First Dynamo attempts

Hans Hubers September - December 2014

Contents

[Introduction 3](#_Toc411611269)

[Newcastle Dynamo workshop (September 2014) 4](#_Toc411611270)

[The wave wall exercise 6](#_Toc411611271)

[Starting with Python 8](#_Toc411611272)

[Conditional statements 8](#_Toc411611273)

[What you can do with strings 8](#_Toc411611274)

[Lists, tuples, sets and dictionaries 9](#_Toc411611275)

[Loops 10](#_Toc411611276)

[Functions 11](#_Toc411611277)

[Object classes 12](#_Toc411611278)

[Exception handling 12](#_Toc411611279)

[Loop over two or more sequences at the same time (ZIP and MAP) 13](#_Toc411611280)

[Application of Python (the Weave component) 15](#_Toc411611281)

[What is this Lambda function? 21](#_Toc411611282)

[What is this Recursion 21](#_Toc411611283)

[The Weave component result 23](#_Toc411611284)

[The Python script in Dynamo is slightly different: 26](#_Toc411611285)

[Continuing with the wave wall exercise 29](#_Toc411611286)

# Introduction

This document serves as a report of the research activities of Hans Hubers in 2014. In 2013 he initiated a research called pCOLAD together with Michela Turrin, Irem Erbas and Ioannis Chatzikonstantinou. It was aimed at developing a method, prototype and case study for parametric COLaborative Architectural Design. pCOLAD was well received and presented at the eCAADe2014 conference in NewCastle. It used Grasshopper and VB.net as software. In 2013 this software was the best choice for a parametric solution. However in the course of 2014 a generative parametric solution became useful for Revit, called Dynamo. Since Revit is the most used BIM software, it seems adequate to investigate if the results of the pCOLAD project can be converted to Revit/Dynamo and if a way can be found to use the VB.net prototype in that environment or if adaption is needed or even a new prototype. Before this research can be executed more knowledge has to be gained about Revit and Dynamo. Hans followed a course in Revit Family development in 2014 at Revitopleidingen in Hilversum, The Netherlands. A [separate report](../../Education/Revit%20Family%20Cursus/Wat%20we%20geleerd%20hebben%20bij%20Revitopleidingen%20in%20Hilversum.docx) about this 2 day course has been made. Here we start with the report on the Dynamo workshop and work our way through the examples, focussing on ways to transfer the typical functionality we used in pCOLAD in Grasshopper towards the Dynamo and Python environment.

# Newcastle Dynamo workshop (September 2014)

Ctrl+W gives notes, you can also rename the nodes

Esc hold -> navigate background

Esc empties search box. Then just start typing search string and hit Enter if found or click on other. The node is put in the centre of the canvas.

F5 = run

Dynamo controls Revit, but not the other way around

Use Select to get Revit elements in Dynamo

To get e.g. a point you also need the Geometry Node

Use the Watch node to check elements

Double click canvas -> Code block hit enter after ; for next outputs

You can use formula’s in a Code block (capital sensitive: Math.Sin(x);)

Right click on node -> labels or lacing to choose longest, shortest or cross product

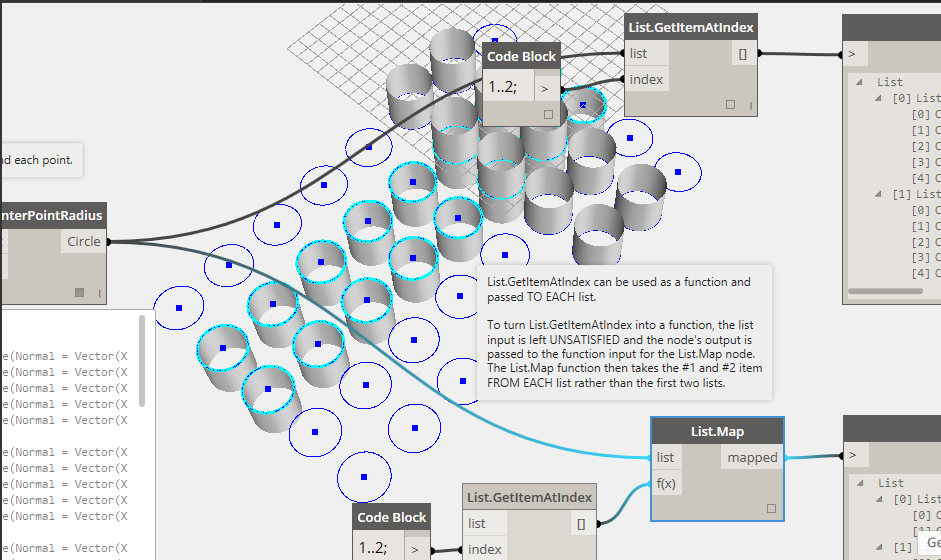
TakeEveryNthItem

RestOfItems chops of the first of the list

Use Map to choose every nth of a list

Use GetItem at index to get every nth index

If you leave input unconnected (this is called turning a node into a function) of a list of lists producing node, you can access the items in the list in the next node. In *Figure 1* the list of lists of circles is accessed at the top by getting the list 1 and 2, while at the bottom with the List.Map the circles of list 1 and 2 are used.



*Figure 1 Way to create a function by leaving the input of a node open*

Use Extrude to get Curve.ExtrudeAsSolid (onder Geometry)

Code block if statement is a ? b : c

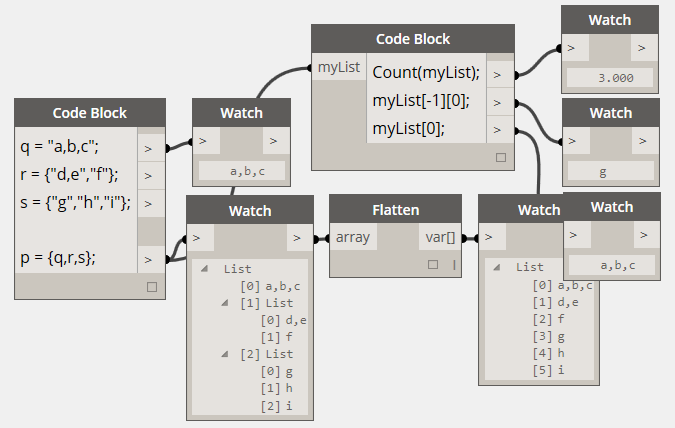
Use Import to ImportInstance.ByGeometries (get geometry in Revit)

Use SelectModel Element

myList[0]; in een codeblock is zelfde als List.GetItemAtIndex

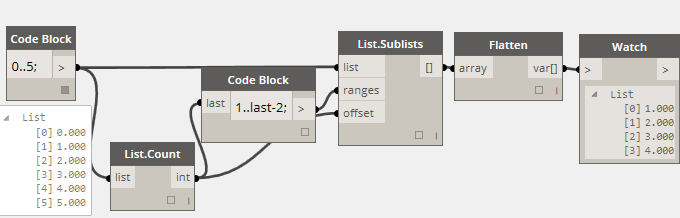
Type PointAtParameter to find Curve.PointAtParameter and use node block 0..1..variable;

A list has the format {“item1”, “item2”, “item3”}. Mind the “” to make a list of strings – otherwise they would be interpreted as variables. {1,2,3} would result in a list of three separated integers. A list of lists can be made by putting lists into variables: itemA = {“item1, item2, item3”} and then put the variables into a list {intemA, itemB etc.} (*Figure 2*). Also notice the Count method that takes a list as argument.



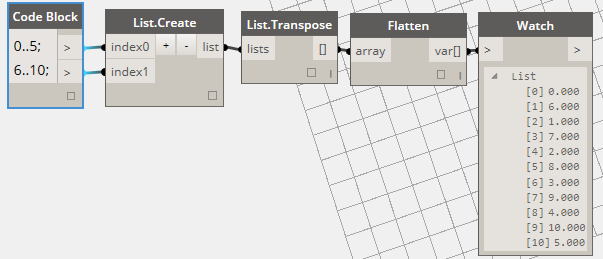
*Figure 2 a ist of lists definied in Code Block*

In order to create a sublist use trick of *Figure 3* or use Slice

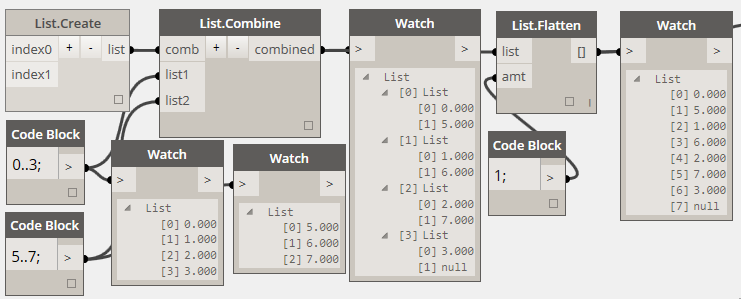


*Figure 3 Creating a sublist*

To weave two lists use trick of *Figure 4*. Or *Figure 5*.



*Figure 4 Weave two lists*



*Figure 5 How List.Combines work (notice List.Create without inputs)*

Attention! In order to write to Excel you need String and Number nodes (a number in a Codeblock doesn’t work). And in order to read a csv file, you have to use floating numbers (so for x,y,z: #.#,#.#,#.#). And for some reason you have to divide by 1000.

About Code Block nodes: <http://dynamobim.org/cbns-for-dummies/>

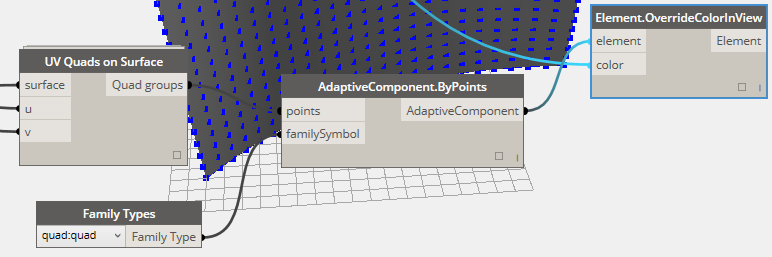
Difference between actions, query and create. As can be seen in left menu in Dynamo.

Create takes (); actions like add takes only what you want to add between ();

Query nodes don’t take () e.g. myPoint.X

You can make custom nodes by right clicking the canvas and choose New Node From Selection, after selection some nodes of course. There is no way back, though you can select the content when you double click to edit the custom node and then copy everything you need in your home solution. You recognize a custom node by the two frames behind it. You can edit a custom node by double clicking. An extra tab will open with an X next to it. Click the X to close after editing. Use Input and Output nodes to make connections to the Custom Node. Delete them if they are not in the right order and create them again in the right order – or just interchange the variables. Find an example in Panels\_Nodes.dyn, which works with the quad.rfa adaptive family in Revit.

You import a Revit family with the node Revit/Selection/Family Types. It lists all families in the active Revit project (so you first have to put a family in ). You can then select the right family.



*Figure 6 Custom Node and Importing a family from Revit*

# The wave wall exercise

One of the exercises is about weaving geometry, literally as ribbons(Figure 7).

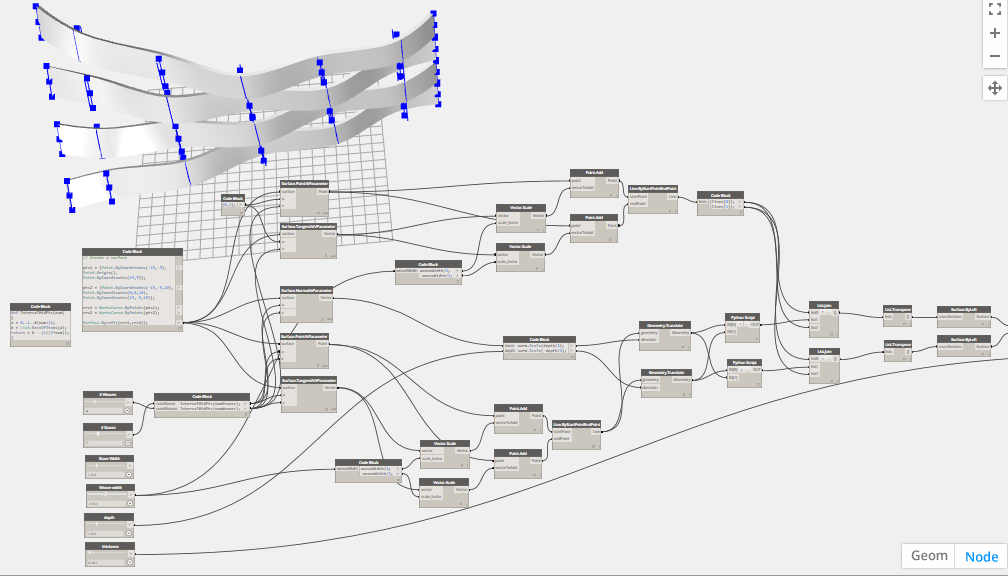


Figure 7 woven surface example

The example deletes parts of unused surfaces. An alternative would be to weave only the right curves that are used to make the ribbons. But a weave node like in Grasshopper doesn’t exist. Wanting to develop one myself I tried to make one by using for or while loops in Design Script. Unfortunately for and while loops don’t work in this Dynamo version 0.7.2.40823. So Python is the only solution. Have to learn Python then, because the weave solutions don’t do what I need.

**Other Resources**

Official sources:

[DynamoBIM.org/learn/](http://DynamoBIM.org/learn/) for video tutorials and a language reference guide

[Dynamo User Forum](http://dynamobim.org/forums/forum/dyn/) for answers to questions

[Dynamo Github site](https://github.com/DynamoDS/Dynamo/issues) to report bugs and track pull requests

Social media:

[DynamoBIM](https://www.facebook.com/DynamoBIM) on Facebook

[@DynamoBIM](https://twitter.com/DynamoBIM) on Twitter

Blogs:

[Archi-Lab](http://archi-lab.net/) by Konrad Sobon

[AEC, you and me](http://aecuandme.wordpress.com/) by Julien Benoit

[The Proving Ground](http://www.theprovingground.org/) by Nate Miller

[The Dark Arts of Revit](http://vasshaug.net/) by Håvard Vasshaug

[Andrzej’s YouTube channel](https://www.youtube.com/channel/UC468uWfsaD8lEcQVlHU7qHA)

[DynamoBIM blog](http://dynamobim.org/blog/)

eCAADe:

[eCAADe Workshop Descriptions](https://www.northumbria.ac.uk/about-us/news-events/events/2014/09/ecaade-2014-conference/workshops/confirmed-workshops/)

<http://simplydynamo.blogspot.co.uk/>

# Starting with Python

Nice introduction to Python can be found [here](http://www.learnpython.org/en/Welcome).

Mind capitals: If is not the same as if, While not the same as while.

A handy quick guide can be found [here](http://www.tutorialspoint.com/python/index.htm).

Description of the standard Python library can be found [here](https://docs.python.org/2/library/).

In order to have debugging information I installed Python Tools for Visual Studio (PTVS) which can be found [here](https://visualstudiogallery.msdn.microsoft.com/9ea113de-a009-46cd-99f5-65ef0595f937). It uses Python 3.4.

Comments start with #, or multiline comments with “”” but should be closed with “”” on a new line if the comment is longer than one line.

Use 4 spaces for every indent (tabs lead to confusion).

Use CapitalCase format for classes and lower\_case\_with\_underscores for function names and methods.

# Conditional statements

(result would be: else) mind the “:” after the statements

a = 20

if a >= 22:

print("if")

elif a >= 21:

print("elif")

else:

print("else")

Python uses C-style string formatting to create new, formatted strings. The "%" operator is used to format a set of variables enclosed in a "tuple" (a fixed size list), together with a format string, which contains normal text together with "argument specifiers", special symbols like "%s" and "%d".

%s - String (or any object with a string representation, like numbers)

%d - Integers

%f - Floating point numbers

%.<number of digits>f - Floating point numbers with a fixed amount of digits to the right of the dot.

%x/%X - Integers in hex representation (lowercase/uppercase)

# This prints out "John is 23 years old."

name = "John"

age = 23

print "%s is %d years old." % (name, age)

# This prints out: A list: [1, 2, 3]

mylist = [1,2,3]

print "A list: %s" % mylist

NOTICE: in Visual Studio you have to put what you want to print between brackets ().

# What you can do with strings

s = "Strings are awesome!"

# Length should be 20

print "Length of s = %d" % len(s)

# First occurrence of "a" should be at index 8

print "The first occurrence of the letter a = %d" % s.index("a")

# Number of a's should be 2

print "a occurs %d times" % s.count("a")

# Slicing the string into bits

print "The first five characters are '%s'" % s[:5] # Start to 5

print "The next five characters are '%s'" % s[5:10] # 5 to 10

print "The twelfth character is '%s'" % s[12] # Just number 12

print "The last five characters are '%s'" % s[-5:] # 5th-from-last to end

# Convert everything to uppercase

print "String in uppercase: %s" % s.upper()

# Convert everything to lowercase

print "String in lowercase: %s" % s.lower()

# Check how a string starts

if s.startswith("Str"):

print "String starts with 'Str'. Good!"

# Check how a string ends

if s.endswith("ome!"):

print "String ends with 'ome!'. Good!"

# Split the string into three separate strings,

# each containing only a word

print "Split the words of the string: %s" % s.split(" ")

# Lists, tuples, sets and dictionaries

myList = [“A”,”B”,”C”]

Make a simple list of integers with the list and range function.

myList = list(range(0,3)) >>> [0,1,2] # notice that 3 is not included in the range

myList = [0]\*5 #or myList[] if you don’t know how many members the list should have, and use myList.append(value)  
number = 16  
second\_number = 10  
first\_array = [1,2,3]  
second\_array = [1,2]  
  
**if** number > 15:  
    myList[0] = "1"  
  
**if** first\_array:  
    myList[1] = "2"  
  
**if** **len**(second\_array) == 2:  
    myList[2] = "3"  
  
**if** **len**(first\_array) + **len**(second\_array) == 5:  
    myList[3] = "4"  
  
**if** first\_array **and** first\_array[0] == 1:  
    myList[4] = "5"  
  
**if** **not** second\_number:  
    myList[5] = "6"  
  
#Assign your output to the OUT variable  
OUT = myList

To check if an item in a list is a list or tuple or set or dictionary:

if isinstance(item,(list,tuple,set,dict)):

Instead of a list you can use a tuple, which is a list with immutable items. You create and access a tuple the same as with a list, only use () to create the list instead of [].

Sets are lists with no doubled items. You can make a set out of list by just putting the list as argument:

myList = [0,1,2,1,0]

mySet = set(myList) >>> [0,1,2]

Dictionarys are lists with keys. You make them with curly brackets : { and }. You get items with square brackets [] and the name of the key.

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

print "dict['Name']: ", dict['Name'];

print "dict['Age']: ", dict['Age'];

When the above code is executed, it produces the following result:

dict['Name']: Zara

dict['Age']: 7

# Loops

# Prints out 3,5,7

for x in xrange(3, 8, 2): # or range(3, 8, 2) # meaning for every integer between 3 and 8 with step = 2 (8 not included)

print x

Break and Continue

# Prints out 0,1,2,3,4

count = 0

while True:

print count

count += 1

if count >= 5:

break

# Prints out only odd numbers - 1,3,5,7,9

for x in xrange(10):

# Check if x is even

if x % 2 == 0: # % is de mod operator which returns the remainder of a division

continue

print x

get the even numbers in the sequence before 319

myList = []  
numbers = [  
    951, 402, 984, 651, 1360, 69, 408, 319, 601, 485, 980, 507, 725, 547, 544,  
    615, 83, 165, 141, 501, 263, 617, 865, 575, 219, 390, 984, 592, 236, 105, 942, 941,  
    386, 462, 47, 418, 907, 344, 236, 375, 823, 566, 597, 978, 328, 615, 953, 345,  
    399, 162, 758, 219, 918, 237, 412, 566, 826, 248, 866, 950, 626, 949, 687, 217,  
    815, 67, 104, 58, 512, 24, 892, 894, 767, 553, 81, 379, 843, 831, 445, 742, 717,  
    958, 609, 842, 451, 688, 753, 854, 685, 93, 857, 440, 380, 126, 721, 328, 753, 470,  
    743, 527  
]  
count = 0  
mynum = numbers [count]  
**while** **not** mynum == 319:  
    mynum = numbers [count]  
    **if** mynum  % 2 == 0:  
        myList.**append**(mynum)  
    count += 1  
  
#Assign your output to the OUT variable  
OUT = myList

more simple would have been:

for number in numbers:

if number == 319:

break

if number % 2 == 1:

continue

# Functions

The keyword [def](https://docs.python.org/3/reference/compound_stmts.html" \l "def) introduces a function definition. It must be followed by the function name and the parenthesized list of formal parameters and :. The statements that form the body of the function start at the next line, and must be indented. Use docstrings to comment on the function (“””here the summary of your definition, then a blank line and then next lines and last quotes on a separate line

”””

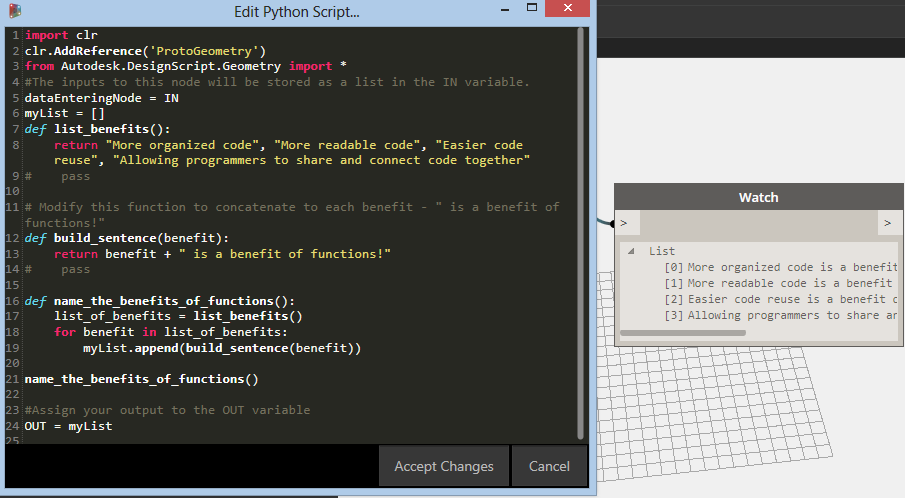


Figure 8 functions in Python

# Object classes

myList = []  
class MyClass:  
    variable = "blah"  
  
    def function(self):  
        myList.append("This is a message inside the class.")  
myobjectx = MyClass()  
myList.append(myobjectx.variable)  
myobjectx.function()  
  
#Assign your output to the OUT variable  
OUT = myList

To find out what is in an imported module (regular expression in this case) or packages use dir(name\_of\_module)

import re  
myList = []  
for thing in dir(re):  
 if "find" in thing:  
  myList.append(thing)          
sorted(myList)  
  
#Assign your output to the OUT variable  
OUT = myList

# Exception handling

def do\_stuff\_with\_number(n):

print n

the\_list = (1, 2, 3, 4, 5)

for i in range(20):

try:

do\_stuff\_with\_number(the\_list[i])

except IndexError: # Raised when accessing a non-existing index of a list

do\_stuff\_with\_number(0)

# Loop over two or more sequences at the same time (ZIP and MAP)

[To loop over two or more sequences](https://docs.python.org/3/tutorial/datastructures.html#tut-loopidioms) at the same time, the entries can be paired with the [zip()](https://docs.python.org/3/library/functions.html#zip) function.

>>>

**>>>** questions = ['name', 'quest', 'favorite color']

**>>>** answers = ['lancelot', 'the holy grail', 'blue']

**>>> for** q, a **in** zip(questions, answers):

**...**  print('What is your {0}? It is {1}.'.format(q, a))

**...**

What is your name? It is lancelot.

What is your quest? It is the holy grail.

What is your favorite color? It is blue.

Probably the [map() function](http://code.activestate.com/recipes/65285-looping-through-multiple-lists/) or a list comprehension is even more interesting.

So maybe next thing would work.

for IN[0]in map(None, IN[1], IN[2]):

collectList.append(IN[0])

well it doesn’t give an error, but the output shows ABC12, DEF34, GHI56

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | a **=** **[**'a1'**,** 'a2'**,** 'a3'**]**  b **=** **[**'b1'**,** 'b2'**]**  *# will iterate 3 times,*  *# the last iteration, b will be None*  **print** "Map:"  **for** x**,** y **in** map**(**None**,** a**,** b**):**  **print** x**,** y  *# will iterate 2 times,*  *# the third value of a will not be used*  **print** "Zip:"  **for** x**,** y **in** zip**(**a**,** b**):**  **print** x**,** y  *# will iterate 6 times,*  *# it will iterate over each b, for each a*  *# producing a slightly different outpu*  **print** "List:"  **for** x**,** y **in** **[(**x**,**y**)** **for** x **in** a **for** y **in** b**]:**  **print** x**,** y |

Using map with "None" as the first argument is discussed in the docs:

"If function is None, the identity function is assumed; if there are multiple list arguments, map() returns a list consisting of tuples containing the corresponding items from all lists (i.e. a kind of transpose operation). The list arguments may be any kind of sequence; the result is always a list."

Note: this returns None for sequences where there is no element. Output:<pre>a1 b1 a2 b2 a3 None</pre>

Zip allows you to iterate over the lists in a similar way, but only up to the number of elements of the smallest list.

Output:<pre>a1 b1 a2 b2</pre>

Python 2.0 introduced list comprehension which explains the rather strange syntax: <pre>[(x,y) for x in a for y in b]</pre> this iterates over the b list for every element in a. These are put into a tuple x, y. We then iterate through that tuple in the outermost for loop.

The result is quite different:<pre>a1 b1 a1 b2 a2 b1 a2 b2 a3 b1 a3 b2</pre>

# Application of Python (the Weave component)

In Dynamo it turned out that a Weave component (like in Grasshopper) was missing. We used it frequently in the pCOLAD case study. Let’s try to program this in Python. The Weave component should combine lists of lists in a patterned way.

E.g. we have two lists of lists: a list of 3 lists: {A,B,C} and {D,E,F} and {G,H,I}. And another list of 3 lists: {1,2} and {3,4} and {5,6}. They will be addressed in Python as IN[0], IN[1], IN[2]…IN[x]. The items from the lists are addressed as IN[x][y]. Now if e.g. we use the pattern {1,2} we should find a loop that adds the content of the 3 lists in such a way that the result will be a list of 3 lists: {{A,1,B,2,C}, {D, 3, E,4, F}, {G, 5, H, 6, I}}.

In Grasshopper one would use Entwine and Weave to realise this (Figure 9). Interesting to see what happens if one of the lists is longer. The Weave component has an option to insert Nulls or skip these combinations, but in either case it repeats the last part of the smallest list.

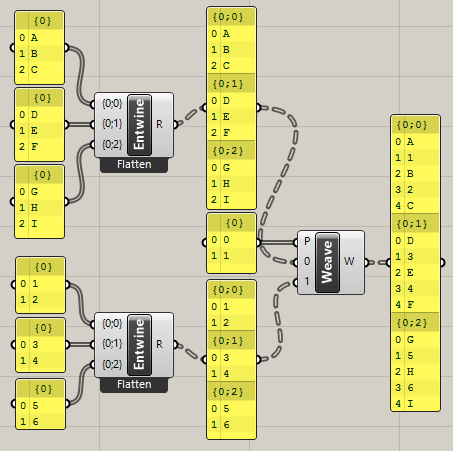


Figure 9 Solution in Grasshopper

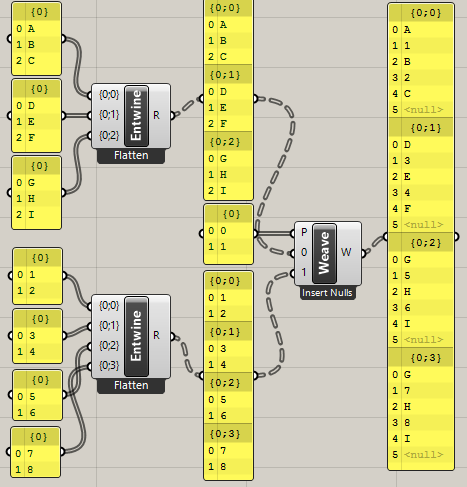


Figure 10 Weave with Insert Nulls option

So in Python we should program the equivalent of the Weave component. Let’s put the pattern as list in IN[0]. {1,2} means that we weave list IN[1] and IN[2] until one of them runs out of items. {1,1,2} would mean that we weave two items from list IN[1] with 1 from list IN[2].

Try again. What you are doing is: you take the longest input list and use it’s count of items as a counter(n) in a repeat loop. Then you check if the nth item of the first input list is a list or an item. If it doesn’t exist or if it is a list, you go to the next input list and do the same. If it is an item you add it to myList and go to the nth item of the next input list. If that is a list you go to the next input list etc. If it is an item you add it to myList. When you finished with the last input list you add myList to collectList and empty myList. Then you go one level deeper(how do I do that?). Again you check what is the highest count of items in the input lists at that level and use that as a counter(n) etc.

Well in fact it is rather simple.

1. You put all the input lists in a list. Dynamo already does that in the IN list.
2. You loop through IN.
   1. You append the item to myList0 (first time myList0 becomes the same as IN).
   2. And if the item is not a list, tuple or set you append it to myList1.
   3. Else you add Null to myList1
3. Add myList1 to collectList.
4. Loop through myList0
   1. And empty myList0
5. Then you loop through that myList0 and put the content in myList2
6. Then set myList1 = myList2
7. And add myList1 to collectList.
8. Then you loop through myList1 and put the content in myList2
9. etc. etc. until there is nothing in mylist1.

You’ll see that 7 and 8 are the same as 2.c and 5 so you can make a loop at 2 and inside that loop you make loop from …

You get an error if you try to add an index outside the range of the list. So use try: and except IndexError: and go to the next loop (automatic). We should use two lists: myList to collect the items and collectList to collect the myList (empty it after that). So we get next pseudo code for inner loop:

myList.append(IN[1][0]) 🡪A

myList.append(IN[2][0]) 🡪1

myList.append(IN[1][1]) 🡪 B

myList.append(IN[2][1]) 🡪 2

myList.append(IN[1][2]) 🡪 C

break because IN[2][2] does not exist and because IN[1] ran out of items the loop stops.

And in the case of {1,1,2} we would get

myList.append(IN[1][0]) 🡪 A

myList.append(IN[1][1]) 🡪 B

myList.append(IN[2][0]) 🡪 1

myList.append(IN[1][2]) 🡪 C

myList.append(IN[1][3]) 🡪 D

myList.append(IN[2][1]) 🡪 2

etc.

And in the case of {1,2,1} we would get

myList.append(IN[1][0]) 🡪 A

myList.append(IN[2][0]) 🡪 1

myList.append(IN[1][1]) 🡪 B

myList.append(IN[1][2]) 🡪 C

myList.append(IN[2][1]) 🡪 2

myList.append(IN[1][3]) 🡪 D

etc.

How do we code this with loops? We see that the first index of IN is looping through the input lists in the order of the pattern until we ran out of all items. So for i in IN[0] should do as inner loop that is started over and over again. The second index of IN runs up for every input list when it is called until we run out of items. So it consists of the items in the input list. That would be an outer loop: for j in IN[i]. That is not possible because i is set in the inner loop. So we can not solve this with nested loops.

for j in IN[0]: # j is 1,2,1

for i in IN[0]: # i is 1,2,1,1,2,1 etc. until we ran out of all items

We get:

myList = []

collectList = []

counter = -1

for i in IN[0]: # i are the pattern items

counter+=1

try:

for j in IN[i]: # j are the input lists

try:

myList.append(j[counter])

except IndexError:

break

collectList.append(myList)

myList = []

except IndexError:

counter = -1

break

My flattenList function in Python :

myList1 = [1,2,3]

myList2 = (4,5,6)

myNestedLists = [myList1, "A", "B", myList2]

def flatIt(anyList):

flatList = []

for item in anyList:

if isinstance(item,(list,tuple,set)):

for subitem in item:

flatList.append(subitem)

else: flatList.append(item)

return flatList

myResult = flatIt(myNestedLists)

for thing in myResult:

print (thing)

It must be possible to rewrite this in list comprehension style. List comprehension is a concise way to create lists on the basis of for and if statements in the same line. It is constructed as follows:

<listname> = [x for x in <iterable> if x <condition>] where the loops and conditions can be nested.

myResult = [item

for item in myNestedLists

if isinstance(item,(list,tuple,set))

else item

for thing in item]

myList1 = [1,2,3]

myList2 = (4,5,6)

myNestedLists1 = [myList1, "A", "B", myList2]

myNestedLists2 = [myList2, "A", "B", myList1]

myDoubleNestedLists = [myNestedLists1, myNestedLists2]

resultList =[item for item in myNestedLists2 if isinstance(item, str)]

print (resultList)

Try again. More general. Imagine we have several lists of different depths.

List1

[0] List1.1

[0] A1

[1] B1

[2] C1

[1]List1.2

[0] D1

[1] E1

[2] F1List2

[0] A2

[1] B2List3

[0] List3.1

[0] A3

[1] List3.1.1

[0] B3

[1] C3

How would we put this together with e.g. a pattern of 1,2,3:

I guess we would start with the first item of List1, which is a list (so we put Null) and then the first item of List2 which is A2 and then the first item of List3, which is a list (so we put Null). Then because we have no more lists, we continue with the second item of List1, which is a list (so we put Null) and then the second item of List2 (B2), and then the second item of List3, which doesn’t exist (so we put Null). Then we continue with another list, because there are no more items on this level. We go 1 level deeper. We start with first item on level2 of List1: A1, then first item on level2 of List2, which doesn’t exist (so we put Null) then first item on level2 of List3 (A3). We continue with second item on level2 of List1(B1), List2 has no second level so we put Null again. Then we add second item of on second level of List3, which is a list (so we put Null). Since there is a third item in List1 on level2 we continue and put C1 <<<so we have to check for every level which is the longest list and repeat for that number>>> the other lists have no third items on level2 so twice a Null. Then we continue with another list, because there are no more items in the longest list on this level. However on this level in the second list of List1 we have D1, E1 and F1. But it doesn’t feel right to just continue the list, because it would be no difference with a flat list A1..F1. <<< so maybe the result should be nested lists? But how? With the earlier example [[A,B,C], [D,E,F], [G,H,I]] to be combined with [[1,2], [3,4], [5,6]] we expected a list of three lists as outcome: [[A,1,B,2,C], [D,3,E,4,F], [G,5,H,6,I]]. So for every list in a list we start a new list. No nested lists as outcome.>>> So third list is first item in second list on level2: D1, and since there are no other second lists on level2, we put Null, Null behind an do the same for second and third item in second list on level2. <<<so we have to find out what is the maximum number of items on every level, and make that number of lists>>> Then we go to third level. Only List3 has a third level and one list on that level. So the fourth list will have Null, Null, B3, Null, Null, C3. And then there are no levels anymore and we are finished.

firstList: Null, A2, Null, Null, B2, Null

secondList: A1, Null, A3, B1, Null, Null, C1, Null, Null

thirdList: D1,Null, Null, E1, Null, Null, F1, Null, Null

fourthList: Null, Null, B3, Null, Null, C3

In pseudo code:

Check how many levels <<< how?>>> go through every list and add 1 to a listCounter if you find a list, and again if in that list you find a list etc. (so you need [recursion](http://interactivepython.org/courselib/static/pythonds/Recursion/recursionsimple.html)\*\*) If there are no more lists, set the levelCounter to the listCounter and the listCounter to 0, go to the next item in the list and do the same. But maybe better solution\*.

Repeat for every level

Check how long is the longest list in this level

Repeat for the pattern

Repeat for every inputList

Repeat for the longest list length

If an item of a list is a (list, tulip or set) then add Null to this loopList

Otherwise add the item to the loopList<<<easier would be to start at the deepest level and then set the collectList to collectList.append(collectList)<<<no that would not work<<<how do you create lists in lists?<<<well in fact we output a list of lists, so only two levels, so we have a loopList that we add every time to the collectList>>>

Add the loopList to the collectList

Empty the loopList

Depending on the Boolean input we delete the Null entries

And how for pattern : 1,1,3,2

firstList : Null, Null, Null, A2, Null, Null, Null, B2

secondList : A1, B1, A3, Null, C1, Null, Null, Null

thirdList : D1, E1, Null, Null, F1, Null, Null, Null

fourthList : Null, Null, B3, Null, Null, Null, C3, Null

1. Get the next item from the lists according to the pattern, until all the lists are exhausted
2. Go one level deeper, and do the same, until the levels are exhausted.

In fact “go one level deeper” is not the right expression, because in List1 that would be A1 – F1, while we want to treat A1-C1 and D1-F1. So we should find a way to use List 1.1 and List 1.2 separately. Maybe start at the bottom of the levels?

Or remember that you found list1.1 and 1.2? Or start always at the last treated level, and process the lists of that level. If there are no lists then put Null as long as the pattern runs.

Are there items in this list at this level?

If so how many times does the pattern fit? And for the other lists at this level? Repeat the pattern for the max of that (maxPatternFit).

You could also say:

1. There are three lists.
2. Every list has 1 or more levels of nesting.
3. Maximum number of levels in this case is 3.
4. Every list consists of single values and/or lists.
5. Collect items from the lists per level in a certain pattern.
6. If an item doesn’t exist or if it is a list then put Null.
7. If at a level all the lists are processed, go to the next level.

So you repeat for the number of levels (or if you recursively go down in the levels until there are no lists in any of the lists). Then you take the pattern and start collecting items from the lists, until the lists are exhausted (that sounds like recursion too). That is if item[x] does not exist in all lists at that level. If an item is a list we put Null. If an item does not exist in a list, but does exist in another list, we also put Null.

# What is this Lambda function?

\* depth = lambda L: isinstance(L, list) and max(map(depth, L))+1

Many programmers (e.g. at <http://stackoverflow.com>) argue against this way of using lambda, because it is not an anonymous function if you give it a name and it is unreadable for programmers that don’t use lambda often. However to understand what is going on here you need to know the following.

lambda is a short way to write a function. f = lambda a,b: a+b returns the sum of a and b. You call the function as follows: g = f(1,2). So here maxDepth = depth(myList) returns L (which is max of the map function) if it is an instance of a list AND if the map function returns a value, which in its turn uses the same depth function (so recursively) over the lists of lists, while max is a function that returns the maximum value of the map function. You add 1 because every list has at least a depth of 1.

**map**(*function*, *iterable*, *...*)

Return an iterator that applies *function* to every item of *iterable*, yielding the results. If additional *iterable* arguments are passed, *function* must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted. For cases where the function inputs are already arranged into argument tuples, see [itertools.starmap()](https://docs.python.org/3/library/itertools.html" \l "itertools.starmap" \o "itertools.starmap).

**max**(*iterable*, *\**[, *key*, *default*])

**max**(*arg1*, *arg2*, *\*args*[, *key*])

Return the largest item in an iterable or the largest of two or more arguments.

If one positional argument is provided, it should be an *[iterable](https://docs.python.org/3/glossary.html" \l "term-iterable)*. The largest item in the iterable is returned. If two or more positional arguments are provided, the largest of the positional arguments is returned.

There are two optional keyword-only arguments. The *key* argument specifies a one-argument ordering function like that used for [list.sort()](https://docs.python.org/3/library/stdtypes.html" \l "list.sort" \o "list.sort). The *default* argument specifies an object to return if the provided iterable is empty. If the iterable is empty and *default* is not provided, a[ValueError](https://docs.python.org/3/library/exceptions.html#ValueError) is raised.

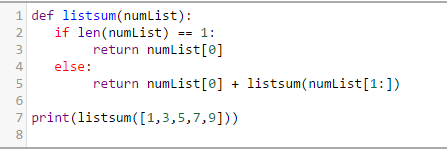
If multiple items are maximal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such assorted(iterable, key=keyfunc, reverse=True)[0] and heapq.nlargest(1, iterable, key=keyfunc).

# What is this Recursion

\*\* [Recursion](http://interactivepython.org/courselib/static/pythonds/Recursion/recursionsimple.html) has three major rules:

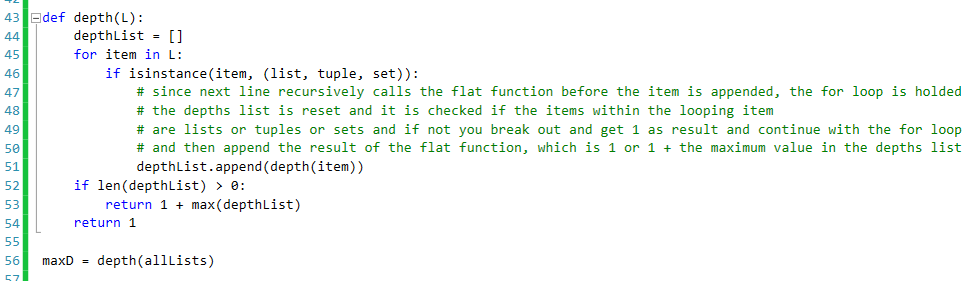
1. A recursive algorithm must have a **base case**.
2. A recursive algorithm must change its state and move toward the base case.
3. A recursive algorithm must call itself, recursively.

The basic form therefore is:



Mind the listsum recursive call of the function on line 5 with a list that is shortened by leaving out [0].

If we translate this to our problem of finding the depth of nested lists we get:



Try again for pattern = [1,1,3,2] writing exactly what you would do by hand in clear programmable steps.

1. Look at every level and figure out how many times you have to execute the pattern.
2. Then collect the items according to the pattern, skipping the items you already collected.
3. Put the collected items in a list of lists per list per level.

Maybe I can build a list that can be processed in the next loop? This sounds like recursion… That temporary list should consist of lists from the actual list but not to be treated as lists. So not recursion, but a second loop within the loop. The difficulty is what list to use in the next outer loop???

Or one could also say that every loop the input lists are flattened with one level. That would be recursion. NO!!! if you flatten you lose the fact that A1-C1 is another list then D1-F1.

In the first loop we add the items according to the pattern (and how many times) and we build the list for the second loop: tempList.append [[List1.1, List 1.2], [None], [List3.1]]. Then we process in the next loop this tempList according to the pattern (recursion) and we build the list for the next loop: tempList.append[[None,None],[None],[List 3.1.1]]. Then we process this list and build the next loop: [[None],[None],[None]]. Then we stop because there are not lists any more. You can check that with a function or by keeping track of a variable that switches when you add lists to the temporary list.

Still remains what you do with the items. You have to figure out how many times to execute the pattern. I worked on that already. That is the x\_pattern function in module1.py. But wasn’t finished. The problem was that if two or more lists should be processed per input list. But with a temprorary list it should be easier. Just count the items per list and divide by the number of times it appears in the pattern.

A nice looping trick is using enumerate: for (i,item) in enumerate(L): this gives you at the same time the index and the item of the list L.

# The Weave component result

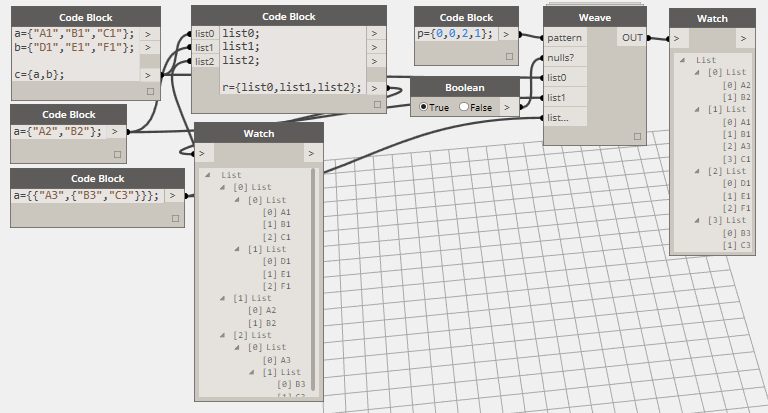


Figure 11 Final solution in Dynamo

The Python script in Visual Studio is as follows (it took 10 trials and about 4 full time equivalent weeks of trial and error while learning many solutions for Python programming).

inputList0 = [0,0,2,1] # the weave pattern

inputList1 = ["true"] # remove null items?

inputList2 = [["A1","B1","C1"],["D1","E1","F1"]]

inputList3 = ["A2","B2"]

inputList4 = [["A3",["B3","C3"]]]

IN = [inputList0, inputList1, inputList2, inputList3, inputList4]

# A1,B1,C1,D1,E1,F1,A2,B2,A3,B3,C3 = ""

""" more visual display of allLists

List1

[0] List1.1

[0] A1

[1] B1

[2] C1

[1]List1.2

[0] D1

[1] E1

[2] F1

List2

[0] A2

[1] B2

List3

[0] List3.1

[0] A3

[1] List3.1.1

[0] B3

[1] C3

result should be:

[[Null, Null, Null, A2, Null, Null, Null, B2],

[A1, B1, A3, Null, C1, Null, Null, Null],

[D1, E1, Null, Null, F1, Null, Null, Null],

[Null, Null, B3, Null, Null, Null, C3, Null]]

or without Nulls: [[A2,B2],[A1,B1,A3,C1],[D1,E1,F1],[B3,C3]]

"""

# above is for testing in Visual Studio, it should be deleted when used in Dynamo

dataEnteringNode = IN

# make a list of the inputs except IN[0] which is used for weave patern

# and without IN[1] which is a boolean for deleting Nulls

pattern = IN[0]

allLists = IN[2:]

collectList = []

def x\_pattern(L,pattern):

# L must have the format of singel lists within a list: [[],[],[]]

# Check how many times the pattern should be executed.

# e.g. p = 0,0,2,1 then check how many items you have in List1 and

# divide by the number of times it appears in the pattern and round up.

# do the same for consequent lists

# return the max of these, so build a list and return max of that list

n = []

for (i,item) in enumerate(L):

q = pattern.count(i)

r = len(item)

# r//q is floor division so round down;

# round up if you add (r % q > 0 ) which evaluates True (=1) if

# there is a remainder of the quotient

n.append(r//q + (r % q > 0 ) )

return max(n)

def make\_list(L,pattern):

# for every list process the items if they are not lists, so make thisList with the items

# in next round process for every list the lists, so recursively use this function with nextList

thisList = []

for item in L:

thisItem = []

for subItem in item:

if not isinstance(subItem,(list,tuple,set)):

thisItem.append(subItem)

thisList.append(thisItem)

nextList = []

for item in L:

nextItem = []

for subItem in item:

if isinstance(subItem,(list,tuple,set)):

nextItem.append(subItem)

nextList.append(nextItem)

altNextList = []

# promote the first items and subitems in nextList

for item in nextList:

# the not lists in items are already removed

# promote first sub items

nextItem = []

for (i,subItem) in enumerate(item):

if i == 0: # only promote the first sub items unless they are lists

for (s,subSubItem) in enumerate(subItem):

if isinstance(subSubItem,(list,tuple,set)) and s > 0:

nextItem.append([subSubItem]) # only promote the first sub sub list

else:

nextItem.append(subSubItem)

else:

nextItem.append(subItem)

altNextList.append(nextItem)

# Process thisList

loopList = []

for n in range(x\_pattern(thisList,pattern)): #n = 0,1 the number of times the pattern can be processed at this level

for (i,p) in enumerate(pattern): # pattern = 0,0,2,1

# process the items in thisList according to the pattern

# and number of times it should be executed

# if e.g. n = 2 and pattern is 0,0,2,1 then next loop should start

# with third item (item[[2]) of list 1 and second item of list 3 and 2 (if any, otherwise None)

# so x (index of sub item) in loop n must be augmented with the number of times

# x appears in pattern, but also with the number of times it was already processed

# in this loop. So x = done.count(p) + n\*pattern.count(p)

done = pattern[:i]

x = done.count(p) + n\*pattern.count(p) # x augments if same list reappears in pattern

# thisList[p][x+n-1] may not exist

try:

loopList.append(thisList[p][x])

except IndexError:

loopList.append(None)

collectList.append(loopList)

# there is no point in having a first empty list in case you weave lists in lists

if collectList[0]==[]:

del(collectList[0])

for item in altNextList:

# if all the items in the altNextList are [] empty then return the collectList

if item == []:

empty = True

else:

empty = False

break

if empty: return collectList

make\_list(altNextList,pattern)

# check if pattern fits with inputs

# test = list(range(0,len(allLists)))

if not set(pattern).issubset(list(range(0,len(allLists)))):

collectList = ["Pattern doesn't fit with input."]

else:

# here is the actual basic command that starts the whole thing

make\_list(allLists,pattern)

def remove\_None(L):

returnList = []

for item in L:

temp = []

# only treat lists

if isinstance(item,(list,tuple,set)):

for subItem in item:

if subItem is not None:

temp.append(subItem)

returnList.append(temp)

else:

returnList = L

return returnList

#Assign your output to the OUT variable

# If nulls ? is false

if IN[1]==False:

OUT = remove\_None(collectList)

else:

OUT = collectList

print("OUT = ",OUT)

# The Python script in Dynamo is slightly different:

**import** clr  
clr.**AddReference**('ProtoGeometry')  
**from** Autodesk.DesignScript.Geometry **import** \*  
dataEnteringNode = IN  
# make a list of the inputs except IN[0] which is used for weave patern  
# and without IN[1] which is a boolean for deleting Nulls  
pattern = IN[0]  
allLists = IN[2:]  
collectList = []  
  
*def* **x\_pattern**(L,pattern):   
# L must have the format of singel lists within a list: [[],[],[]]  
# Check how many times the pattern should be executed.  
# e.g. p = 0,0,2,1 then check how many items you have in List1 and   
# divide by the number of times it appears in the pattern and round up.  
# do the same for consequent lists  
# return the max of these, so build a list and return max of that list   
    n = []  
    **for** (i,item) **in** **enumerate**(L):  
        q = pattern.**count**(i)  
        r = **len**(item)  
        # r//q is floor division so round down;   
        # round up if you add (r % q > 0 ) which evaluates True (=1) if  
        # there is a remainder of the quotient  
        n.**append**(r//q + (r % q > 0 ) )   
    return **max**(n)  
  
  
*def* **make\_list**(L,pattern):  
# for every list process the items if they are not lists, so make thisList with the items  
# in next round process for every list the lists, so recursively use this function with nextList  
    thisList = []  
    **for** item **in** L:  
        thisItem = []  
        **for** subItem **in** item:  
            **if** **not** **isinstance**(subItem,(list,tuple,set)):  
                thisItem.**append**(subItem)  
  
        thisList.**append**(thisItem)  
  
    nextList = []  
    **for** item **in** L:  
        nextItem = []  
        **for** subItem **in** item:  
            **if** **isinstance**(subItem,(list,tuple,set)):  
                nextItem.**append**(subItem)  
  
        nextList.**append**(nextItem)  
  
    altNextList = []  
    # promote the first items and subitems in nextList  
    **for** item **in** nextList:    
        # the not lists in items are already removed  
        # promote first sub items  
        nextItem = []  
        **for** (i,subItem) **in** **enumerate**(item):  
            **if** i == 0: # only promote the first sub items unless they are lists  
                **for** (s,subSubItem) **in** **enumerate**(subItem):  
                    **if** **isinstance**(subSubItem,(list,tuple,set)) **and** s > 0:  
                        nextItem.**append**([subSubItem]) # only promote the first sub sub list  
                    **else**:  
                        nextItem.**append**(subSubItem)  
  
            **else**:  
                nextItem.**append**(subItem)                                  
  
        altNextList.**append**(nextItem)  
  
    # Process thisList   
    loopList = []  
    **for** n **in** **range**(**x\_pattern**(thisList,pattern)): #n = 0,1 the number of times the pattern can be processed at this level  
        **for** (i,p) **in** **enumerate**(pattern): # pattern = 0,0,2,1  
            # process the items in thisList according to the pattern   
            # and number of times it should be executed  
            # if e.g. n = 2 and pattern is 0,0,2,1 then next loop should start  
            # with third item (item[[2]) of list 1 and second item of list 3 and 2 (if any, otherwise None)  
            # so x (index of sub item) in loop n must be augmented with the number of times  
            # x appears in pattern, but also with the number of times it was already processed  
            # in this loop. So x = done.count(p) + n\*pattern.count(p)  
            done = pattern[:i]  
            x = done.**count**(p) + n\*pattern.**count**(p)  # x augments if same list reappears in pattern  
  
            # thisList[p][x+n-1] may not exist  
            **try**:  
                loopList.**append**(thisList[p][x])  
            **except** IndexError:  
                loopList.**append**(None)  
  
    collectList.**append**(loopList)  
    # there is no point in having a first empty list in case you weave lists in lists  
    **if** collectList[0]==[]:  
        **del**(collectList[0])  
  
    **for** item **in** altNextList:  
        # if all the items in the altNextList are [] empty then return the collectList  
        **if** item == []:  
            empty = True  
        **else**:  
            empty = False  
            break  
  
    **if** empty: return collectList          
  
    **make\_list**(altNextList,pattern)  
  
# check if pattern fits with inputs  
# test = list(range(0,len(allLists)))  
**if** **not** **set**(pattern).**issubset**(**list**(**range**(0,**len**(allLists)))):  
    collectList = ["Pattern doesn't fit with input."]  
**else**:   
    # here is the actual basic command that starts the whole thing  
    **make\_list**(allLists,pattern)  
  
  
*def* **remove\_None**(L):  
    returnList = []  
    **for** item **in** L:  
        temp = []  
        # only treat lists  
        **if** **isinstance**(item,(list,tuple,set)):  
            **for** subItem **in** item:  
                **if** subItem **is** **not** None:  
                    temp.**append**(subItem)  
  
            returnList.**append**(temp)  
        **else**:  
            returnList = L  
  
    return returnList  
  
#Assign your output to the OUT variable  
# If nulls ? is false  
**if** IN[1]==False:   
    OUT = **remove\_None**(collectList)  
**else**:  
    OUT = collectList

What helped was visualising the lists and wished result in Excel in a small window (left in Figure 12) and many print commands to see intermediate results in the Output window of VS. The window with black background is Python running in the background in VS (showed it in foreground to make it clear).

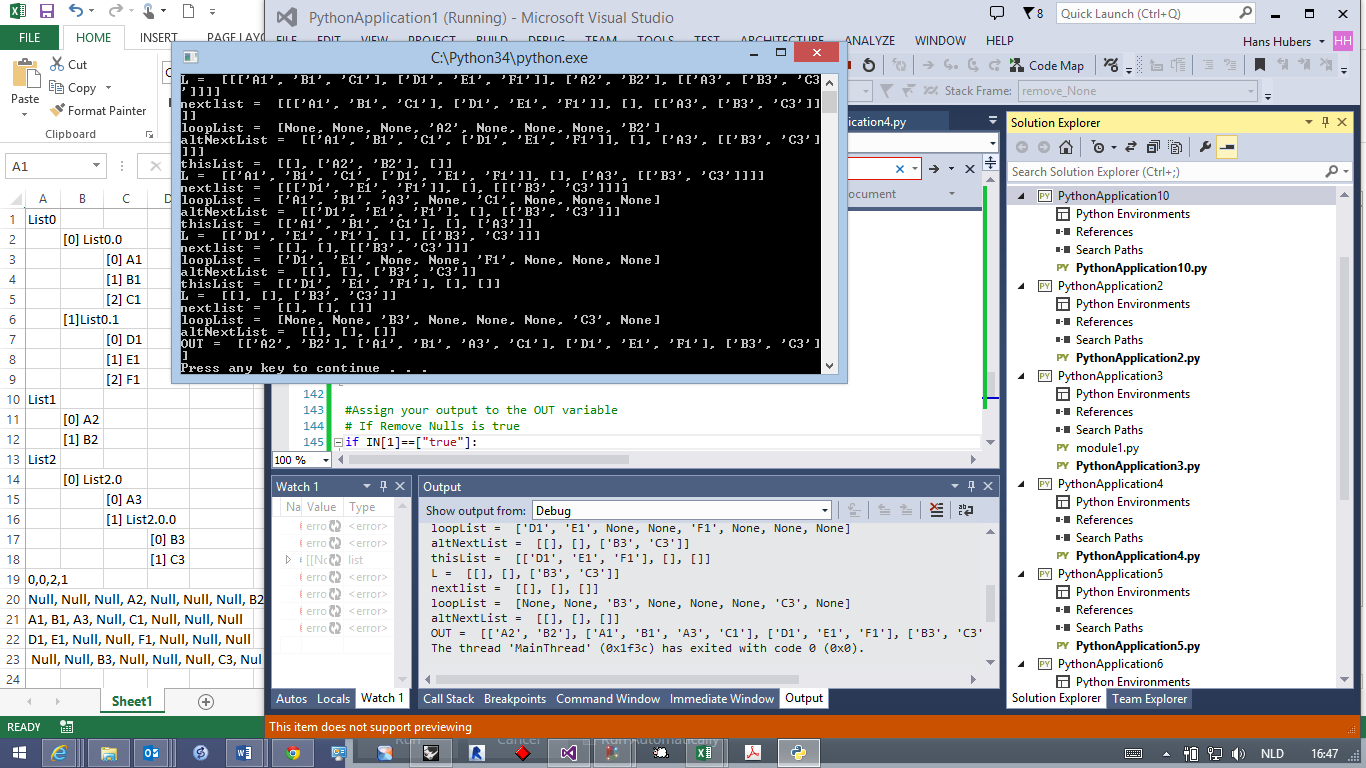


Figure 12 Working in Visual Studio and Excel

Continuing with the wave wall exercise

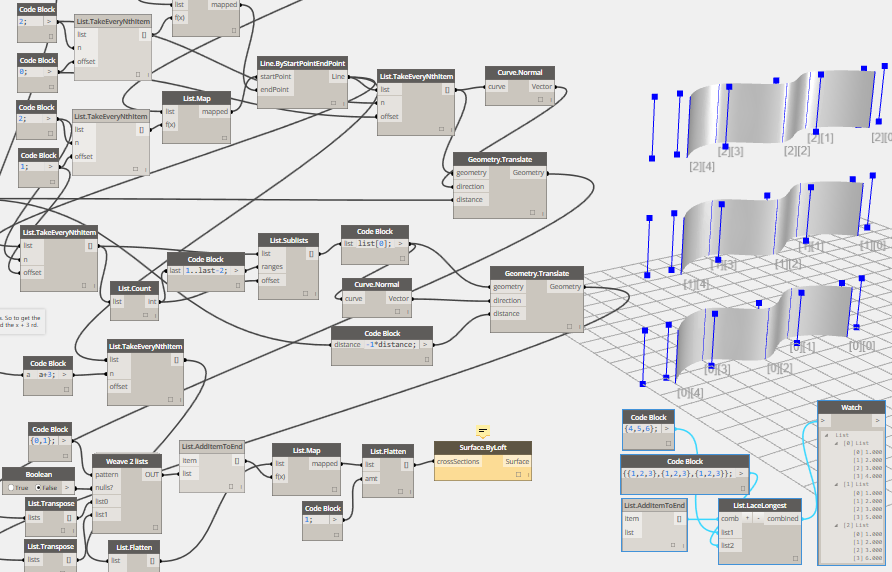


Figure 13 Continuing with the wave wall exercise

Figure 13 shows the application of the weave Python script. But I ran into a problem where I wanted to add the start curves of ribbons to beginning and end. To find the solution two things helped. Right click a node and activate ‘Show Labels’. This displays the list arguments of the parts in 3D and helps figuring out which part of the lists goes where. And a simple example as in Figure 14 where I found how to add items from a list to lists in lists. Again by using a node as a function. In this case List.AddItemToEnd as function in the List.LaceLongest node.

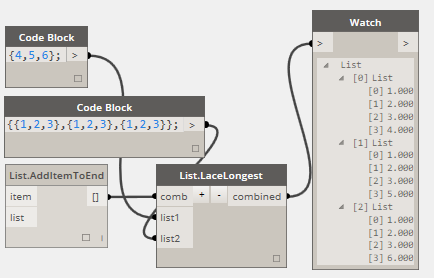


Figure 14 Using List.AddItemToEnd as function in the List.LaceLongest node

From here we should start investigating the use of VB.net prototype we made for pCOLAD in the Dynamo environment. [There is a way](https://github.com/DynamoDS/Dynamo/wiki/How-To-Create-Your-Own-Nodes#recursion) to simply insert nodes through dll’s using the Zero Touch interface. However it is not using VB.net but C#. Does a VB.net dll work anyway, or do we have to translate pCOLAD nodes to C#? We will start a separate document about this, because in the meanwhile we are in january 2015.