

SD322695

# EXPLORING PYTHON NODES IN DYNAMO

Tadeh Hakopian

HKS

BIM Manager and Design Technology Specialist

## Learning Objectives

- Learn about the Python node in Dynamo and how to configure it
- Learn how to execute code in Python with statements and conditions
- Learn how to debug your Python script when errors appear
- Learn how to create Python scripts for Dynamo that can create new editing possibilities unavailable with regular Dynamo or Revit tools

## Description

Learn how to use Python in Dynamo to enable more editing options than ever before. Python is one of the fastest-growing scripting languages and can be used for daily tasks in your own projects. Dynamo can let you directly input Python code into your scripts to do things regular nodes can't. This course will lead you through how to plan, edit, and execute your own scripts with Python for Dynamo. Learn about the essentials of setting up your own Python script, and edit geometry, sort data lists, write content to Revit software, and much more. With Python, you can unleash the potential in your projects so come and see what's possible.

## Speaker(s)

**Tadeh Hakopian** leverages BIM, VDC and Design Technology to provide his teams with impactful tools for project success. He has over 8 years of experience in the AEC field developing methods and practices to enhance project outcomes. With a background in Architecture he has worked with designers, engineers and contractors in all phases of building design and construction. Over the years he has been a part of large, complex projects in Commercial, Sports, Education, Healthcare and Residential sectors. His current focus is on design automation, data insights in projects and comprehensive workflows that come full circle in planning project life cycles. He is an active speaker at conferences including Trimble Dimensions, ATG Midwest University, BILT NA, BIM Forum and his local community meetups. Current Professional Goals Help move the AEC profession into new horizons using value driven solutions and innovative research.

## About this Class

Everyone should be able to try out coding so I created this class as a starting point. Sometimes the barrier feels high and you don't know where to start which is where Dynamo comes into the picture.

For those in the AEC field who are interested in coding you can use the Dynamo plug in which is a great utility that makes the most out of visual scripting with Revit. With Dynamo you can create your own scripts and execute the commands and write to Revit. This includes Python language support which is easy to learn and has an active community around it.

This class is meant to be a a showcase of how Python can be used in Dynamo for practical everyday tasks. This includes design exploration, modifying your model, integrating software tools, data reporting, document setup and more. After taking this class for yourself you will see what it takes to get started with Python including the essentials of syntax and deploying code.

It's wonderful that we have all these tools available for us and a lot of open source content to go with it. Hopefully with this class you can get a good start and a sense of what is possible for your own projects. Trust me it's easy.



# **Learning Objectives**

## **Introduction**

This class is meant to orient a novice with using Python code for Dynamo. That includes project setup, configuring your test environment, working through examples and additional context for problem solving.

## **Learn about the Python node in Dynamo and how to configure it**

The Python node has virtually all the functionality of the Revit API at its disposal. This class will show you what the properties of the Python node and how you can use it in the Dynamo environment.

## **Learn how to debug your Python script when errors appear**

Coding comes with debugging. We will go through examples of what goes right and wrong in code and typical fixes. Debugging is an important part of the process as errors are routine and problem solving is essential to successful coding.

## **Learn how to execute code in Python with statements and conditions**

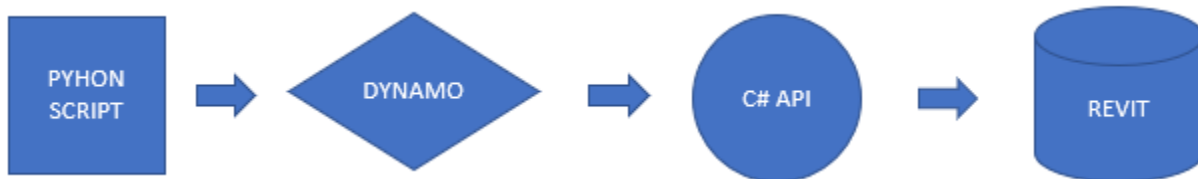
Like all coding languages you have to set certain parameters to execute your script. Python makes this process straight forward and intuitive. We'll dive into how to set up statements and how to use them in a comprehensive way with improvised and boiler plate code.

## **Learn how to create Python scripts for Dynamo that can create new editing possibilities unavailable with regular Dynamo or Revit tools**

The point of this class is to take all that Python coding knowledge to make your own scripts and solve problems in your projects. We will go into different use cases of Python in Dynamo, solutions to modeling and editing dilemmas and what is possible with Python that might not be so easy with typical Dynamo and Revit workflows.

## Why Use Python in Dynamo

1. Expanding the capabilities of Dynamo with custom scripts
2. Consolidate actions with repeatable commands (loops)
3. Package code into smaller and easier to read scripts
4. Python can be used beyond Dynamo into every imaginable Software workflow



## Prerequisites

Ideally you would be prepared with some knowledge of how Dynamo and Python coding functions before taking this class. The training includes basic examples of each. To get the most out of your time try out these courses to get a grounding in the class before starting.

1. Python Basics from Google - <https://developers.google.com/edu/python/set-up>
2. Dynamo Basics - <https://www.autodesk.com/autodesk-university/class/Dynamo-Dummies-Intro-Dynamo-and-How-It-Interacts-Revit-2014>
3. Python in Dynamo Basics - <https://www.autodesk.com/autodesk-university/class/Untangling-Python-Crash-Course-Dynamos-Python-Node-2017>

## Computer and Software Setup

This class is structured as a demonstration but you can follow along with your own computer. It is recommended to do this after the course since it will be recorded and meant to be repeatable.

1. Revit 2019 or higher and a computer that can run the software.
2. Dynamo 2.0 or later
3. Python 3.0 or later (IronPython comes with Dynamo)
4. Python IDLE or another Code IDE like Sublime Text

## Exercise Prep

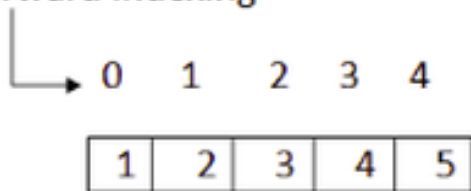
This is a list of all the topics covered in the handout. The course is meant to build up skills from basics to more complex examples based upon the prior example while incorporating concepts as the examples advance. There is also a supplement portion that explain the process that goes with any scenario like problem solving and debugging. The Exercises start at 0 to reinforce the Python index numbering system which also starts at 0.

- 0 - Python Node essentials – The Python Node, Boilerplate code, Strings Concatenation
- 1 - Lists – Basics of lists, Appending values to a list
- 2 - For Loop – iterate variables, nested loops, append to list
- 3 - Writing to Revit – Using Dynamo nodes, inputs to code, writing to Revit
- 4 - Using Definitions – Definitions and functions, writing to python
- 5 - Unwrap elements – examples of wrap and unwrap, modifying the code for Revit Services
- 6 - Try and Except – Example of code and use case
- 7 - Loop Coordinates – Example of code with existing graph

\*\*\* Supplemental Training \*\*\*

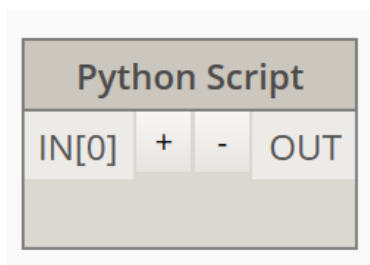
- Problem solving with Documentation
- Distribute
- Debugging

Forward indexing



## Exercise 0 - Essentials

- Dynamo already has the Python Node built into it
- This version of Python is Iron Python which works with C# compatible for the Revit API
- This node allows you to code within Dynamo without switching to the Revit API
- The node configuration is very simple and starts with some basic code in it
- Like other nodes it has inputs and outputs though you can modify all these attributes
- The key thing to remember when using Python nodes is to understand what your starting point is, the inputs and the outputs just like a regular Dynamo graph



```
Python Script
1 # Enable Python support and load DesignScript library
2 import clr
3 clr.AddReference('ProtoGeometry')
4 from Autodesk.DesignScript.Geometry import *
5
6 # The inputs to this node will be stored as a list in the IN variables.
7 dataEnteringNode = IN
8
9 # Place your code below this line
10
11 # Assign your output to the OUT variable.
12 OUT = 0
```

- The Boiler plate code provides essentials and some examples of how to get started
- Add References you need to work with the packages
- Add an Input
- Add an Output
- All the rest of your code goes in between
- That is the essentials of how any Python node works

```
1 import clr
2 clr.AddReference('ProtoGeometry')
3 from Autodesk.DesignScript.Geometry import *
4 dataEnteringNode = IN
5 OUT = 0
```

## Exercise 0 – Inputs and Outputs

This exercise will orient us to the Python node and how it works.

Open Dynamo and create a code block and a Python Node.

Python Node needs inputs from external nodes to work so you have to write them into the node.

We want to concatenate (combine) strings to make one string.

In this example our code block has text strings as inputs into each section of the Python node.

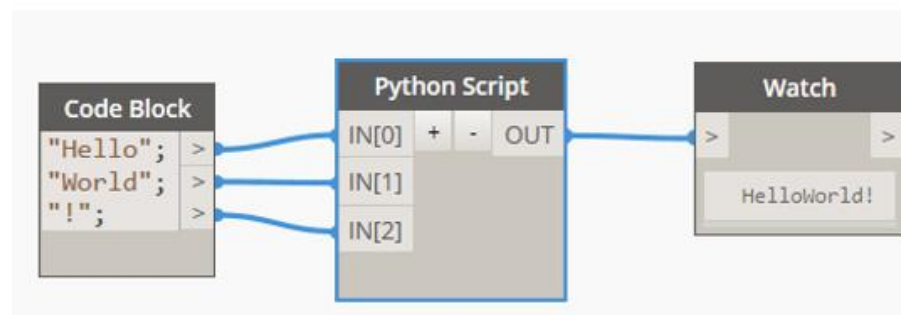
In Python all you have to do is list the output based on inputs.

By creating the instructions in Python you are all set to add your strings and get a combined result. This process is the most basic example of how all scripting in Python works in Dynamo. Clear the template

1. Add a value for each input
2. Set your OUT list to a combination of the values
3. Save and Run
4. Add your Code block to each input
5. Check your results

### Python Script

```
1 #set your inputs
2
3 string1 = IN[0]
4 string2 = IN[1]
5 string3 = IN[2]
6
7 #set your output
8 OUT = string1 + string2 + string3
```



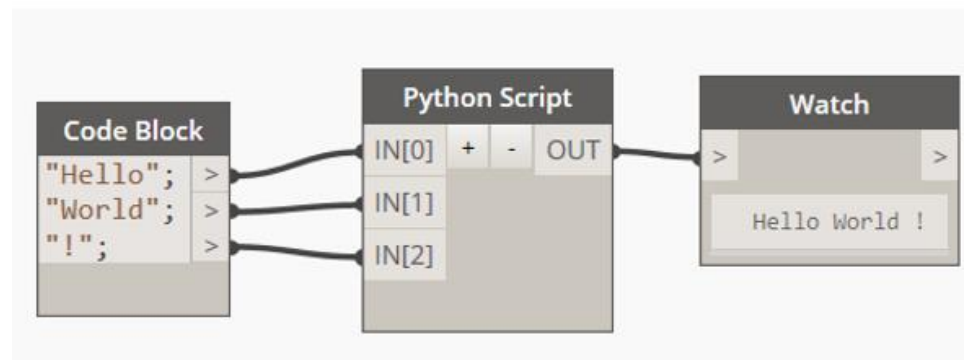


## Exercise 0 – Inputs and Outputs

With the Python node you can put together strings in different ways.  
This is another way to concatenate strings

1. Clear the template
2. Set your OUT list to a combination of the Inputs
3. Save and Run
4. Add your Code block to each input
5. Check your results

```
Python Script
1 # Concatenate the strings as your output
2 OUT = IN[0] + " " + IN[1] + " " + IN[2]
```

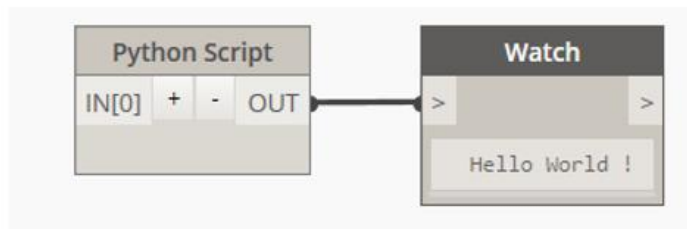


## Exercise 0 – Inputs and Outputs

The entire code can be written into the Python script without external inputs

1. Clear the template
2. Add a value for each statement as a string
3. Set your OUT list to a combination of the values
4. Save and Run
5. Check your results

```
Python Script
1 #set your inputs
2
3 string1 = "Hello "
4 string2 = "World "
5 string3 = "!"
6
7 #set your output
8 OUT = string1 + string2 + string3
```



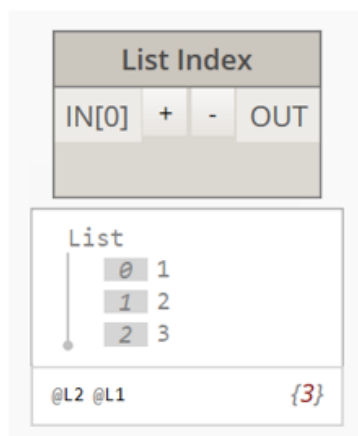
## Exercise 1 – Lists

- Lists are a common output for Python nodes
- You can create a list variable with brackets “[]” or “list()”
- The list can be called whatever you want and appended or modified however you want
- Lists can be made of strings, numbers, element inputs or anything else that can be input to the code
- Any Generic Python editor can create lists

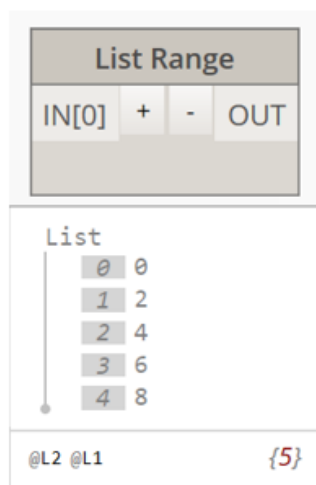
```
example = [] # empty list
example = list() # empty list
example = [1,2,3] # list with three elements
example = [0, "zero"] # elements can be of mixed types
```

Examples of an index and range appended to a list in Python

```
1 list = [1,2,3]
2
3 OUT = list
```

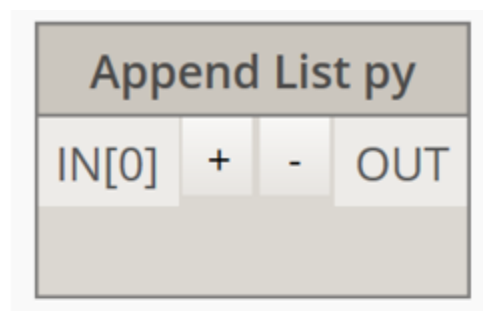


```
1 list = range(0,10,2)
2
3 OUT = list
```



## Exercise 1 – Lists

1. Create as many lists as you want
2. Either internal to the code or external from other nodes
3. Set a list value before creating your code
4. Then use the **append** operation to add the inputs to your list
5. The list can be called whatever you want and appended or modified however you want
6. You can place the list with another value as the OUT = list



```
1 # define variables
2 string1 = "The beginning"
3 string2 = ["one fish", "two fish", "red fish", "blue fish"]
4 string3 = "The end"
5 numbers = [1, 2, 3]
6
7 # out variables
8 list = []
9
10 # Build the list with appended variables
11
12 list.append(string1)
13 list.append(string2)
14 list.append(numbers)
15
16 #results to be returned|
17 OUT = list, string3
```

The result is each list is appended with the string from the values stated above. These lists are indexed as well with 0 being the list itself then each appended value nested.

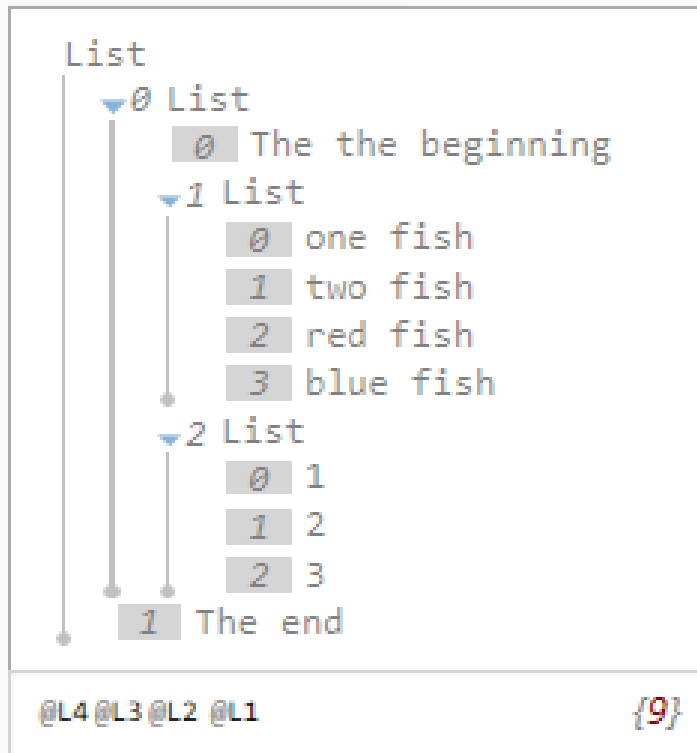
## Append PY

IN[0]

+

-

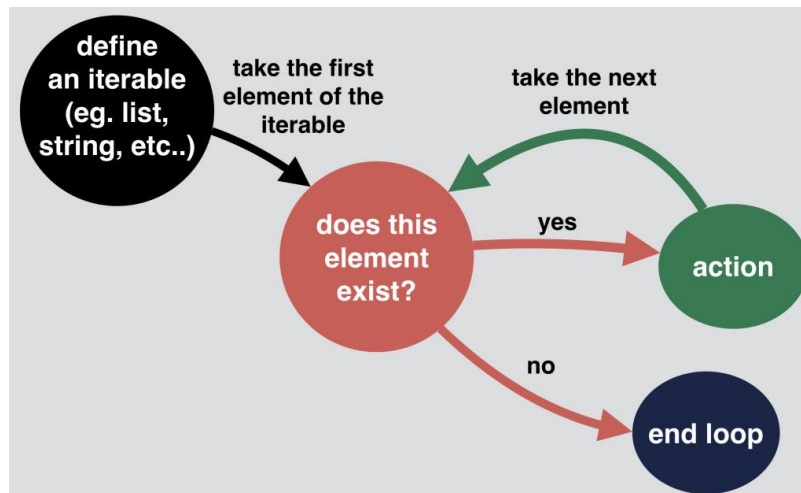
OUT



If the value is not part of an index or range then the nested ends with the value itself.  
Should there be an index or range then each value inside is another nested list.  
These lists can be manipulated like regular dynamo lists.

## Exercise 2 – Loops

- Loops are fundamental to any coding and make the Python node especially useful
- Same concepts as before with values and lists structured to create outputs
- Loops iterate (repeat) over a sequence that you can assign in your code
- Start > Provide Condition > Create Statement > Increment > End
- You don't code a 'loop' it's a description of the process
- You need a 'for' statement followed by a variable you assign which can be whatever you want
- Then followed by the function like list or range followed by whatever is in that function's parentheses
- The end of the line needs a colon then you set the execution like print after that or it won't work.



Python 3.6

```
1 for i in range(0,11,2):
2     print(i)
```

[Edit this code](#)

Print output (drag lower right corner to re

2  
4  
6  
8  
10

FramesObjects

## Exercise 2 – Loops

- Every time you indent the loop it becomes nested meaning it takes lower priority
- Once the loop iterations are over you can place the variables to a list or function and print
- Using variables in your statements means the loop can do a lot of work in a few lines of code
- Operations like loops shows why coding is so useful compared to keystrokes or visual scripting because a few lines can do the work of a lot of repeat commands

Python 3.6

```
→ 1 total = 0
   2
   3 for i in range(0,4):
   4     for j in range(0,4):
   5         for k in range(0,4):
   6             total = (i, j, k)
   7
   8 print(total)
```

[Edit this code](#)

Print output (drag lower right corner to resize)

Frames      Objects

In this regular Python example the values are nested below each other and print to the total with the variables set to each position in the range.

Print output (drag lower right corner to resize)

Frames      Objects

Global frame

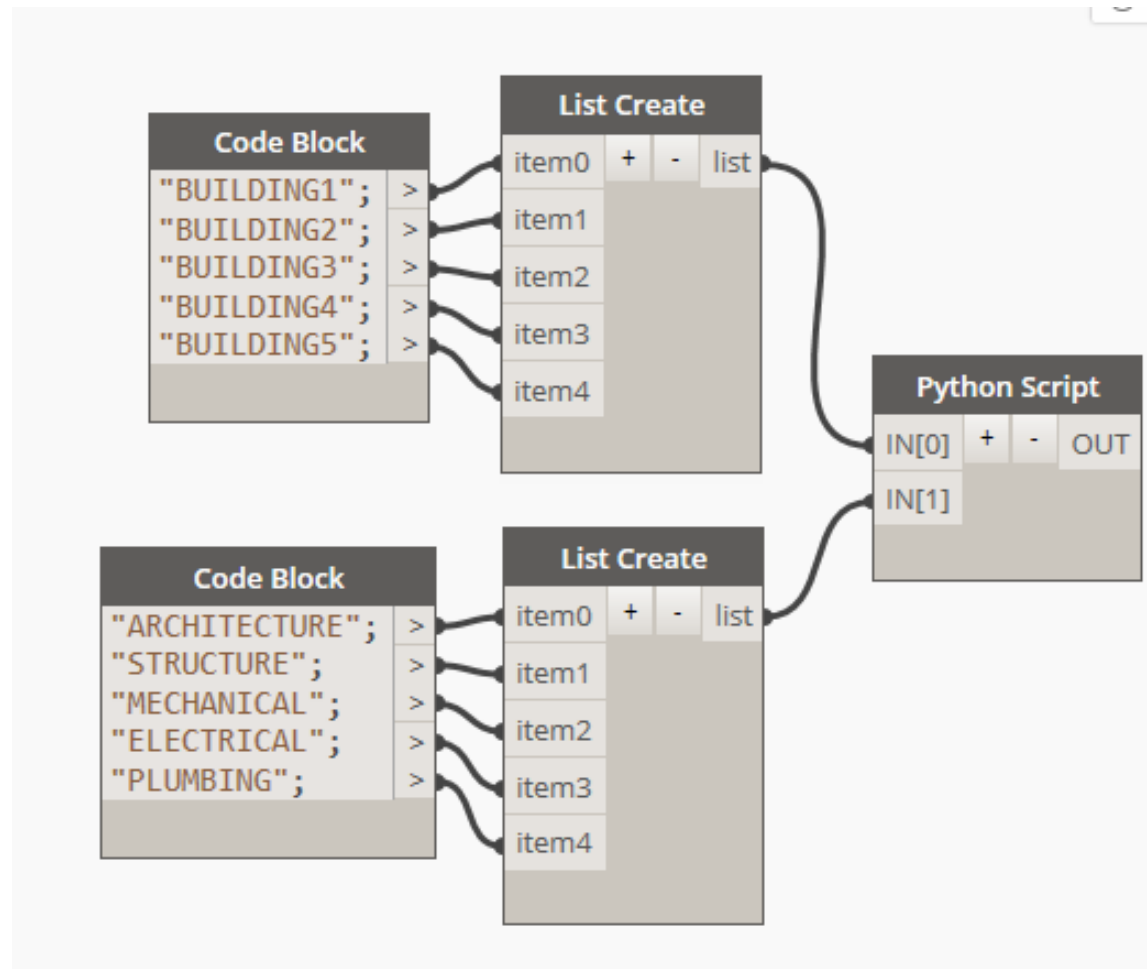
total	
i	3
j	3
k	3

tuple

0	1	2
3	3	3

## Exercise 2 – Loops

- With Dynamo you can create content like strings in code blocks and set a list
- Then feed that list into the script as an input you can loop through
- This example will show how to match all the Building names to the Discipline names

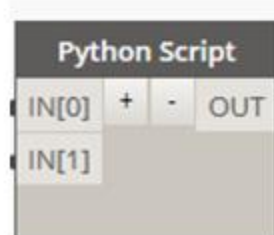




1. Your Outlist will be the list with parentheses meaning you want all the results
2. Start with for 'X-variable' in Value-1
3. Then add colon :
4. Indent
5. Add another for statement ending with colon
6. End of this loop will Append to your list these values within the parentheses
7. OUT is your Outlist

### Python Script

```
1 BuildingNames = IN[0]
2 DisciplineNames = IN[1]
3
4 Outlist = list()
5
6 for BN in BuildingNames:
7     for DN in DisciplineNames:
8         Outlist.append(BN + "_" + DN)
9
10 OUT = Outlist
```



The image shows a Python Script node on the left and a Watch window on the right. The Python Script node contains the following code:

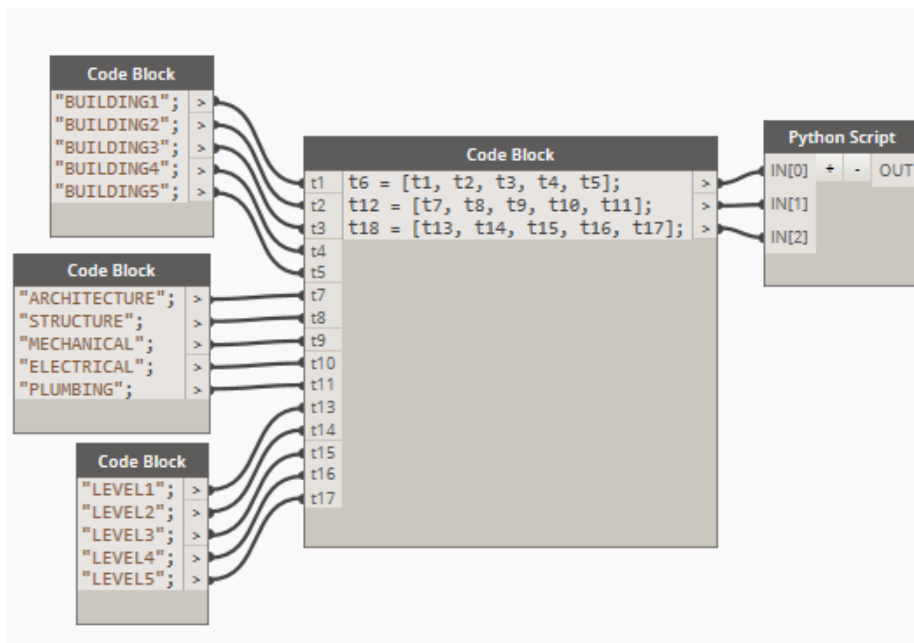
```
1 BuildingNames = IN[0]
2 DisciplineNames = IN[1]
3
4 Outlist = list()
5
6 for BN in BuildingNames:
7     for DN in DisciplineNames:
8         Outlist.append(BN + "_" + DN)
9
10 OUT = Outlist
```

The Watch window displays the output of the script, which is a list of 16 strings. The strings are organized by building and then by discipline. The first four items are for Building 1, the next four for Building 2, the next four for Building 3, and the last four for Building 4. Each building has four disciplines: Architecture, Structure, Mechanical, and Electrical. The strings are concatenated with an underscore between the building name and the discipline name.

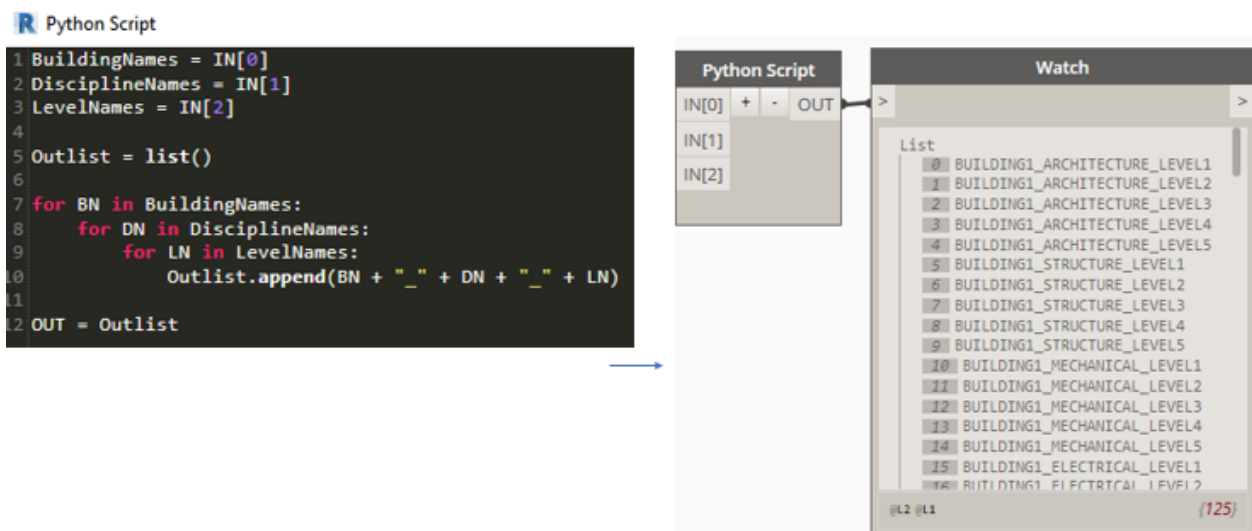
Index	Value
0	BUILDING1_ARCHITECTURE
1	BUILDING1_STRUCTURE
2	BUILDING1_MECHANICAL
3	BUILDING1_ELECTRICAL
4	BUILDING1_PLUMBING
5	BUILDING2_ARCHITECTURE
6	BUILDING2_STRUCTURE
7	BUILDING2_MECHANICAL
8	BUILDING2_ELECTRICAL
9	BUILDING2_PLUMBING
10	BUILDING3_ARCHITECTURE
11	BUILDING3_STRUCTURE
12	BUILDING3_MECHANICAL
13	BUILDING3_ELECTRICAL
14	BUILDING3_PLUMBING
15	BUILDING4_ARCHITECTURE
16	BUILDING4_STRUCTURE

The Watch window also shows the list's length as 16 and the total number of elements as 25.

1. For every value you add that's another loop you can run
2. Assign a value per category and add another indented loop to provide an output
3. Benefit of using code is that you can change the append to other things
4. You can add string values or numbers then use it to write back into Revit without adding another node which keeps your code compact

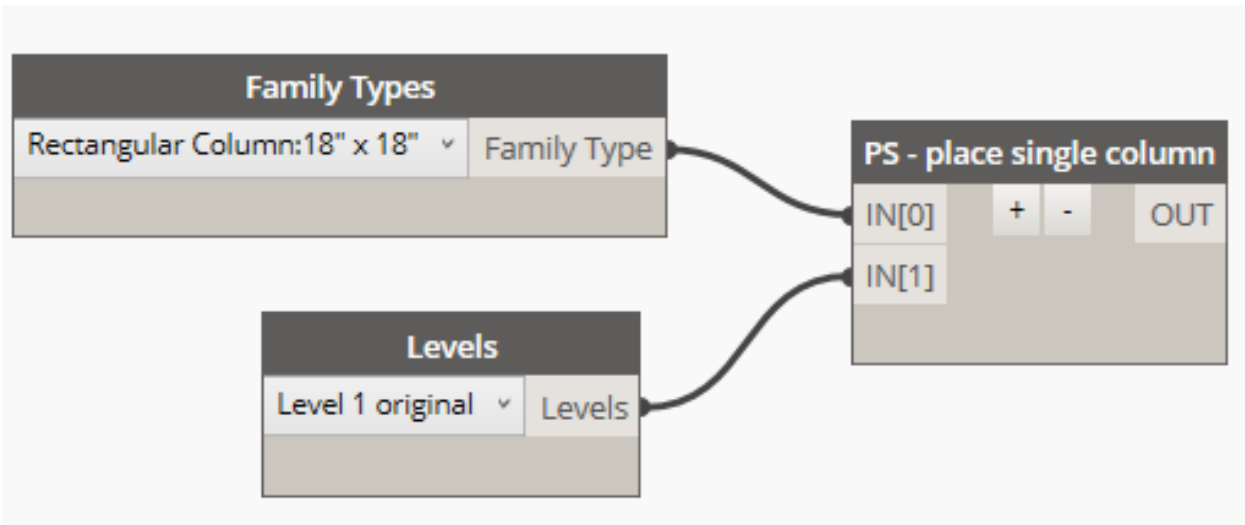


More list inputs can be added and the same logic applies by nested each listed in the order you want it to be appended to the Outlist.

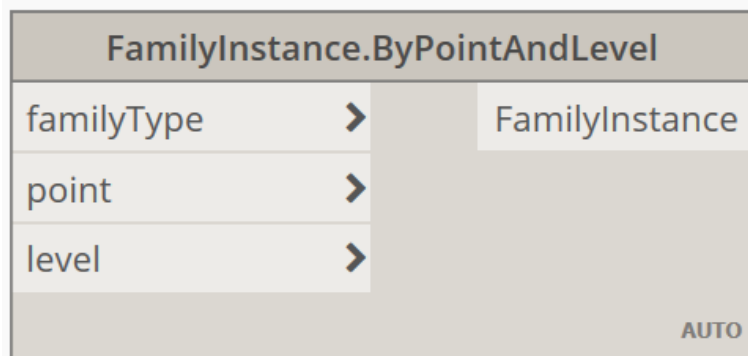


### Exercise 3 – Write to Revit

- With the use of values, lists, append and loops you can use Python to get work done
- For example by placing a family into Revit based on the values you assign
- To start you need a family and a level to place it on assuming it is a level constrained family



Dynamo nodes like the one below can be coding in to the Python script which will be demonstrated.

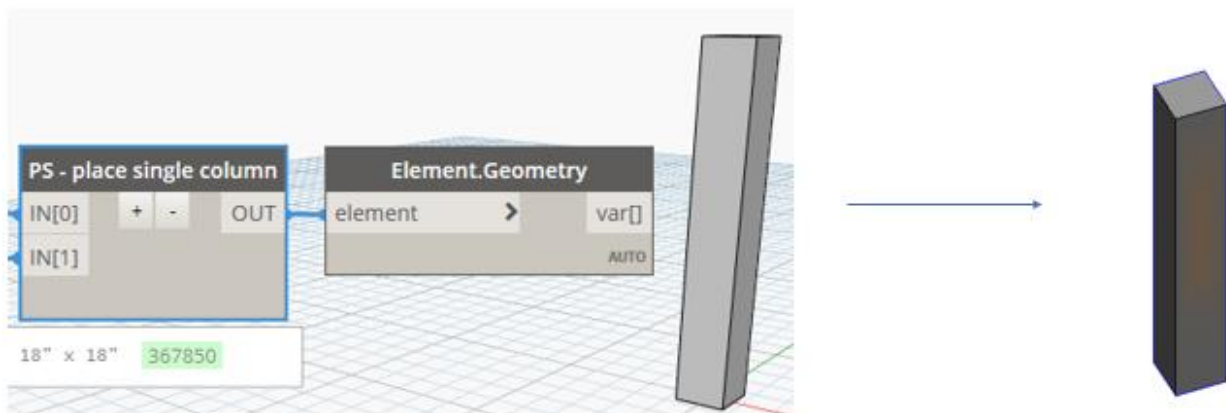


1. Start with the template
2. Make sure ProtoGeometry and RevitNodes are imported
3. Add an input for family type
4. Add an input for the level
5. Create a value for the operation Point.ByCoordinates followed by the origin position
6. Create a value for the family you want to place followed by range of the first 3 values
7. Output is the family
8. Run the script and you get a single column at the coordinates

#### PS - place single column

```
1 # Enable Python support and load DesignScript library
2 import clr
3 clr.AddReference('ProtoGeometry')
4 from Autodesk.DesignScript.Geometry import *
5 clr.AddReference('RevitNodes')
6 from Revit.Elements import *
7
8 famtype = IN[0]
9 level = IN[1]
10 pbc = Point.ByCoordinates(0,0,0)
11 col = FamilyInstance.ByPointAndLevel(famtype,pbc,level)
12
13 # output element based on the variable you assigned
14 OUT = col
```

The result is a single column will be placed at the 0,0,0 position



Next step is adding more columns with adjustments to the code

1. Expand the concept to multiple members
2. Keep the first 3 values but replace the output with a list instead of the family so we can get more than one result
3. Now we use a loop with a range
4. Value pbc has the x value added to it in 10 foot increments based on the range added
5. Columns are then placed starting from 0 then the pbc looped range and the level
6. Output is your list
7. Now you have line of columns

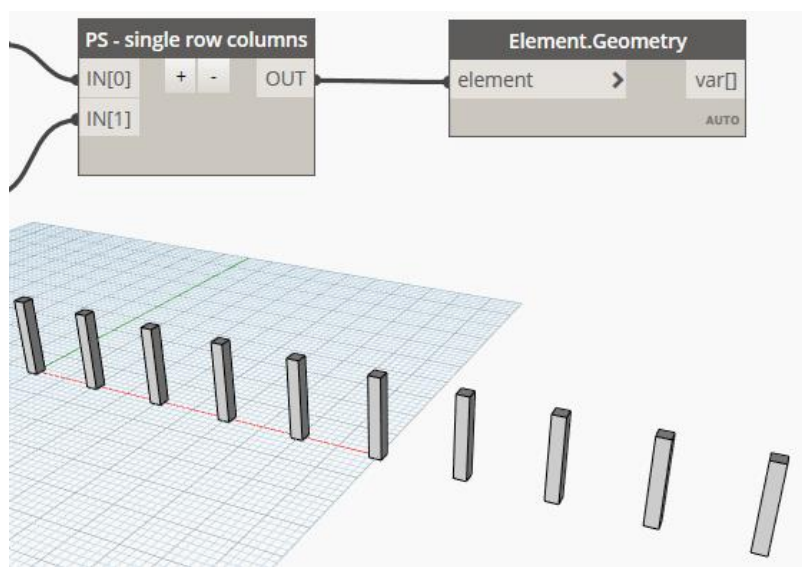
#### PS - single row columns

```

1 # Enable Python support and load DesignScript library
2 import clr
3 clr.AddReference('ProtoGeometry')
4 from Autodesk.DesignScript.Geometry import *
5 clr.AddReference('RevitNodes')
6 from Revit.Elements import *
7
8 famtype = IN[0]
9 level = IN[1]
10 pbc = Point.ByCoordinates(0,0,0)
11 output = []
12
13 for x in range(0, 100, 10):
14     pbc = Point.ByCoordinates(x,0,0)
15     col = FamilyInstance.ByPointAndLevel(famtype,pbc,level)
16     output.append(col)
17
18 OUT = output

```

Now we have a row of columns due to the for loop we added which goes through the range.

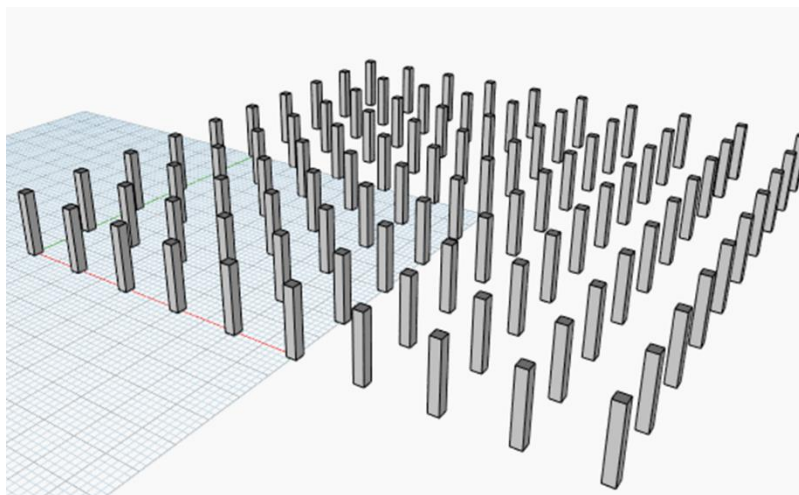


1. You can replicate cross product with another variable in the loop
2. Keep all the values the same
3. Add an indent and another variable in the loop
4. By adding a loop you can then list all points in the x and y range
5. The pbc now gets the x and y values appended to it
6. The remaining code is the same
7. Now you have line of columns
8. Remember the loop with the point by coordinates is the function that made the line work

```
PS - XY matrix column

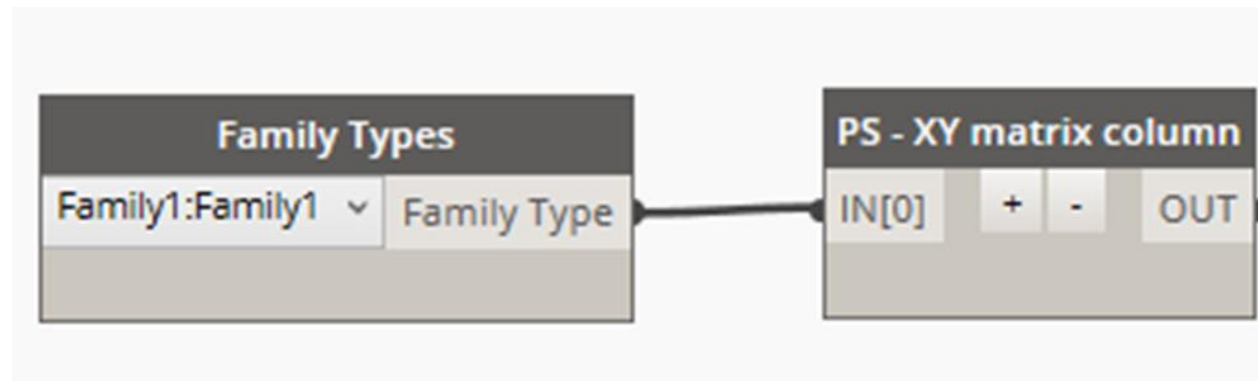
1 # Enable Python support and load DesignScript library
2 import clr
3 clr.AddReference('ProtoGeometry')
4 from Autodesk.DesignScript.Geometry import *
5 clr.AddReference('RevitNodes')
6 from Revit.Elements import *
7
8 famtype = IN[0]
9 level = IN[1]
10 pbc = Point.ByCoordinates(0,0,0)
11 output = []
12
13 for x in range(0, 100, 10):
14     for y in range(0, 100, 10):
15         pbc = Point.ByCoordinates(x,y,0)
16         col = FamilyInstance.ByPointAndLevel(famtype,pbc,level)
17         output.append(col)
18
19 OUT = output
```

Same as last time we get more rows that are distributed according to the range we set.

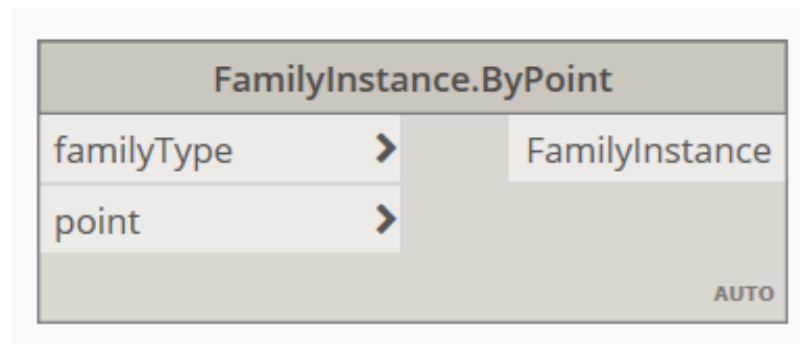


### Exercise 3 – Write to Revit

- If you want a different output then the easy way to figure that out is to just load a node and use its features
- In this example we can use FamilyInstance.Bypoint to do a similar operation to the column placement



Replace the last node with this new one in the script because we do not need the level parameter anymore as this family will be unhosted.



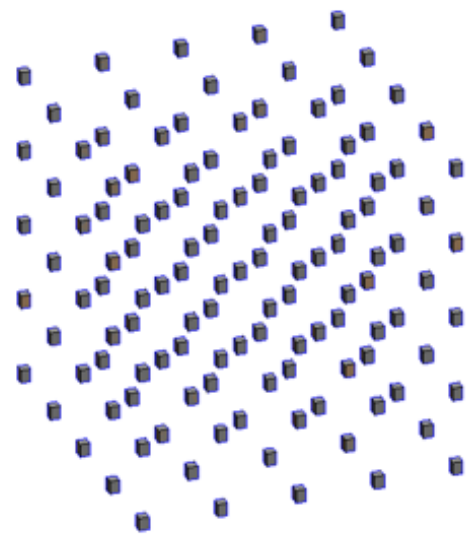
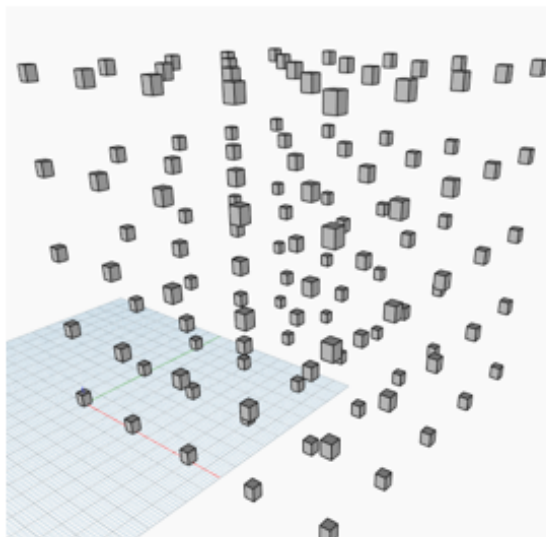


1. The script is fundamentally the same
2. Only difference is we removed the level input since this family doesn't have a level constraint
3. Add one more loop for the z loop value and add that to the pbc range
4. Output the list of points
5. Now you have a grid that produces all the families from the loop an inputs
6. If you run this result then it will write to Revit these families at the coordinates

#### PS - XYZ matrix Family PY

```
1 # Enable Python support and load DesignScript library
2 import clr
3 clr.AddReference('ProtoGeometry')
4 from Autodesk.DesignScript.Geometry import *
5 clr.AddReference('RevitNodes')
6 from Revit.Elements import *
7
8 famtype = IN[0]
9 pbc = Point.ByCoordinates(0,0,0)
10 output = []
11
12 for x in range(0, 100, 20):
13     for y in range(0, 100, 20):
14         for z in range(0, 100, 20):
15             pbc = Point.ByCoordinates(x,y,z)
16             col = FamilyInstance.ByPoint(famtype,pbc)
17             output.append(col)
18
19 OUT = output
```

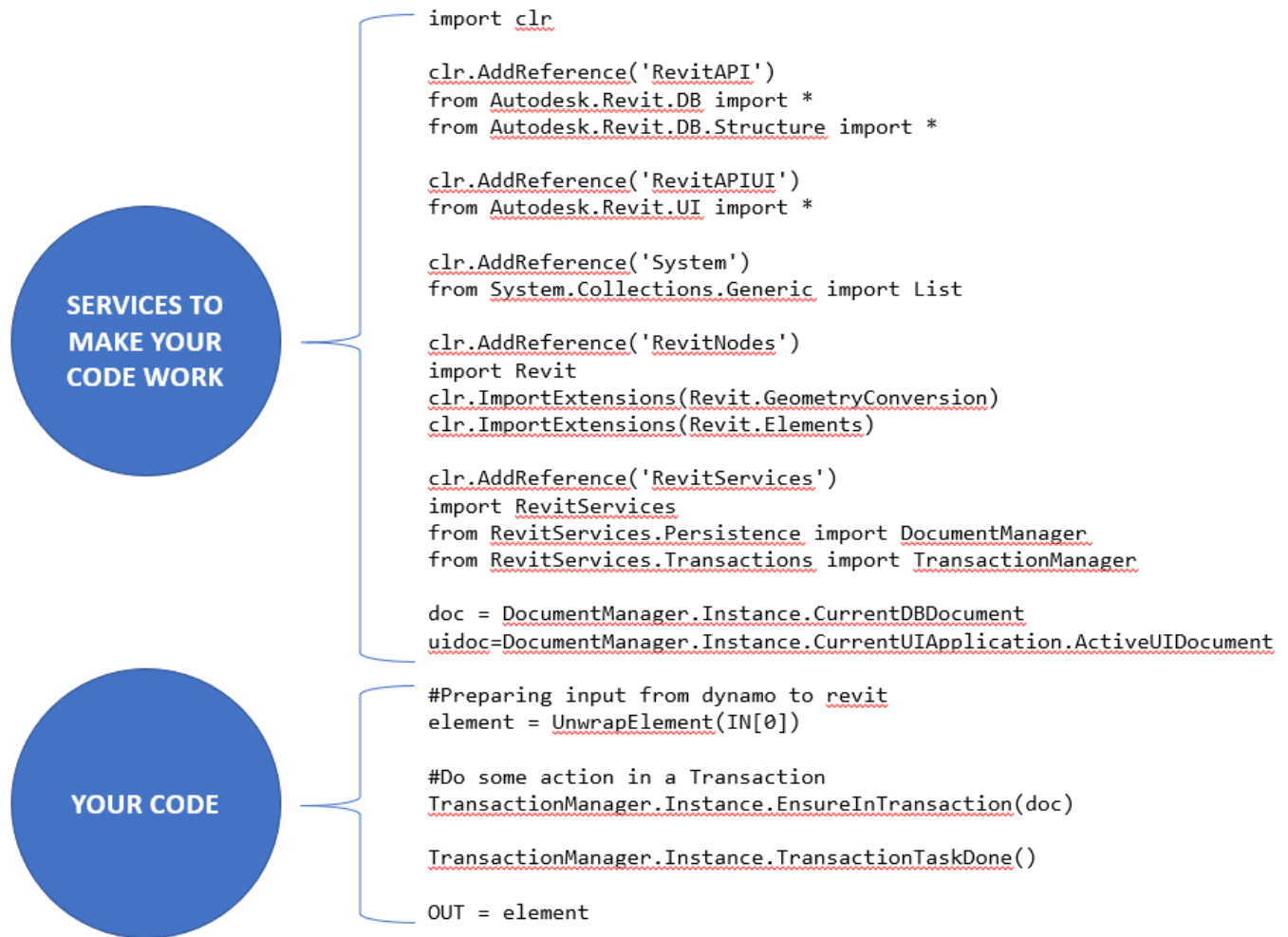
With these nested loops and the node written into the code you can create a lot of elements that can be written directly into Revit saving you time from manual placement.



## Exercise 4 – Unwrap Elements

- Another way to use the Python Node is to bypass restrictions you normally encounter with Revit
- Unwrapping takes the Revit native version of the elements allowing you to manipulate them without going through Dynamo's interface
- Requires transaction manager to work which you load from RevitServices import
- Import that feature and write in the transaction code before and after your main script
- The reason you need transactions is for when you have to modify elements directly
- Not necessary if you are creating brand new elements like the prior example

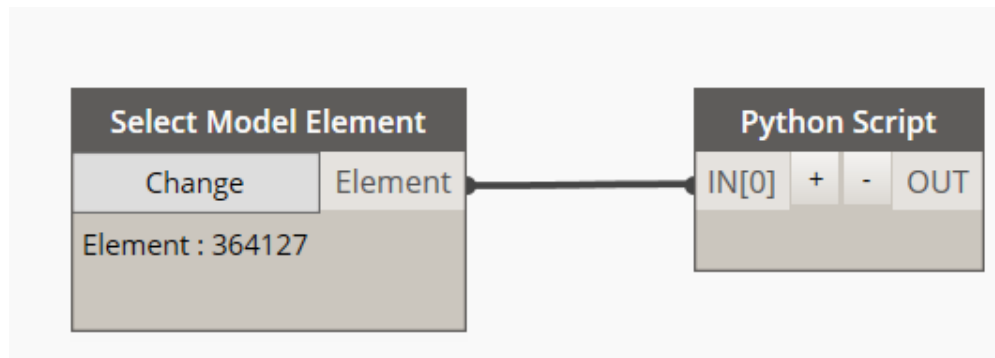
```
1 import clr
2
3 # Add services for document and transactions|
4
5 clr.AddReference('RevitServices')
6 import RevitServices
7 from RevitServices.Persistence import DocumentManager
8 from RevitServices.Transactions import TransactionManager
9
10 # set document manager value =
11
12 doc = DocumentManager.Instance.CurrentDBDocument
13
14 # input elements unwrapped in parentheses
15
16 elements = UnwrapElement(IN[0])
17
18 # Begin Transaction
19
20 TransactionManager.Instance.EnsureInTransaction(doc)
21
22 # code here
23
24 # End Transaction
25
26 TransactionManager.Instance.TransactionTaskDone()
27
28 OUT = 0
```



Importing modules is an important part of your code which adds functions to your script. If you don't add the modules then the functions you call in the code won't work. Modules are a common part of coding and can be part of your template so you don't have to remember what to add each time you edit.

## Exercise 4 – Unwrap Elements

1. In this example the code iterates with a one line for loop to delete all element IDs input
2. That can be done with a filter or a selection node input
3. Code to add includes `doc = DocumentManager.Instance.CurrentDBDocument`
4. Input value needs to go into parentheses as a function of the Unwrapped elements you want to edit
5. Start with ensure Transaction
6. Then code for the elements
7. Finish with Transaction done then OUT



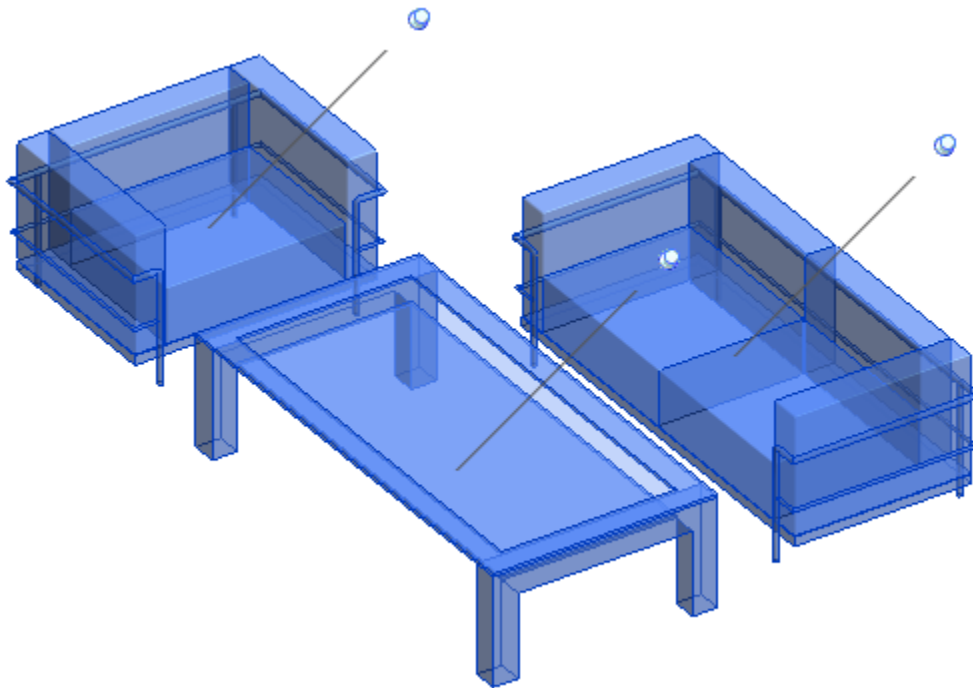
The script is straight forward with a loop before and after the transaction function. Because we are editing elements directly we have to use this function.

### Python Script

```
1 import clr
2
3 clr.AddReference('RevitServices')
4 import RevitServices
5 from RevitServices.Persistence import DocumentManager
6 from RevitServices.Transactions import TransactionManager
7
8 doc = DocumentManager.Instance.CurrentDBDocument
9
10 elements = UnwrapElement(IN[0])
11
12 TransactionManager.Instance.EnsureInTransaction(doc)
13
14 for e in elements:
15     doc.Delete(e.Id)
16
17 TransactionManager.Instance.TransactionTaskDone()
18
19 OUT = 'done'
```

## Exercise 4 – Unwrap Elements

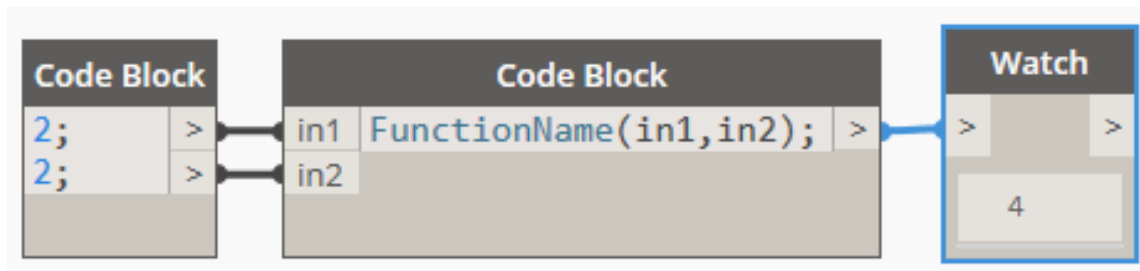
- Even if the ID is pinned you can still delete it
- Convenient for removing a lot of content quickly without guessing which elements are pinned or not



## Exercise 5 – Definitions and Functions

- Another way to use the Python Node is to include definitions and functions
- These defined functions can exist without loops or inputs and work as part of the script
- Works similarly in the code block version of definitions and functions where you need a return but the curly brace '{}' is not required for Python code
- BTW if you know designsript in Dynamo then you're most of the way to using Python node because the concepts are similar

```
Code Block
/*This is a multi-line comment,
which continues for
multiple lines*/
def FunctionName(in1,in2)
{
//This is a comment
sum = in1+in2;
return sum;
};
```

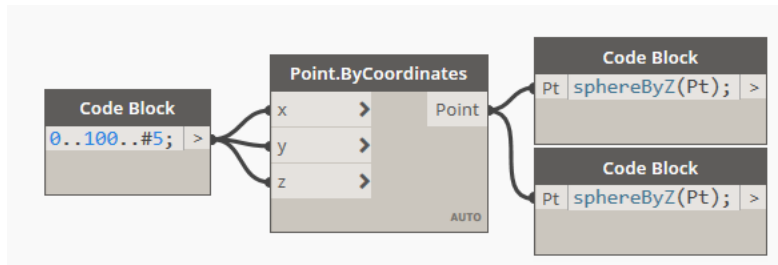


In the following example below you can see how the designsript code can be replicated in the Python node producing the same geometry

```

Code Block
def sphereByZ(inputPt)
{
sphereRadius=inputPt.Z/10;
sphere=Sphere.ByCenterPointRadius(inputPt,sphereRadius);
return = sphere;
};

```

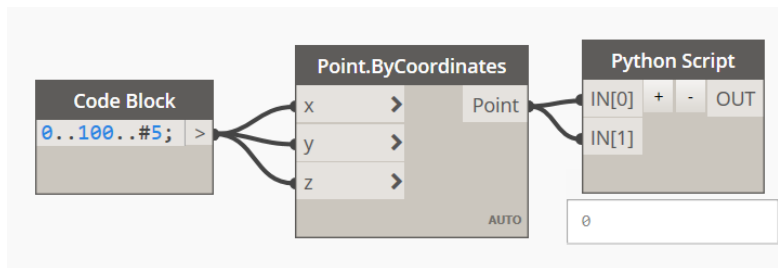


Python Version:

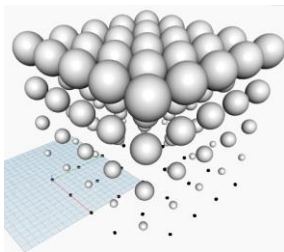
```

1 import clr
2 clr.AddReference('ProtoGeometry')
3 from Autodesk.DesignScript.Geometry import *
4
5 def sphereByZ(inputPt1):
6     sphereRad=Input1.Z/10
7     sphere = Sphere.ByCenterPointRadius(input1, sphereRad)
8     return sphere
9
10 inputPt1 = IN[0]
11 sphereRad = IN[1]
12
13 OUT = 0

```

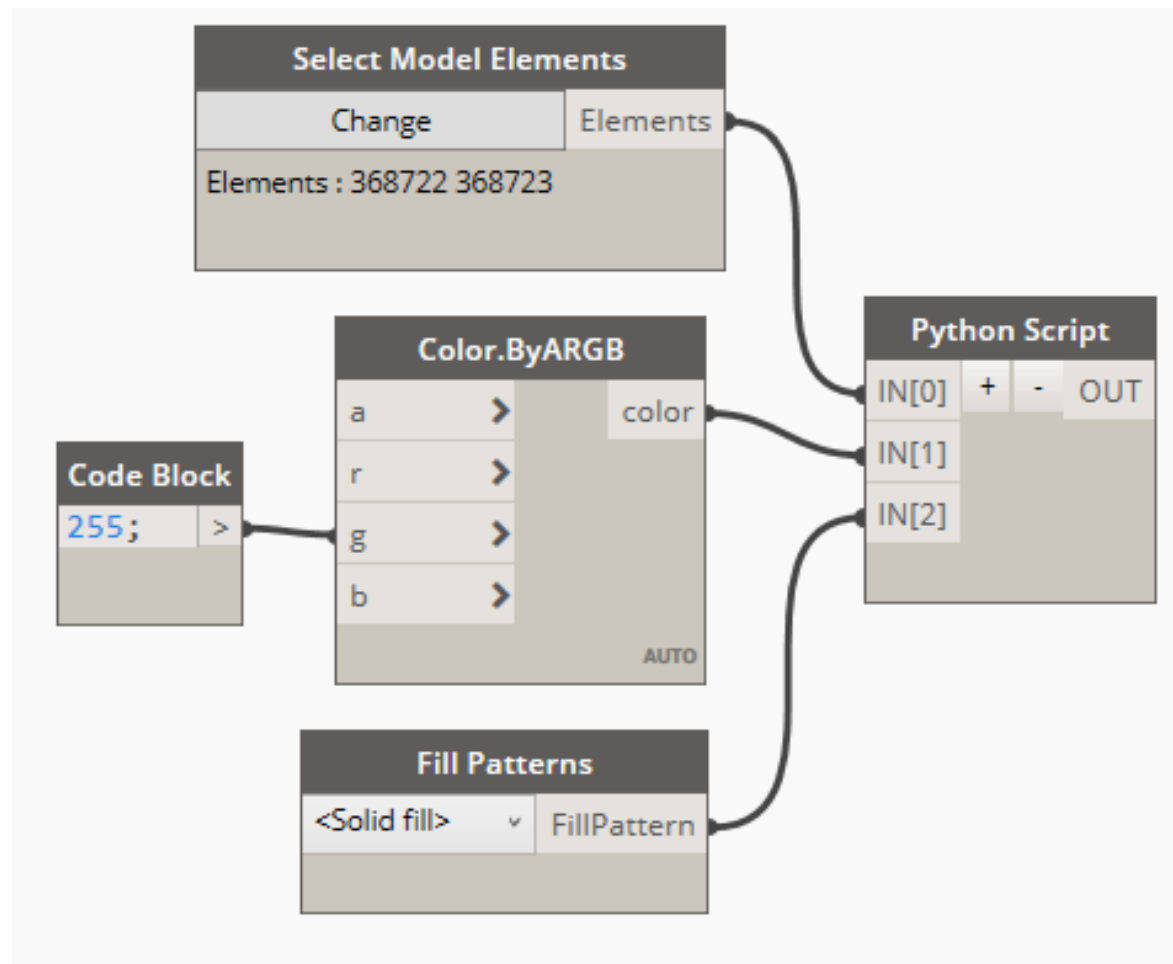


Both designsript and Python create the same geometry



## Exercise 5 – Definitions and Functions

1. In this example the script will use definitions and function to modify settings to the model
2. The color and pattern inputs will override the settings in the selected Revit elements
3. The results can be updated live
4. Import the services necessary for the process
5. Create a definition with def
6. After that add the function
7. Then add the return which stipulates what modifications are supposed to happen
8. You have to know what you need to adjust in order to make the correct change to the elements and parameters in use

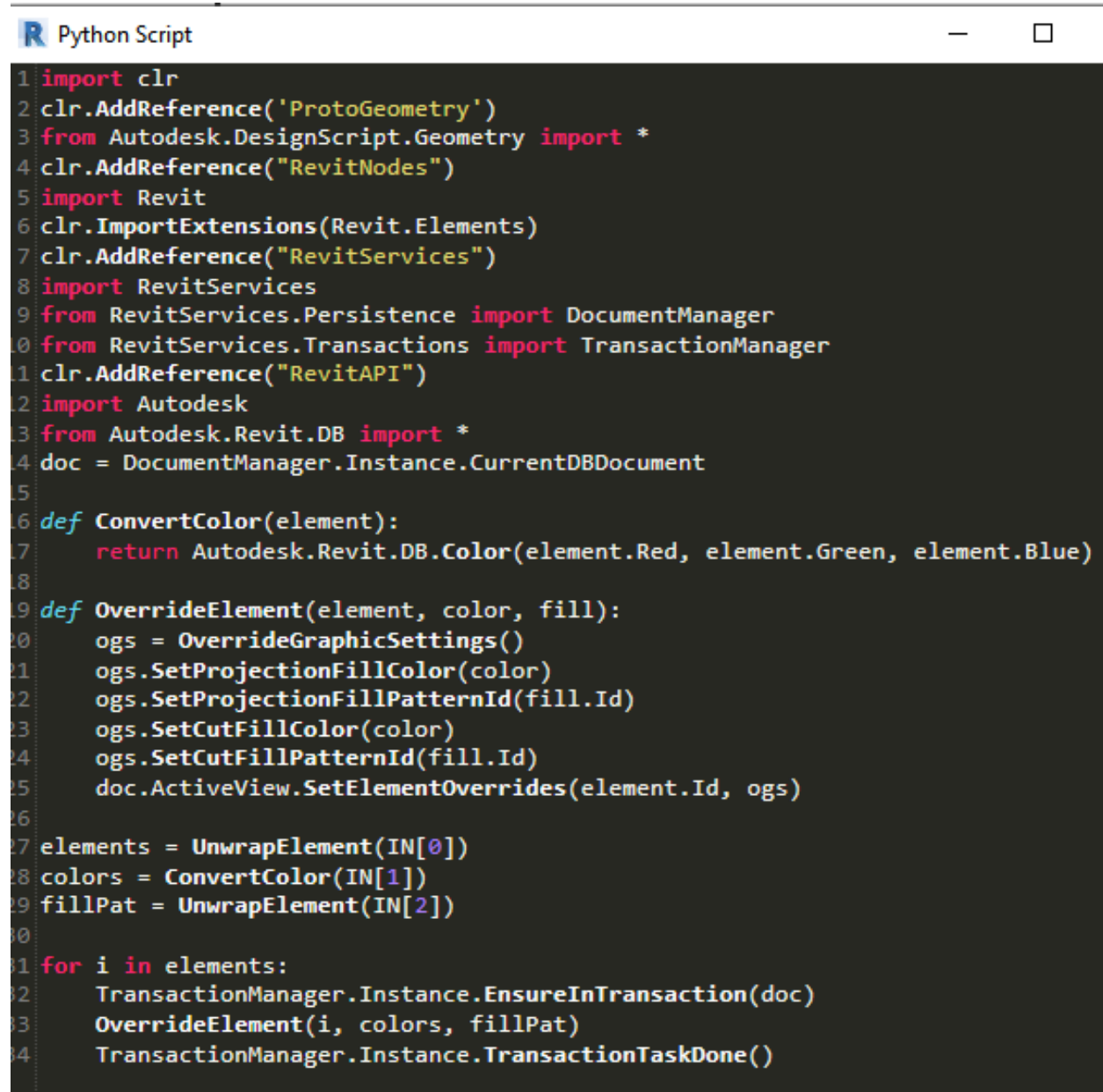




Notice how the definitions are set below.

The values and loop code are normally written though we have no out list.

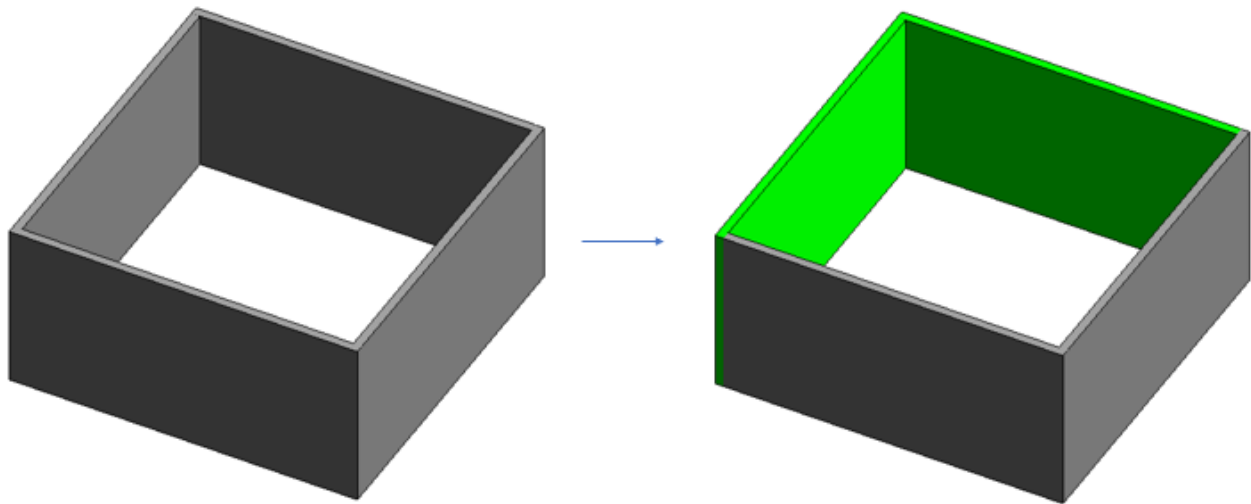
That is because the definitions are running and affecting the element inputs so an out list is unnecessary in this case.



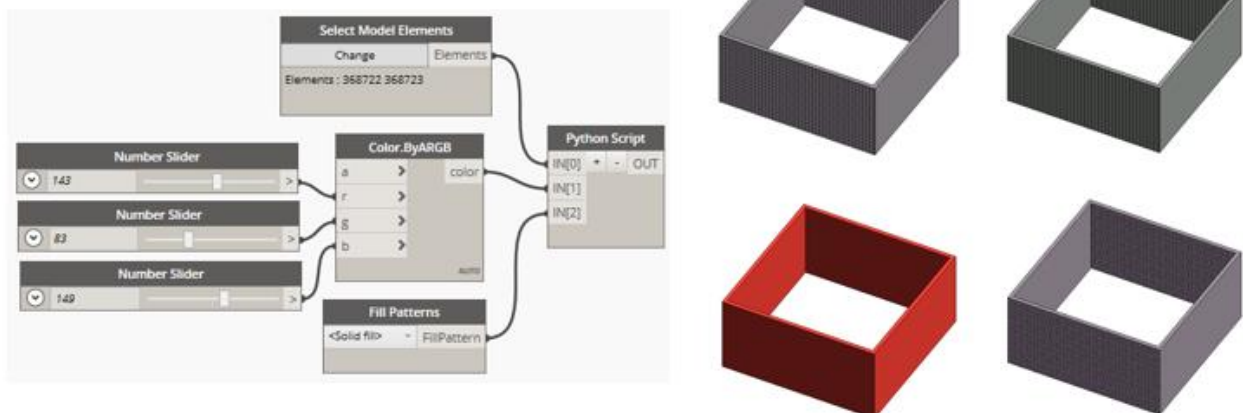
```
Python Script
1 import clr
2 clr.AddReference('ProtoGeometry')
3 from Autodesk.DesignScript.Geometry import *
4 clr.AddReference("RevitNodes")
5 import Revit
6 clr.ImportExtensions(Revit.Elements)
7 clr.AddReference("RevitServices")
8 import RevitServices
9 from RevitServices.Persistence import DocumentManager
10 from RevitServices.Transactions import TransactionManager
11 clr.AddReference("RevitAPI")
12 import Autodesk
13 from Autodesk.Revit.DB import *
14 doc = DocumentManager.Instance.CurrentDBDocument
15
16 def ConvertColor(element):
17     return Autodesk.Revit.DB.Color(element.Red, element.Green, element.Blue)
18
19 def OverrideElement(element, color, fill):
20     ogs = OverrideGraphicSettings()
21     ogs.SetProjectionFillColor(color)
22     ogs.SetProjectionFillPatternId(fill.Id)
23     ogs.SetCutFillColor(color)
24     ogs.SetCutFillPatternId(fill.Id)
25     doc.ActiveView.SetElementOverrides(element.Id, ogs)
26
27 elements = UnwrapElement(IN[0])
28 colors = ConvertColor(IN[1])
29 fillPat = UnwrapElement(IN[2])
30
31 for i in elements:
32     TransactionManager.Instance.EnsureInTransaction(doc)
33     OverrideElement(i, colors, fillPat)
34     TransactionManager.Instance.TransactionTaskDone()
```

Credit: DannySBentley

Running the script can let you change color and texture as those are parameters defined as what the function can change in each element.

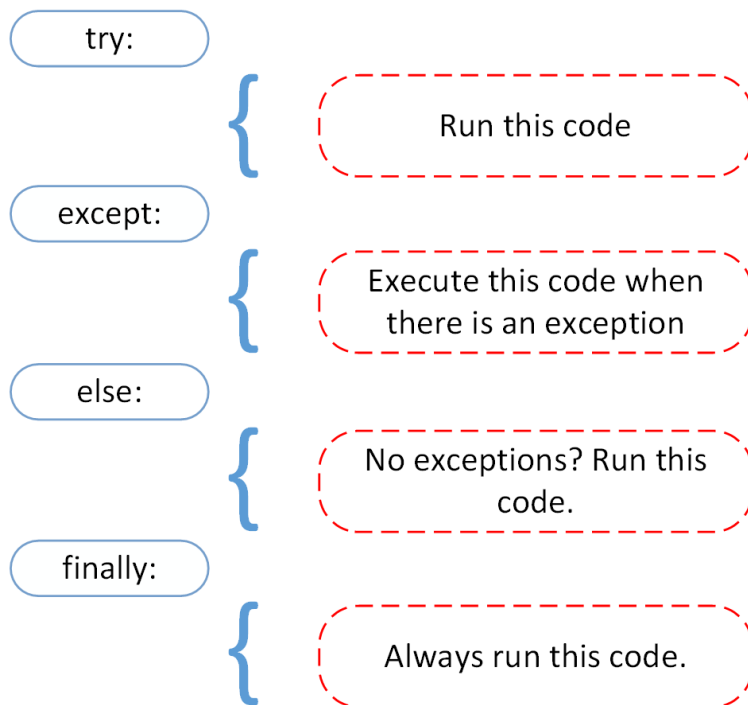


Adding sliders and changing the settings can show results in real time so you can find the nuance you want in the output



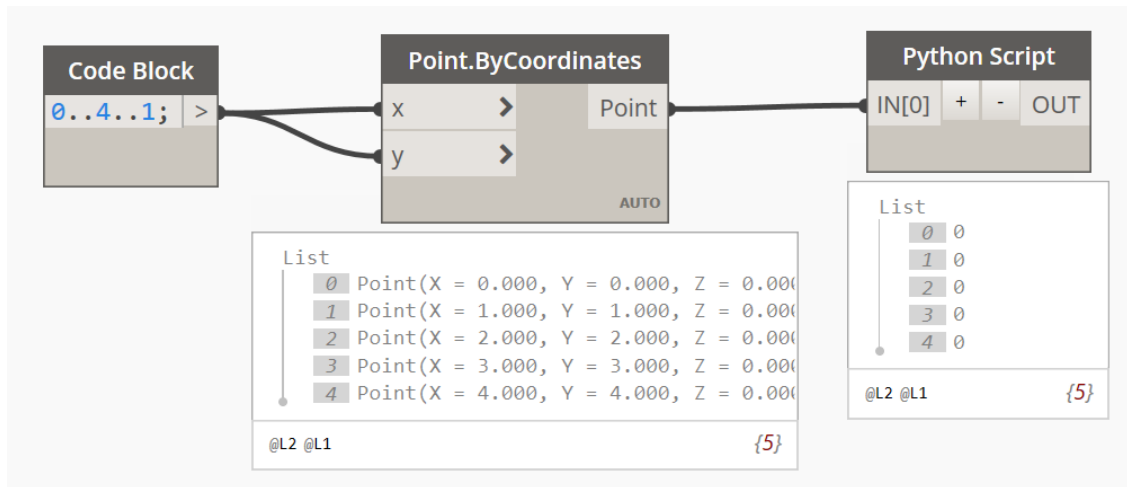
## Exercise 6 – Try and Except

- Sometimes when you run loops there are errors that can't be ignored
- What you want to do is break out of that code and continue the operation
- You can make a switch with an except function



This isn't the first thing you would write into code.  
It usually comes up when your initial idea didn't work out as planned.  
The exception makes for a convenient way to move on from a problem index and continue working.

In the example below there is a list range which creates coordinates without issue since every point has an X and Y which is what the Python script is looking for.  
If there is no Y value then a message appears to notify you to add a Y value but the script will not fail due to one index not meeting the criteria.

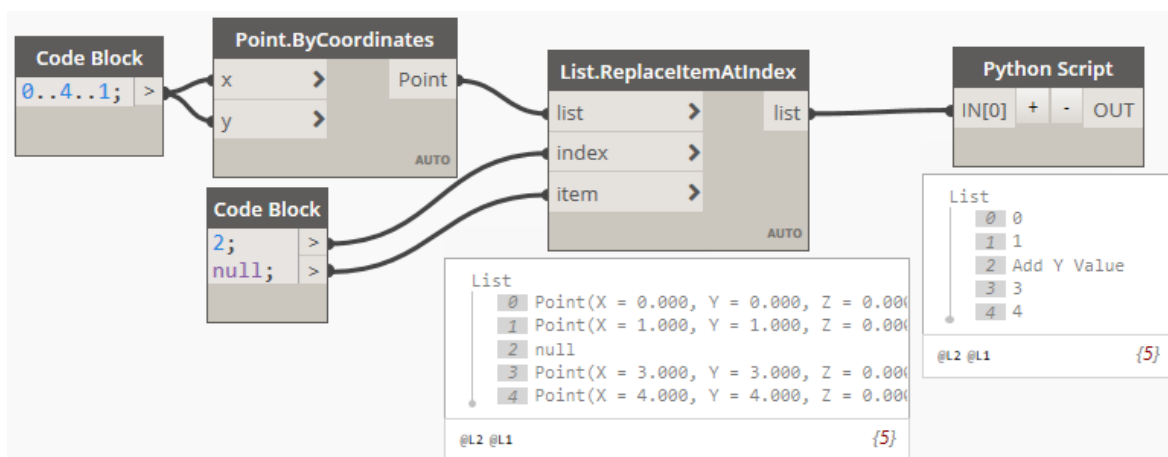


```

1 import clr
2 clr.AddReference('ProtoGeometry')
3 from Autodesk.DesignScript.Geometry import *
4
5 PTS = IN[0]
6 outList = []
7
8 for PT in PTS:
9     try:
10        outList.append(PT.Y)
11    except:
12        outList.append("Add Y Value")
13
14 OUT = outList

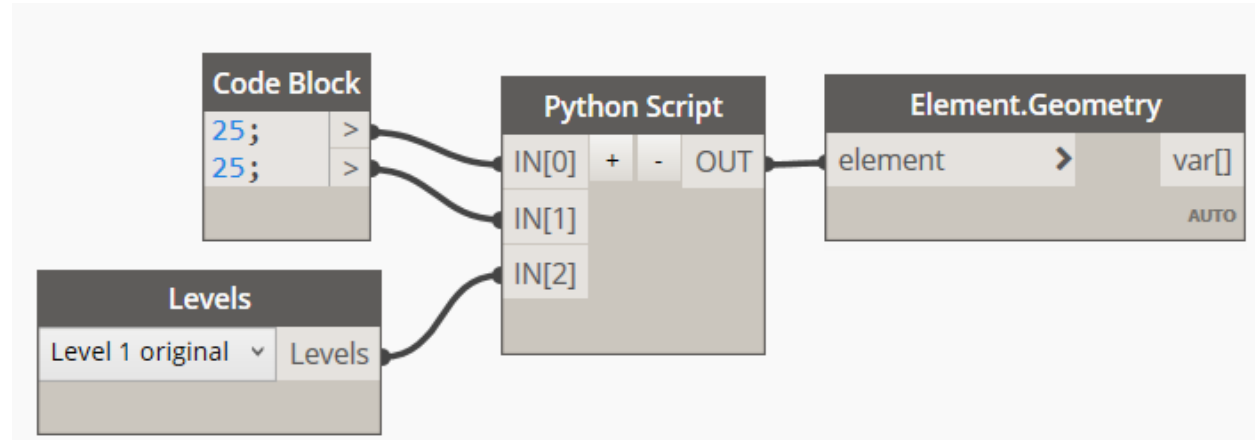
```

In the modification below the 2<sup>nd</sup> index is completely removed but the script runs because we accommodated the exception with the except function.



## Exercise 6 – Try and Except

- Try and except ends up being used when you need to continue from iterations that won't complete operations successfully
- You probably won't realize it's necessary until you run into problems with your script
- Even a simple output like connecting points to make a wall can be challenging if you can't get the outputs functional



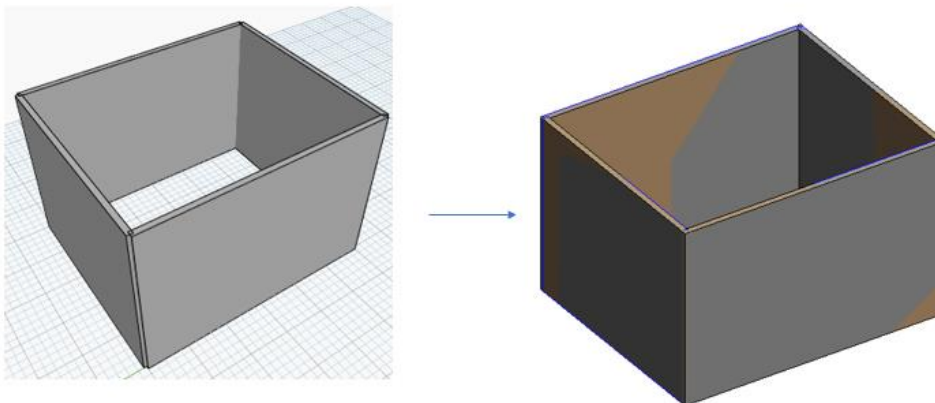
1. In this code we want to create a 4 corner wall
2. Inputs and values are stated
3. Then define the list
4. We want to enumerate points with a range function using a for loop
5. The except function will allow the break and provide a different option when the code is run to complete the process
6. Then the final lines of code finish the transaction which completes the list

```

Python Script
1 import clr
2
3 clr.AddReference("RevitNodes")
4 import Revit
5 clr.ImportExtensions(Revit.Elements)
6
7 clr.AddReference("RevitServices")
8 import RevitServices
9 from RevitServices.Persistence import DocumentManager
10 from RevitServices.Transactions import TransactionManager
11 doc = DocumentManager.Instance.CurrentDBDocument
12
13 clr.AddReference('RevitAPI')
14 from Autodesk.Revit.DB import XYZ, Line, Wall
15
16 width = IN[0]
17 height = IN[1]
18 level = UnwrapElement(IN[2])
19
20 pt1 = XYZ(0, 0, 0)
21 pt2 = XYZ(width, 0, 0)
22 pt3 = XYZ(width, height, 0)
23 pt4 = XYZ(0, height, 0)
24
25 pts = [pt1, pt2, pt3, pt4]
26 walls = []
27
28 TransactionManager.Instance.EnsureInTransaction(doc)
29 for n, pt in enumerate(pts):
30     try:
31         wall_line = Line.CreateBound(pt, pts[n+1])
32     except IndexError:
33         wall_line = Line.CreateBound(pt, pts[0])
34     wall = Wall.Create(doc, wall_line, level.Id, False)
35     walls.append(wall.ToDSType(False))
36
37 TransactionManager.Instance.TransactionTaskDone()
38
39 OUT = walls

```

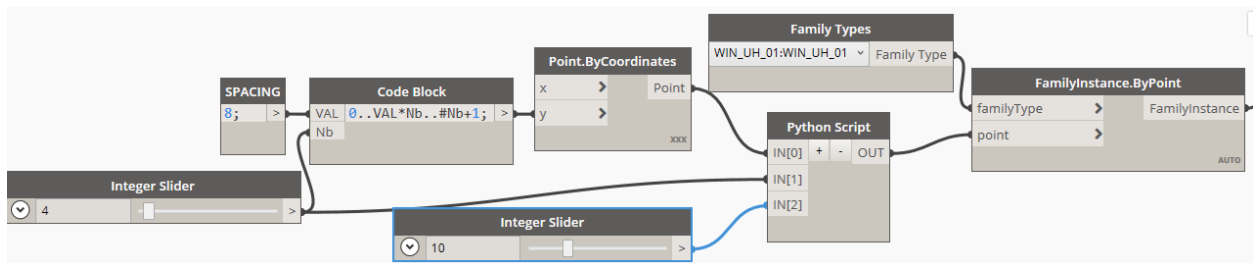
The more restraints an element has the more you must consider in your program



Sometime experimentation and alternative approaches are necessary to make the code work

## Exercise 7 – Graphs

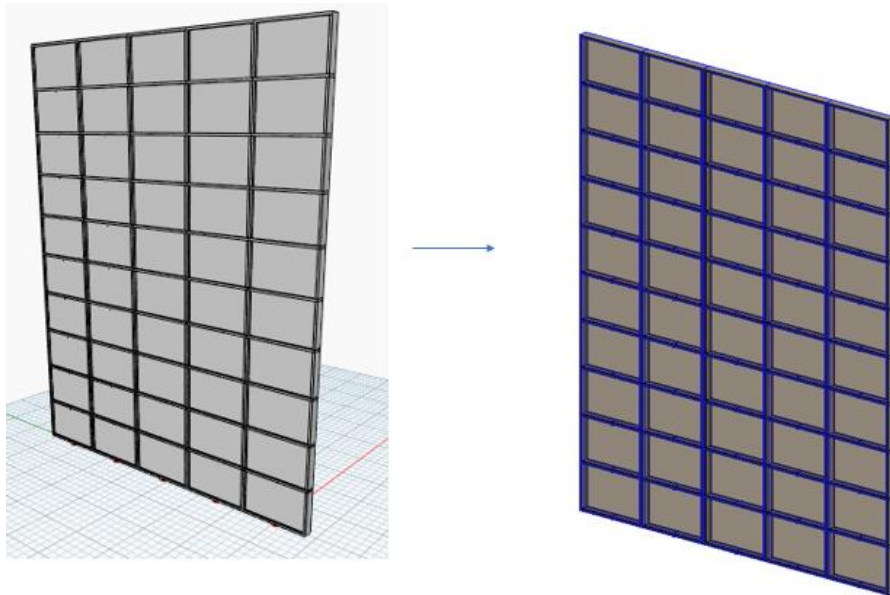
- Python nodes don't have to be standalone with a few inputs
- Often they can be part of a larger script serving a single purpose
- The key is to make sure the entire node graph is easy to follow and your Python node inputs are clearly defined for its purpose in the rest of the graph
- With modifications you can do some design review in real time
- Adding sliders can provide some live results as you experiment with your code
- Depending on how you create your code you can have it all contained within the Python node or distributed throughout different nodes
- Consider the process that is easier to replicate and if someone else can modify it



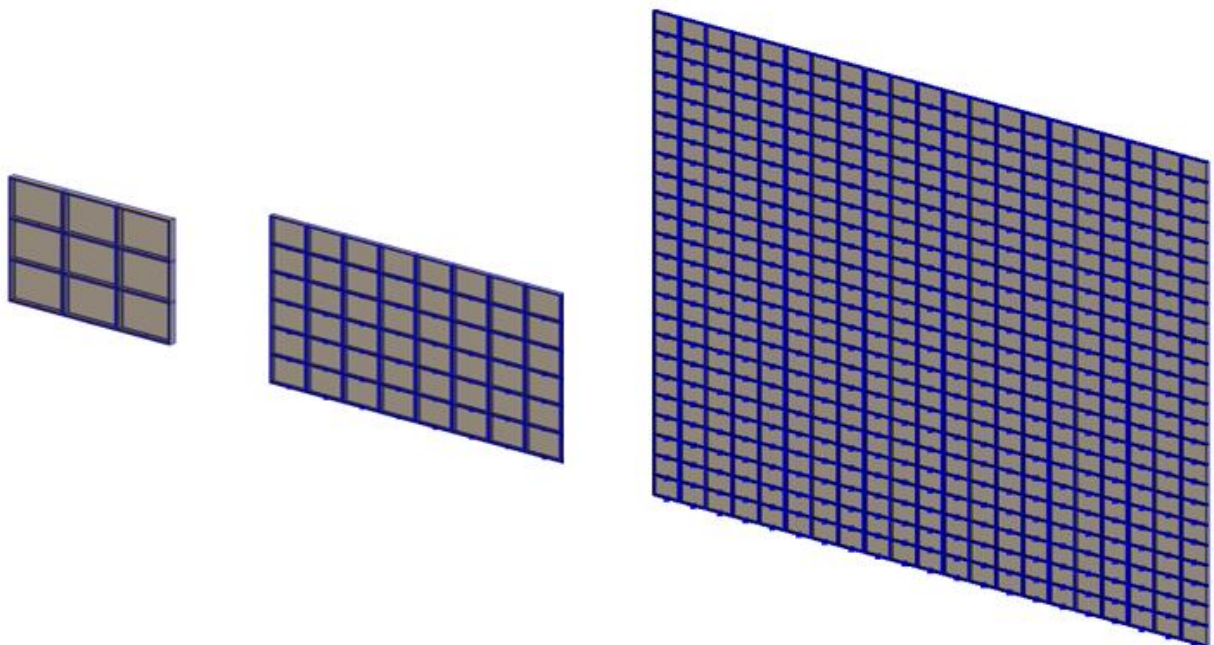
1. A code block is part of the values before entering the Python node
2. This example uses the inputs to loop new points for a design study
3. You can see variables like j and D being used in different ways to create the design study
4. The number sliders can quickly adjust the input and see a different result
5. If necessary you can modify the code on the fly to try different variations

```
1 # Enable Python support and load DesignScript library
2 import clr
3 clr.AddReference('ProtoGeometry')
4 from Autodesk.DesignScript.Geometry import *
5
6 # The inputs to this node will be stored as a list in the IN variables.
7 NodeList = IN[0]
8 CopyNumber = IN[1]
9 NumberNodes = IN[2]
10
11 NewPoints = []
12
13 Vertical = 5
14
15 # Place your code below this line
16 for j in range(0, CopyNumber):
17     for i in range(0, NumberNodes+1):
18         D = NodeList[i];
19         E = Geometry.Translate(D,0,0,Vertical*j);
20         NewPoints.append(E)
21
22 # Assign your output to the OUT variable.
23 OUT = NewPoints
```

Based on the inputs and the way the code is written you can modify the position of the elements depending on the family input size and shape allowing for more flexibility.



You can experiment with different functions, formulas and settings to change the results.



The important thing to remember is to keep the code clear and write notes and comments as necessary to explain what you are doing in case anyone else has to work on it. Or if you forget.

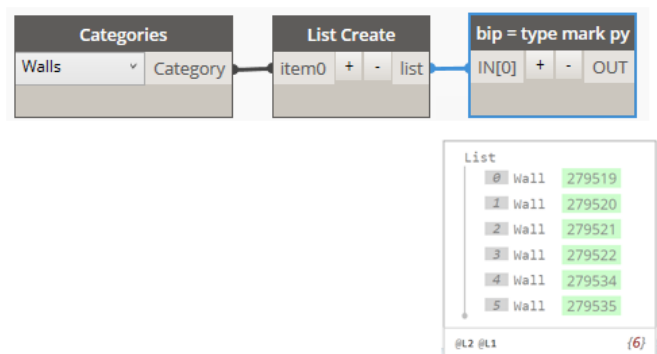
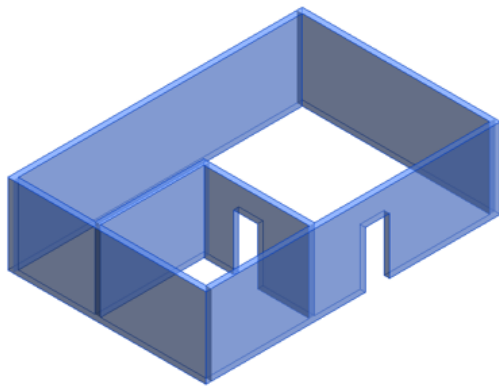


## Problem Solving

- When you create code for the first time it is not always obvious what you have to use as a function
- Or you want to modify existing code to do something else but not sure where to start
- Revit API docs is where you should start looking
- Find the function you want to change and search for it in the Website



1. Typical scenario is you get a script that you did not author and you want to modify it to something different
2. For example this script looks up an element's Type Mark parameter but you want to see the comment parameter
3. In this example the script finds the wall Type Mark but we don't want all the walls just the ones with comments
4. If you don't change the way the code reads the parameter you can't select what you want



1. Open the script
2. You see that the only input is coming from the built in parameter
3. This refers to the Type Mark
4. That parameter is being evaluated for the string 'A6' and return the list of elements which contain it
5. If we can change the built in parameter and string we can change the results



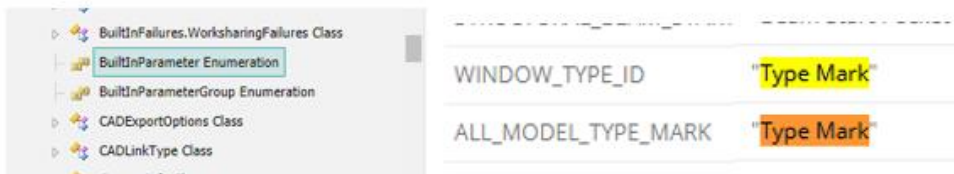
```
1 import clr
2 clr.AddReference('RevitAPI')
3 from Autodesk.Revit.DB import *
4 import Autodesk
5
6 clr.AddReference('RevitNodes')
7 import Revit
8 clr.ImportExtensions(Revit.GeometryConversion)
9 clr.ImportExtensions(Revit.Elements)
10
11 clr.AddReference('RevitServices')
12 from RevitServices.Persistence import DocumentManager
13 from RevitServices.Transactions import TransactionManager
14
15 doc = DocumentManager.Instance.CurrentDBDocument
16
17 bip = BuiltInParameter.ALL_MODEL_TYPE_MARK
18 provider = ParameterValueProvider(ElementId(bip))
19 evaluator = FilterStringEquals();
20 rule = FilterStringRule(provider, evaluator, "A6", False);
21 filter = ElementParameterFilter(rule);
22 walls = FilteredElementCollector(doc).OfClass(Wall).WherePasses
23 (filter).ToElements()
24
25 OUT = walls
```

1. In this case you are looking for the 'BuiltInParameter' function because that is what is running through all the element ID's for them mark parameter value
2. Go to Revitapidocs.com for the documentation
3. Search for the BuiltInParameter and you'll get a result to click on
4. Search the list for mark parameter to see if it exists
5. Then do the same for the comment parameter
6. You now see the function for comments to copy into the code

```

7 bip = BuiltInParameter.ALL_MODEL_TYPE_MARK
8 provider = ParameterValueProvider(ElementId(bip))
9 evaluator = FilterStringEquals();
10 rule = FilterStringRule(provider, evaluator, "A6", False);

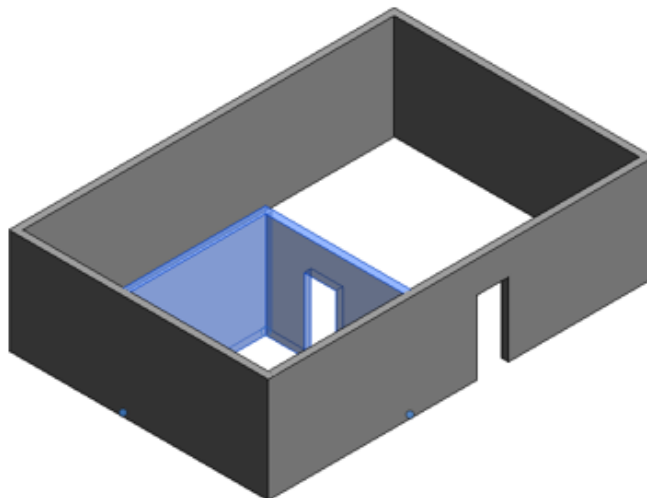
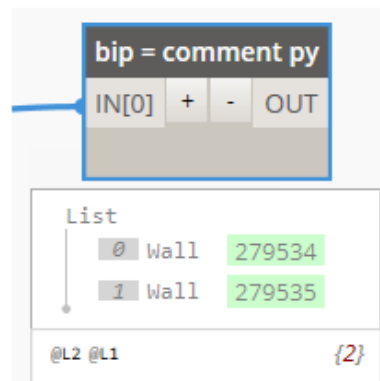
```



ALL_MODEL_MANUFACTURER	"Manufacturer"
ALL_MODEL_INSTANCE_COMMENTS	"Comments"
ALL_MODEL_TYPE_COMMENTS	"Type Comments"

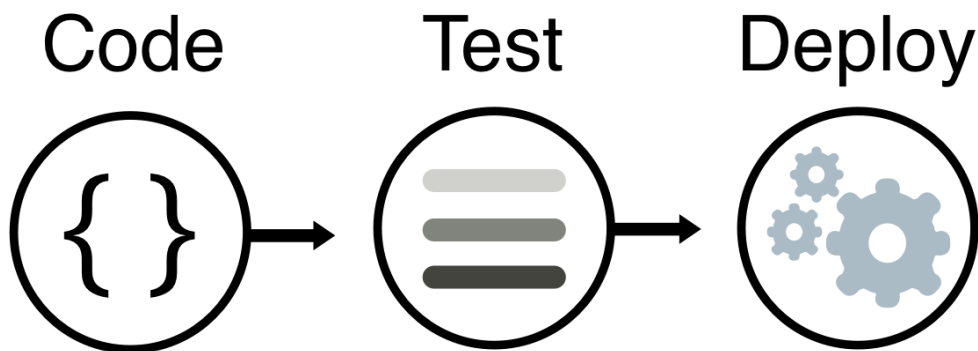
1. Copy Paste the comment parameter into the code replacing the Type Mark parameter
2. Look for the text contained in the Revit element comments field and write it in the Python code string value
3. Run the script and you'll see the elements listed with that value
4. Modifications like this are typical for getting one purpose made code to do another task based on document research

```
7 bip = BuiltInParameter.ALL_MODEL_INSTANCE_COMMENTS  
8 provider = ParameterValueProvider(ElementId(bip))  
9 evaluator = FilterStringEquals();  
10 rule = FilterStringRule(provider, evaluator, "Review", False);
```

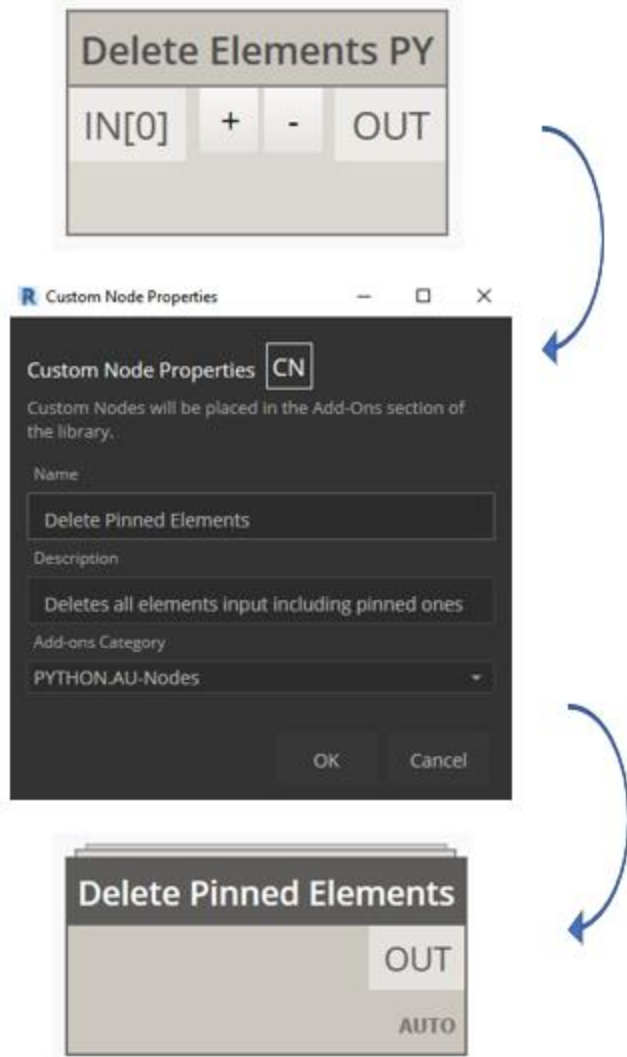


## Distribution

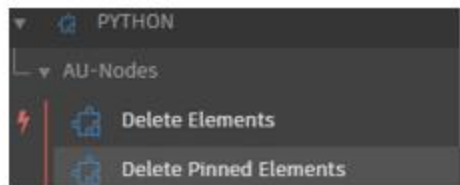
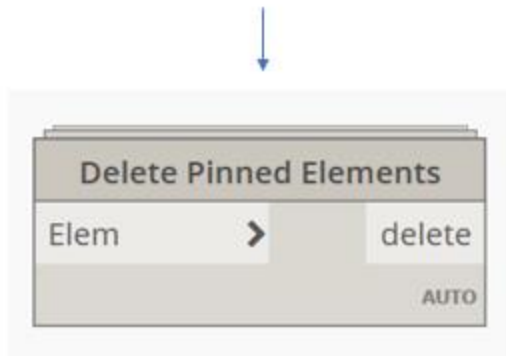
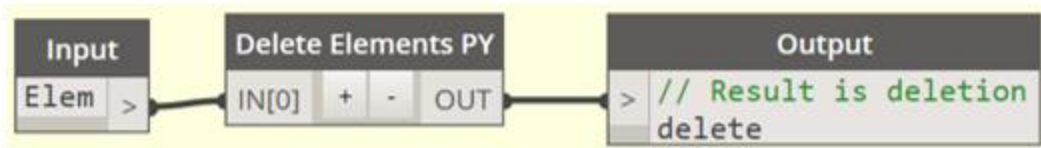
- Ideally you share the code you write since it can save someone else the trouble of doing it since you solved a problem in that code.
- There are many ways to share code but you want a consistent process that will ensure the contents are copied over reliably and available to others without hassle.
- Covered here is the Dynamo package distribution and 3<sup>rd</sup> party websites to support that distribution



1. If you want to keep a copy of the script in your library or share it then make your own node
2. It's an easy process with a given selection of nodes even just one like the python script
3. Select the node, right click outside the node then choose to create a custom node
4. Input the Name, description and Category from an existing list or type in your own



1. A new tab will appear in Dynamo where you can edit that node
2. You want to assign an input and output so the node knows to take either
3. Then it will load in your new node with inputs and outputs
4. You can create as many inputs as you want so long as you assign what kind of inputs they are
5. Then you can see your node in the library and load it anytime



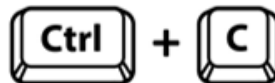
The next step would be to distribute it on the Dynamo package manager. You can search and share for the online database of packages with that system. Keep in mind that the packages need to be regularly updated or they won't be useful after a while.

1. Or just host it on Github
2. All your Dynamo and Python code can be hosted here for distribution and collaboration
3. The most important skill for coding is finding solutions and adding them to your project



20 lines (16 sloc) | 471 Bytes

```
1 # Enable Python support and load DesignScript library
2 import clr
3 clr.AddReference('ProtoGeometry')
4 from Autodesk.DesignScript.Geometry import *
5 clr.AddReference('RevitNodes')
6 from Revit.Elements import *
7
8 famtype = IN[0]
9 pbc = Point.ByCoordinates(0,0,0)
10 output = []
11
12 for x in range(0, 100, 20):
13     for y in range(0, 100, 20):
14         for z in range(0, 100, 20):
15             pbc = Point.ByCoordinates(x,y,z)
16             col = FamilyInstance.ByPoint(famtype,pbc)
17             output.append(col)
18
19 OUT = output
```



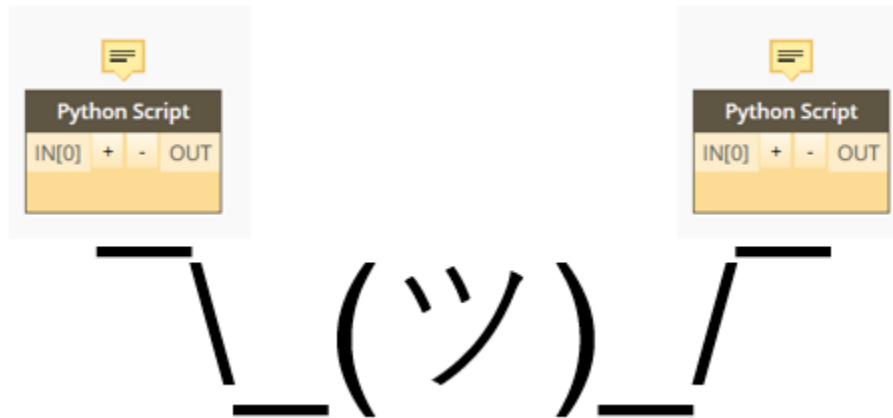
PS - XYZ matrix Family PY

```
1 # Enable Python support and load DesignScript library
2 import clr
3 clr.AddReference('ProtoGeometry')
4 from Autodesk.DesignScript.Geometry import *
5 clr.AddReference('RevitNodes')
6 from Revit.Elements import *
7
8 famtype = IN[0]
9 pbc = Point.ByCoordinates(0,0,0)
10 output = []
11
12 for x in range(0, 100, 20):
13     for y in range(0, 100, 20):
14         for z in range(0, 100, 20):
15             pbc = Point.ByCoordinates(x,y,z)
16             col = FamilyInstance.ByPoint(famtype,pbc)
17             output.append(col)
18
19 OUT = output
```



## Debug

1. Debugging will probably take up more time than you realize
2. Python will tell you what the issue is and which line to find it on
3. Syntax – colons, brackets, parentheses, quotations, indents, etc all effect your code
4. Punctuation – capital and lowercase makes a big difference in your code especially for variables
5. Import – modules and services need to be called or code may not execute
6. Operators – using the correct operators in the right place may not always be obvious so try them in isolation to check a run



## **Limitations to Consider**

As you continue to use Python and Dynamo you will be able to create your own code from the ground up. It is worth mentioning that not everything is going to be easy to solve. Sometimes what you want to do might end up being more complex than you realize.

Here's some things of what you can expect:

### **Dynamo is Iterative**

With every release or update of Revit the API can change a little bit. For anyone used to coding plugins or scripts for Revit this is a known issue. If the API function changes then your scripts have to be updated in order to work with newer releases.

### **Python in Dynamo relies on the Revit API**

Dynamo lets you interface with the Revit API and Python nodes can let you use the API functions directly in the code. As much previously the API content can change and your code will need updates. However if you are coding then you always need to keep up to date on the Revit API releases and how they function.

### **Python node script is different from regular Python script**

While mostly the same there are a few limitations with Python in Dynamo compared to regular scripting. First there is no 'print' command just an 'OUT' command. Because this is all visual scripting all content made has to exist in a node which has inputs and outputs. Python is no exception so there is an OUT which you need to address. Not a big hurdle you just have to remember to have a different execution process with Python in Dynamo but that can lead to more debugging issues.

### **Script deployment can be an issue at larger organizations**

Making scripts for yourself or a small group is easy to maintain. The problem is when you have hundreds of people across multiple computer types and offices. You may need to create script that are adapted to scaling in an environment like that.

### **You may be creating this code solo within a given team**

There is a big community online of people who are coding with Dynamo but you may be a community of one in your office. If scripts don't work that usually means you are the only one fixing problems so be prepared for that workload.

## Additional Learning

Dynamo Primer - [https://primer.dynamobim.org/10\\_Custom-Nodes/10-4\\_Python.html](https://primer.dynamobim.org/10_Custom-Nodes/10-4_Python.html)

Dynamo Forums - <https://forum.dynamobim.com/>

Autodesk University - <https://www.autodesk.com/autodesk-university/au-online?query=PYTHON>

LinkedIn Learning - <https://www.linkedin.com/learning/dynamo-for-revit-python-scripting-2/jumping-into-python-and-dynamo>

YouTube Channels (Danny Bentley) - <https://www.youtube.com/channel/UC1Dx-iGyRbwHzZ8ZyGWF5w>

Automate the boring stuff with Python - <https://automatetheboringstuff.com/>

## Contact Information for Questions

Twitter: [@tadeh\\_hakopian](https://twitter.com/tadeh_hakopian)

LinkedIn: <https://www.linkedin.com/in/thakopian/>

Github: <https://github.com/thakopian>

