Exploring Python Nodes in Dynamo

SD322695

Tadeh Hakopian

AUTODESK UNIVERSITY

INTRODUCTION

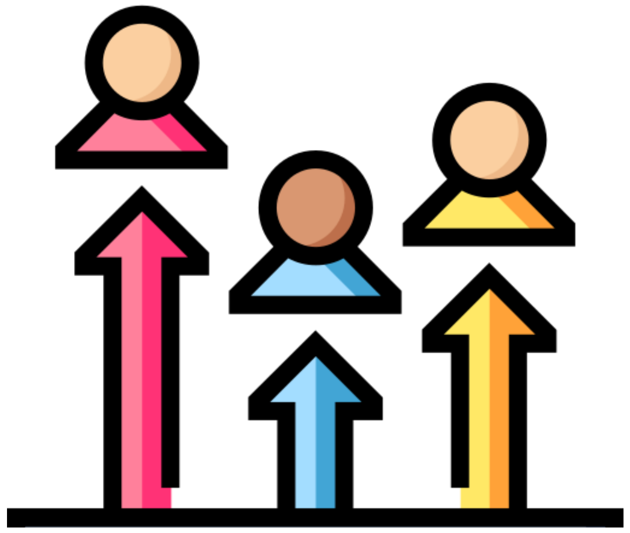
CLASS COMPOSITION

What do you do?

Who knows Dynamo scripting?

Who knows any kind of coding?

Who knows Python coding?



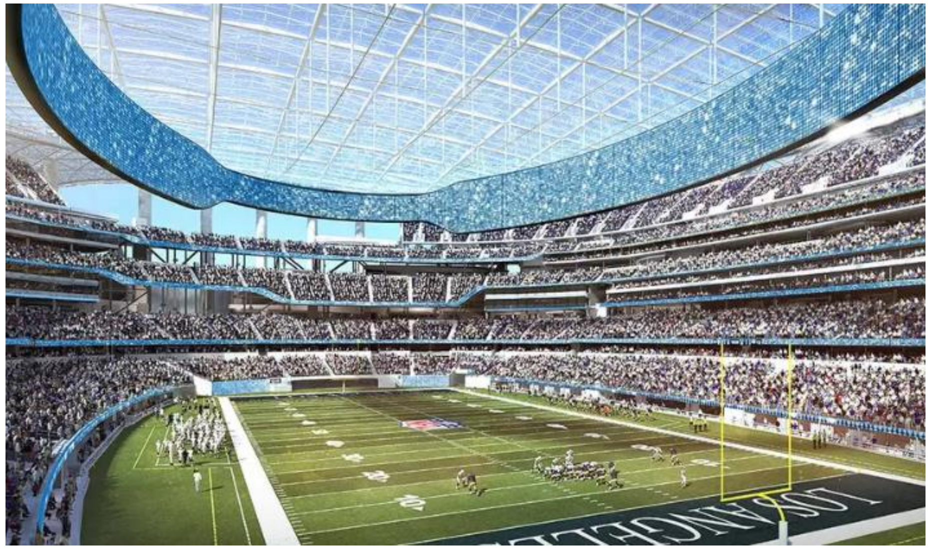
Bitly and QR code to latest version of this course

http://bit.ly/2DyIX2i

About the speaker

Tadeh (Todd-A) Hakopian

Tadeh 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 and his local community meetups. Current Professional Goals Help move the AEC profession into new horizons using value driven solutions and innovative research.



HKS Architects

Large complex projects which take advantage of every technique and tool we have to help us manage the model data

Tools like Dynamo and coding have made a difference for us to perform and deliver on our designs.

LEARNING OBJECTIVES

1

Learn about the python node in dynamo and how to configure it

2

Learn how to debug your python script when errors appear

3

Learn how to execute code in python with statements and conditions

4

Create python scripts for new editing and designs capabilities

Description

What this class is about

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.

Purpose of this Class

What you can get out of learning

- Since Dynamo has been released the community around it has grown year after year.
- There has been a lot of interest in scripting graphs and increasingly writing code.
- Now that people are providing coding examples there is a new way to get work done in Dynamo with Python code.
- One gap I have noticed is that the average user isn’t familiar enough with the Python code to use the examples out there.
- That motivated me to create instruction on how to use Python at an introductory level for Dynamo with examples.
- Therefore this class will go over essentials of using Python in Dynamo with examples to help orient users on the methods to try the code themselves.
- Most of the examples are meant to be accessible by the average user to promote the understanding of basic Python coding use cases in Dynamo.

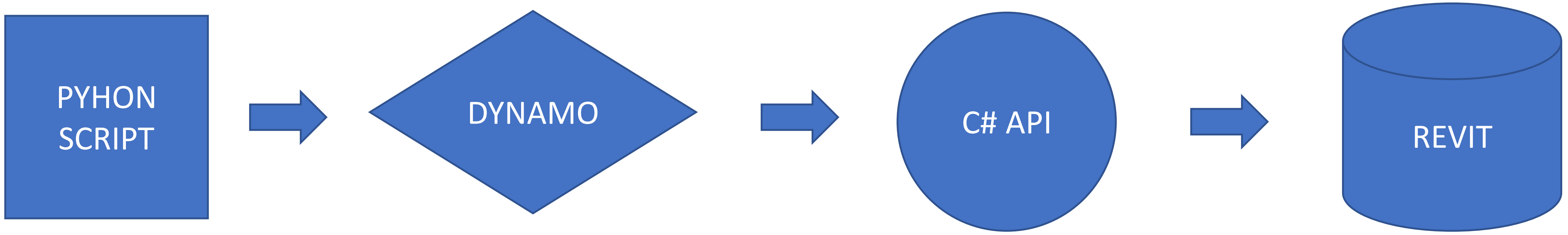
PREPARATION

WHY USE PYTHON IN DYNAMO

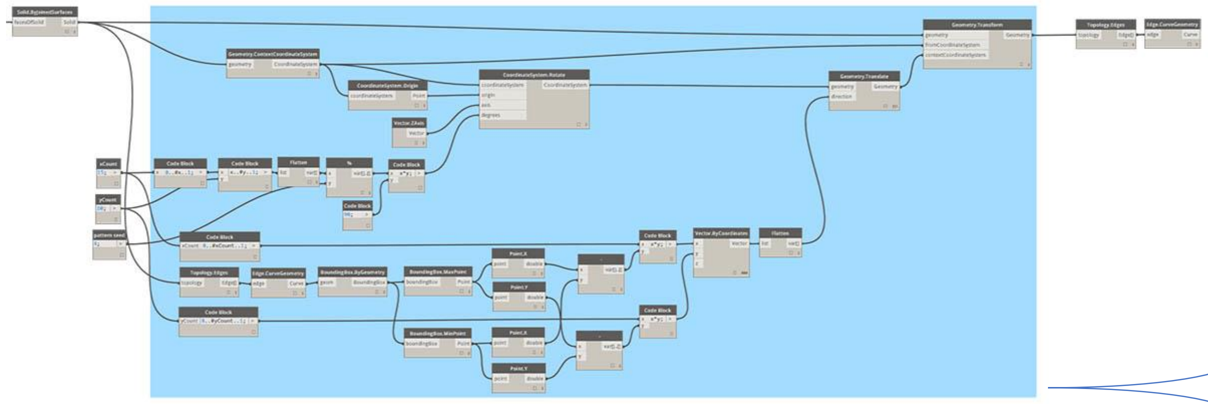
?

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



WHY USE PYTHON IN DYNAMO



```
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

solid = IN[0]
seed = IN[1]
xCount = IN[2]
yCount = IN[3]

solids = []

yDist = solid.BoundingBox.MaxPoint.Y-solid.BoundingBox.MinPoint.Y
xDist = solid.BoundingBox.MaxPoint.X-solid.BoundingBox.MinPoint.X

for i in xRange:
   for j in yRange:
      fromCoord = solid.ContextCoordinateSystem
      toCoord =
fromCoord.Rotate(solid.ContextCoordinateSystem.Origin,Vector.ByCoordinates
(0,0,1),(90*(i+j%val)))
      vec = Vector.ByCoordinates((xDist*i),(yDist*j),0)
      toCoord = toCoord.Translate(vec)
      solids.append(solid.Transform(fromCoord,toCoord))

OUT = solids
```

COMPUTER AND SOFTWARE SETUP

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 is class is structured as a demonstration not a lab 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

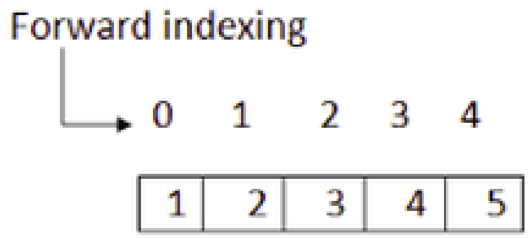
EXERCISES

EXERCISE PREP

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 ***

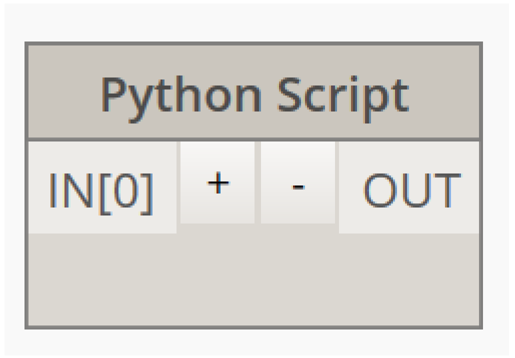
- Problem solving with Documentation
- Distribute
- Debugging



EXERCISE 0
ESSENTIALS

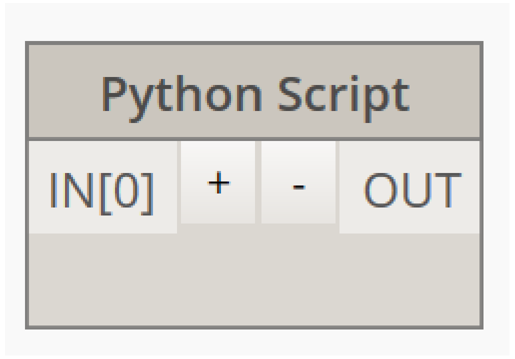
EXERCISE 0 - ESSENTIALS

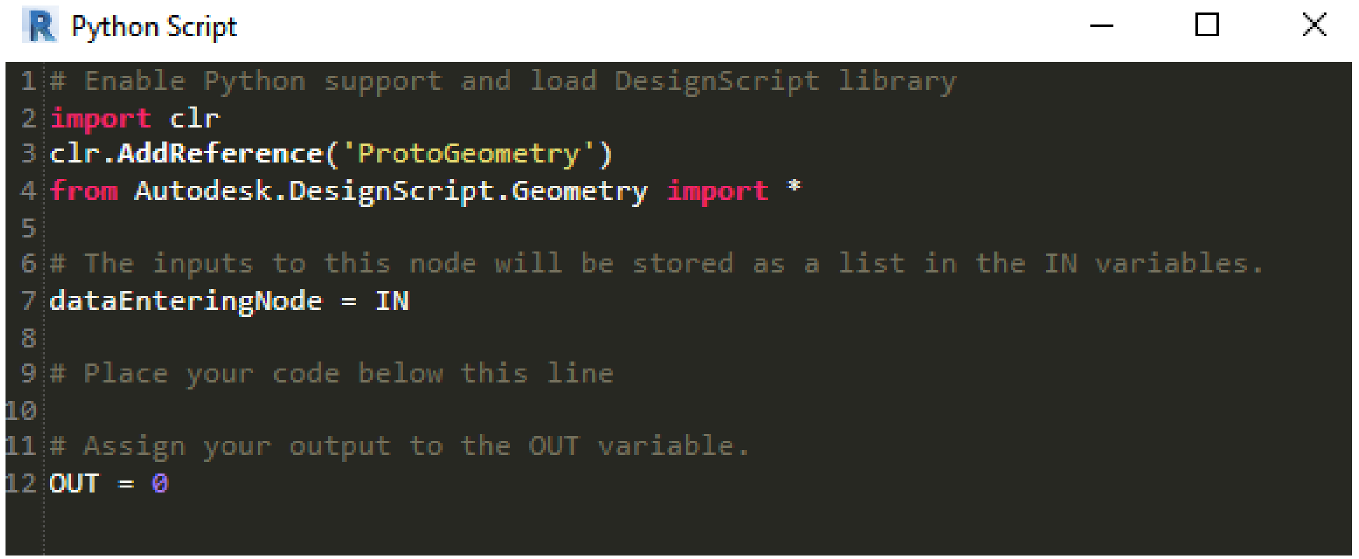
- Dynamo already has the Python Node built into it
- This version of Python is Iron Python which works with C# perfect for the Revit API
- This node allows you to code within Dynamo without switching to the Revit API



EXERCISE 0 - ESSENTIALS

- 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 of 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

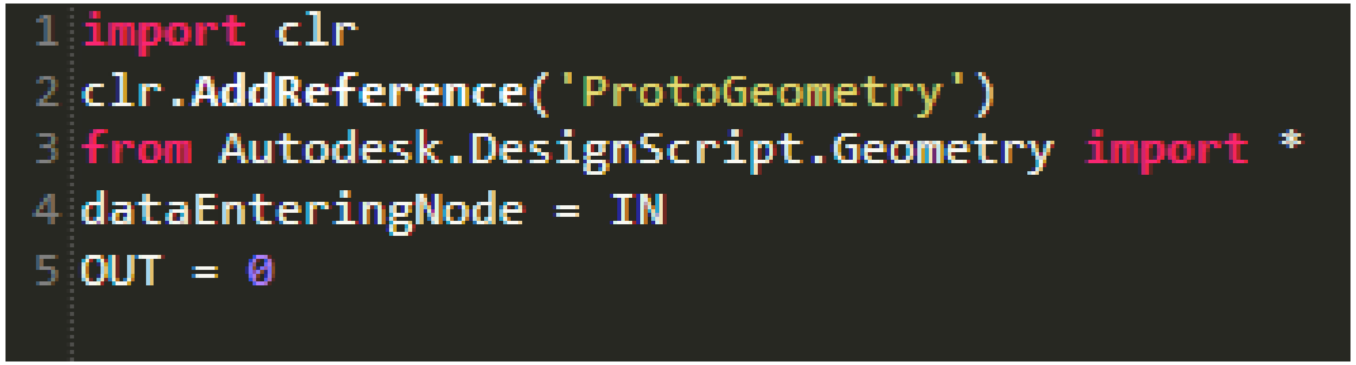




```
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

# The inputs to this node will be stored as a list in the IN variables.
dataEnteringNode = IN

# Place your code below this line

# Assign your output to the OUT variable.
OUT = 0
```

EXERCISE 0 - ESSENTIALS

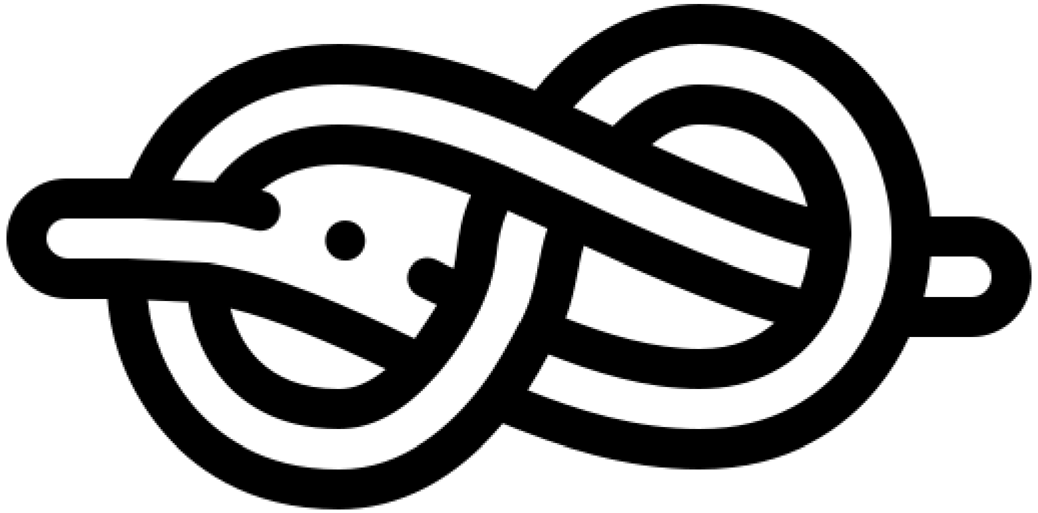
- The Boiler plate code provides essentials and some examples of how to get started
- Add References you need to work with the packages
- Add an Input
- Add an Output
- All the rest of your code goes in between
- That is the essentials of how any Python node works



```
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
dataEnteringNode = IN
OUT = 0
```

EXERCISE 0 - ESSENTIALS

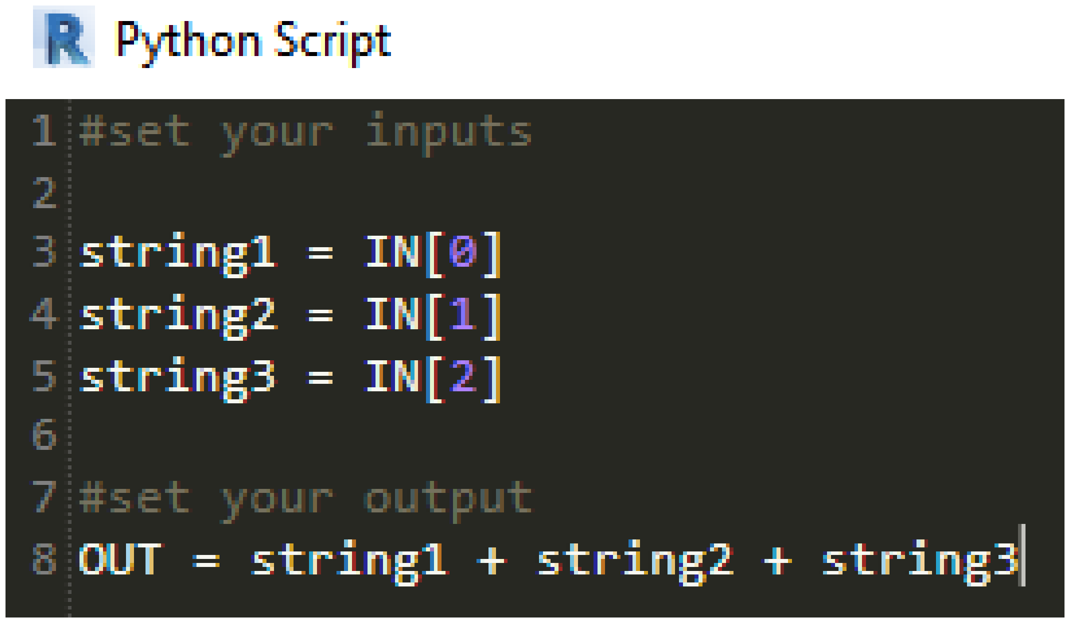
```
import clr

# Bring in the packages you need

clr.AddReference('ProtoGeometry')

from Autodesk.DesignScript.Geometry import *

# Any input to the Python Node itself

dataEnteringNode = IN

# Place code in the next section

for x in elements():

       ……

# The result

OUT = 0
```



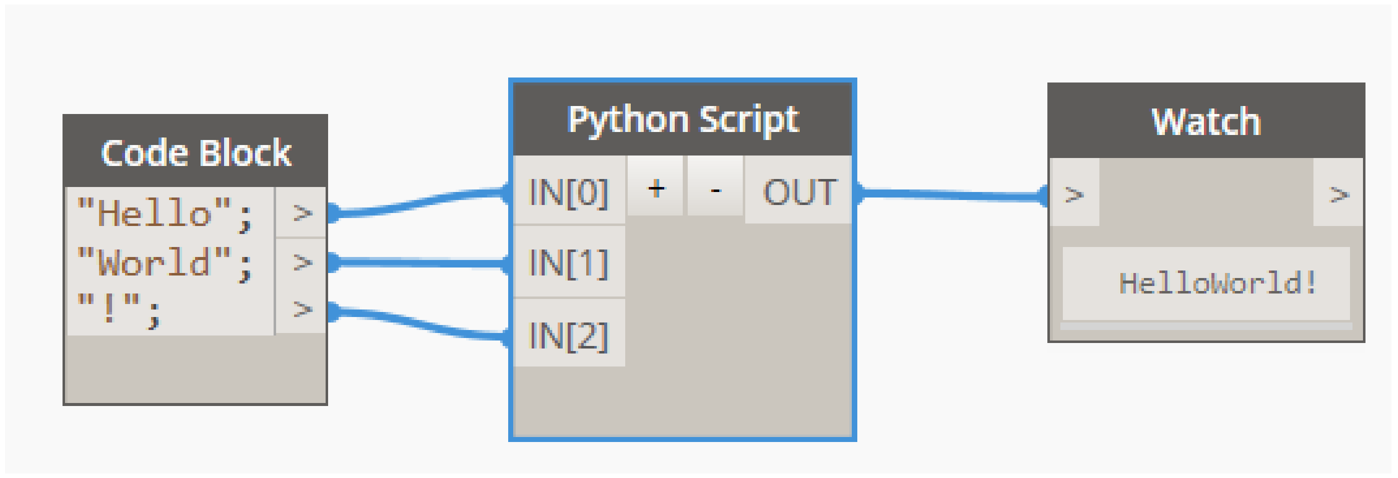
Let’s Make Some Strings!

EXERCISE 0 – INPUTS AND OUTPUTS

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


Python Script

```
#set your inputs

string1 = IN[0]
string2 = IN[1]
string3 = IN[2]

#set your output
OUT = string1 + string2 + string3
```



EXERCISE 0 – INPUTS AND OUTPUTS

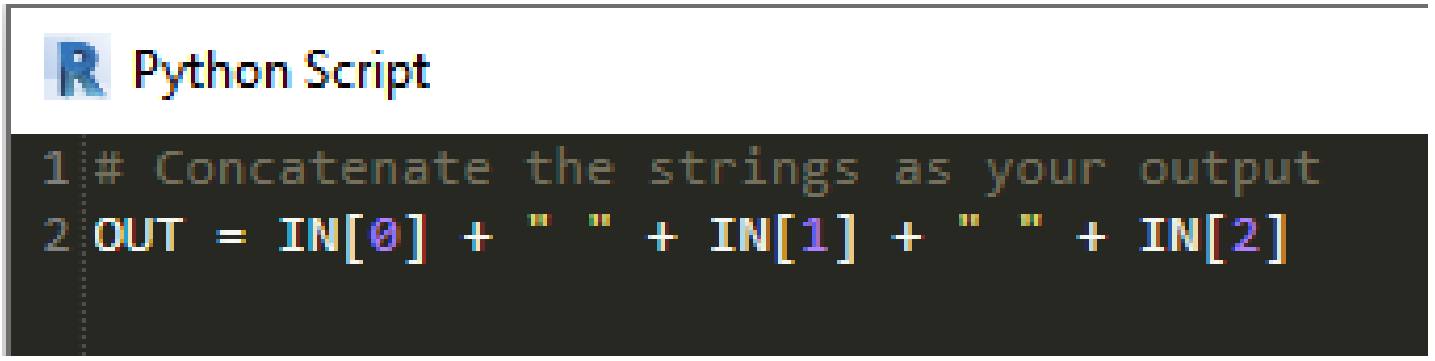
1. Clear the template

2. Set your OUT list to a combination of the Inputs

3. Save and Run

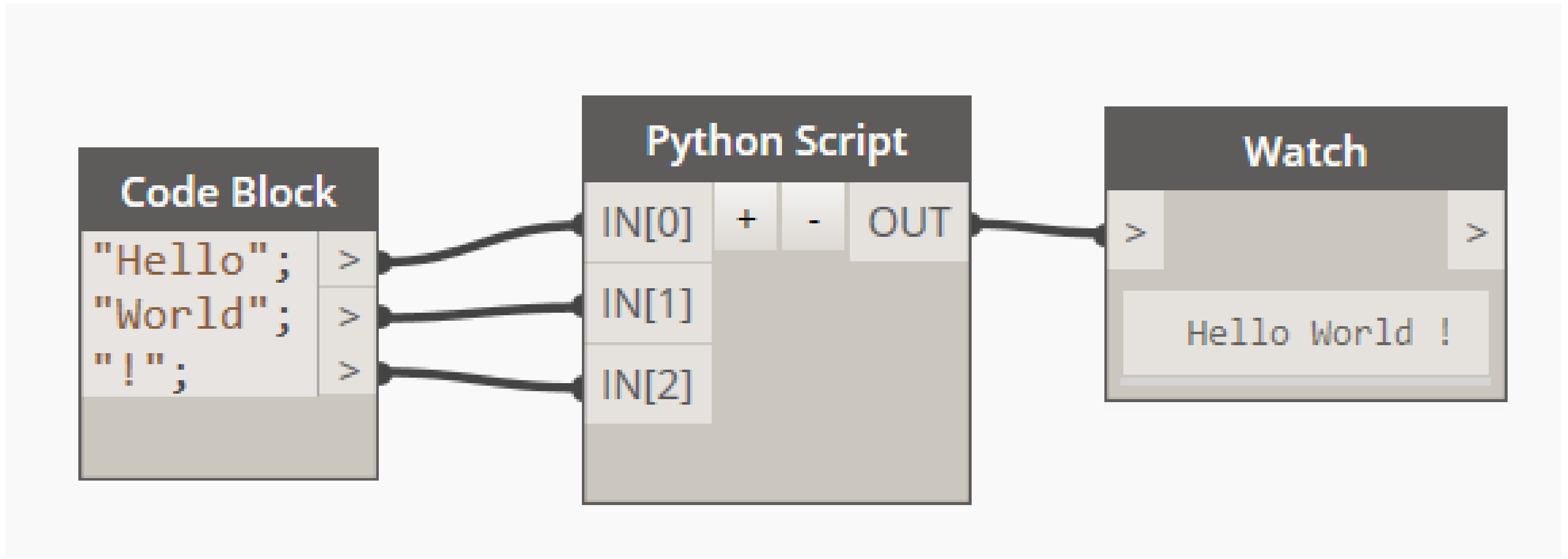
4. Add your Code block to each input

5. Check your results



Python Script

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



EXERCISE 0 – INPUTS AND OUTPUTS

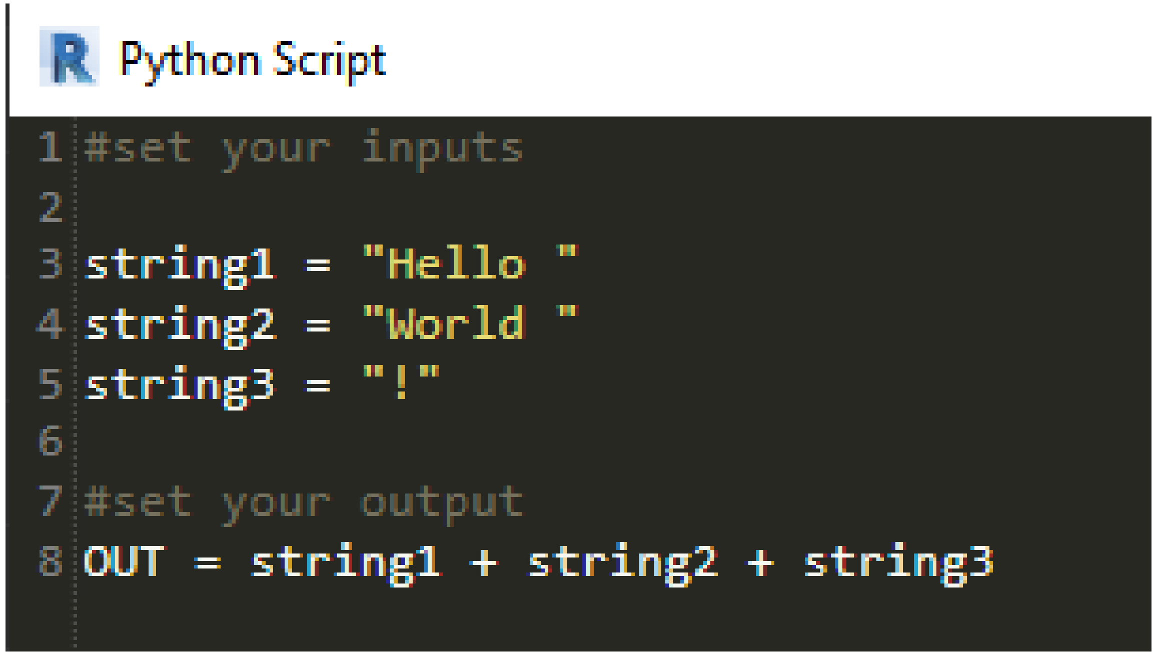
1. Clear the template

2. Add a value for each statement as a string

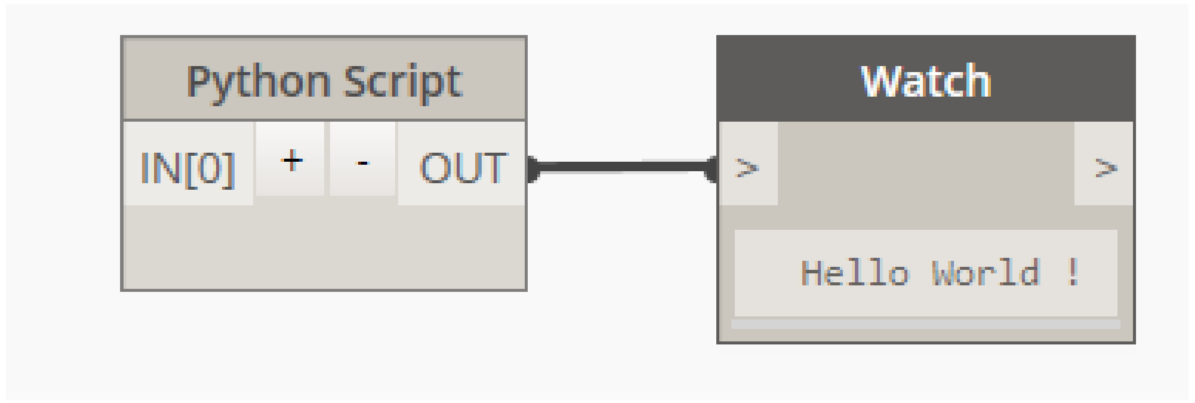
3. Set your OUT list to a combination of the values

4. Save and Run

5. Check your results



```
#set your inputs

string1 = "Hello "
string2 = "World "
string3 = "!"

#set your output
OUT = string1 + string2 + string3
```



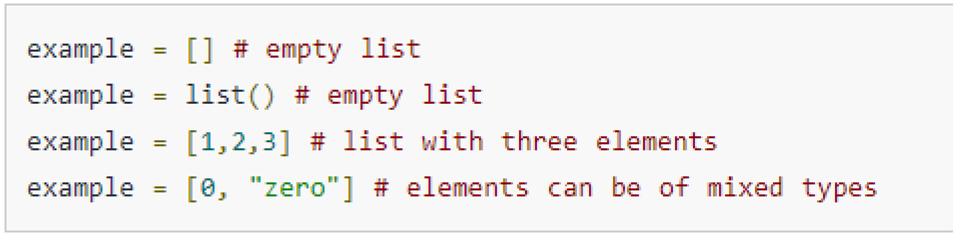
```
Hello World !
```

EXERCISE 1
LISTS

EXERCISE 1 - LISTS

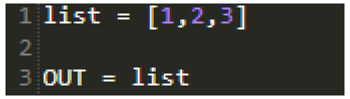
EXERCISE 1 - LISTS

- 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

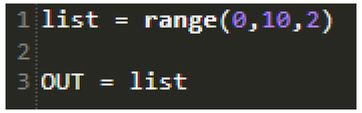


```
example = [] # empty list
example = list() # empty list
example = [1,2,3] # list with three elements
example = [0, "zero"] # elements can be of mixed types
```

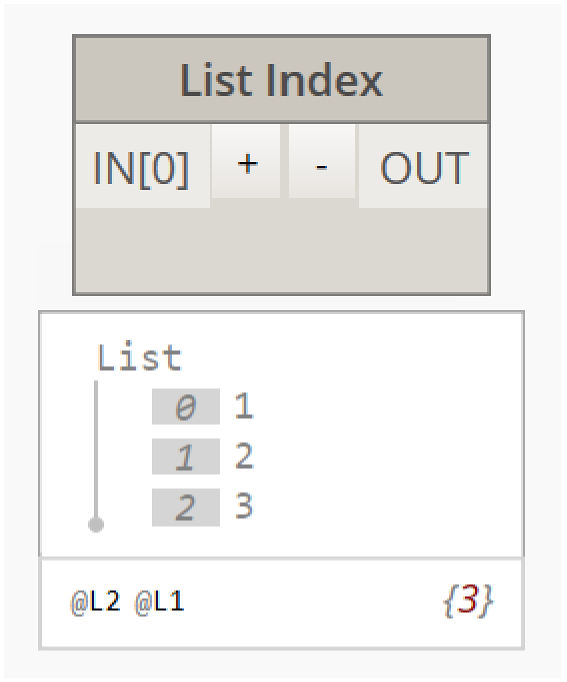
EXERCISE 1 - LISTS



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

OUT = list
```



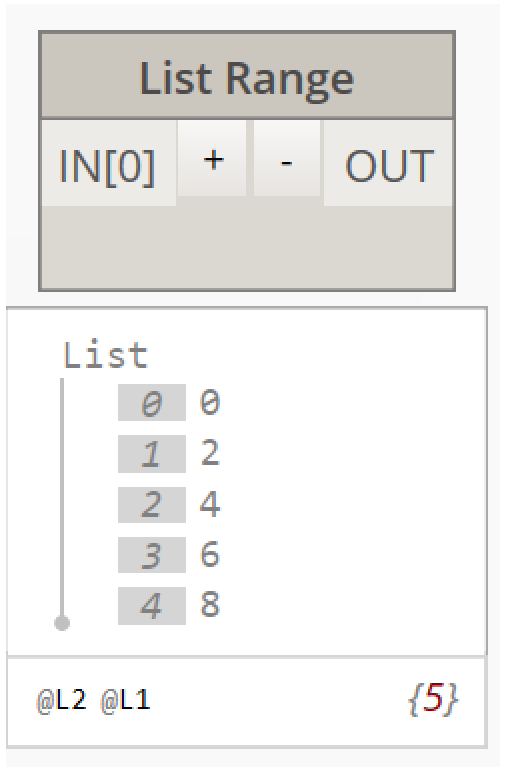
```
list = range(0,10,2)

OUT = list
```



List Index

IN[0] + - OUT

```
List
  0 1
  1 2
  2 3
@L2 @L1        {3}
```



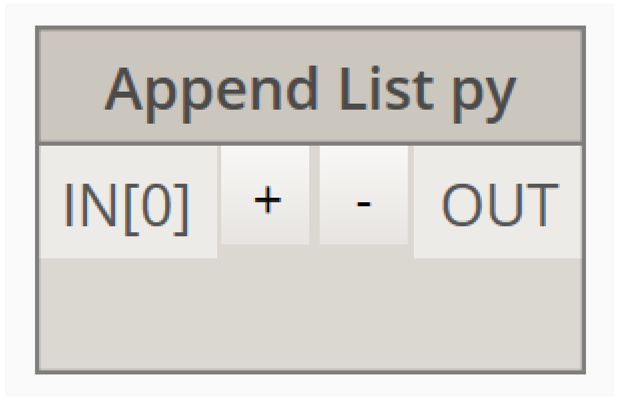
List Range

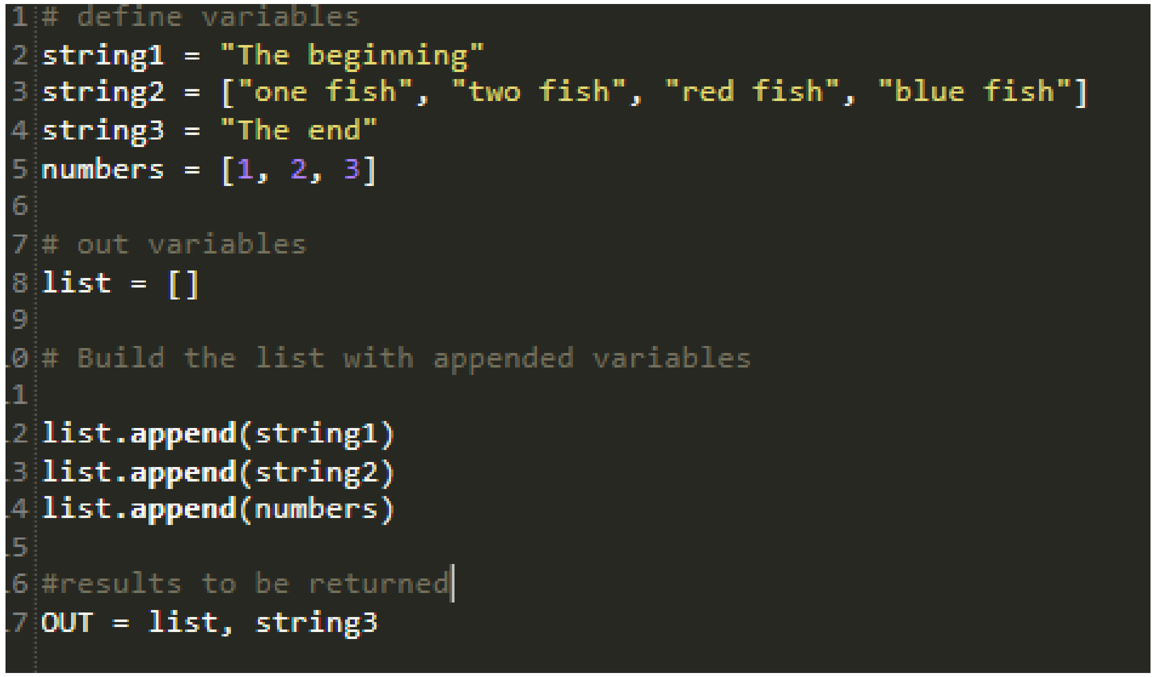
IN[0] + - OUT

```
List
  0 0
  1 2
  2 4
  3 6
  4 8
@L2 @L1        {5}
```

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





```
# define variables
string1 = "The beginning"
string2 = ["one fish", "two fish", "red fish", "blue fish"]
string3 = "The end"
numbers = [1, 2, 3]

# out variables
list = []

# Build the list with appended variables

list.append(string1)
list.append(string2)
list.append(numbers)

#results to be returned
OUT = list, string3
```

EXERCISE 1 - LISTS

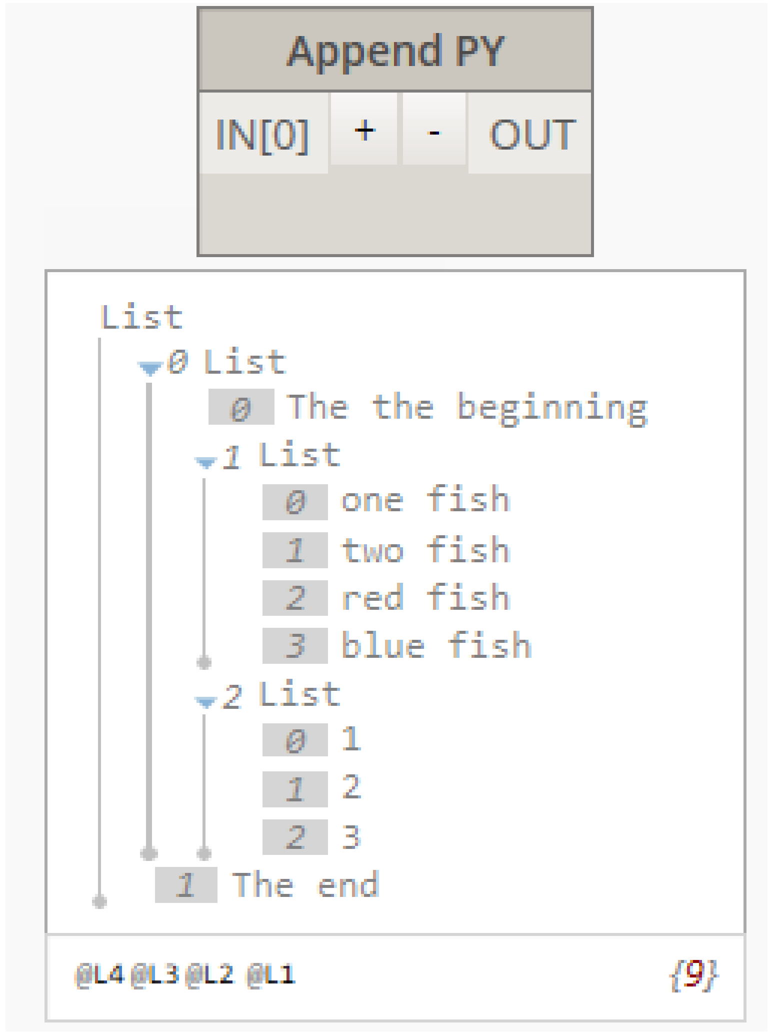
1. Lists can be used to create information from strings and inputs

2. Lists can be nested depending on how you appended them

3. From the prior example we set 3 lists to be appended with strings and numbers

4. Then add "The End" string as part of the output

5. The same concept is applied for Revit content



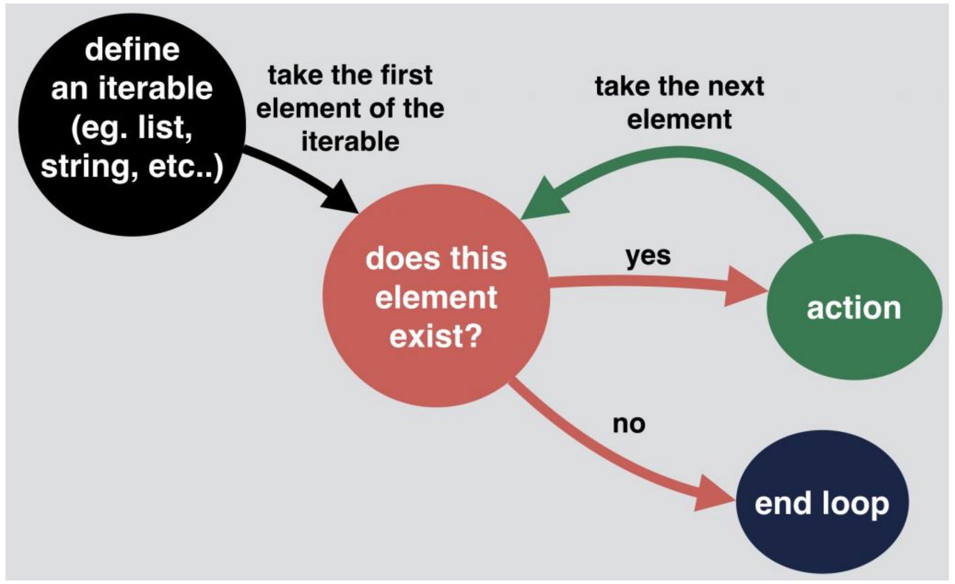
```
List
  0 List
    0 The the beginning
    1 List
      0 one fish
      1 two fish
      2 red fish
      3 blue fish
    2 List
      0 1
      1 2
      2 3
  1 The end
@L4 @L3 @L2 @L1                {9}
```

EXERCISE 2
LOOPS

EXERCISE 2 - LOOPS

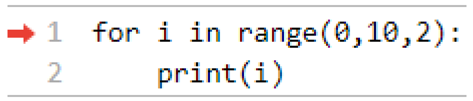
EXERCISE 2 - LOOPS

- 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

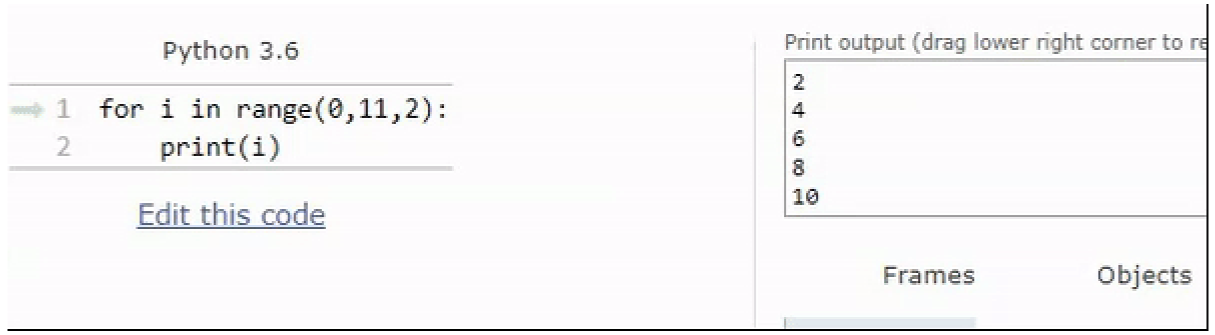

define an iterable (eg. list, string, etc..)
take the first element of the iterable
take the next element
does this element exist?
yes
action
no
end loop

EXERCISE 2 - LOOPS

- 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



```
for i in range(0,10,2):
    print(i)
```



Python 3.6

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

```
2
4
6
8
10
```

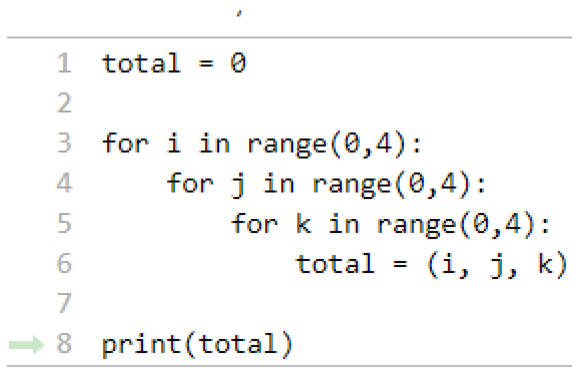
EXERCISE 2 - LOOPS

- Every time you indent the loop becomes nested meaning it takes lower priority

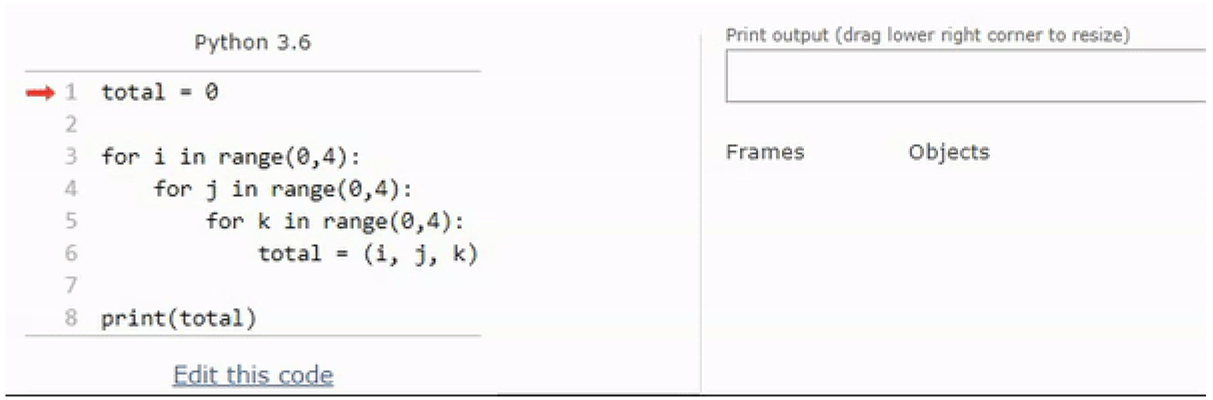
- Once the loop iterations are over you can place the variables to a list or function and print

- 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

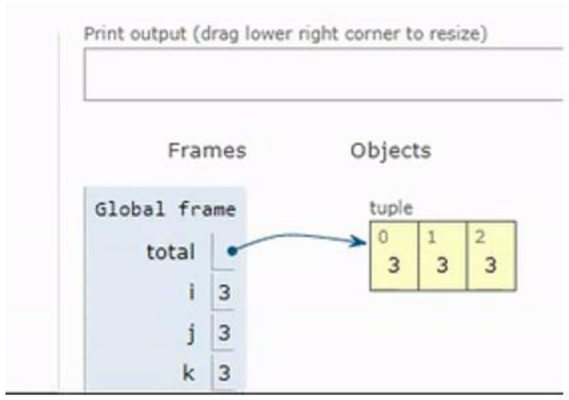


```
total = 0

for i in range(0,4):
    for j in range(0,4):
        for k in range(0,4):
            total = (i, j, k)

print(total)
```



Python 3.6

```
total = 0

for i in range(0,4):
    for j in range(0,4):
        for k in range(0,4):
            total = (i, j, k)

print(total)
```



Global frame

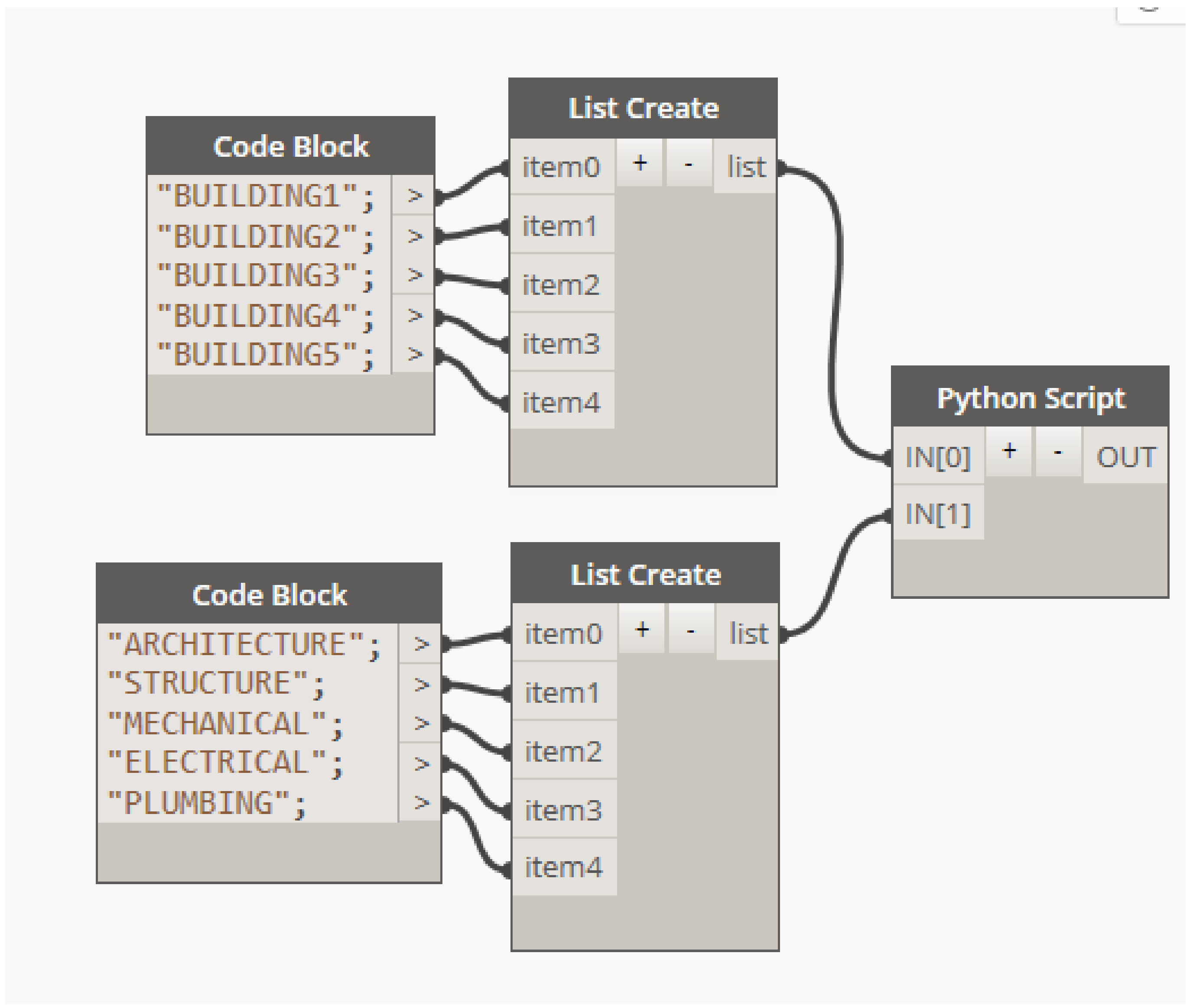
total	
i	3
j	3
k	3

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



```
"BUILDING1";
"BUILDING2";
"BUILDING3";
"BUILDING4";
"BUILDING5";
```

```
"ARCHITECTURE";
"STRUCTURE";
"MECHANICAL";
"ELECTRICAL";
"PLUMBING";
```

EXERCISE 2 - LOOPS

1. Your Outlist will be the **list** with parentheses meaning you want all the results

2. Start with **for** 'X-variable' **in** Value-1

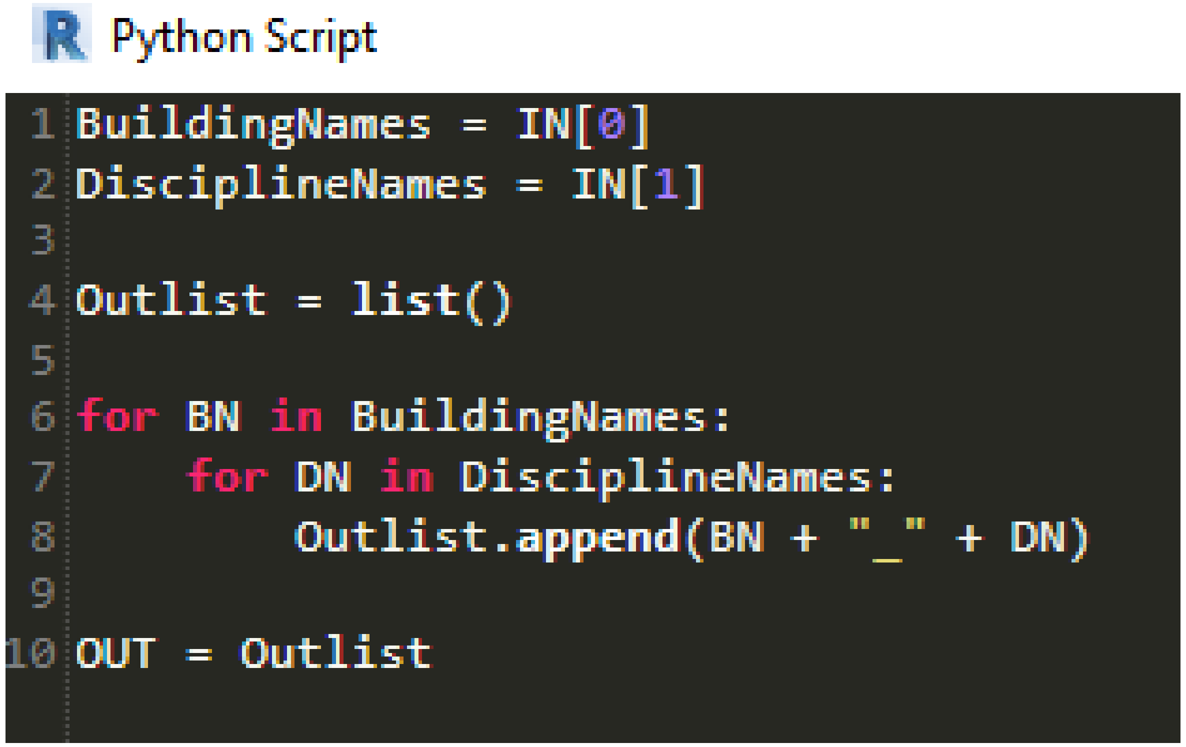
3. Then add colon **:**

4. Indent

5. Add another for statement ending with colon

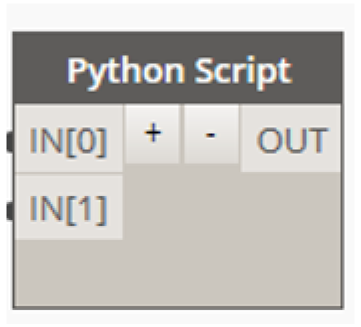
6. End of this loop will **Append** to your list these values within the parentheses

7. **OUT** is your Outlist

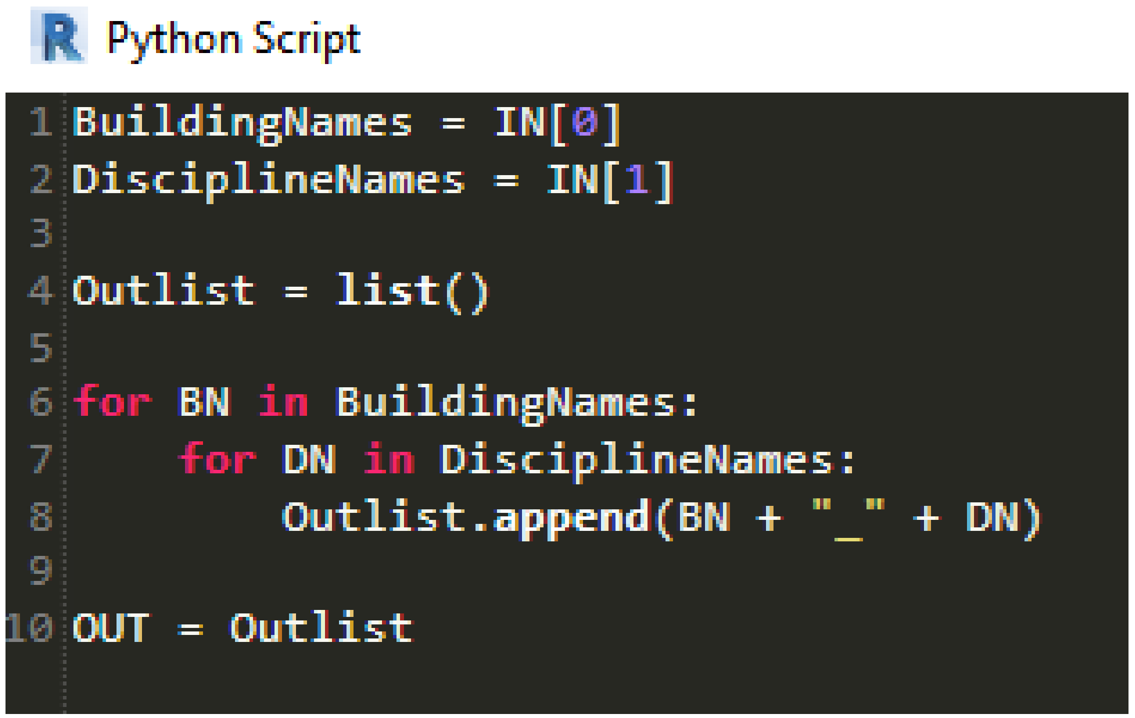


Python Script

```python
BuildingNames = IN[0]
DisciplineNames = IN[1]

Outlist = list()

for BN in BuildingNames:
    for DN in DisciplineNames:
        Outlist.append(BN + "_" + DN)

OUT = Outlist
```

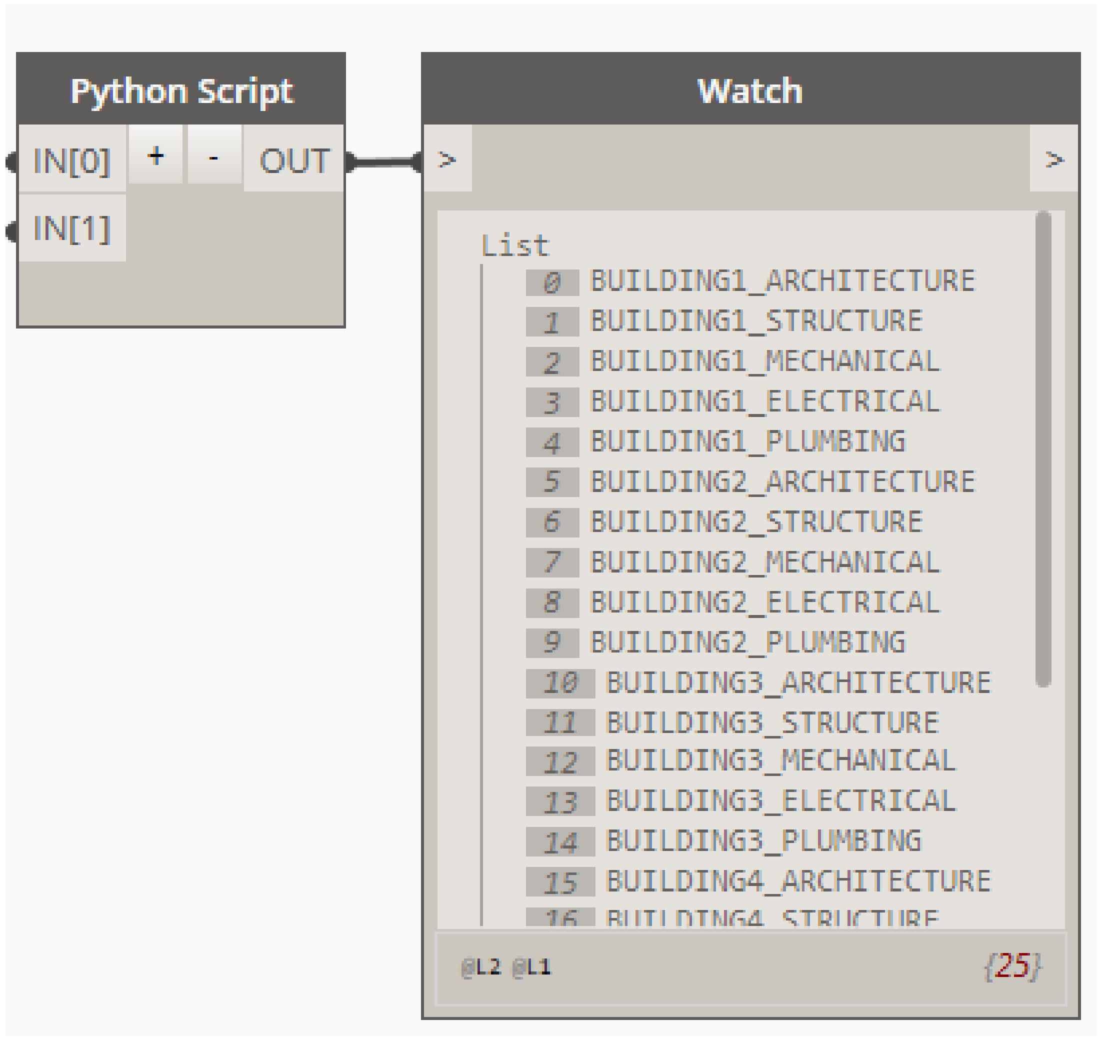


EXERCISE 2 - LOOPS



Python Script

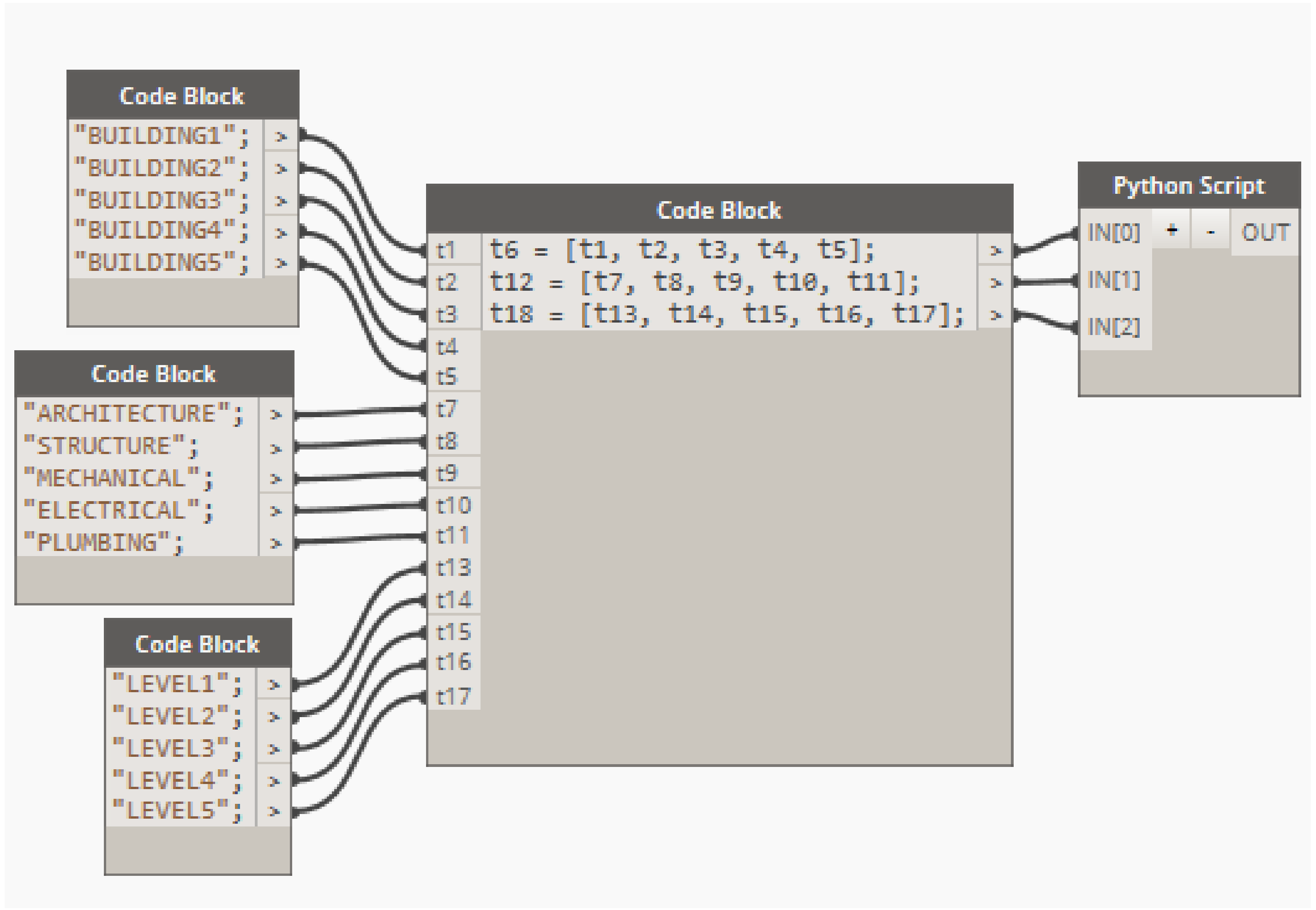
```python
BuildingNames = IN[0]
DisciplineNames = IN[1]

Outlist = list()

for BN in BuildingNames:
    for DN in DisciplineNames:
        Outlist.append(BN + "_" + DN)

OUT = Outlist
```



```
List
  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
@L2 @L1                    {25}
```

EXERCISE 2 - LOOPS

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



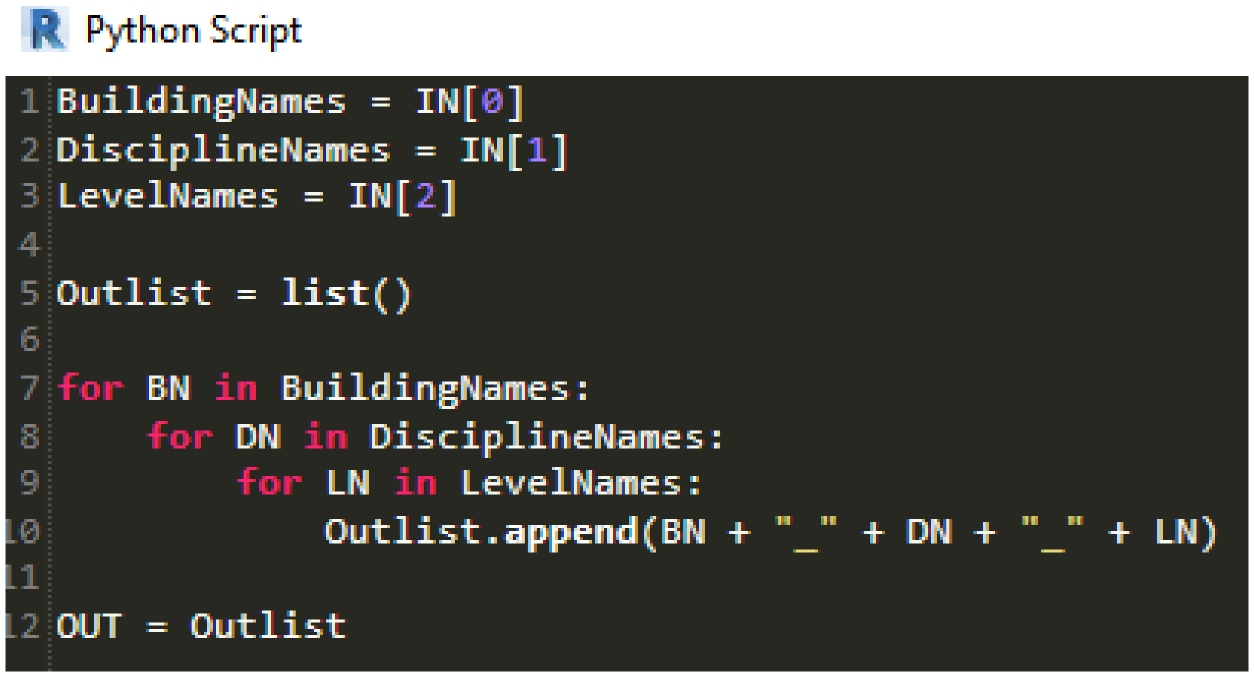
```
"BUILDING1";
"BUILDING2";
"BUILDING3";
"BUILDING4";
"BUILDING5";
```

```
"ARCHITECTURE";
"STRUCTURE";
"MECHANICAL";
"ELECTRICAL";
"PLUMBING";
```

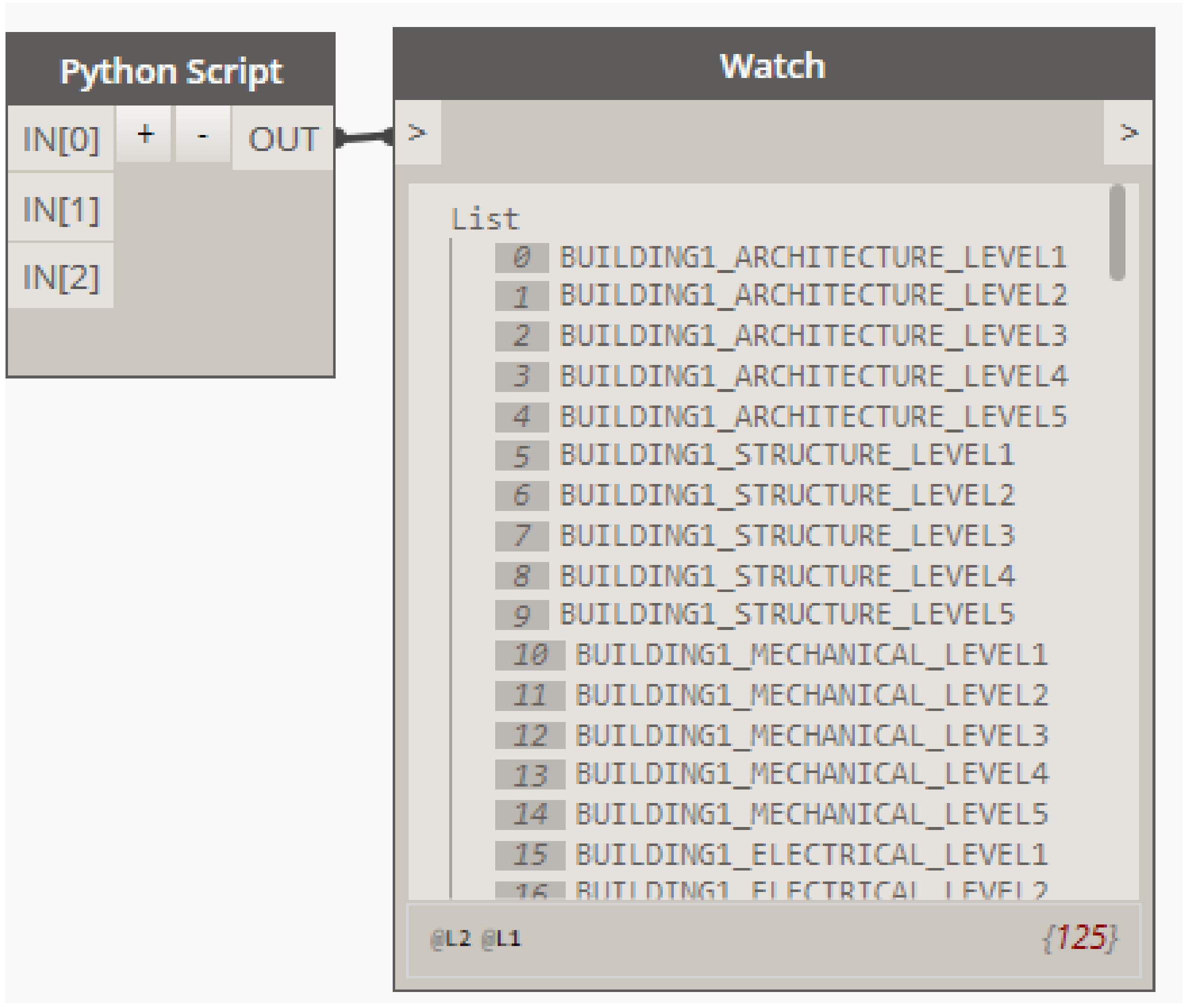
```
"LEVEL1";
"LEVEL2";
"LEVEL3";
"LEVEL4";
"LEVEL5";
```

```
t6 = [t1, t2, t3, t4, t5];
t12 = [t7, t8, t9, t10, t11];
t18 = [t13, t14, t15, t16, t17];
```

EXERCISE 2


Python Script

```
BuildingNames = IN[0]
DisciplineNames = IN[1]
LevelNames = IN[2]

Outlist = list()

for BN in BuildingNames:
    for DN in DisciplineNames:
        for LN in LevelNames:
            Outlist.append(BN + "_" + DN + "_" + LN)

OUT = Outlist
```



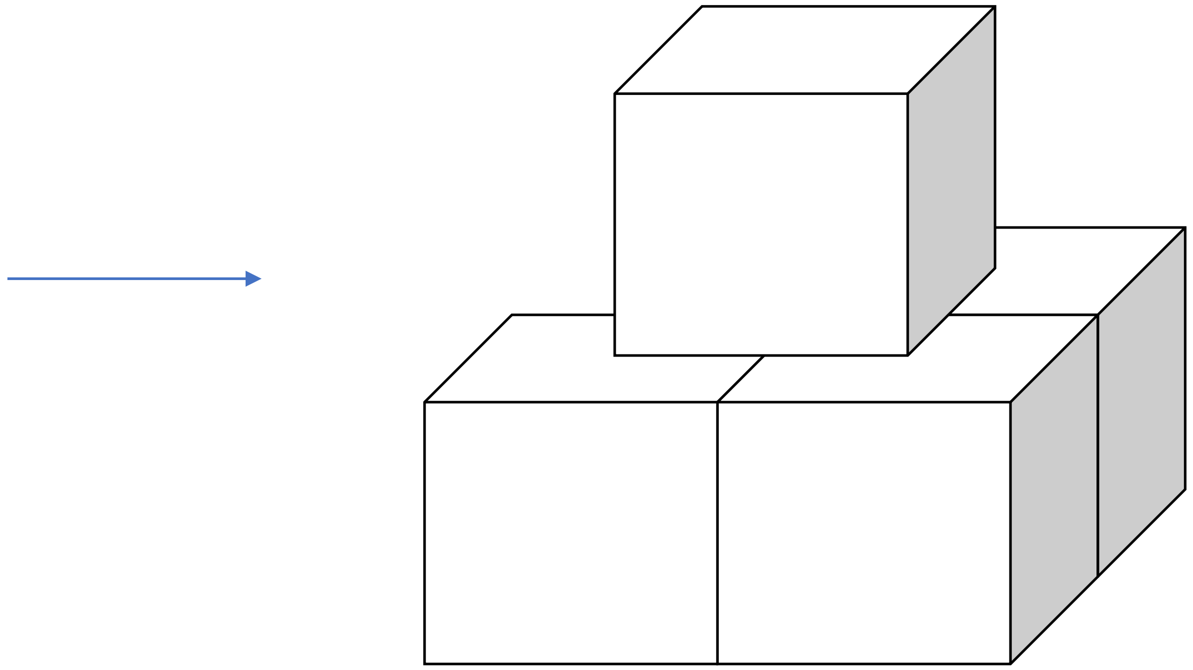
```
List
  0 BUILDING1_ARCHITECTURE_LEVEL1
  1 BUILDING1_ARCHITECTURE_LEVEL2
  2 BUILDING1_ARCHITECTURE_LEVEL3
  3 BUILDING1_ARCHITECTURE_LEVEL4
  4 BUILDING1_ARCHITECTURE_LEVEL5
  5 BUILDING1_STRUCTURE_LEVEL1
  6 BUILDING1_STRUCTURE_LEVEL2
  7 BUILDING1_STRUCTURE_LEVEL3
  8 BUILDING1_STRUCTURE_LEVEL4
  9 BUILDING1_STRUCTURE_LEVEL5
  10 BUILDING1_MECHANICAL_LEVEL1
  11 BUILDING1_MECHANICAL_LEVEL2
  12 BUILDING1_MECHANICAL_LEVEL3
  13 BUILDING1_MECHANICAL_LEVEL4
  14 BUILDING1_MECHANICAL_LEVEL5
  15 BUILDING1_ELECTRICAL_LEVEL1
  16 BUILDING1_ELECTRICAL_LEVEL2
@L2 @L1                          {125}
```

X-INDENT

EXERCISE 3
WRITE

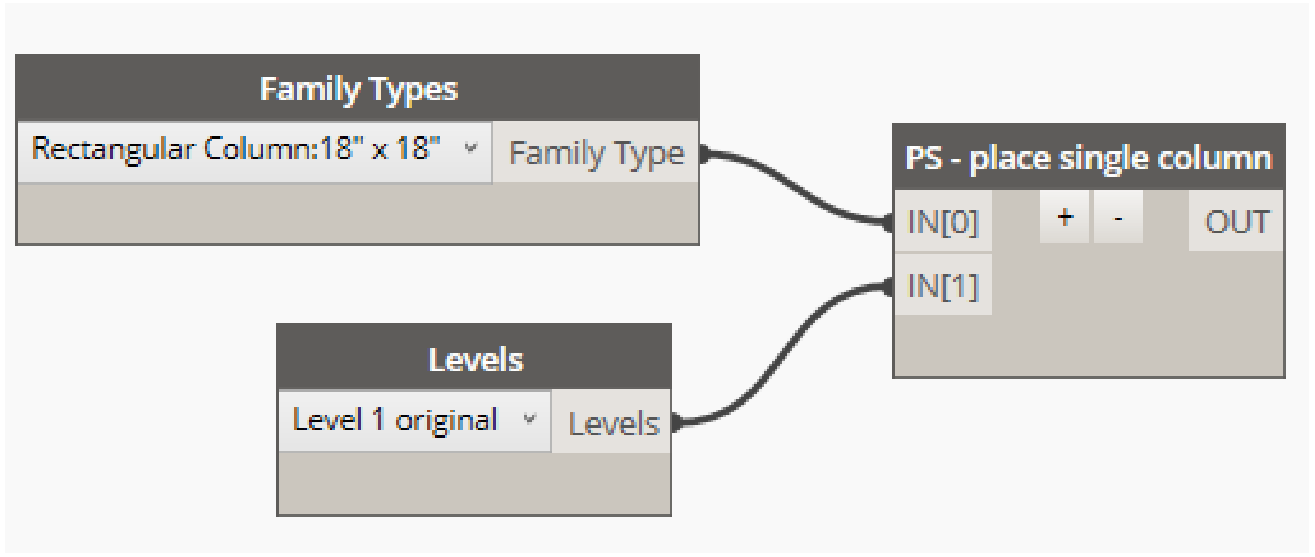
EXERCISE 3 – WRITE TO REVIT

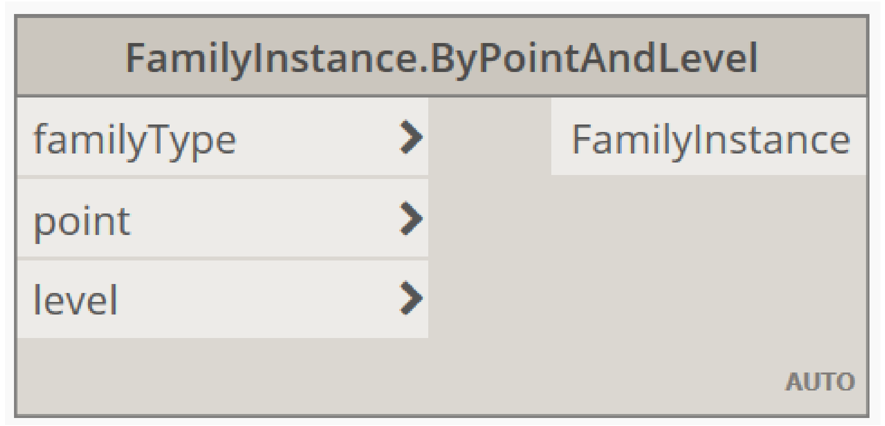
```
for x in elements():
    list(x)

OUT = list
```



EXERCISE 3 – WRITE TO REVIT

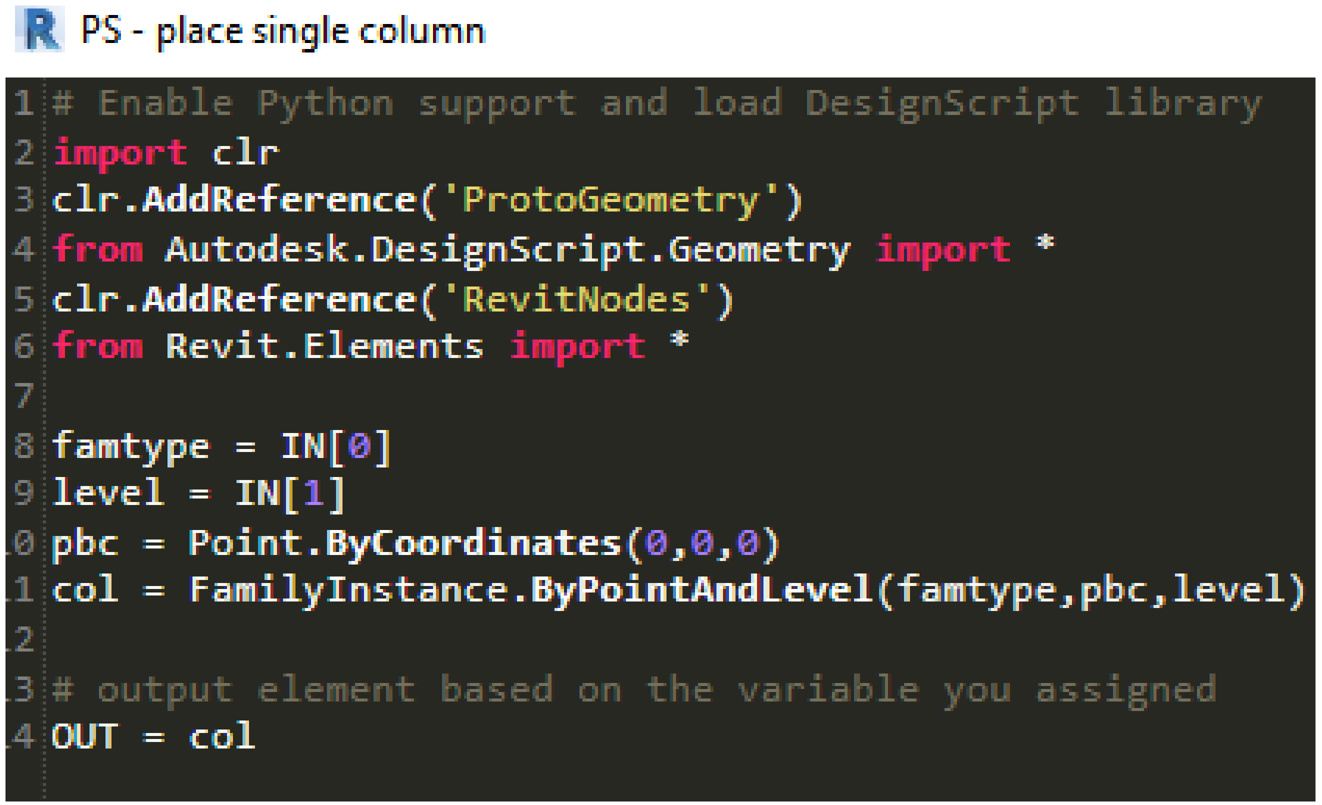
- With the use of values, lists, append and loops you can use Python to get work done
- For example by placing a family into Revit based on the values you assign
- To start you need a family and a level to place it on assuming it is a level constrained family





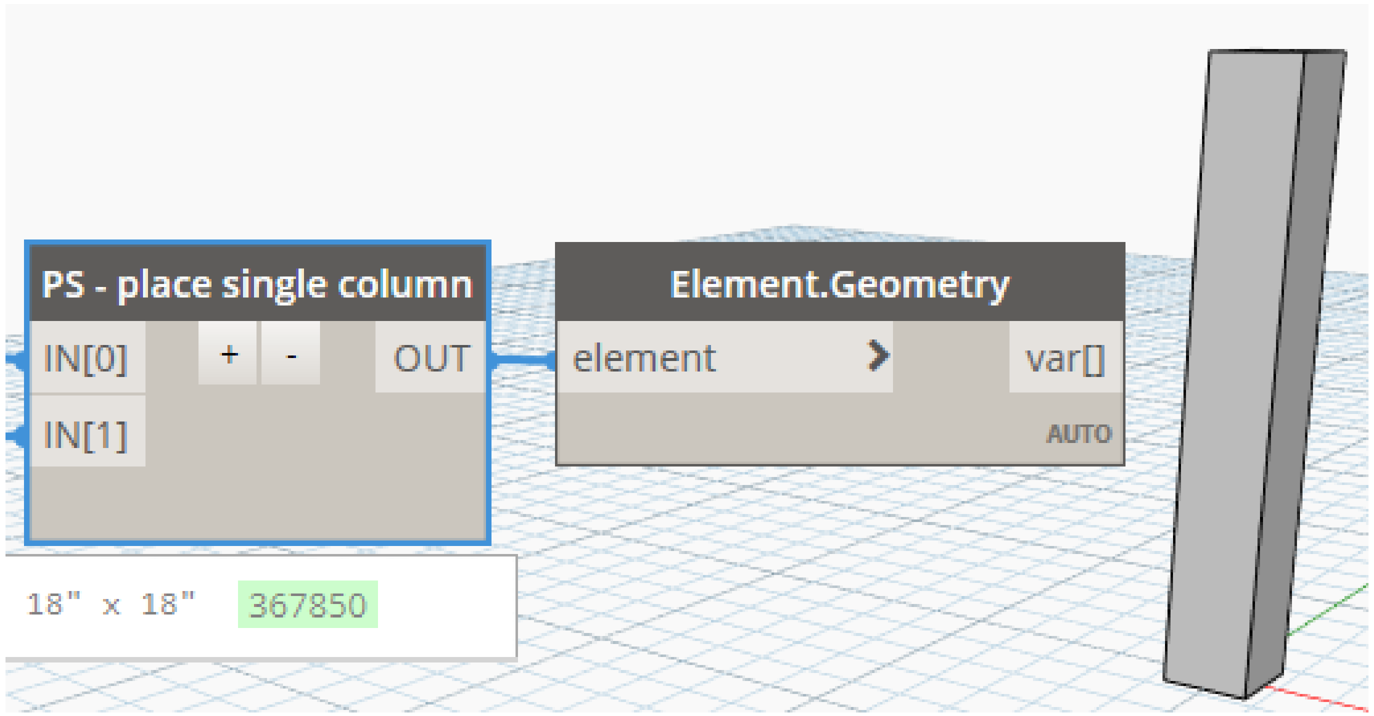
EXERCISE 3 – WRITE TO REVIT

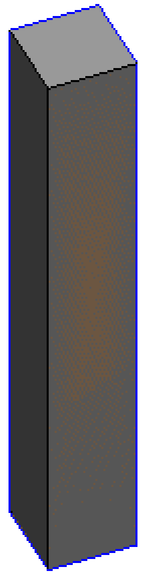
1. Start with the template
2. Make sure ProtoGeometry and RevitNodes are imported
3. Add an input for family type
4. Add an input for the level
5. Create a value for the operation Point.ByCoordinates followed by the origin position
6. Create a value for the family you want to place followed by range 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

```python
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
clr.AddReference('RevitNodes')
from Revit.Elements import *

famtype = IN[0]
level = IN[1]
pbc = Point.ByCoordinates(0,0,0)
col = FamilyInstance.ByPointAndLevel(famtype,pbc,level)

# output element based on the variable you assigned
OUT = col
```

EXERCISE 3 – WRITE TO REVIT





EXERCISE 3 – WRITE TO REVIT

1. Expand the concept to multiple members

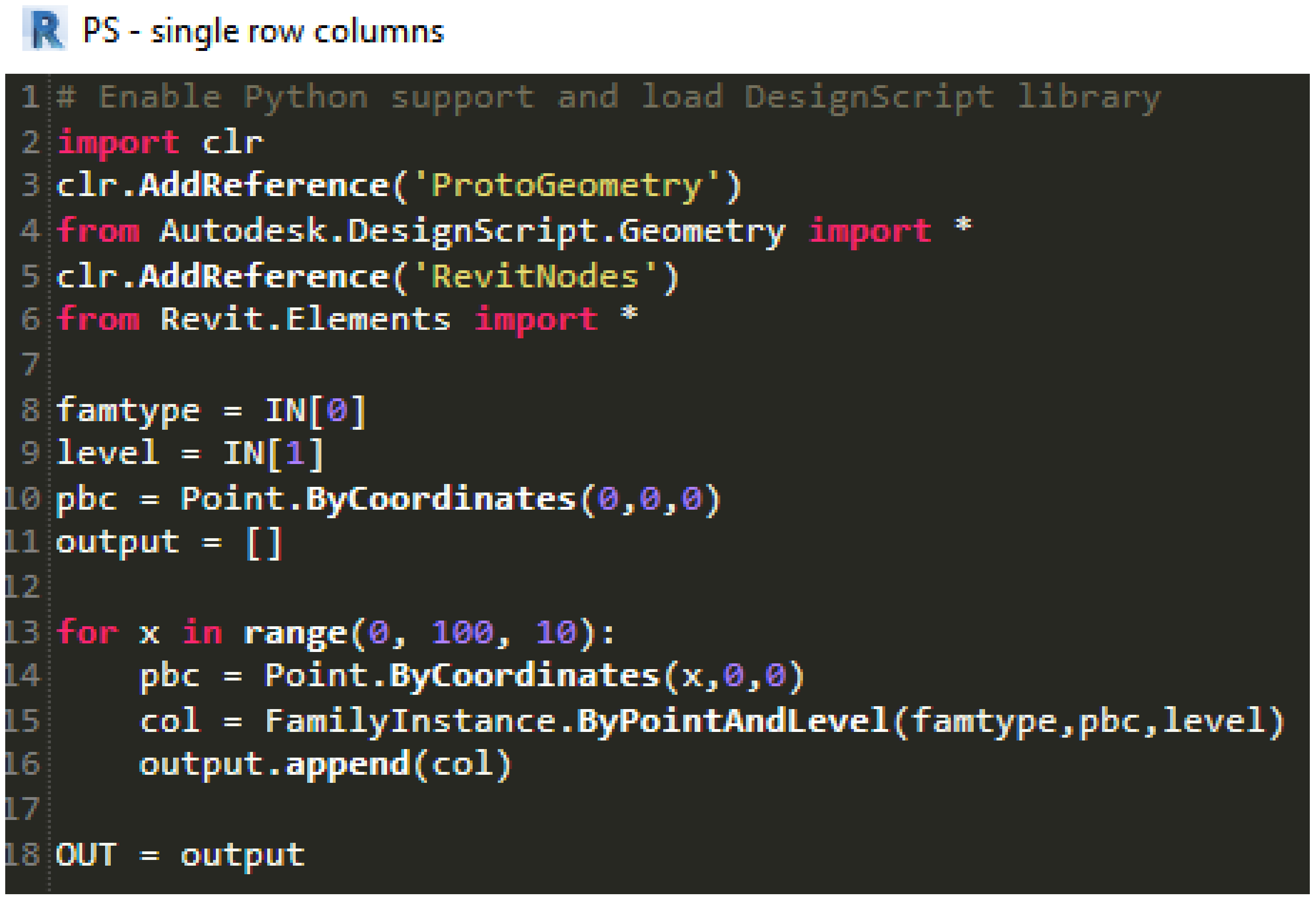
2. Keep the first 3 values but replace the output with a list instead of the family so we can get more than one result

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

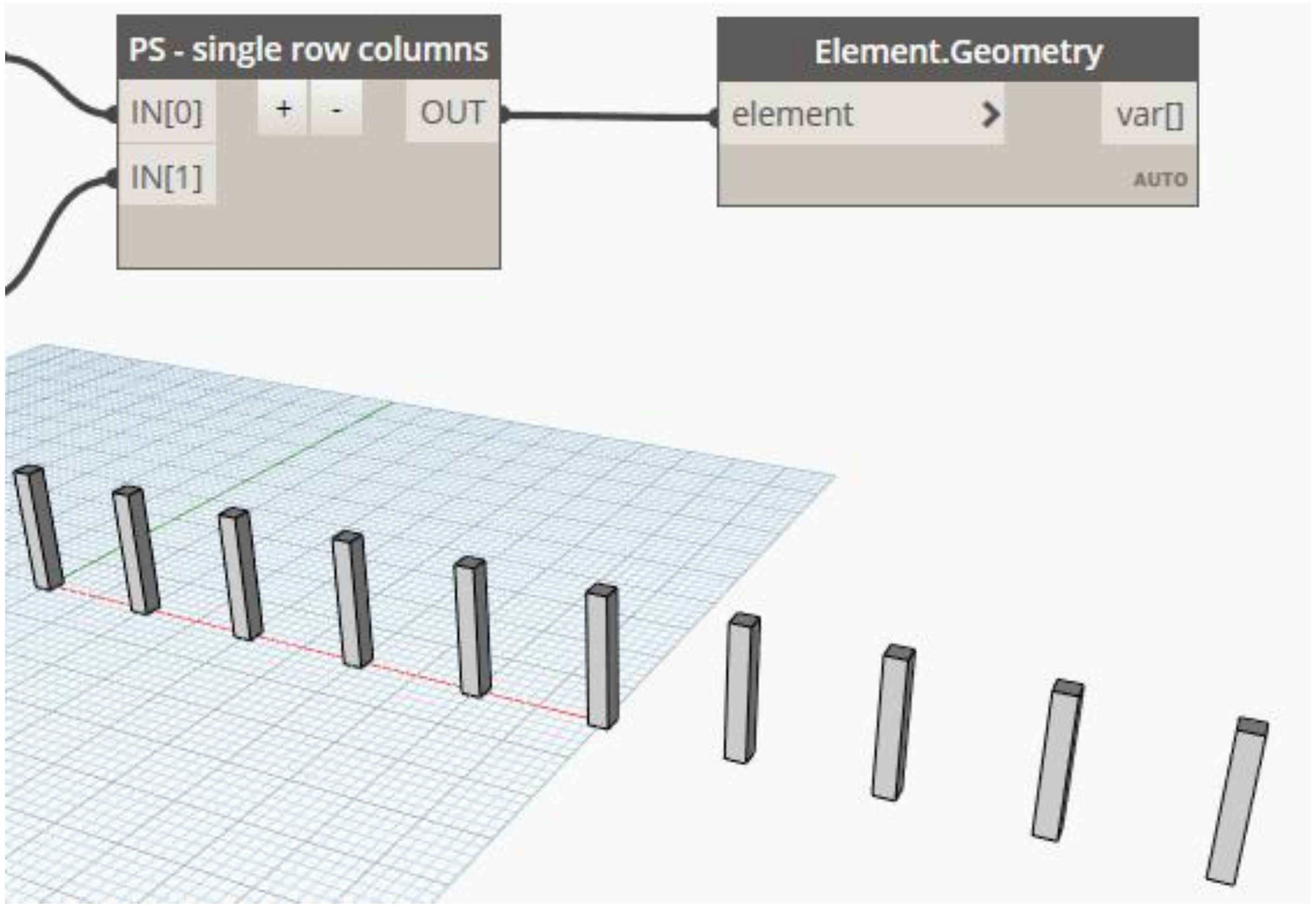
6. Output is your list


PS - single row columns

```python
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
clr.AddReference('RevitNodes')
from Revit.Elements import *

famtype = IN[0]
level = IN[1]
pbc = Point.ByCoordinates(0,0,0)
output = []

for x in range(0, 100, 10):
    pbc = Point.ByCoordinates(x,0,0)
    col = FamilyInstance.ByPointAndLevel(famtype,pbc,level)
    output.append(col)

OUT = output
```

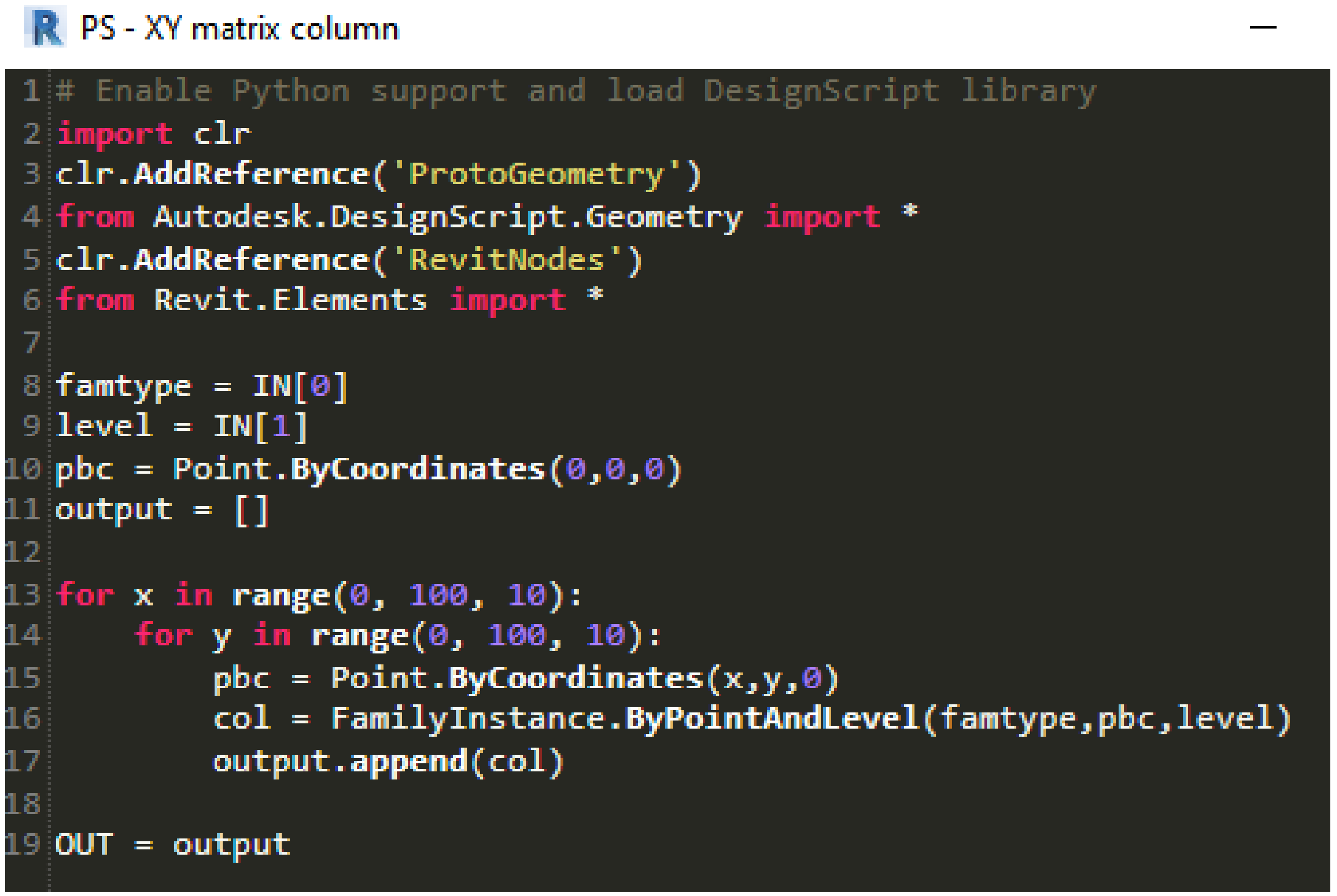
EXERCISE 3 – WRITE TO REVIT

- Now you have line of columns
- Remember the loop with the point by coordinates is the function that made the line work



EXERCISE 3 – WRITE TO REVIT

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

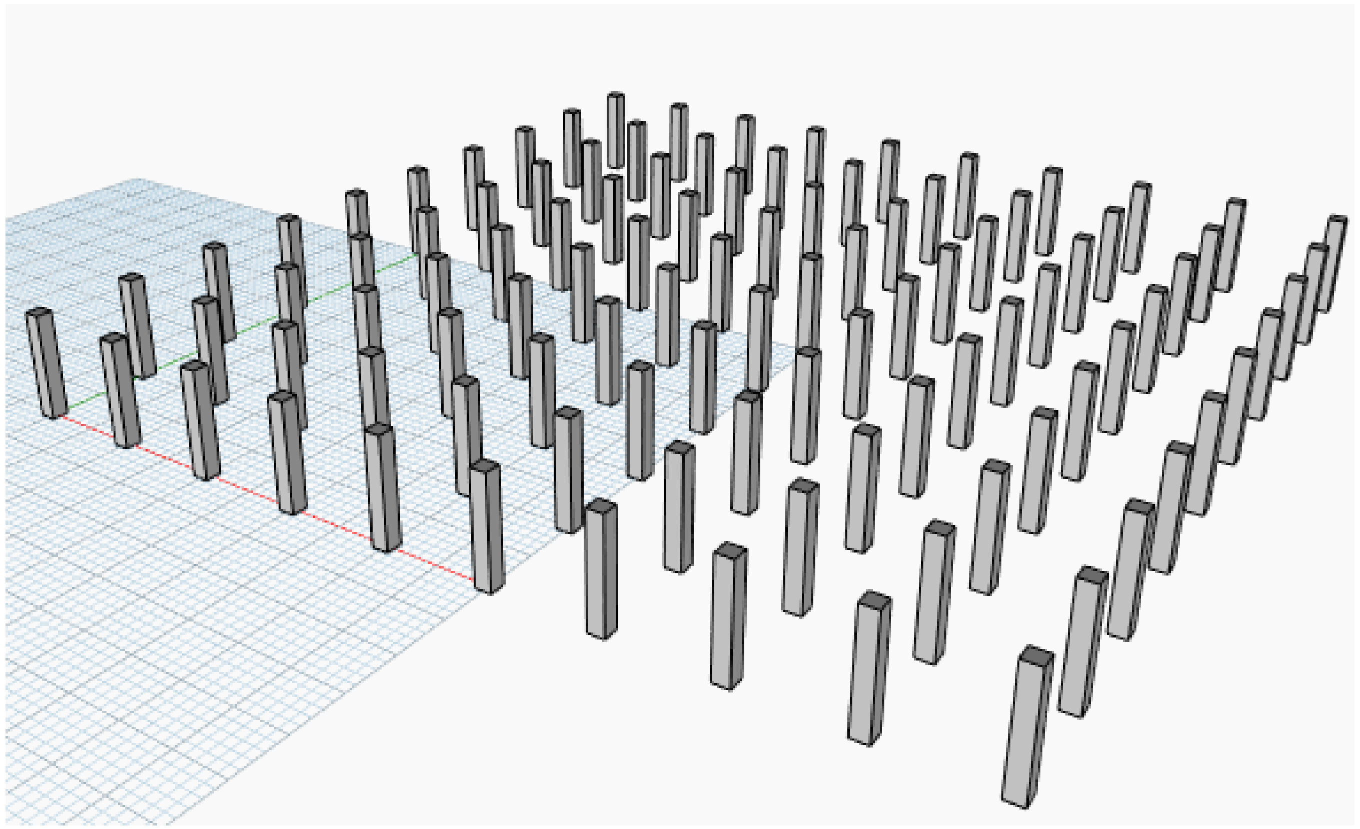


```python
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
clr.AddReference('RevitNodes')
from Revit.Elements import *

famtype = IN[0]
level = IN[1]
pbc = Point.ByCoordinates(0,0,0)
output = []

for x in range(0, 100, 10):
    for y in range(0, 100, 10):
        pbc = Point.ByCoordinates(x,y,0)
        col = FamilyInstance.ByPointAndLevel(famtype,pbc,level)
        output.append(col)

OUT = output
```

X-COLON

EXERCISE 3 – WRITE TO REVIT

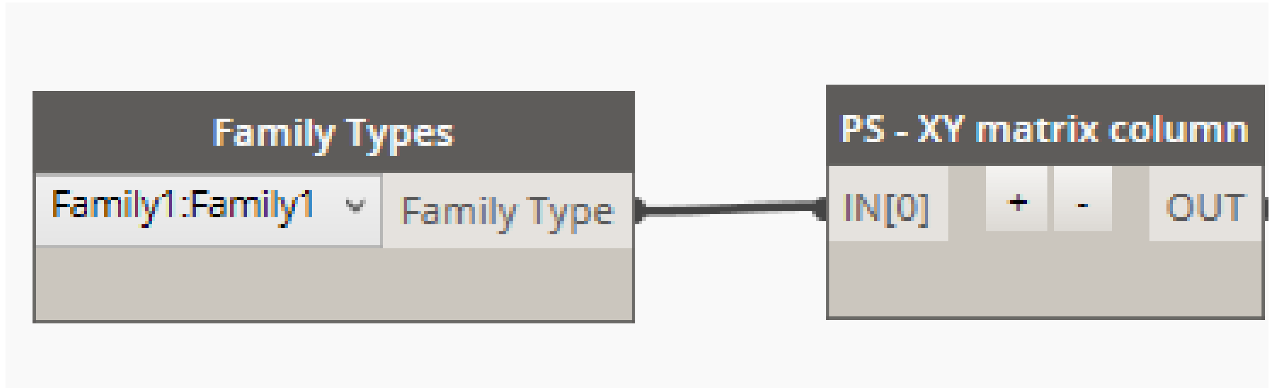
- You end up with a result the same as cross-lacing the position with more control within one node
- The loop adds a column per point
- The range and loops can all be adjusted for different outputs in the same code

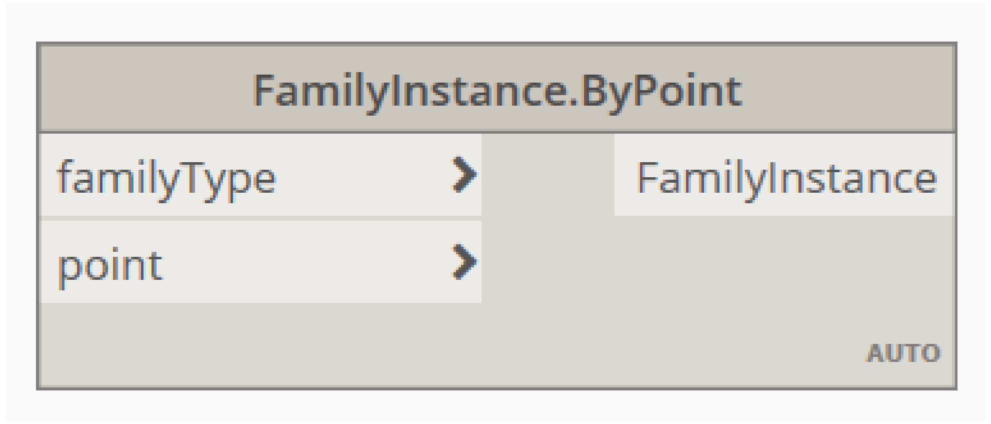


EXERCISE 3 – WRITE TO REVIT

- 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





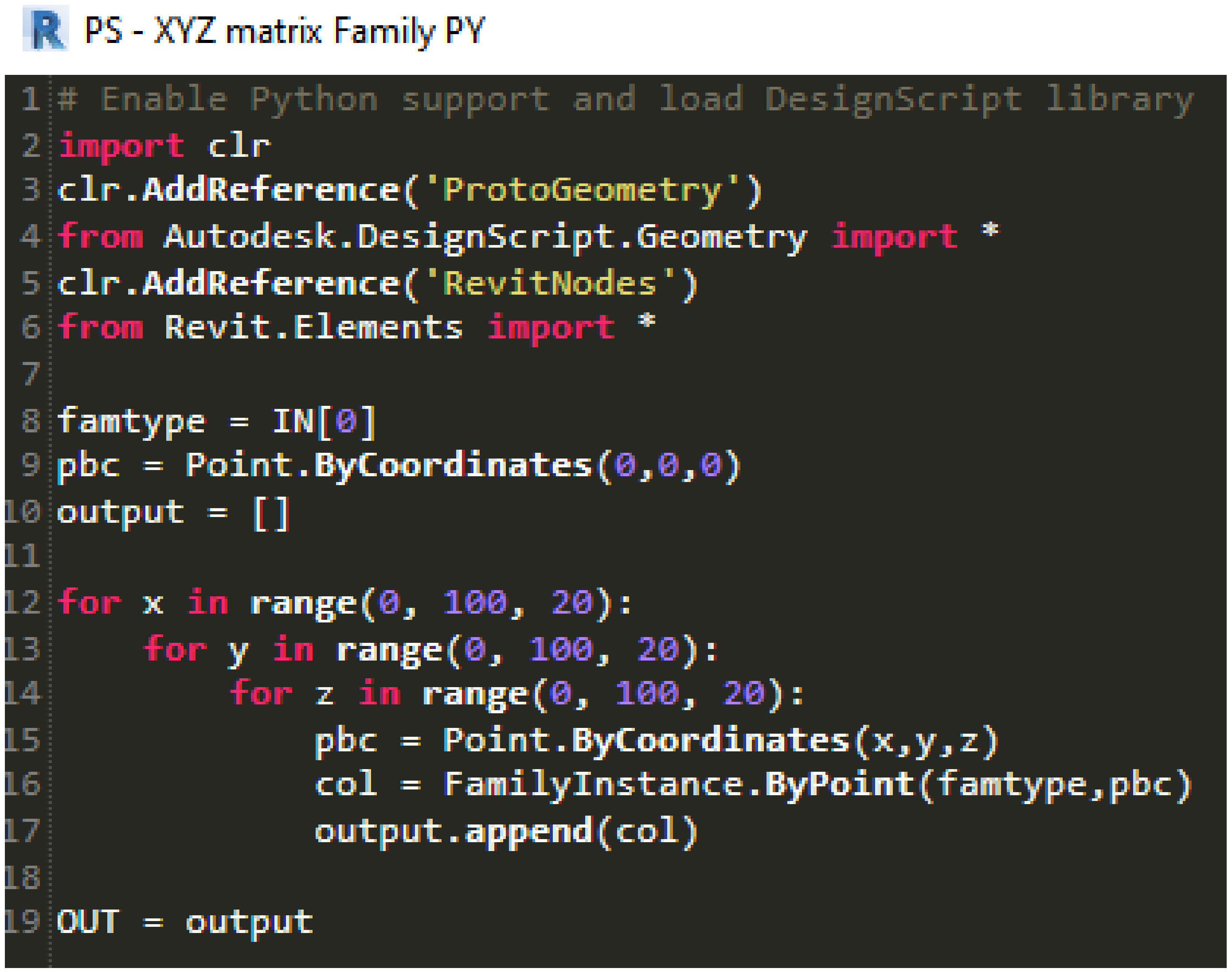
EXERCISE 3 – WRITE TO REVIT

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

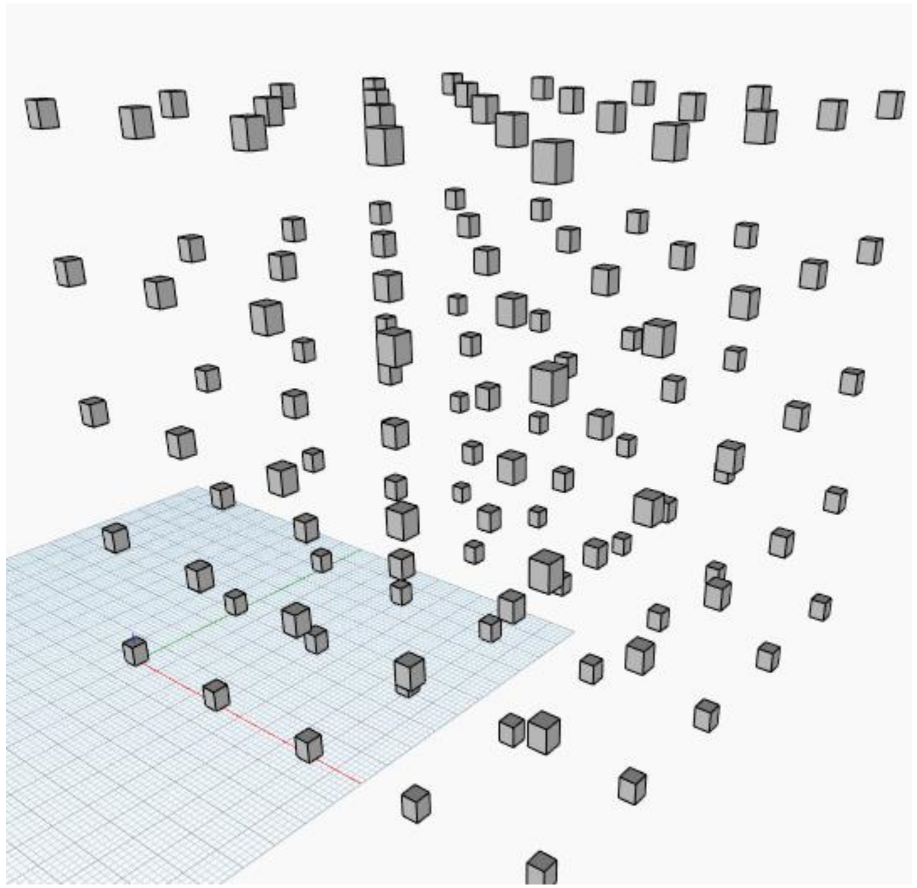
4. Output the list of points

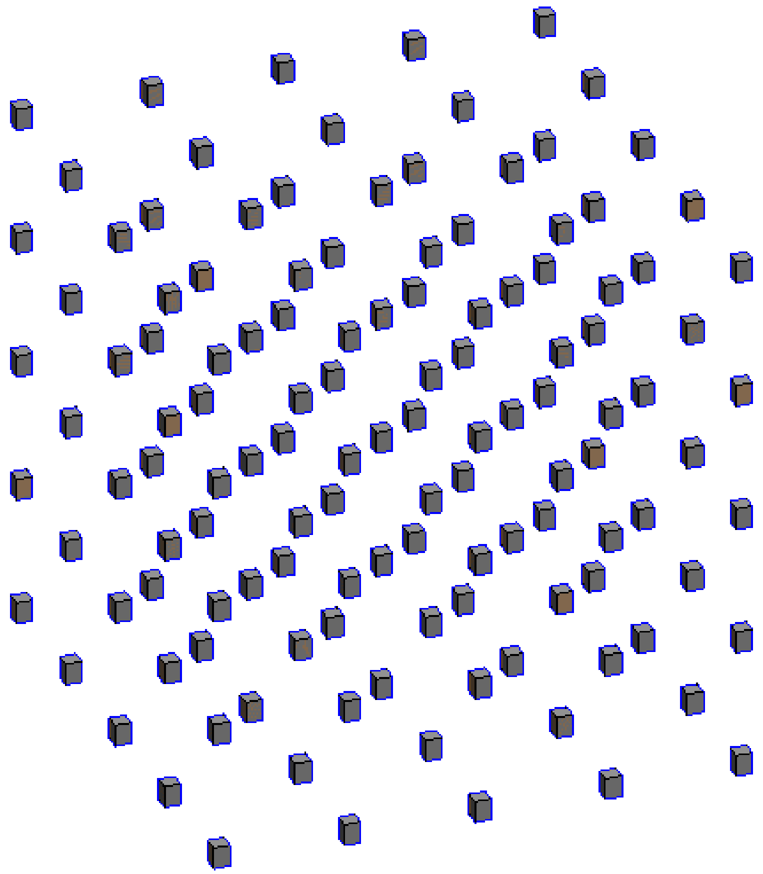


PS - XYZ matrix Family PY

```python
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
clr.AddReference('RevitNodes')
from Revit.Elements import *

famtype = IN[0]
pbc = Point.ByCoordinates(0,0,0)
output = []

for x in range(0, 100, 20):
    for y in range(0, 100, 20):
        for z in range(0, 100, 20):
            pbc = Point.ByCoordinates(x,y,z)
            col = FamilyInstance.ByPoint(famtype,pbc)
            output.append(col)

OUT = output
```

EXERCISE 3 – WRITE TO REVIT



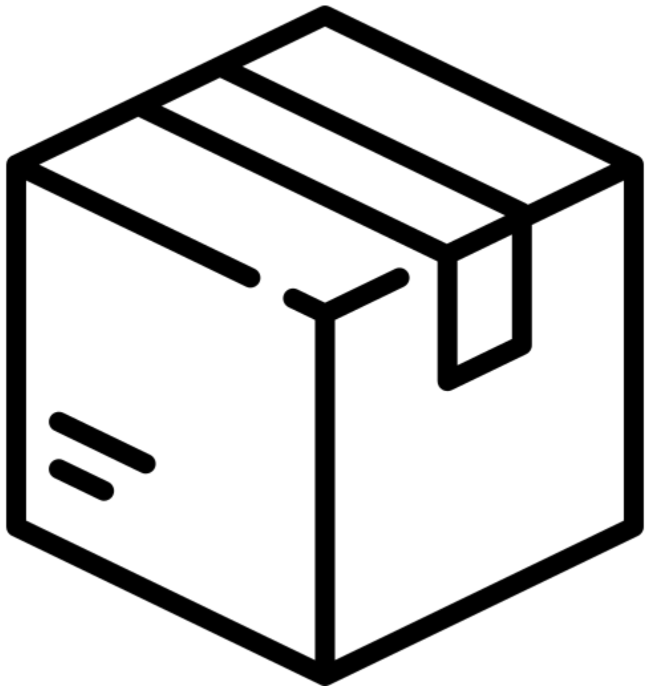


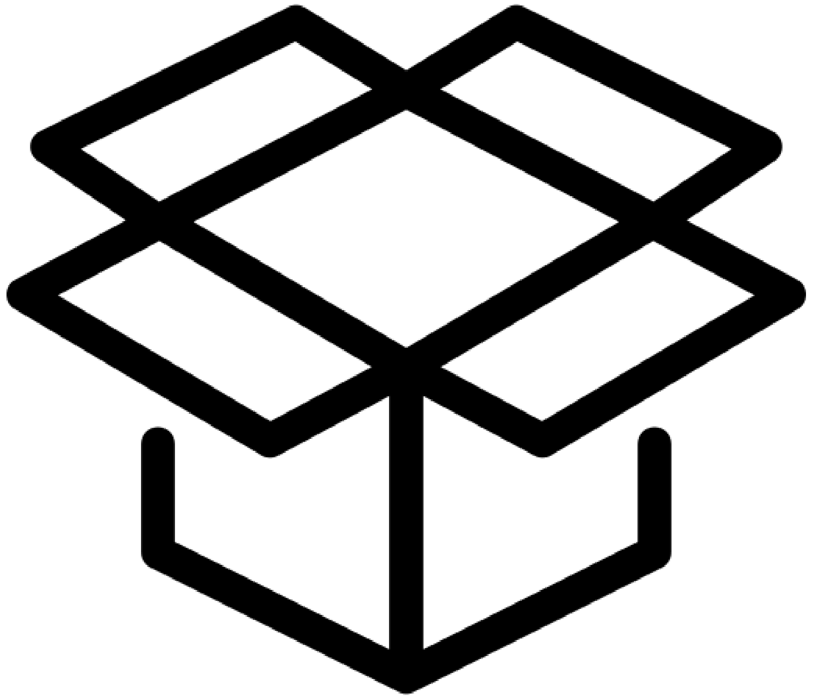
Now you have a grid that produces all the families from the loop an inputs

If you run this result then it will write to Revit these families at the coordinates

EXERCISE 4
WRAP

EXERCISE 4 – UNWRAP ELEMENTS





EXERCISE 4 – UNWRAP ELEMENTS

SERVICES TO MAKE YOUR CODE WORK

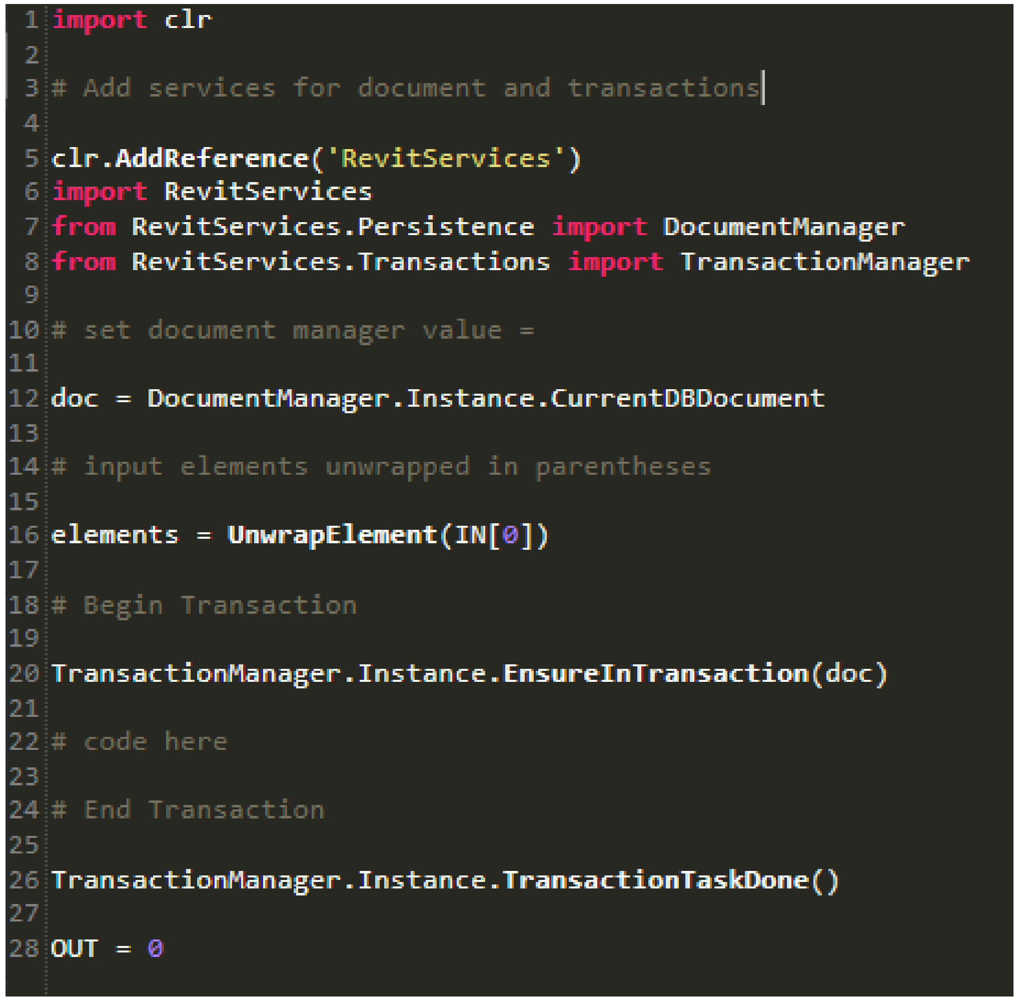
```
import clr

clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *
from Autodesk.Revit.DB.Structure import *

clr.AddReference('RevitAPIUI')
from Autodesk.Revit.UI import *

clr.AddReference('System')
from System.Collections.Generic import List

clr.AddReference('RevitNodes')
import Revit
clr.ImportExtensions(Revit.GeometryConversion)
clr.ImportExtensions(Revit.Elements)

clr.AddReference('RevitServices')
import RevitServices
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager

doc = DocumentManager.Instance.CurrentDBDocument
uidoc=DocumentManager.Instance.CurrentUIApplication.ActiveUIDocument
```

YOUR CODE

```
#Preparing input from dynamo to revit
element = UnwrapElement(IN[0])

#Do some action in a Transaction
TransactionManager.Instance.EnsureInTransaction(doc)

TransactionManager.Instance.TransactionTaskDone()

OUT = element
```

EXERCISE 4 – UNWRAP ELEMENTS

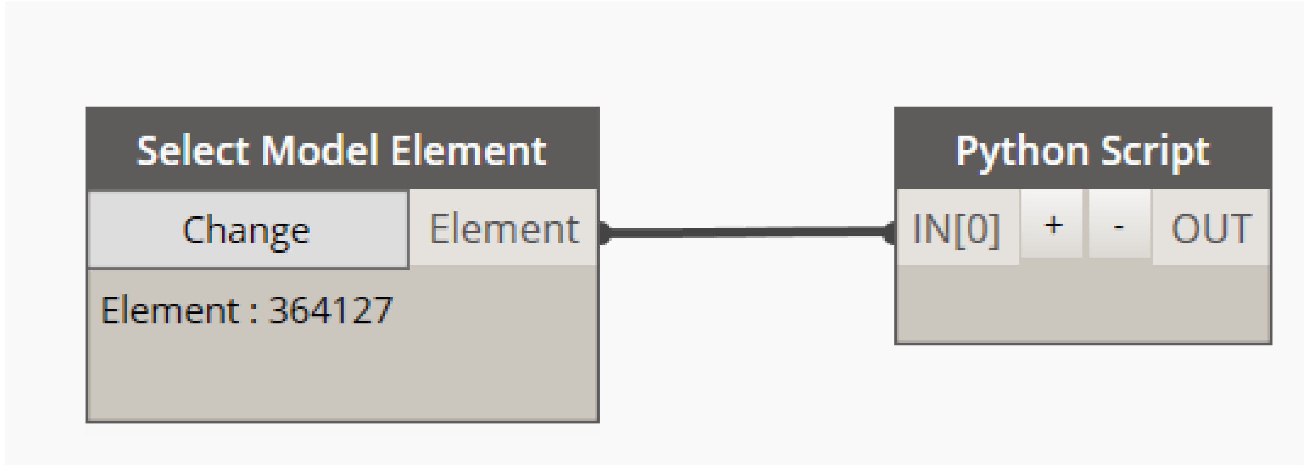
- Another way to use the Python Node is to bypass restrictions you normally encounter with Revit
- Unwrapping takes the Revit native version of the elements allowing you to manipulate them without going through Dynamo's interface
- Requires transaction manager to work which you load from RevitServices 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 prior example

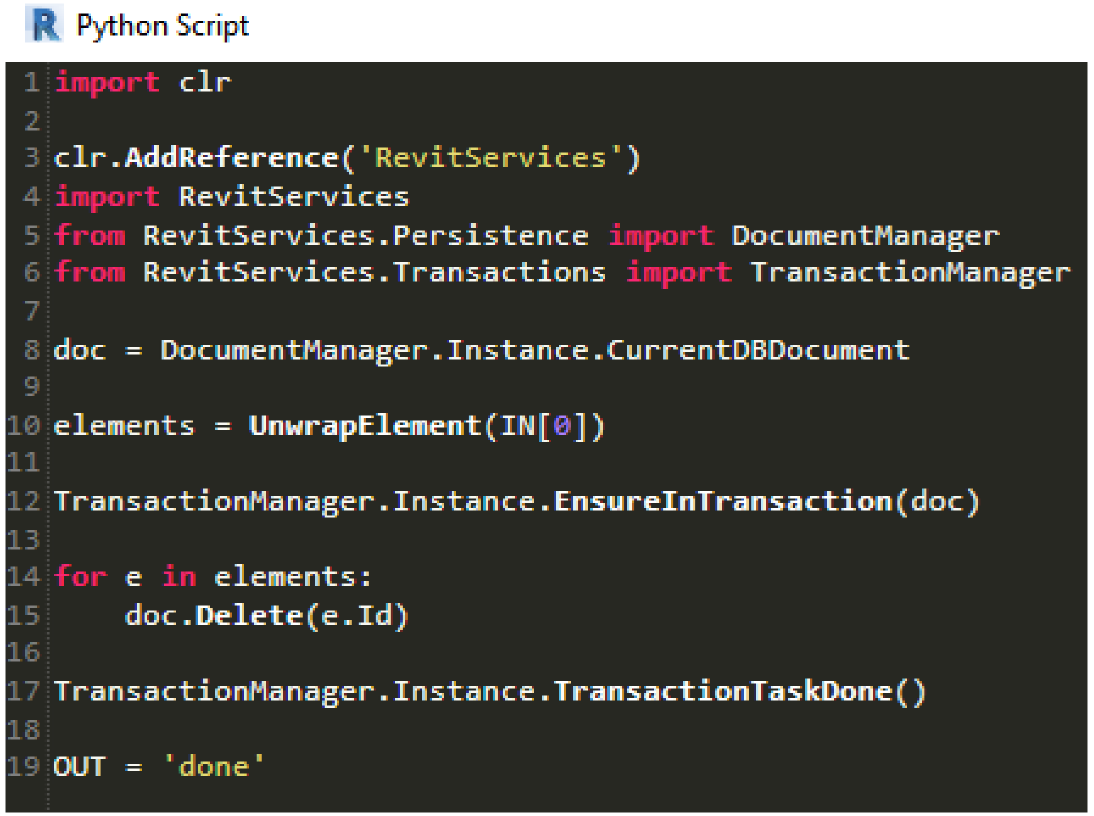


```python
import clr

# Add services for document and transactions

clr.AddReference('RevitServices')
import RevitServices
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager

# set document manager value =

doc = DocumentManager.Instance.CurrentDBDocument

# input elements unwrapped in parentheses

elements = UnwrapElement(IN[0])

# Begin Transaction

TransactionManager.Instance.EnsureInTransaction(doc)

# code here

# End Transaction

TransactionManager.Instance.TransactionTaskDone()

OUT = 0
```

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

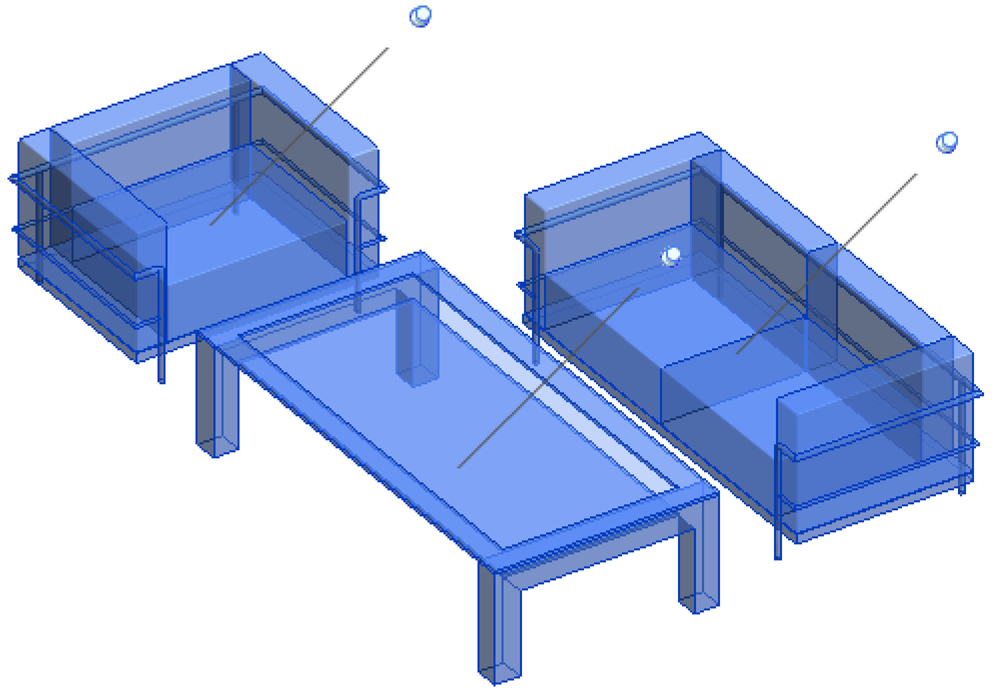




```python
import clr

clr.AddReference('RevitServices')
import RevitServices
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager

doc = DocumentManager.Instance.CurrentDBDocument

elements = UnwrapElement(IN[0])

TransactionManager.Instance.EnsureInTransaction(doc)

for e in elements:
    doc.Delete(e.Id)

TransactionManager.Instance.TransactionTaskDone()

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
FUNCTIONS

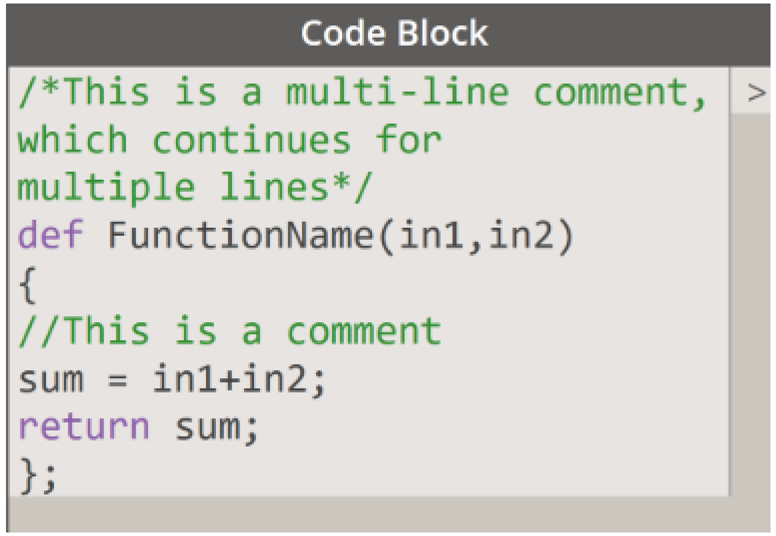
func·tion

/ˈfəNG(k)SH(ə)n/

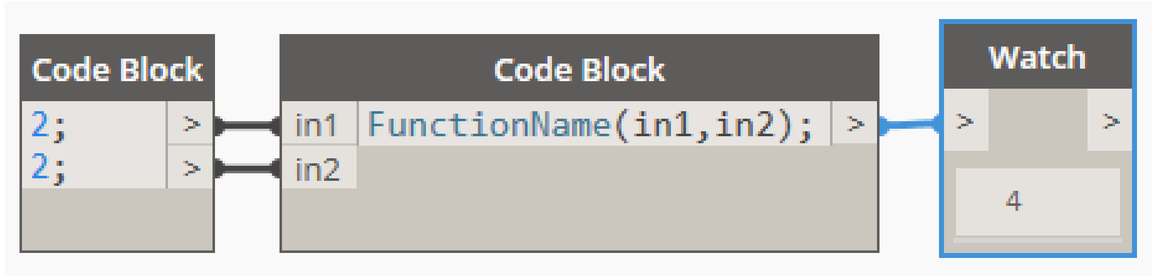
$$y = f(x)$$

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 designscript in Dynamo then you’re most of the way to using Python node because the concepts are similar



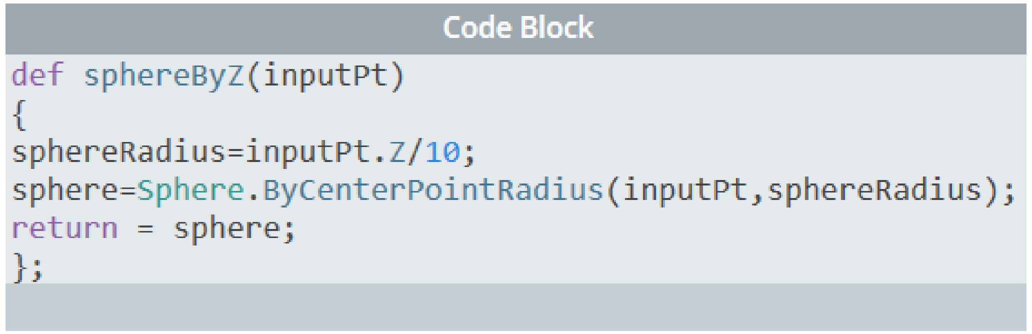
```
/*This is a multi-line comment,
which continues for
multiple lines*/
def FunctionName(in1,in2)
{
//This is a comment
sum = in1+in2;
return sum;
};
```



```
2;
2;
```

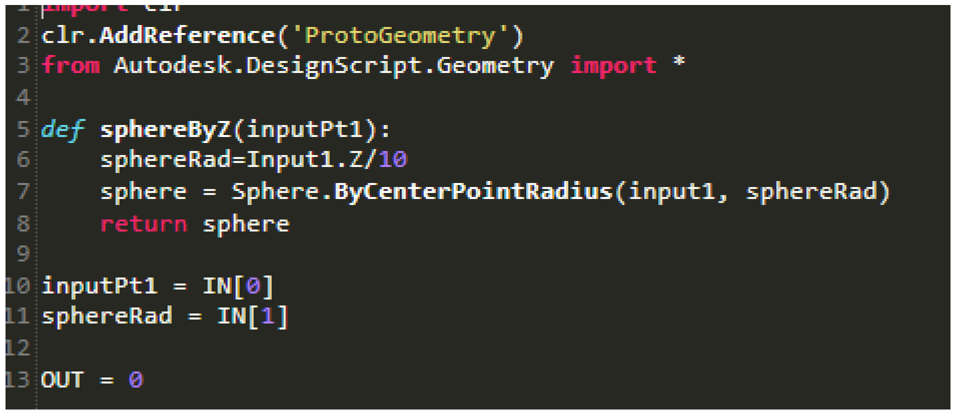
```
FunctionName(in1,in2);
```

EXERCISE 5 DEFINITIONS AND FUNCTIONS

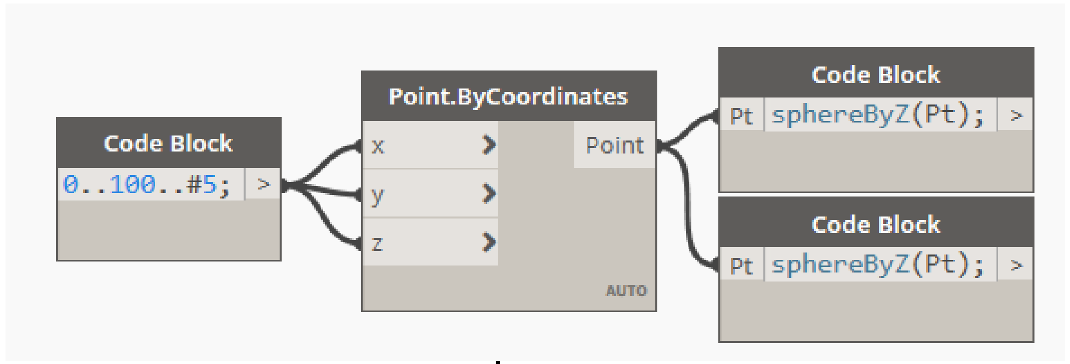


```
def sphereByZ(inputPt)
{
sphereRadius=inputPt.Z/10;
sphere=Sphere.ByCenterPointRadius(inputPt,sphereRadius);
return = sphere;
};
```

≅



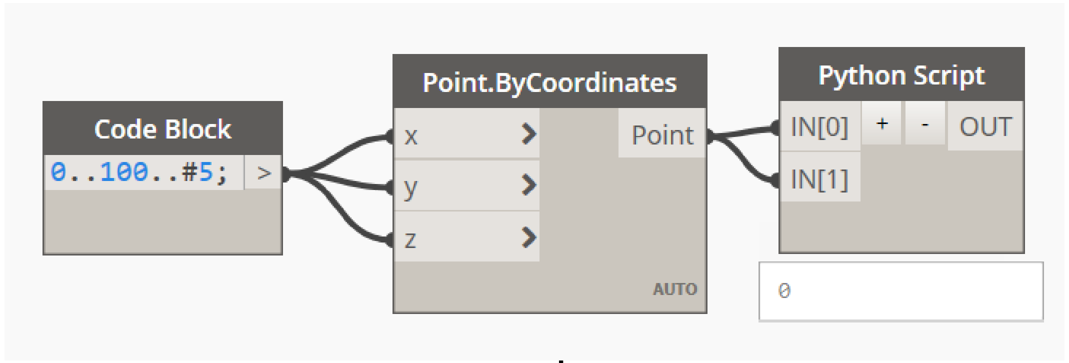
```
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

def sphereByZ(inputPt1):
    sphereRad=Input1.Z/10
    sphere = Sphere.ByCenterPointRadius(input1, sphereRad)
    return sphere

inputPt1 = IN[0]
sphereRad = IN[1]

OUT = 0
```



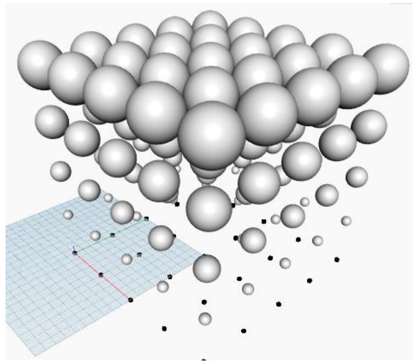
```
0..100..#5;
```

```
sphereByZ(Pt);
```

```
sphereByZ(Pt);
```



```
0..100..#5;
```

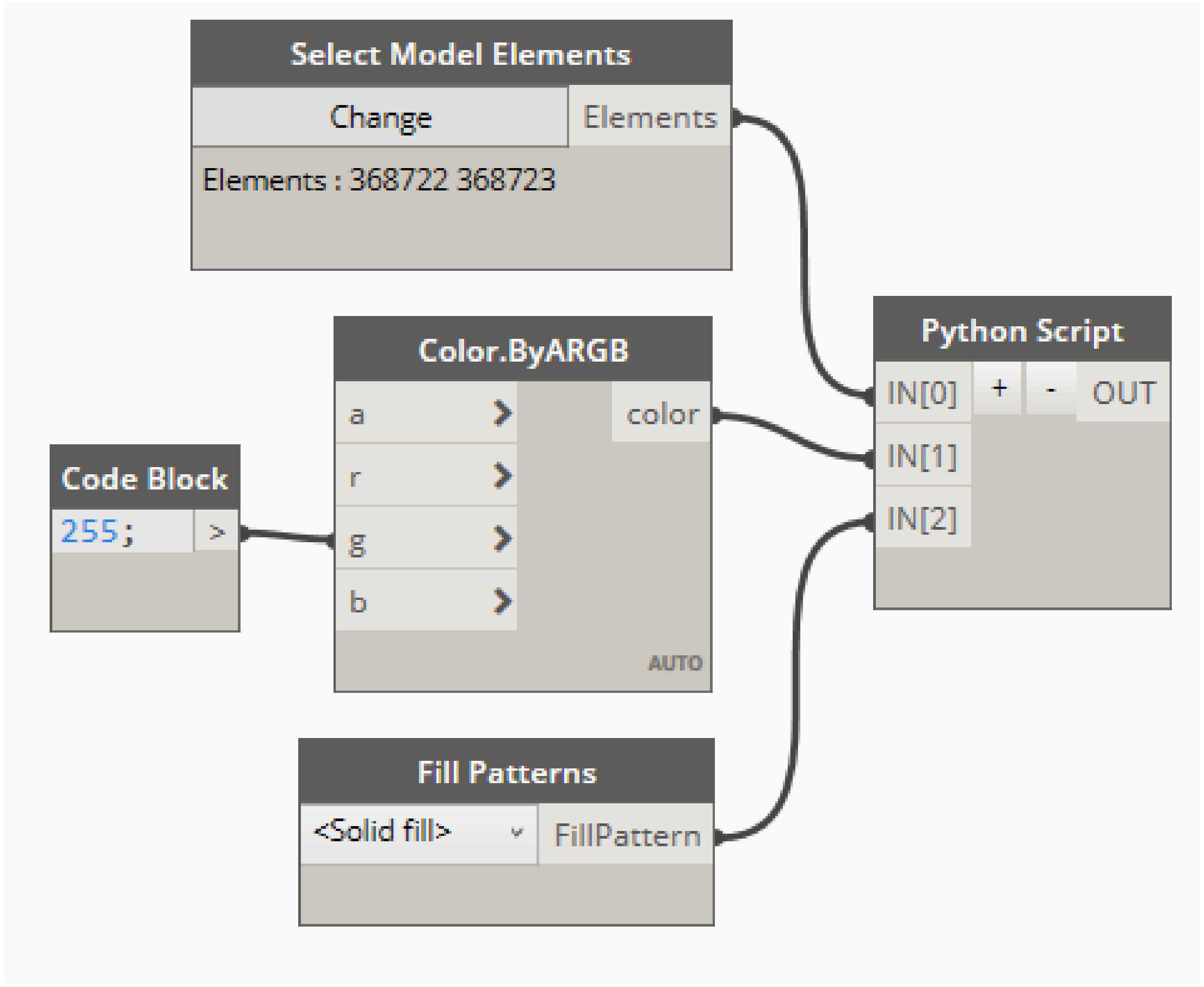


EXERCISE 5 DEFINITIONS AND FUNCTIONS

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

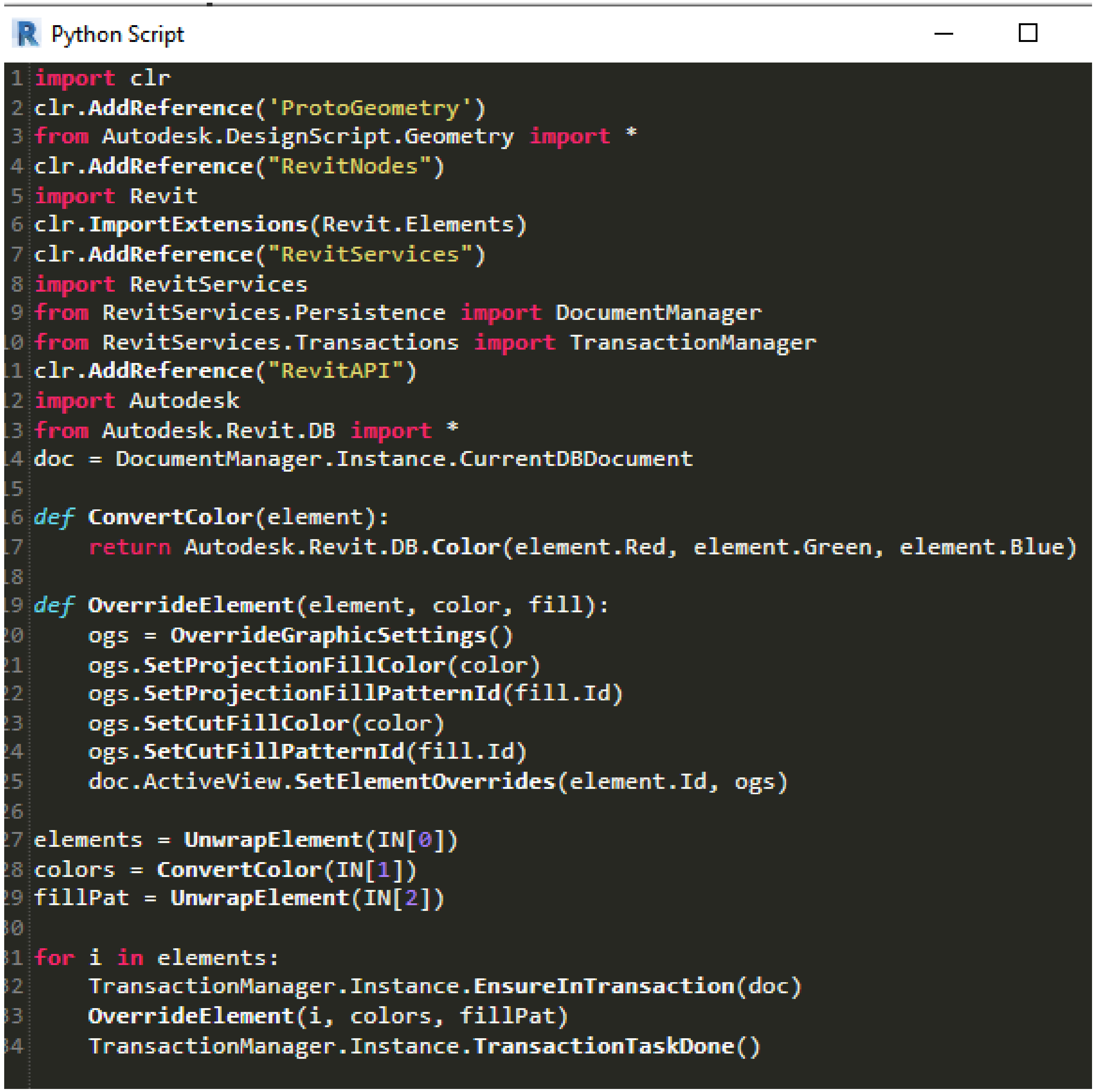
3. The results can be updated live



EXERCISE 5 DEFINITIONS AND FUNCTIONS

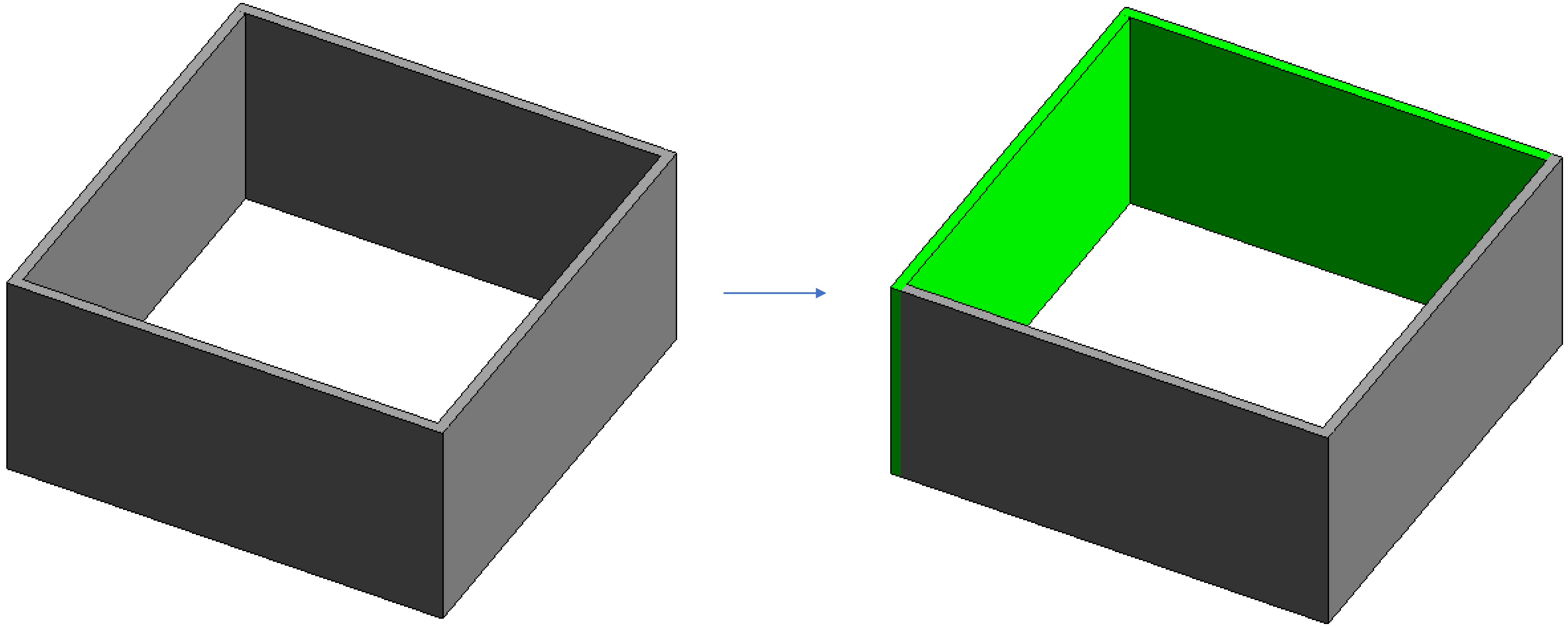
1. Import the services necessary for the process
2. Create a definition with *def*
3. After that add the function
4. Then add the return which stipulates what modifications are supposed to happen
5. You have to know what you need to adjust in order to make the correct change to the elements and parameters in use

Credit: DannySBentley



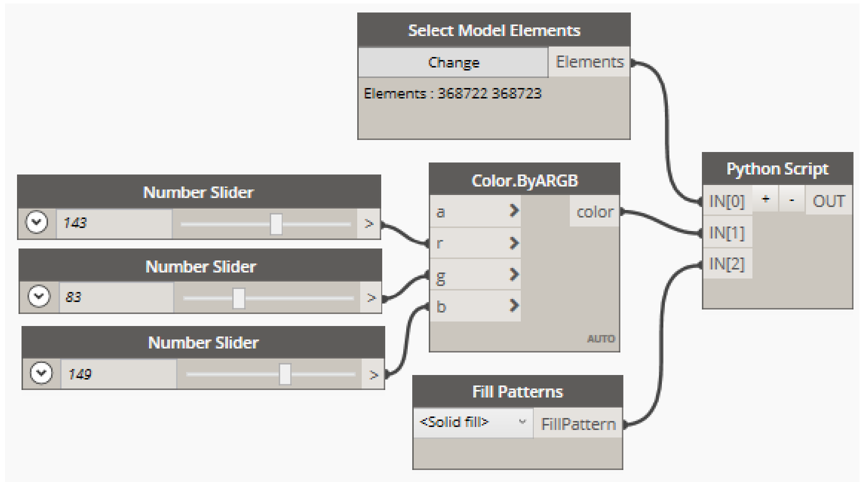
```python
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
clr.AddReference("RevitNodes")
import Revit
clr.ImportExtensions(Revit.Elements)
clr.AddReference("RevitServices")
import RevitServices
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager
clr.AddReference("RevitAPI")
import Autodesk
from Autodesk.Revit.DB import *
doc = DocumentManager.Instance.CurrentDBDocument

def ConvertColor(element):
    return Autodesk.Revit.DB.Color(element.Red, element.Green, element.Blue)

def OverrideElement(element, color, fill):
    ogs = OverrideGraphicSettings()
    ogs.SetProjectionFillColor(color)
    ogs.SetProjectionFillPatternId(fill.Id)
    ogs.SetCutFillColor(color)
    ogs.SetCutFillPatternId(fill.Id)
    doc.ActiveView.SetElementOverrides(element.Id, ogs)

elements = UnwrapElement(IN[0])
colors = ConvertColor(IN[1])
fillPat = UnwrapElement(IN[2])

for i in elements:
    TransactionManager.Instance.EnsureInTransaction(doc)
    OverrideElement(i, colors, fillPat)
    TransactionManager.Instance.TransactionTaskDone()
```

EXERCISE 5 DEFINITIONS AND FUNCTIONS



OUT (print)

EXERCISE 5 DEFINITIONS AND FUNCTIONS



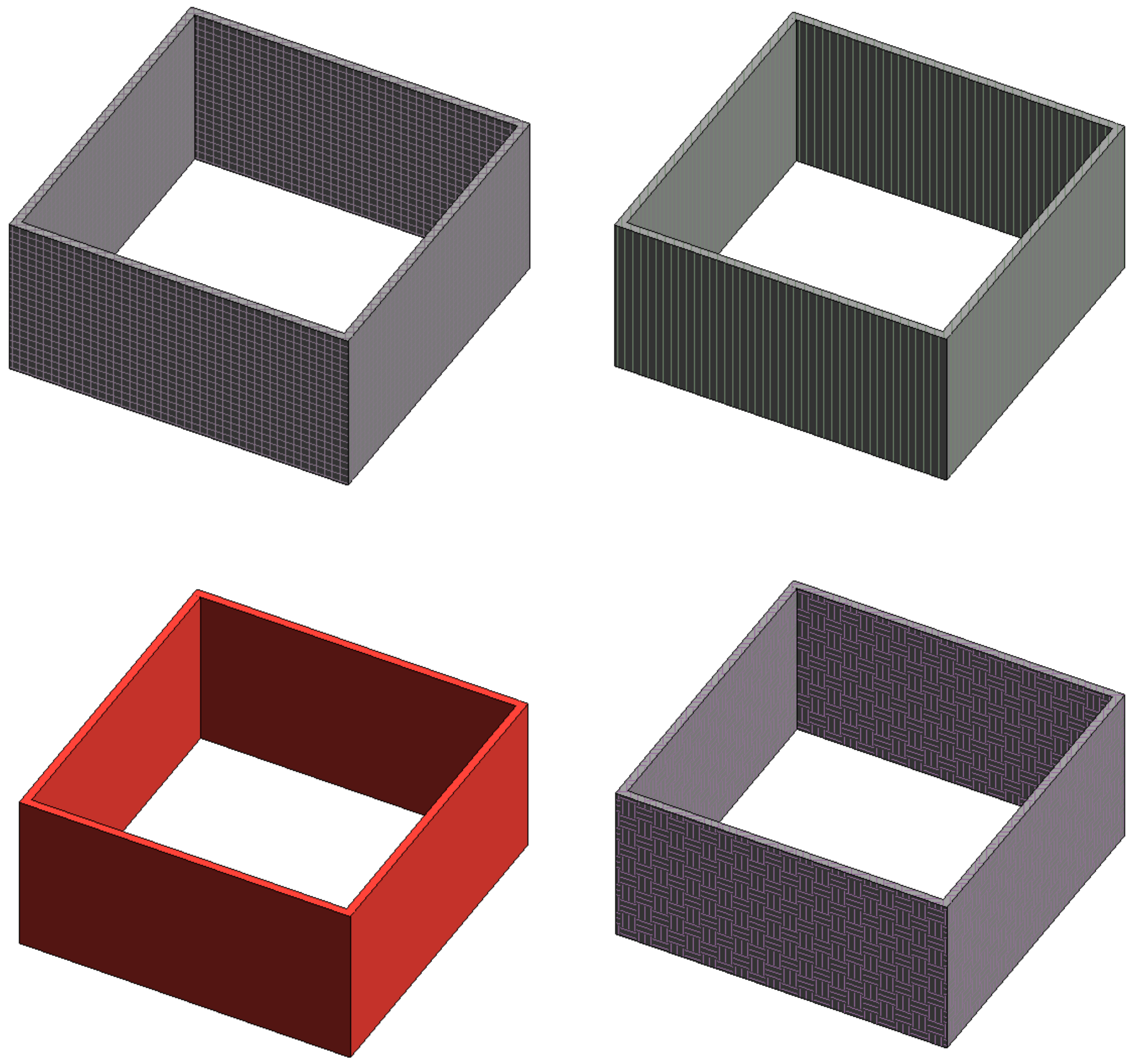
Select Model Elements
Change | Elements
Elements : 368722 368723

Number Slider 143
Number Slider 83
Number Slider 149

Color.ByARGB
a | r | g | b | color

Fill Patterns
<Solid fill> | FillPattern

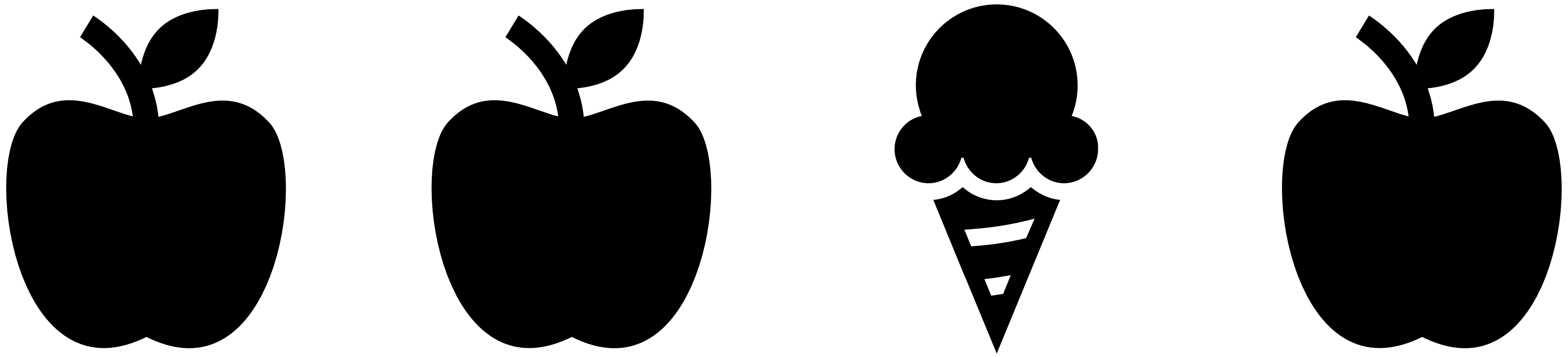
Python Script
IN[0] | IN[1] | IN[2] | OUT



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

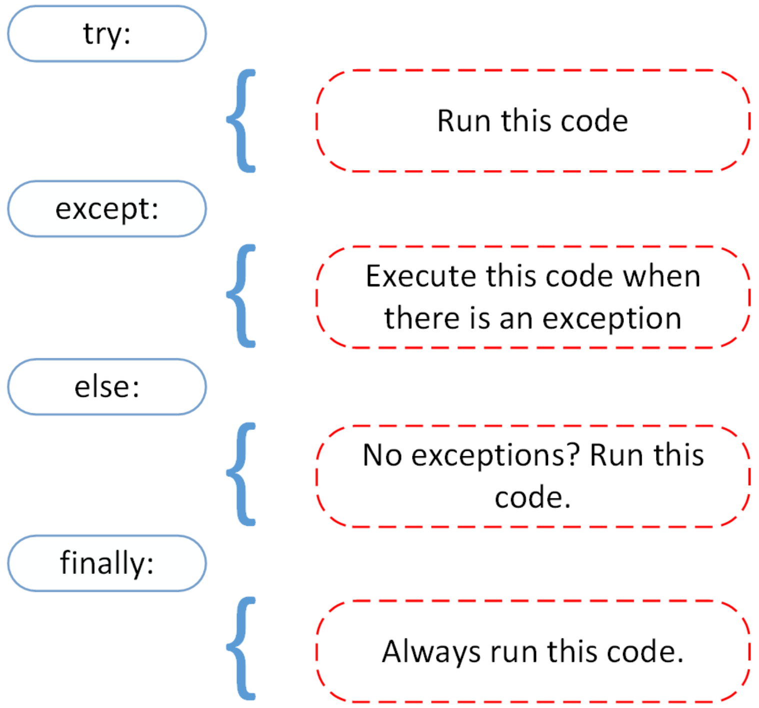
EXERCISE 6
TRY & EXCEPT

EXERCISE 6 – Try and Except

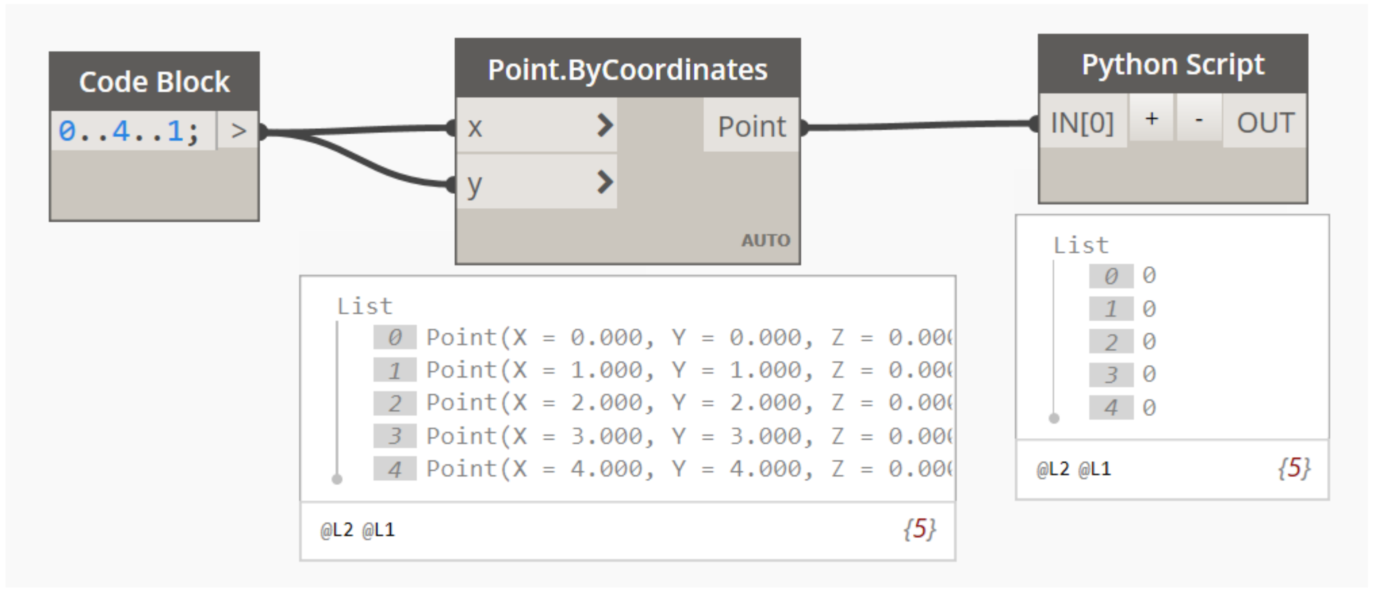


EXERCISE 6 – Try and Except

- 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 break with an except function

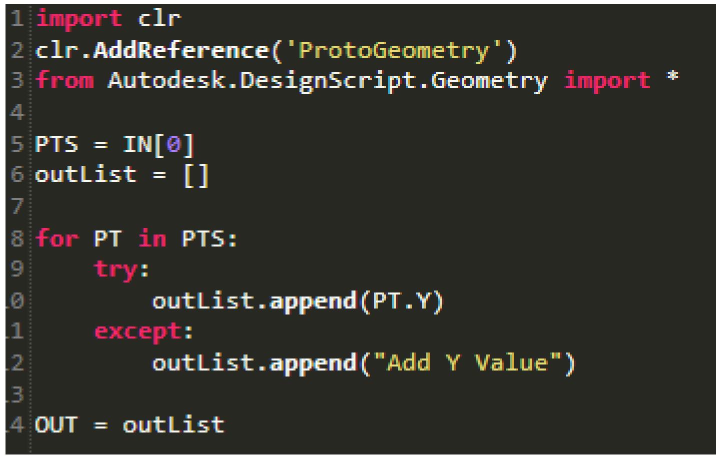

try:
Run this code
except:
Execute this code when there is an exception
else:
No exceptions? Run this code.
finally:
Always run this code.

EXERCISE 6

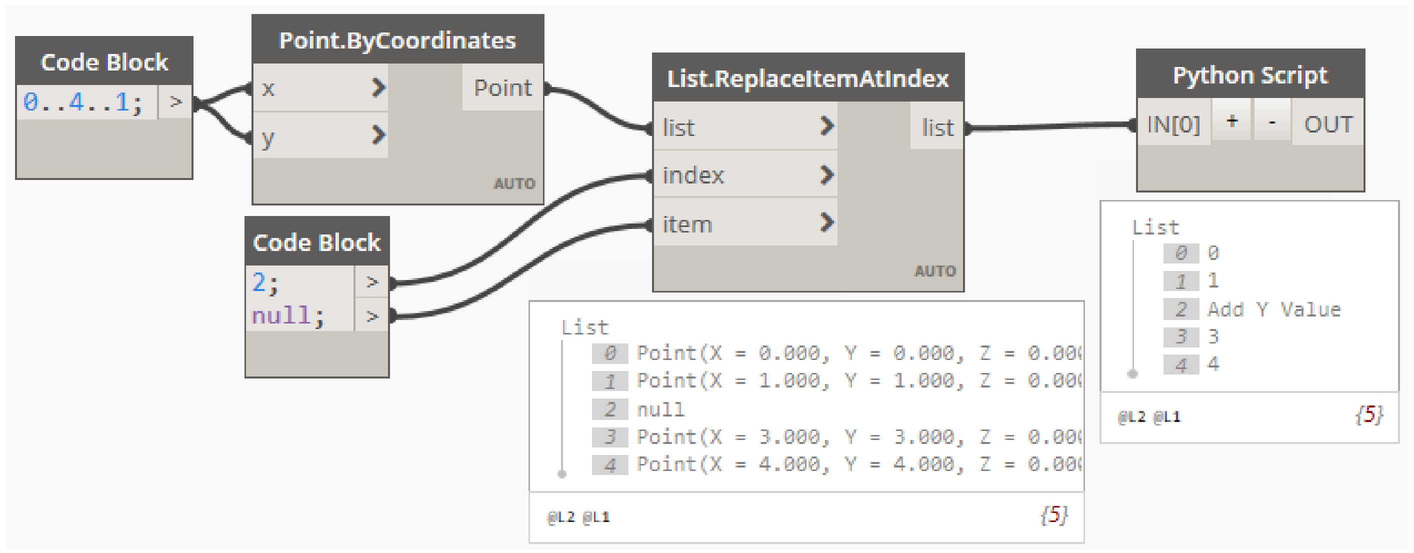


```
List
  0  Point(X = 0.000, Y = 0.000, Z = 0.00
  1  Point(X = 1.000, Y = 1.000, Z = 0.00
  2  Point(X = 2.000, Y = 2.000, Z = 0.00
  3  Point(X = 3.000, Y = 3.000, Z = 0.00
  4  Point(X = 4.000, Y = 4.000, Z = 0.00
@L2 @L1                                {5}
```

```
List
  0  0
  1  0
  2  0
  3  0
  4  0
@L2 @L1      {5}
```



```python
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

PTS = IN[0]
outList = []

for PT in PTS:
    try:
        outList.append(PT.Y)
    except:
        outList.append("Add Y Value")

OUT = outList
```

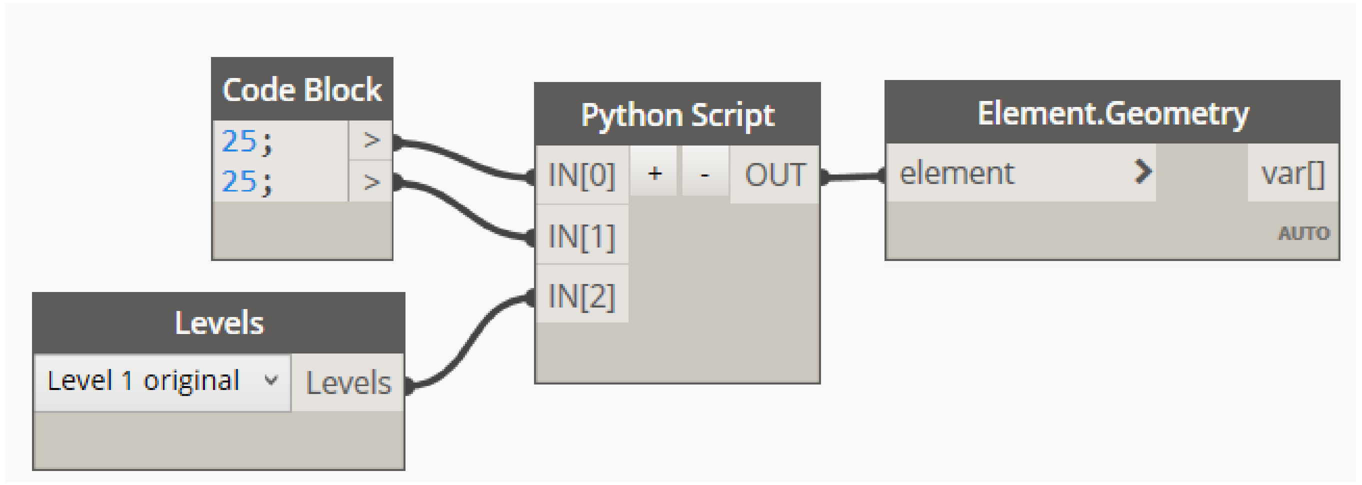


```
List
  0  Point(X = 0.000, Y = 0.000, Z = 0.00
  1  Point(X = 1.000, Y = 1.000, Z = 0.00
  2  null
  3  Point(X = 3.000, Y = 3.000, Z = 0.00
  4  Point(X = 4.000, Y = 4.000, Z = 0.00
@L2 @L1                                {5}
```

```
List
  0  0
  1  1
  2  Add Y Value
  3  3
  4  4
@L2 @L1      {5}
```

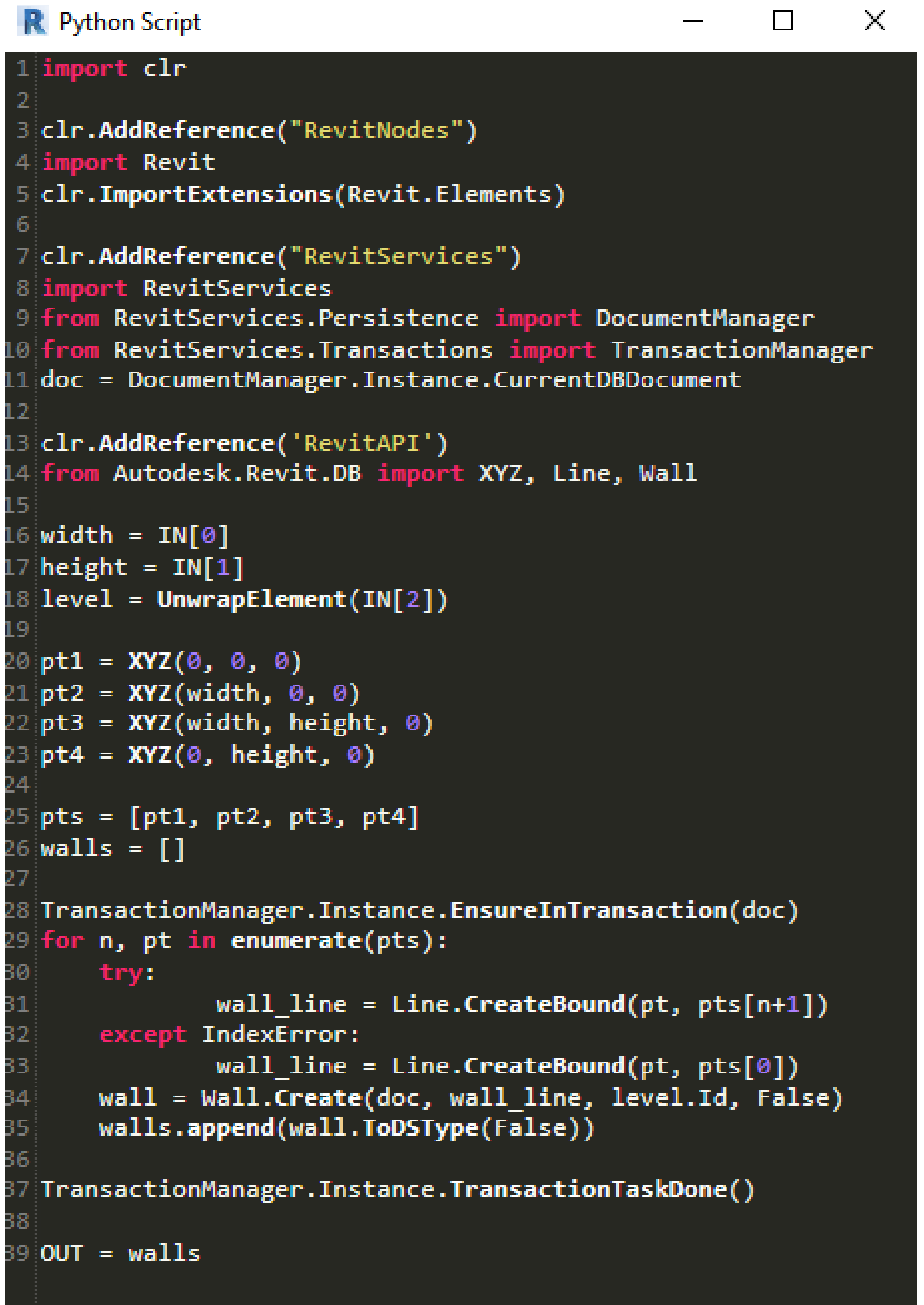
EXERCISE 6

- 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



EXERCISE 6

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

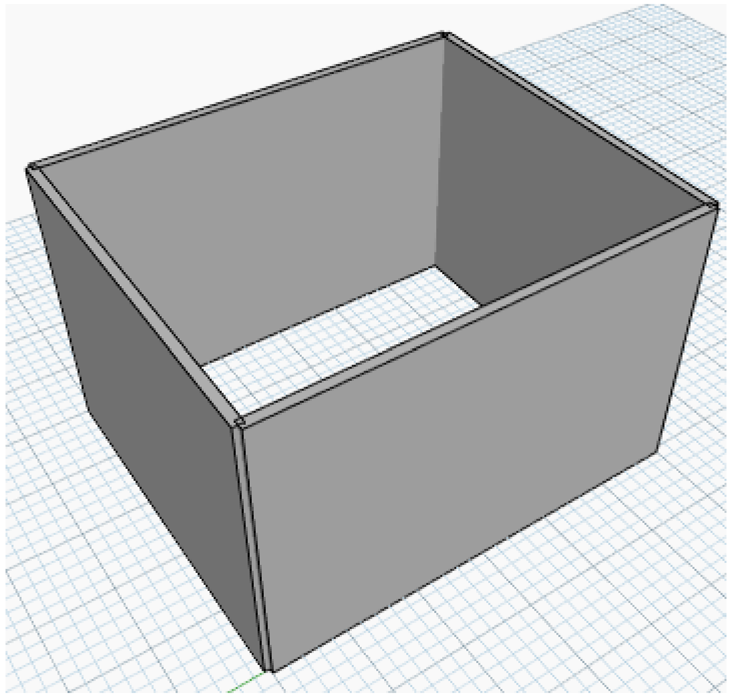


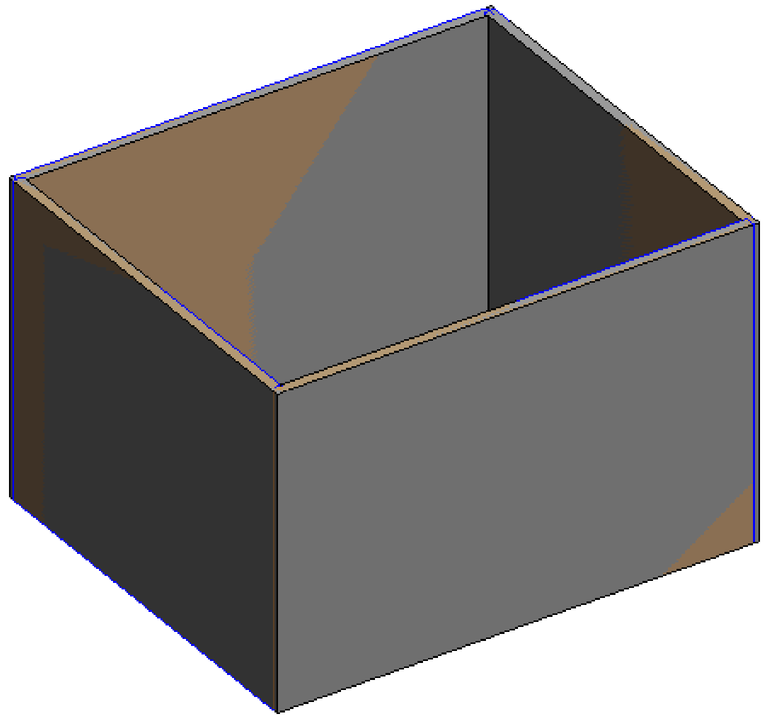
Python Script

```python
import clr

clr.AddReference("RevitNodes")
import Revit
clr.ImportExtensions(Revit.Elements)

clr.AddReference("RevitServices")
import RevitServices
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager
doc = DocumentManager.Instance.CurrentDBDocument

clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import XYZ, Line, Wall

width = IN[0]
height = IN[1]
level = UnwrapElement(IN[2])

pt1 = XYZ(0, 0, 0)
pt2 = XYZ(width, 0, 0)
pt3 = XYZ(width, height, 0)
pt4 = XYZ(0, height, 0)

pts = [pt1, pt2, pt3, pt4]
walls = []

TransactionManager.Instance.EnsureInTransaction(doc)
for n, pt in enumerate(pts):
    try:
            wall_line = Line.CreateBound(pt, pts[n+1])
    except IndexError:
            wall_line = Line.CreateBound(pt, pts[0])
    wall = Wall.Create(doc, wall_line, level.Id, False)
    walls.append(wall.ToDSType(False))

TransactionManager.Instance.TransactionTaskDone()

OUT = walls
```

X-OUTLIST

EXERCISE 6





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

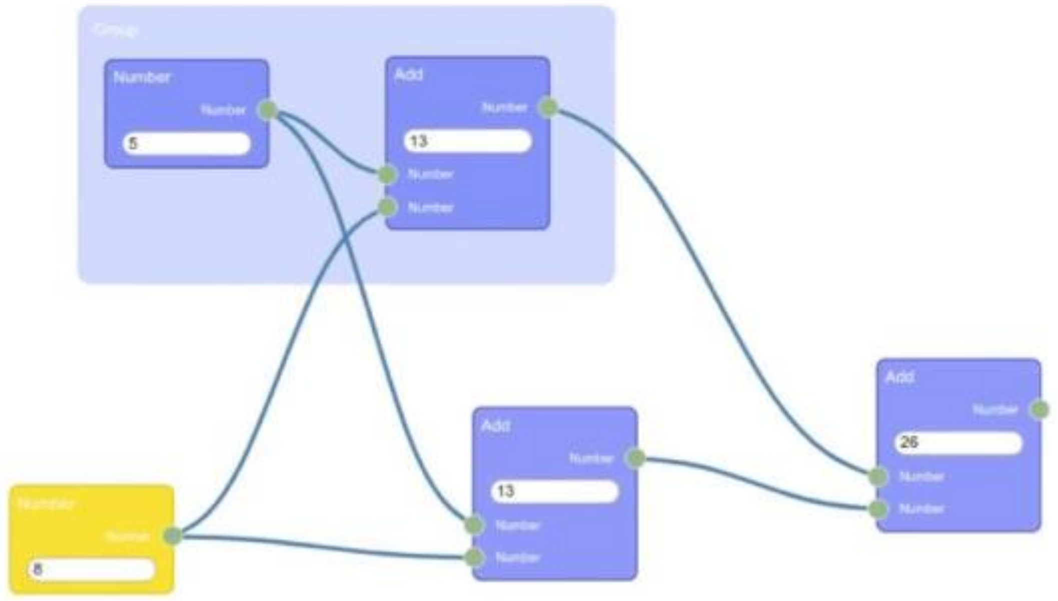
The more restraints an element has the more you must consider in your program so that it functions

EXERCISE 7
GRAPHS

EXERCISE 7 - GRAPHS

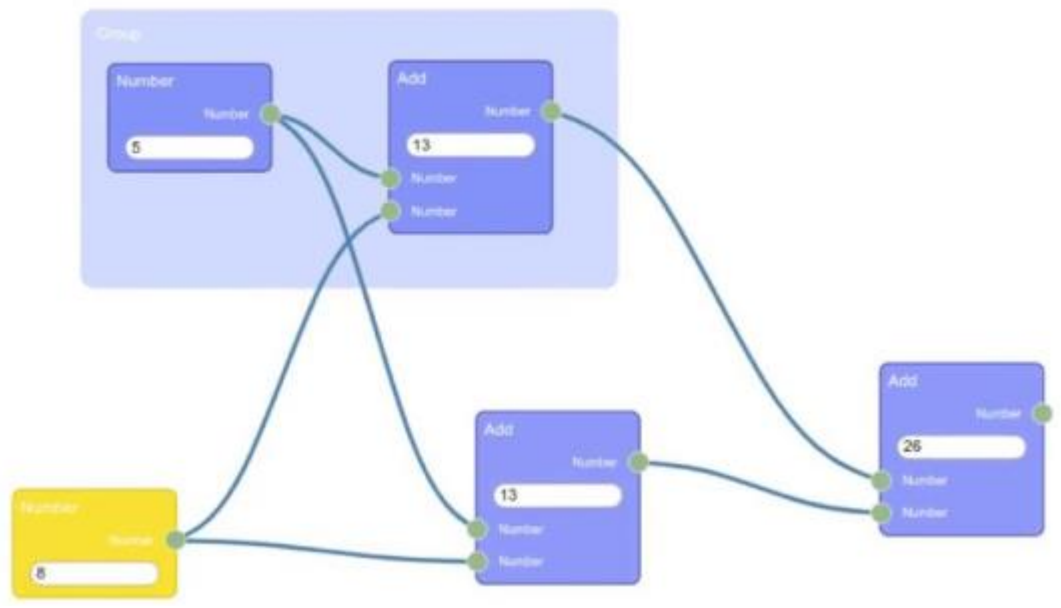
EXERCISE 7 - GRAPHS

- 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

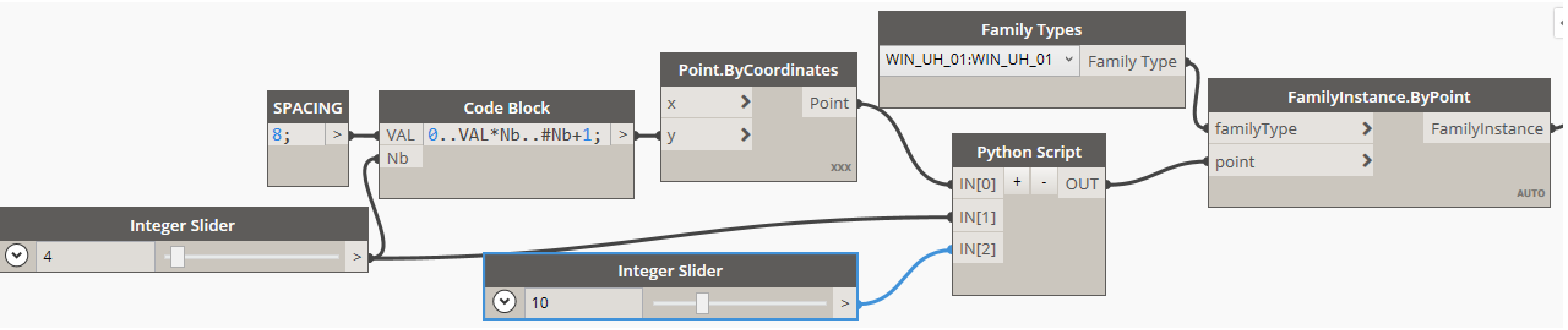


EXERCISE 7 - GRAPHS

- 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

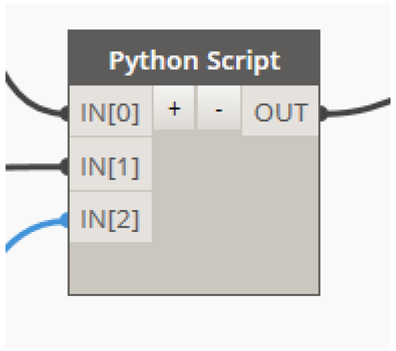


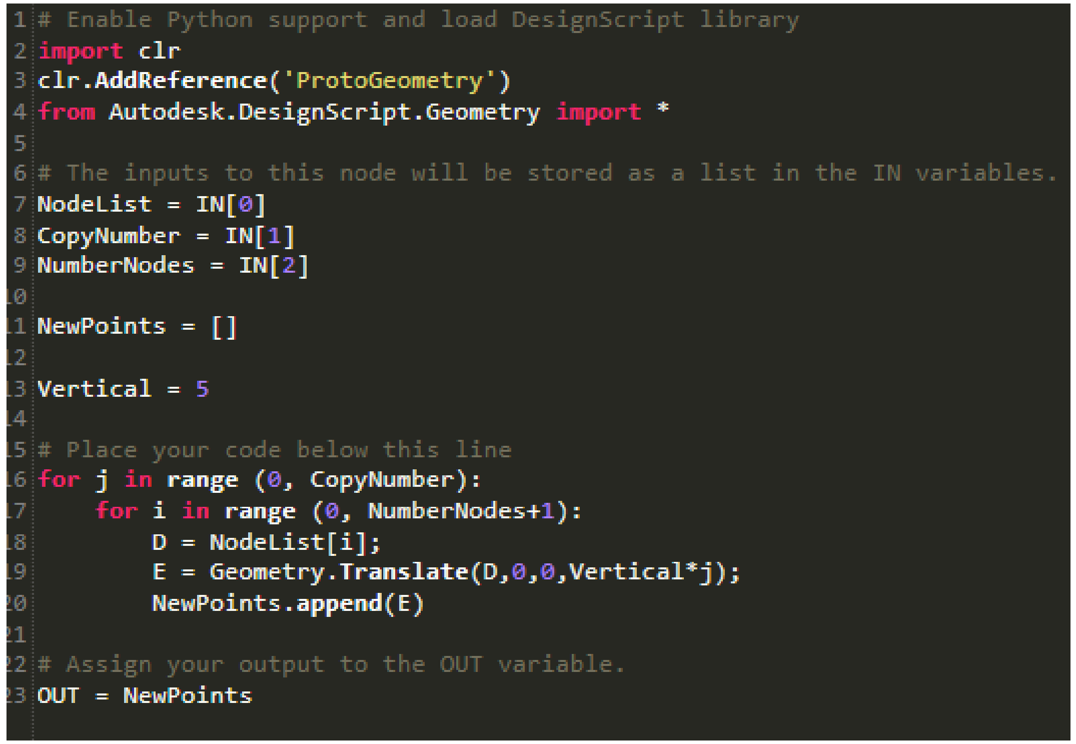
EXERCISE 7 - GRAPHS



EXERCISE 7 - GRAPHS

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

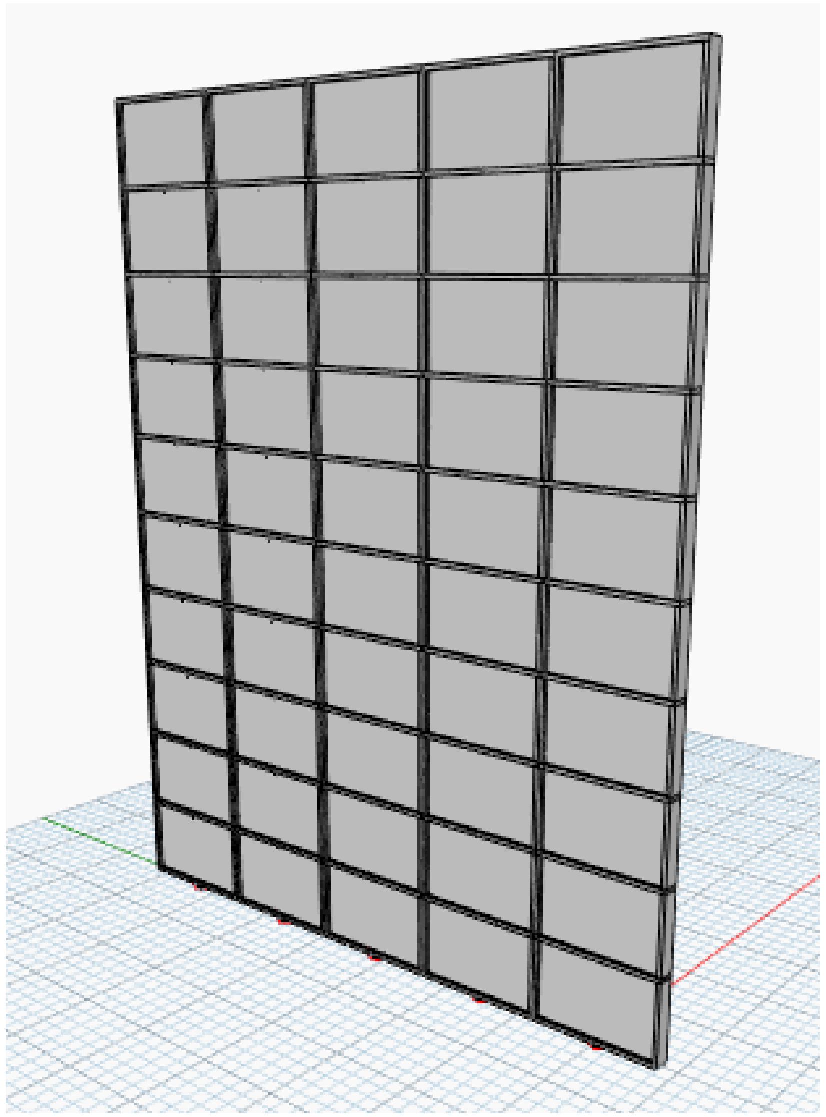


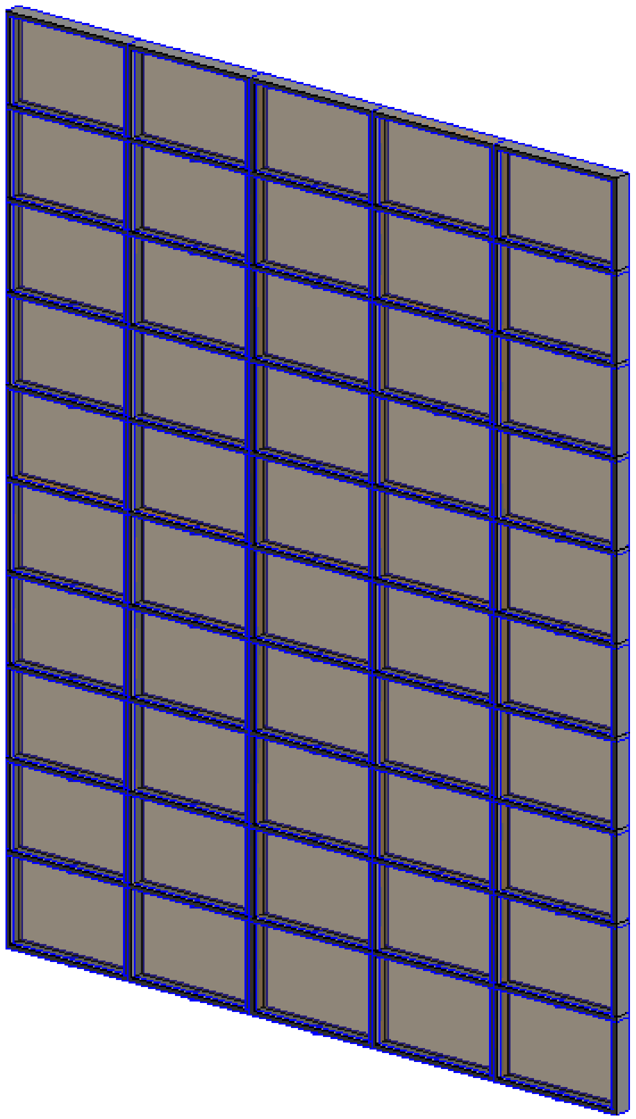


```python
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

# The inputs to this node will be stored as a list in the IN variables.
NodeList = IN[0]
CopyNumber = IN[1]
NumberNodes = IN[2]

NewPoints = []

Vertical = 5

# Place your code below this line
for j in range (0, CopyNumber):
    for i in range (0, NumberNodes+1):
        D = NodeList[i];
        E = Geometry.Translate(D,0,0,Vertical*j);
        NewPoints.append(E)

# Assign your output to the OUT variable.
OUT = NewPoints
```

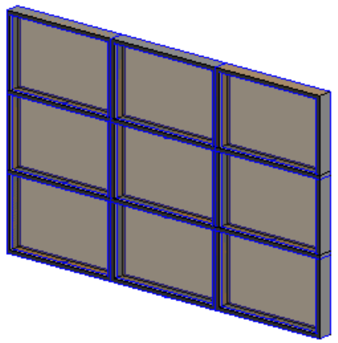
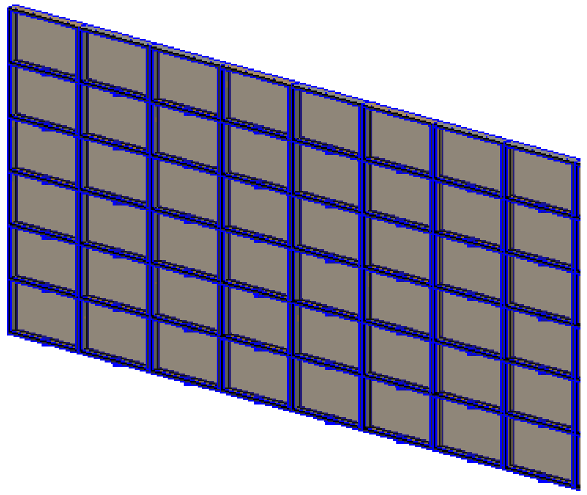
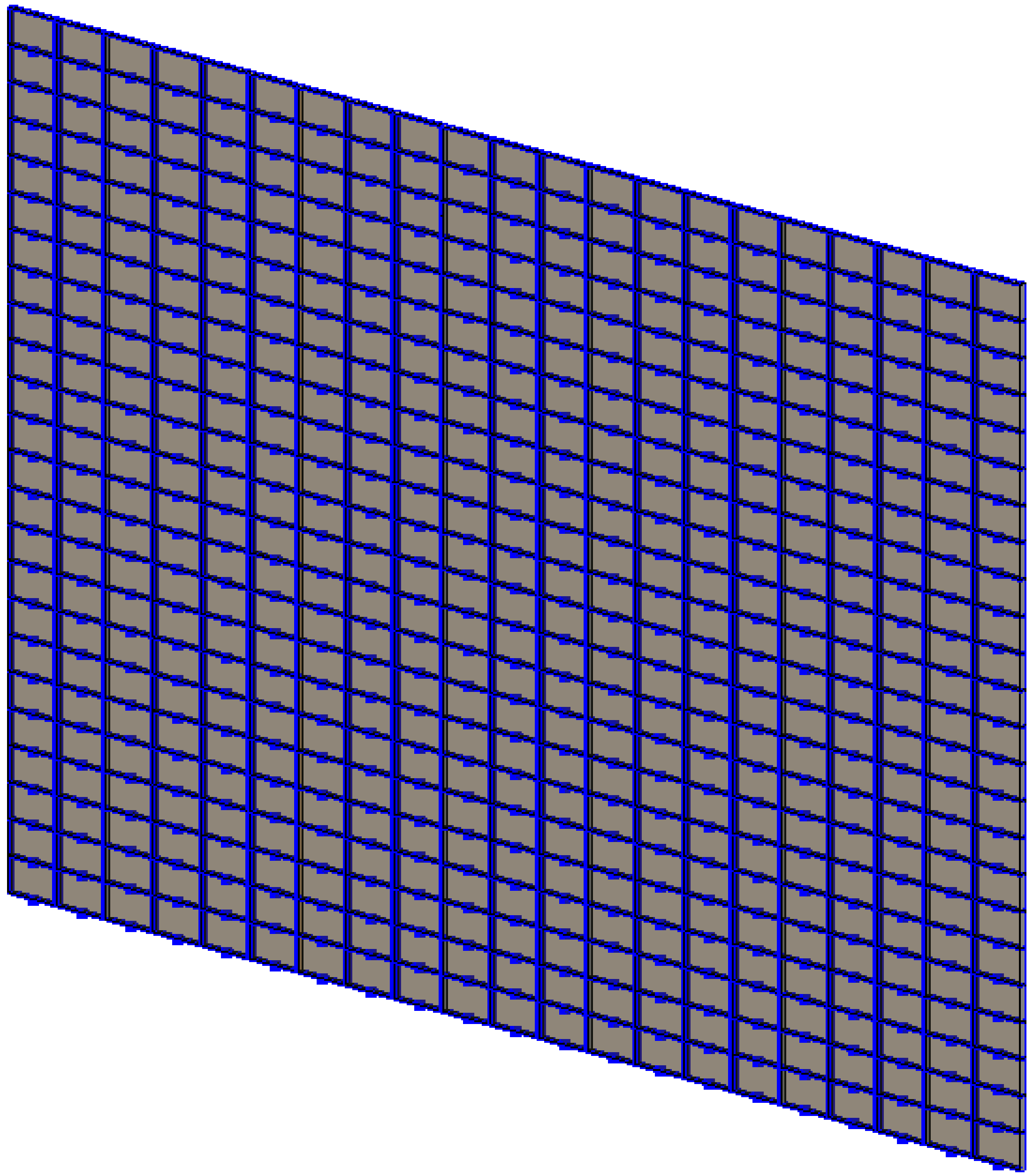
X-[]

EXERCISE 7 - GRAPHS





EXERCISE 7 - GRAPHS

PROBLEM SOLVING

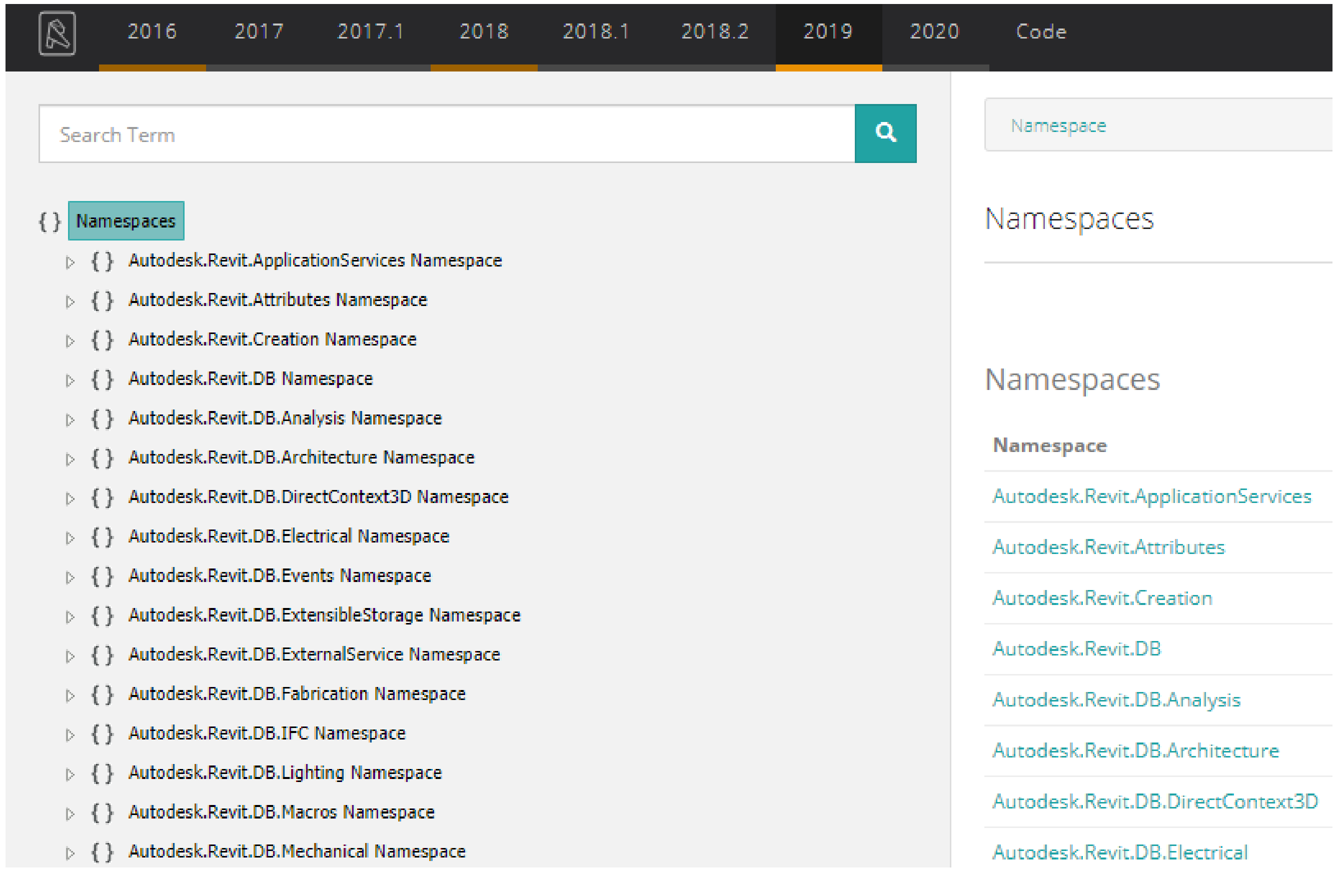
PROBLEM SOLVING

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 is where you should start looking
- Find the function you want to change and search for it in the Website

Revit API Docs

Online Documentation for the Revit API



{ } Namespaces
- { } Autodesk.Revit.ApplicationServices Namespace
- { } Autodesk.Revit.Attributes Namespace
- { } Autodesk.Revit.Creation Namespace
- { } Autodesk.Revit.DB Namespace
- { } Autodesk.Revit.DB.Analysis Namespace
- { } Autodesk.Revit.DB.Architecture Namespace
- { } Autodesk.Revit.DB.DirectContext3D Namespace
- { } Autodesk.Revit.DB.Electrical Namespace
- { } Autodesk.Revit.DB.Events Namespace
- { } Autodesk.Revit.DB.ExtensibleStorage Namespace
- { } Autodesk.Revit.DB.ExternalService Namespace
- { } Autodesk.Revit.DB.Fabrication Namespace
- { } Autodesk.Revit.DB.IFC Namespace
- { } Autodesk.Revit.DB.Lighting Namespace
- { } Autodesk.Revit.DB.Macros Namespace
- { } Autodesk.Revit.DB.Mechanical Namespace

Namespaces

Namespaces

Namespace
Autodesk.Revit.ApplicationServices
Autodesk.Revit.Attributes
Autodesk.Revit.Creation
Autodesk.Revit.DB
Autodesk.Revit.DB.Analysis
Autodesk.Revit.DB.Architecture
Autodesk.Revit.DB.DirectContext3D
Autodesk.Revit.DB.Electrical

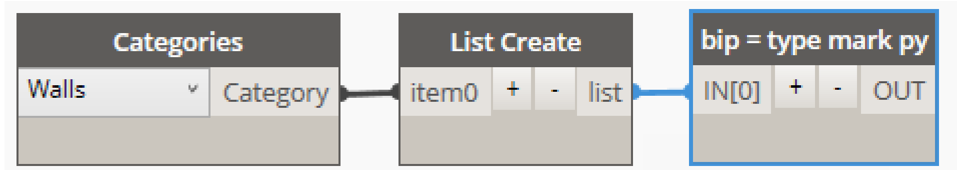
PROBLEM SOLVING

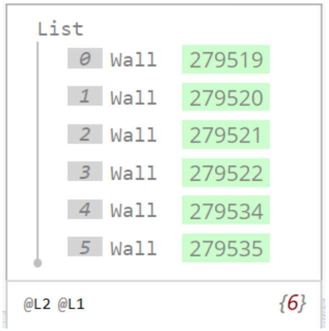
1. Typical scenario is you get a script that you did not author and you want to modify it to something different

2. For example this script looks up an element's Type Mark parameter but you want to see the comment parameter

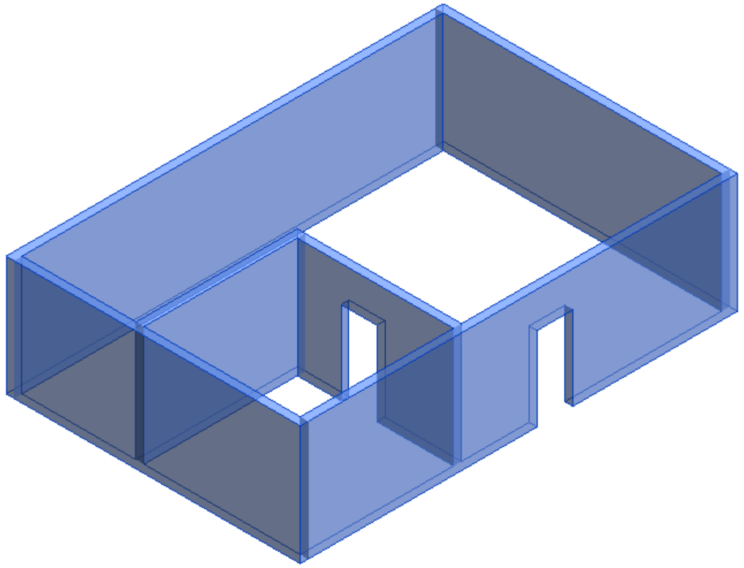
3. In this example the script finds the wall Type Mark but we don't want all the walls just the ones with comments

4. If you don't change the way the code reads the parameter you can't select what you want





```
List
  0 Wall  279519
  1 Wall  279520
  2 Wall  279521
  3 Wall  279522
  4 Wall  279534
  5 Wall  279535
@L2 @L1                {6}
```



PROBLEM SOLVING

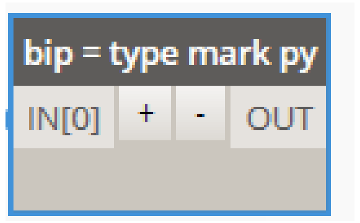
1. Open the script

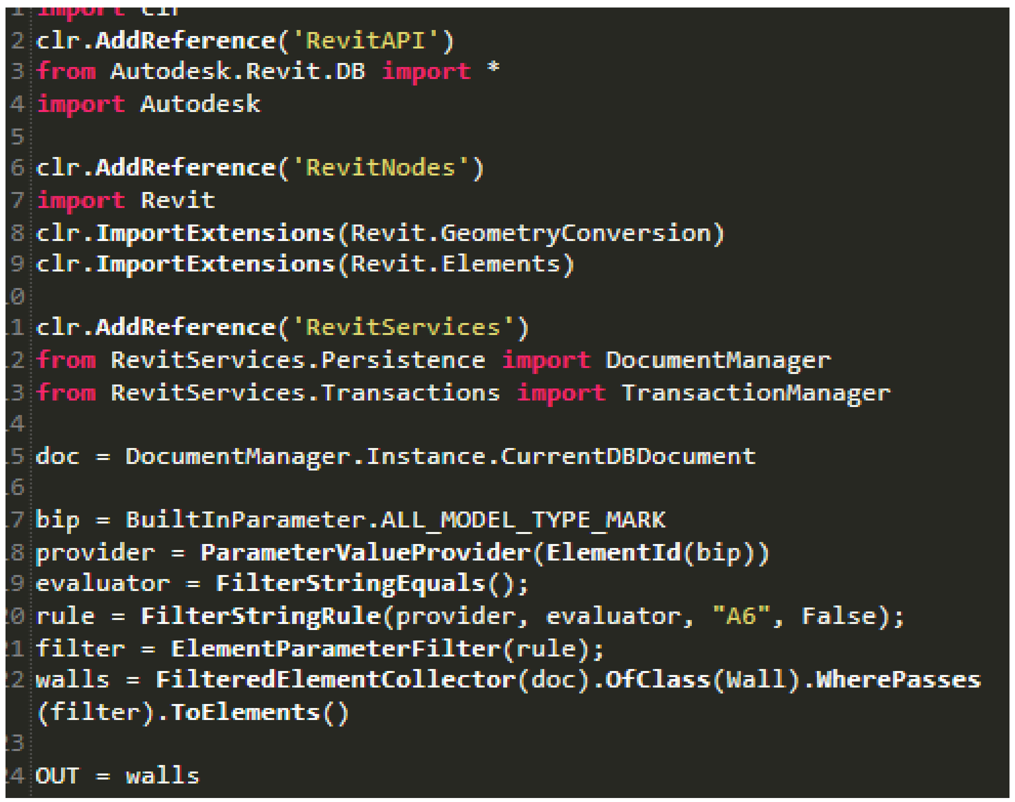
2. You see that the only input is coming from the built in parameter

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





```
import clr
clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *
import Autodesk

clr.AddReference('RevitNodes')
import Revit
clr.ImportExtensions(Revit.GeometryConversion)
clr.ImportExtensions(Revit.Elements)

clr.AddReference('RevitServices')
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager

doc = DocumentManager.Instance.CurrentDBDocument

bip = BuiltInParameter.ALL_MODEL_TYPE_MARK
provider = ParameterValueProvider(ElementId(bip))
evaluator = FilterStringEquals();
rule = FilterStringRule(provider, evaluator, "A6", False);
filter = ElementParameterFilter(rule);
walls = FilteredElementCollector(doc).OfClass(Wall).WherePasses
(filter).ToElements()

OUT = walls
```

PROBLEM SOLVING

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

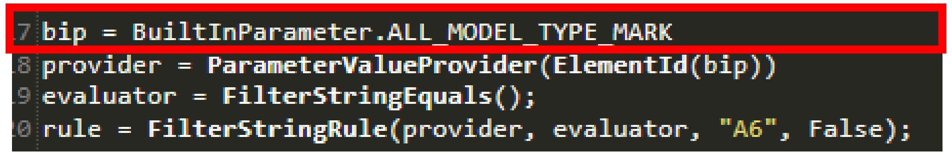
2. Go to Revitapidocs.com for the documentation

3. Search for the BuiltInParameter and you'll get a result to click on

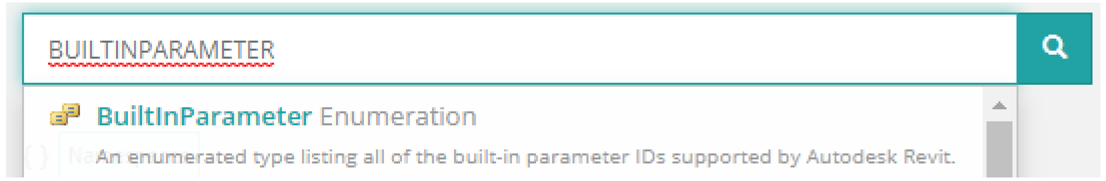
4. Search the list for mark parameter to see if it exists

5. Then do the same for the comment parameter

6. You now see the function for comments to copy into the code



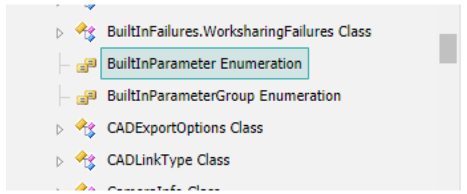
```
bip = BuiltInParameter.ALL_MODEL_TYPE_MARK
provider = ParameterValueProvider(ElementId(bip))
evaluator = FilterStringEquals();
rule = FilterStringRule(provider, evaluator, "A6", False);
```



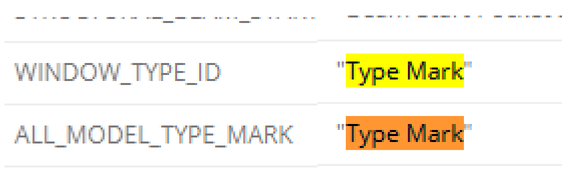
BUILTINPARAMETER

BuiltInParameter Enumeration

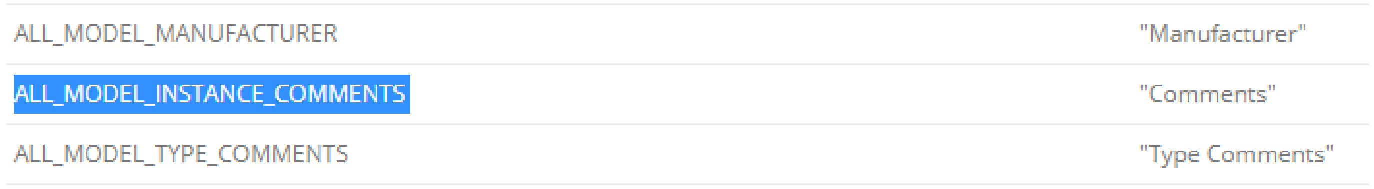
An enumerated type listing all of the built-in parameter IDs supported by Autodesk Revit.



- BuiltInFailures.WorksharingFailures Class
- BuiltInParameter Enumeration
- BuiltInParameterGroup Enumeration
- CADExportOptions Class
- CADLinkType Class



WINDOW_TYPE_ID	"Type Mark"
ALL_MODEL_TYPE_MARK	"Type Mark"



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

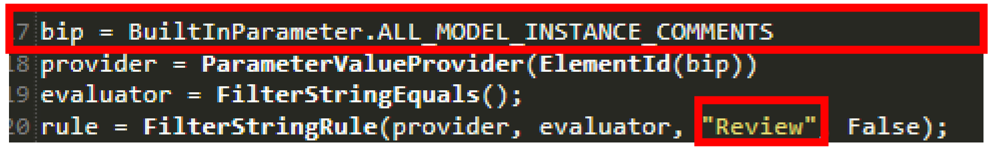
PROBLEM SOLVING

1. Copy Paste the comment parameter into the code replacing the Type Mark parameter

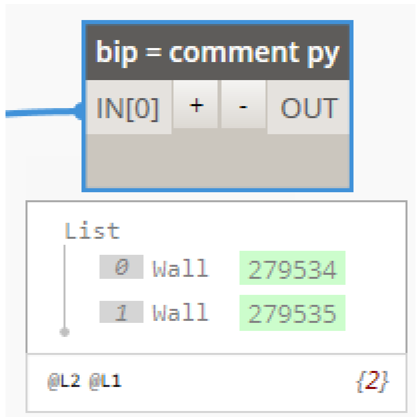
2. Look for the text contained in the Revit element comments field and write it in the Python code string value

3. Run the script and you'll see the elements listed with that value

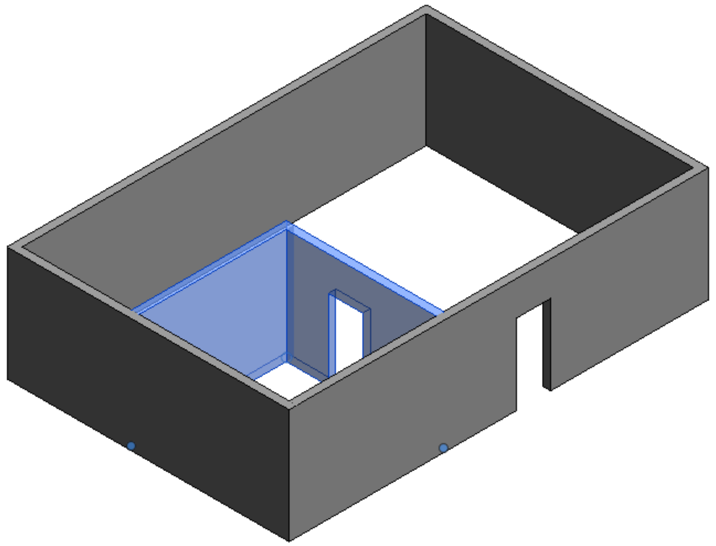
4. Modifications like this are typical for getting one purpose made code to do another task based on document research



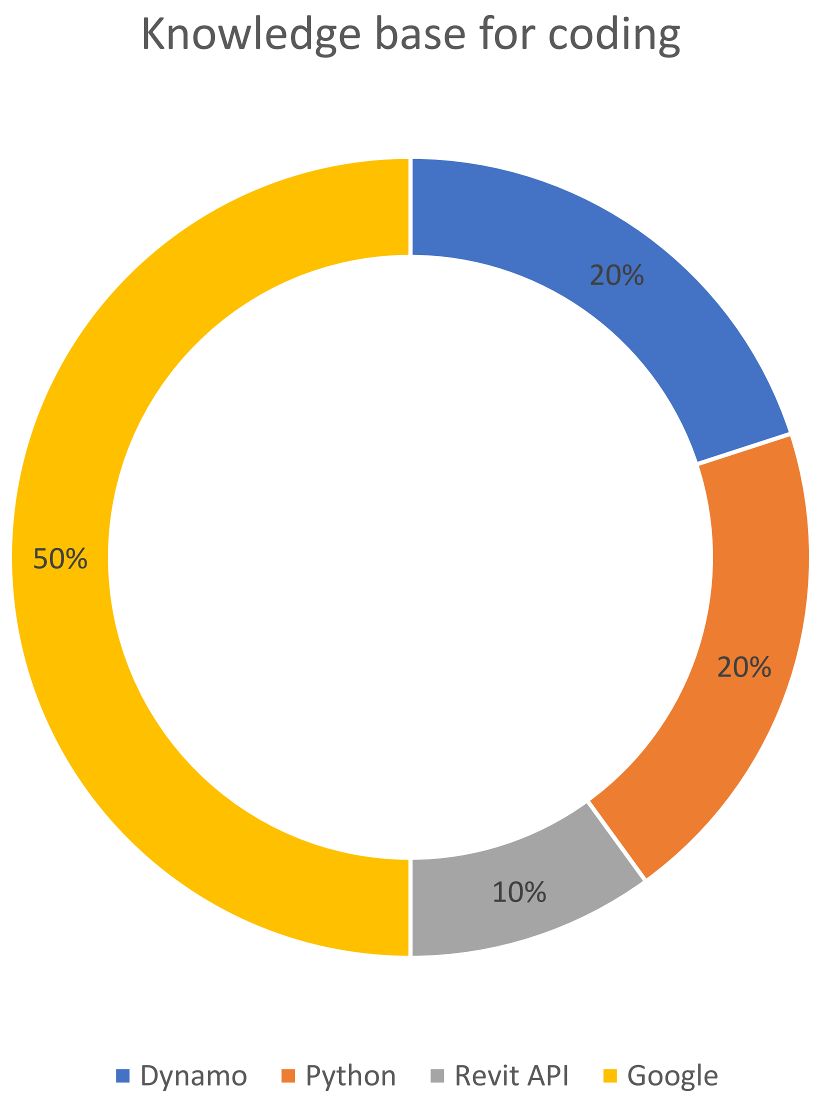
```
bip = BuiltInParameter.ALL_MODEL_INSTANCE_COMMENTS
provider = ParameterValueProvider(ElementId(bip))
evaluator = FilterStringEquals();
rule = FilterStringRule(provider, evaluator, "Review", False);
```



```
List
  0 Wall 279534
  1 Wall 279535
@L2 @L1          {2}
```

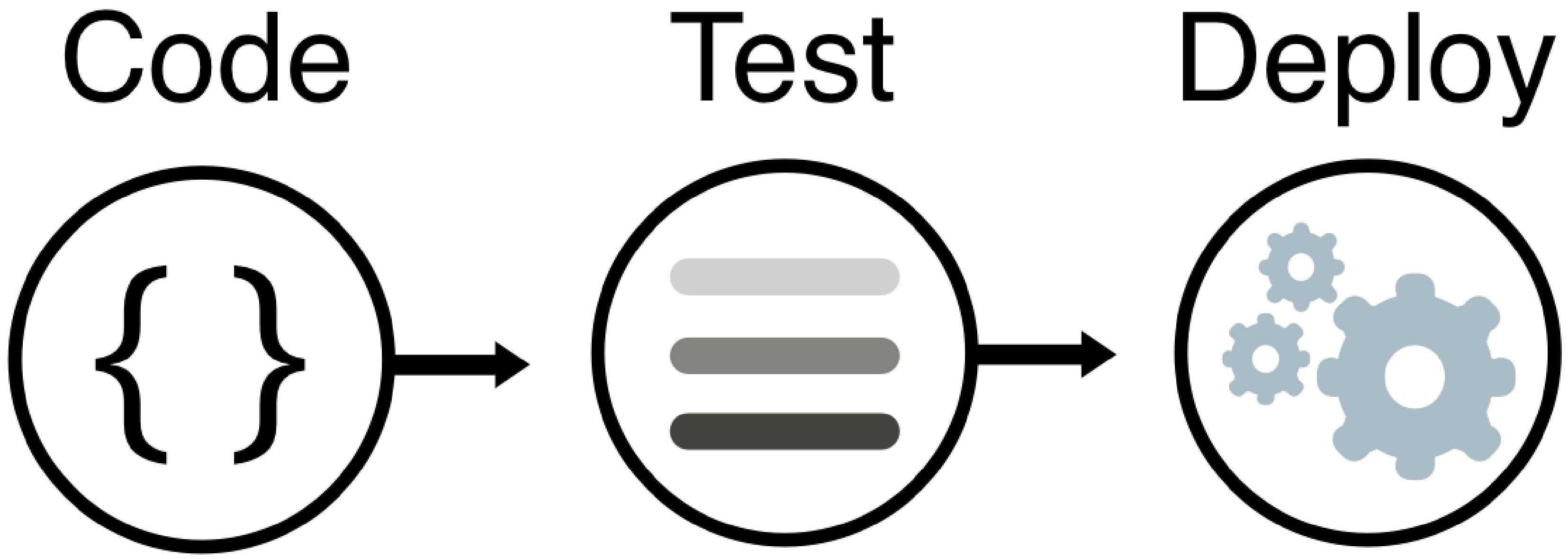


SKILLS YOU NEED



DISTRIBUTE

DISTRIBUTION



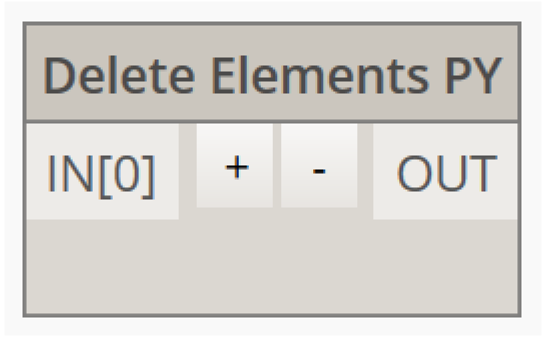
DISTRIBUTION

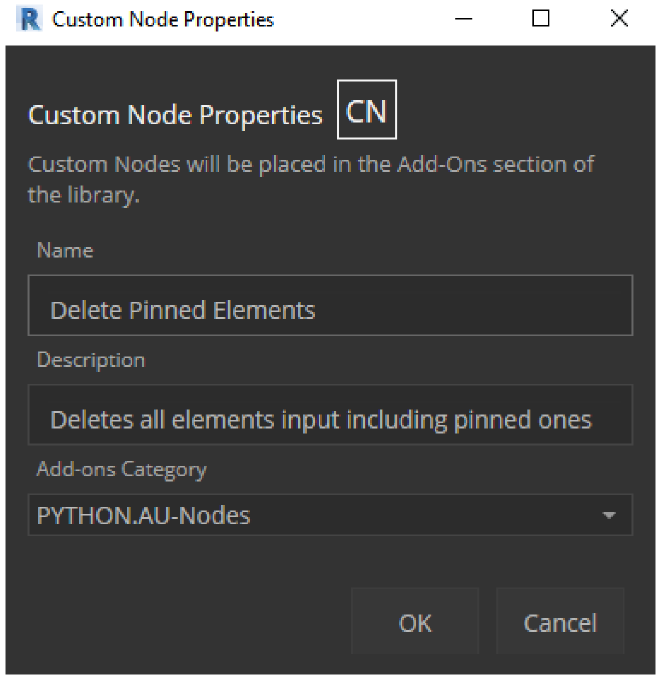
1. If you want to keep a copy of the script in your library or share it then make your own node

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




Custom Node Properties

Custom Node Properties CN

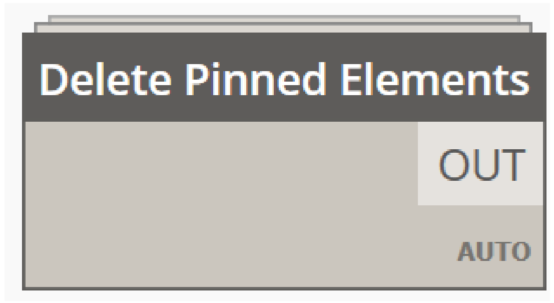
Custom Nodes will be placed in the Add-Ons section of the library.

Name: Delete Pinned Elements

Description: Deletes all elements input including pinned ones

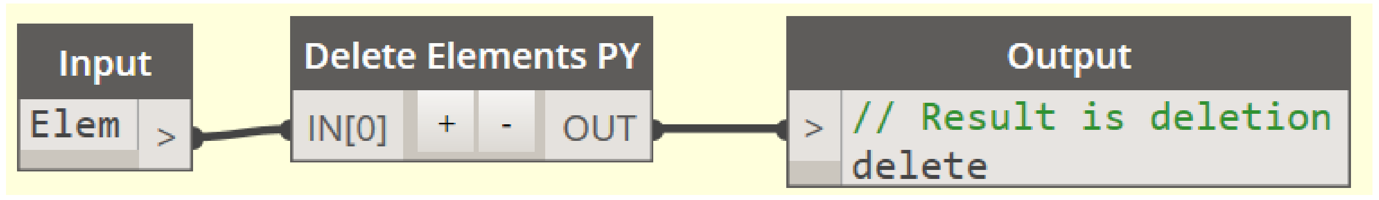
Add-ons Category: PYTHON.AU-Nodes

OK Cancel

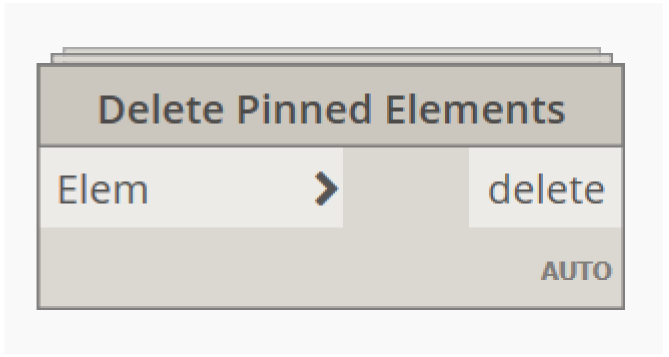


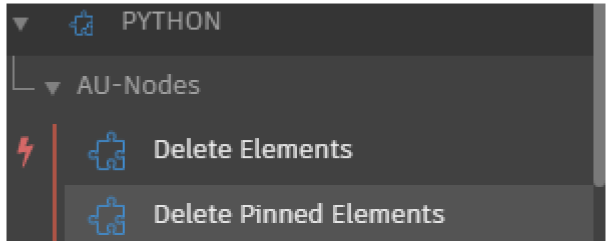
DISTRIBUTION

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 ouputs
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



```
// Result is deletion
delete
```

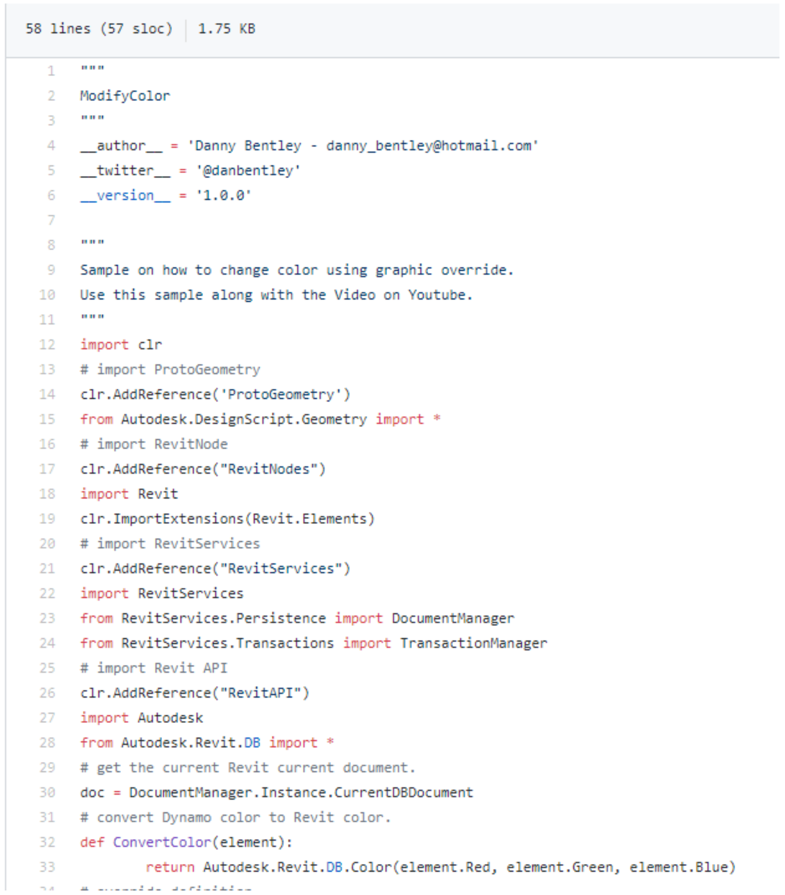




DISTRIBUTION

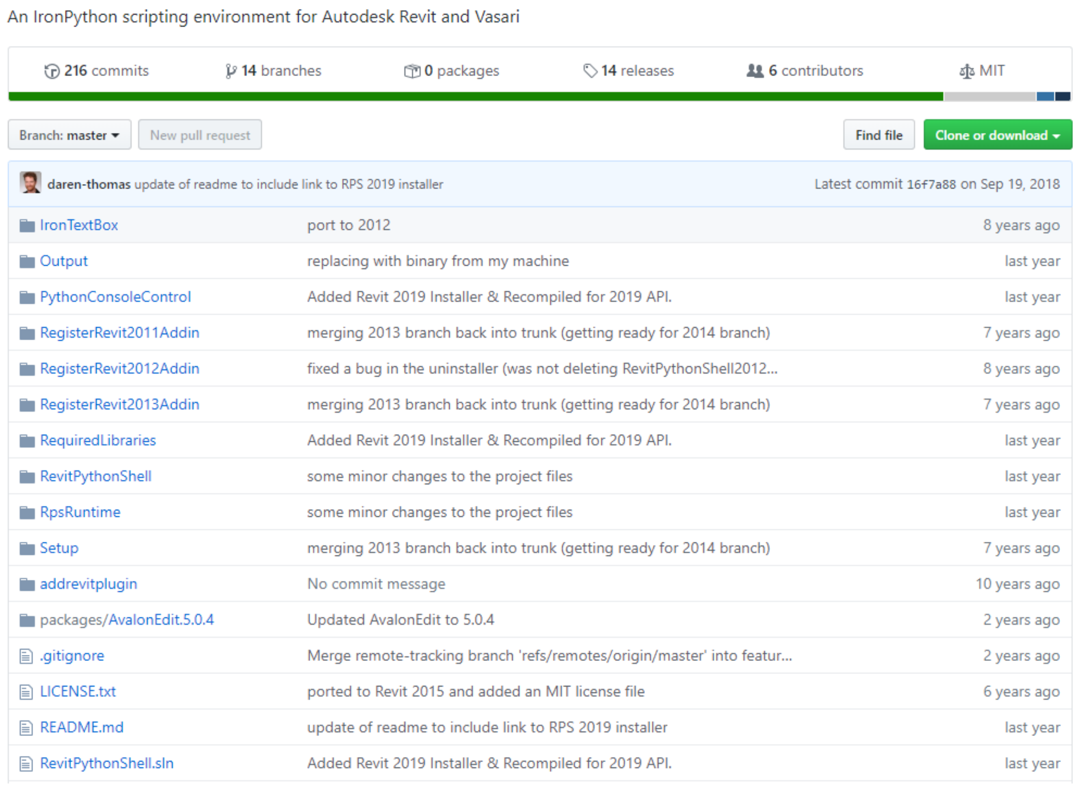
1. Or just host it on Github

2. All your Dynamo and Python code can be hosted here for distribution and collaboration



58 lines (57 sloc) | 1.75 KB

```python
"""
ModifyColor
"""
__author__ = 'Danny Bentley - danny_bentley@hotmail.com'
__twitter__ = '@danbentley'
__version__ = '1.0.0'

"""
Sample on how to change color using graphic override.
Use this sample along with the Video on Youtube.
"""
import clr
# import ProtoGeometry
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
# import RevitNode
clr.AddReference("RevitNodes")
import Revit
clr.ImportExtensions(Revit.Elements)
# import RevitServices
clr.AddReference("RevitServices")
import RevitServices
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager
# import Revit API
clr.AddReference("RevitAPI")
import Autodesk
from Autodesk.Revit.DB import *
# get the current Revit current document.
doc = DocumentManager.Instance.CurrentDBDocument
# convert Dynamo color to Revit color.
def ConvertColor(element):
        return Autodesk.Revit.DB.Color(element.Red, element.Green, element.Blue)
```



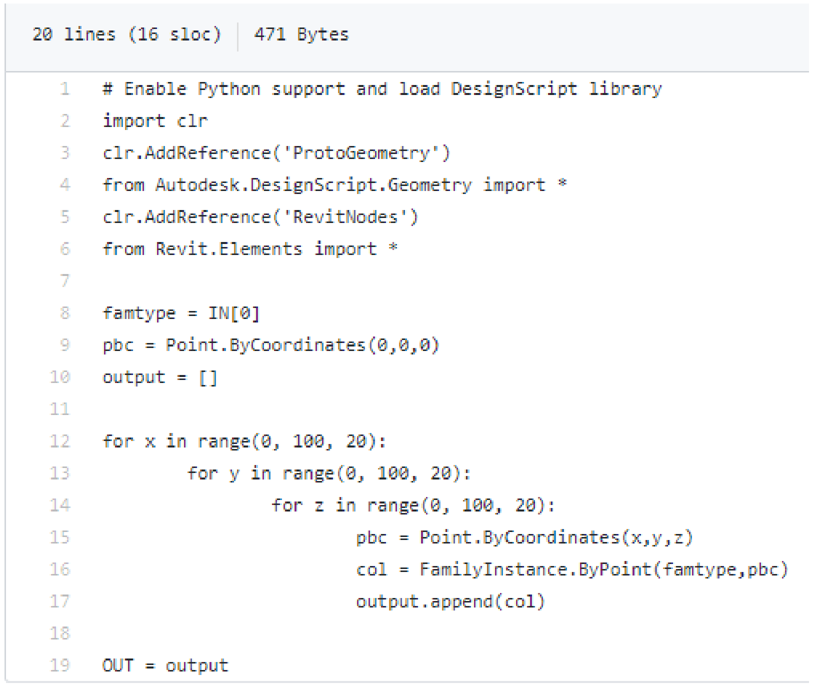
An IronPython scripting environment for Autodesk Revit and Vasari

daren-thomas update of readme to include link to RPS 2019 installer — Latest commit 16f7a88 on Sep 19, 2018

IronTextBox	port to 2012	8 years ago
Output	replacing with binary from my machine	last year
PythonConsoleControl	Added Revit 2019 Installer & Recompiled for 2019 API.	last year
RegisterRevit2011Addin	merging 2013 branch back into trunk (getting ready for 2014 branch)	7 years ago
RegisterRevit2012Addin	fixed a bug in the uninstaller (was not deleting RevitPythonShell2012...	8 years ago
RegisterRevit2013Addin	merging 2013 branch back into trunk (getting ready for 2014 branch)	7 years ago
RequiredLibraries	Added Revit 2019 Installer & Recompiled for 2019 API.	last year
RevitPythonShell	some minor changes to the project files	last year
RpsRuntime	some minor changes to the project files	last year
Setup	merging 2013 branch back into trunk (getting ready for 2014 branch)	7 years ago
addrevitplugin	No commit message	10 years ago
packages/AvalonEdit.5.0.4	Updated AvalonEdit to 5.0.4	2 years ago
.gitignore	Merge remote-tracking branch 'refs/remotes/origin/master' into featur...	2 years ago
LICENSE.txt	ported to Revit 2015 and added an MIT license file	6 years ago
README.md	update of readme to include link to RPS 2019 installer	last year
RevitPythonShell.sln	Added Revit 2019 Installer & Recompiled for 2019 API.	last year

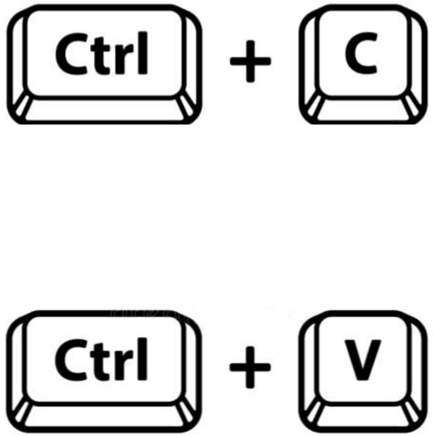
DISTRIBUTION

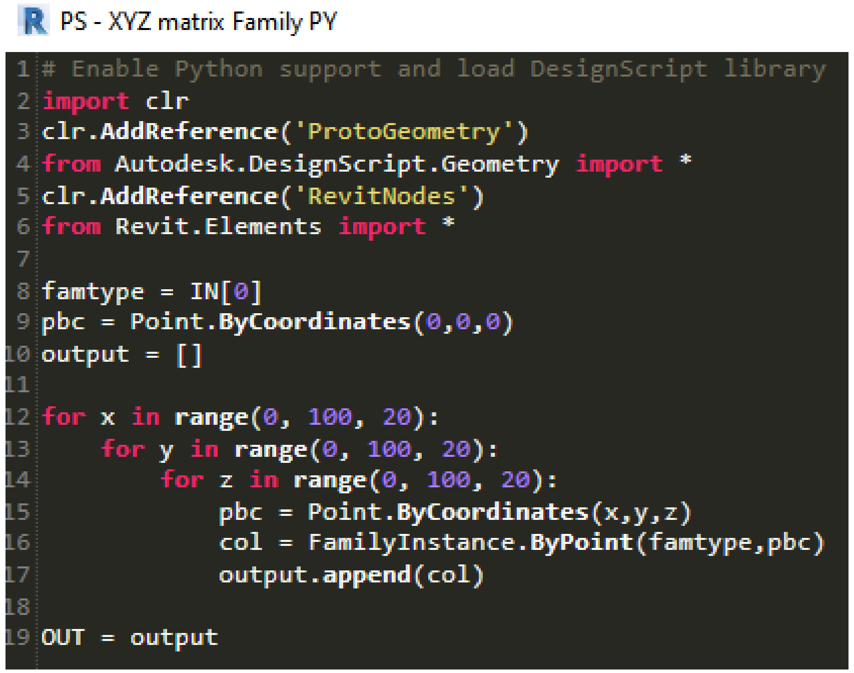
The most important skill for coding



20 lines (16 sloc) | 471 Bytes

```python
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
clr.AddReference('RevitNodes')
from Revit.Elements import *

famtype = IN[0]
pbc = Point.ByCoordinates(0,0,0)
output = []

for x in range(0, 100, 20):
        for y in range(0, 100, 20):
                for z in range(0, 100, 20):
                        pbc = Point.ByCoordinates(x,y,z)
                        col = FamilyInstance.ByPoint(famtype,pbc)
                        output.append(col)

OUT = output
```



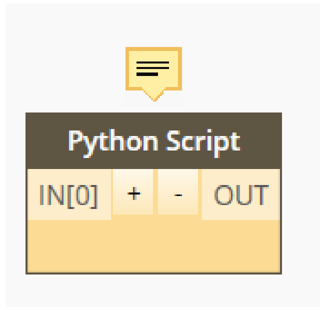


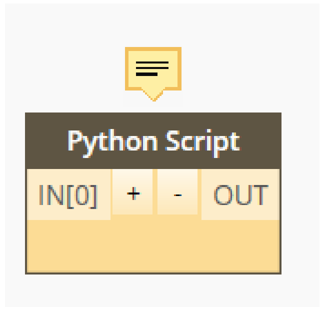
PS - XYZ matrix Family PY

```python
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
clr.AddReference('RevitNodes')
from Revit.Elements import *

famtype = IN[0]
pbc = Point.ByCoordinates(0,0,0)
output = []

for x in range(0, 100, 20):
    for y in range(0, 100, 20):
        for z in range(0, 100, 20):
            pbc = Point.ByCoordinates(x,y,z)
            col = FamilyInstance.ByPoint(famtype,pbc)
            output.append(col)

OUT = output
```

DEBUG

DEBUG





¯_(ツ)_/¯

DEBUG

1. Debugging will probably take up more time than you realize

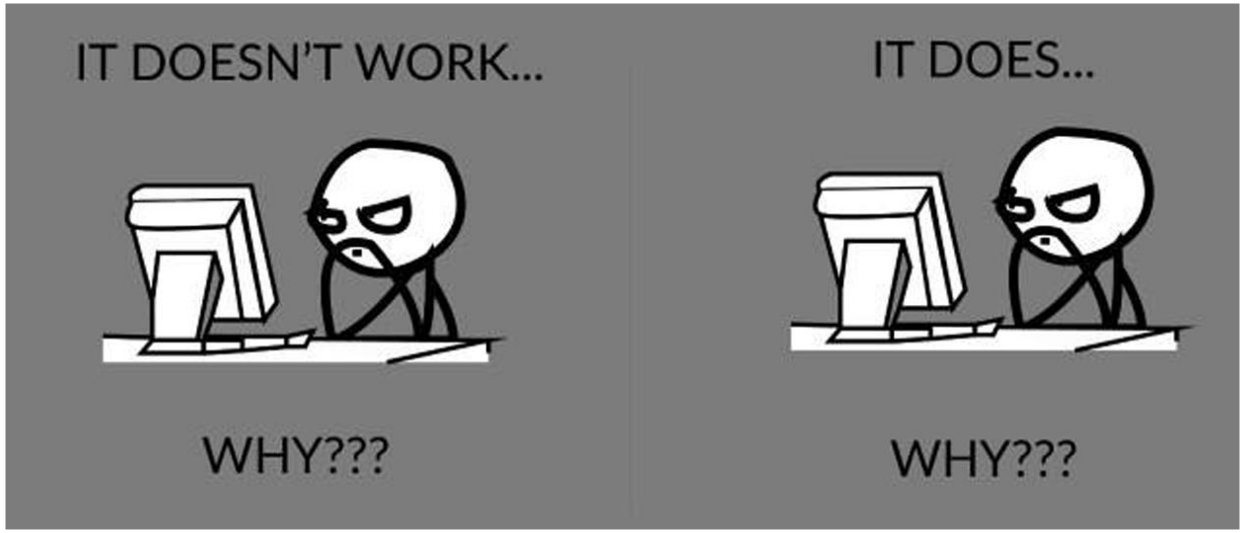
2. Python will tell you what the issue is and which line to find it on

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



FOLLOW UP

REVIEW LEARNING OBJECTIVES

1

Learn about the python node in dynamo and how to configure it

2

Learn how to debug your python script when errors appear

3

Learn how to execute code in python with statements and conditions

4

Create python scripts for new editing and designs capabilities

LIMITATIONS TO CONSIDER

Software Updates

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

With every release or update of Revit the API can change a little bit

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. There is no ‘print’ command just an ‘OUT’ command.

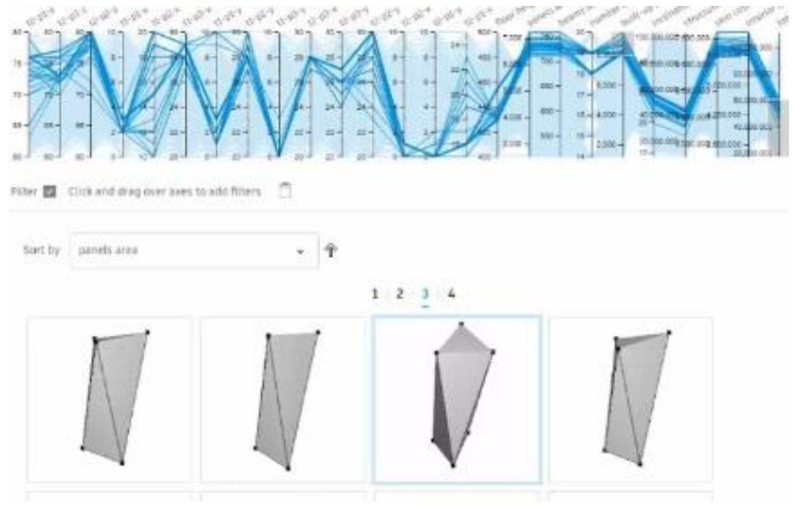
Script deployment in your organization

It is always a learning curve to get the scripts to be distributed regularly and functionally to your team

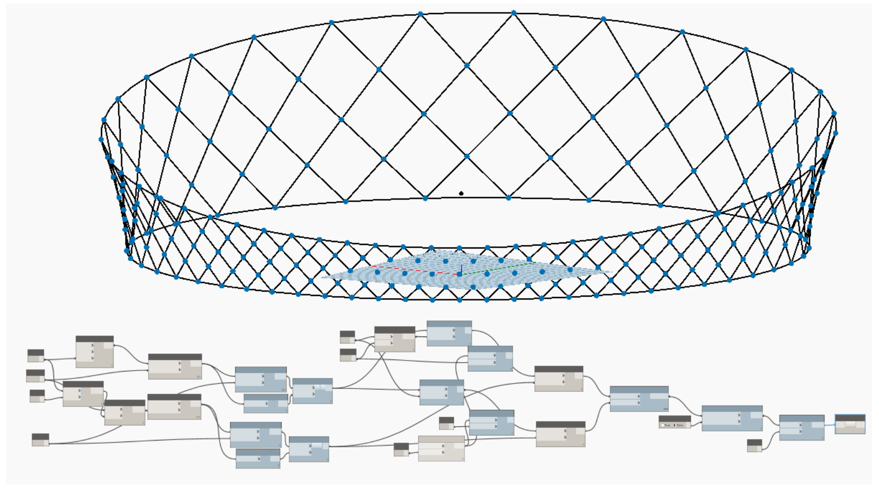
You’re coding solo

If scripts don’t work that usually means you are the only one fixing problems so be prepared for that workload.

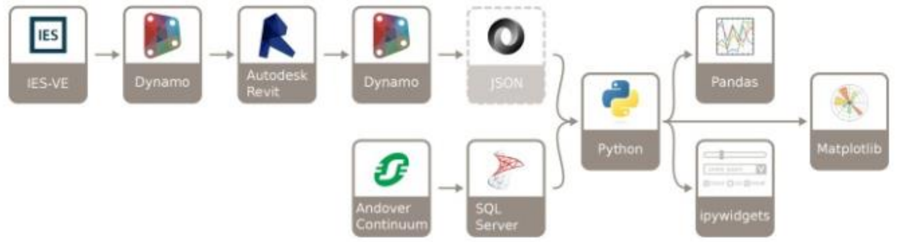
PYTHON POSSIBILITIES FOR YOUR PROJECTS



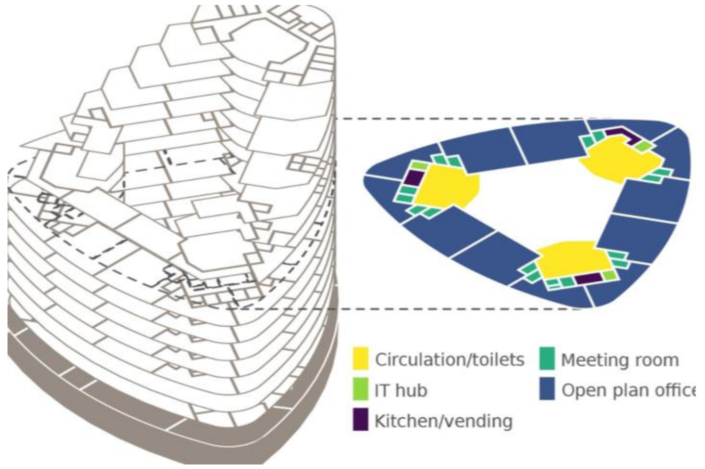
Generative Design



Structural Analysis



Data Integration



Energy Studies

PYTHON POSSIBILITIES BEYOND DYNAMO

django

pandas

$y_it = \beta' x_{it} + \mu_i + \epsilon_{it}$

PyTorch

pyRevit blog

ADDITIONAL LEARNING

Udemy

GitHub

learnpython.org

in LEARNING

AUTODESK® UNIVERSITY

Dynamo

YouTube

DEV
DEVELOP TOGETHER
DEV.TO

stackoverflow

WHO TO FOLLOW FOR PACKAGE DEVELOPMENT

PARALLAX TEAM

Rhythm for Dynamo
sixtysecondrevit.com

Revit API Docs
Online Documentation for the Revit API

BAD MONKEYS

pyRevit blog

SFDUG

HYPAR
Buildings. Generated.

EVOLVE LAB
evolvebim.com

Dynamo

CONTACT INFORMATION

Tadeh Hakopian

Contact Information

Twitter: @tadeh_hakopian

Linkedin: https://www.linkedin.com/in/thakopian/

Github: https://github.com/thakopian

THANK YOU !

AUTODESK®

Make anything™