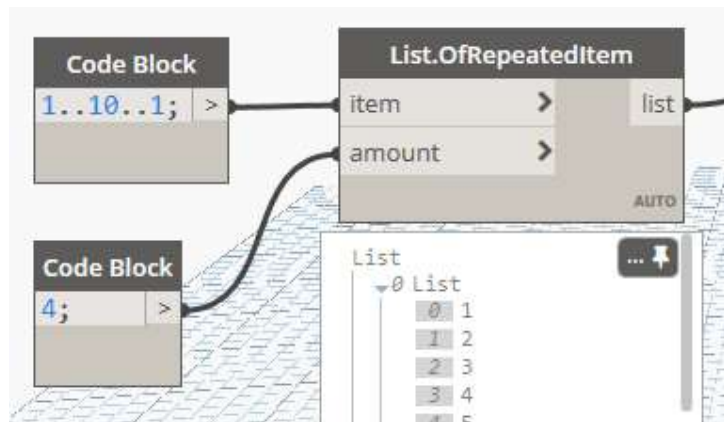
Now we have created the data to export we need to send this data to Excel. The Data.ExportExcel node has a few inputs, the first of which requires you to tell it the location of the excel file, to do this you use a File Path node to Browse to the file you want.

You then supply the name of the worksheet via a Code Block or String Node.

Next up is the starting location of the data in excel, this is the zero index location of the row and cell, so for example A1 is 0,0 and C5 is 4,3.

Now we supply the data to be written which will be supplied as top level lists = rows

for example in the image to the right you can see that there will be 4 lists of 1 to 10, these become 4 rows of 10 columns in Excel like the image below.



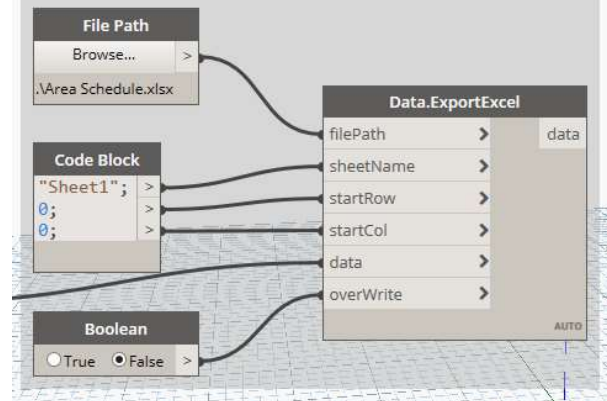
	A	B	C	D	E	F	G	H	I	J
1	1	2	3	4	5	6	7	8	9	10
2	1	2	3	4	5	6	7	8	9	10
3	1	2	3	4	5	6	7	8	9	10
4	1	2	3	4	5	6	7	8	9	10

Finally, you get to choose if you want to overwrite the data or not... so... of course you want to overwrite the data right? Why would you write data but not overwrite data?

Well, apparently just like Worksets being 'editable' in Revit, this terminology is a little confusing.

If you set overwrite data to false then it only overwrites the cells you want to overwrite, if you set it to true it basically clears the worksheet and places the new data into a blank worksheet. Maybe it's just me but that seems... (fill in the blank), so if you want to retain the formatting and existing data around your export you need to set overwrite to false (even though you do want to overwrite the data that you are writing to the excel file)

Send Data to Excel, note that False means you can keep formatting



Python

Getting started with python can seem like a daunting task, if you have no prior experience with programming in C# or similar then it requires picking up a whole new skill. Even if you do have experience with C# Revit API programming you are still going to have to learn a new syntax of talking to the computer.

I suggest taking some training on Python from someone who knows what they are doing, or using some on line references to dip your toes into the Python water.

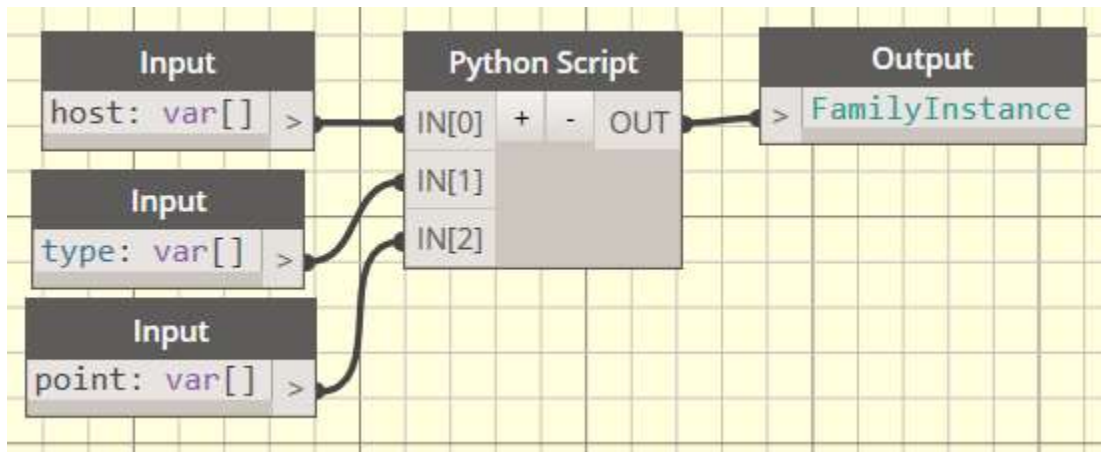
This Handout is not going to teach you how to write python for Dynamo, but I would like to share some of the things I figured out along my way.

Reading other peoples code is a great way to learn

If someone has written some python for Dynamo and it works, and you know what it does, then reading through the code is a great way to get to know the language, be aware that not all code is well written, so there may be code in the example that is not required, or is inefficient.

Spotting these things means that you are progressing 😊

You will find great code snippets on the forums at dynamobim.org and also by investigating packages. A lot of packages that you download are simply wrappers for python code, for example the Springs.HostedInstance.ByPoints is a great Node in the Springs package created by Dimitar Venkov dimitar.ven@gmail.com and while I totally appreciate the effort to create and distribute the package, if you double click on it and take a look at the contents of HostedInstance.ByPoints.dyf you will see the following:



So essentially this is a Python script with 3 inputs and 1 output, the Python code is shown on the next page.


```

Python Script
1 #Copyright(c) 2016, Dimitar Venkov
2 # @Sdevene, dimitar.venkov@autodesk.com
3
4 import clr
5
6 clr.AddReference("RevitServices")
7 import RevitServices
8 from RevitServices.Persistence import DocumentManager
9 from RevitServices.Transactions import TransactionManager
10 doc = DocumentManager.Instance.CurrentDBDocument
11
12 clr.AddReference("RevitAPI")
13 import Autodesk
14
15 clr.AddReference("RevitNodes")
16 import Revit
17 clr.ImportExtensions(Revit.Elements)
18 clr.ImportExtensions(Revit.GeometryConversion)
19
20 def tolist(obj1):
21     if hasattr(obj1, "__iter__"): return obj1
22     else: return [obj1]
23
24 host = UnwrapElement(tolist(IN[0]))
25 ftype = UnwrapElement(tolist(IN[1]))
26 fpts = UnwrapElement(tolist(IN[2]))
27
28 OUT = []
29 strt = Autodesk.Revit.DB.Structure.StructuralType.NonStructural
30 ftp_len = len(ftype) == 1
31 hst_len = len(host) == 1
32
33 TransactionManager.Instance.EnsureInTransaction(doc)
34 for i in xrange(len(fpts)):
35     p = fpts[i].ToXYZ(True)
36     j = 0 if ftp_len else i
37     k = 0 if hst_len else i
38     try:
39         if not ftype[j].IsActive : ftype[j].Activate()
40         level = doc.GetElement(host[k].LevelId)
41         nf = doc.Create.NewFamilyInstance(p, ftype[j], host[k], level, strt)
42         OUT.append(nf.ToDSType(False))
43     except:
44         OUT.append(None)
45 TransactionManager.Instance.TransactionTaskDone()
  
```

Run Save Changes Revert

From reading the code on the page before you can see that:

- References are imported
- A method named tolist is defined
- Inputs are assigned to variables
- Other variables are defined
- A Transaction is opened
- A for loop is used to place the family instances
- The family instances are added to the OUT list
- The Transaction is closed

Being able to read and understand this code means that you can start to replicate code like it, if there are parts of it that you are unsure of then you can simply research that code.