DesignScript

A Robust Dive into the Language Underlying Dynamo

Sol Amour

Curious Human Being

GR CENTRE

LJUBLJANA

11 – 13 OCT 2018

@solamour

amoursol@gmail.com

SOL AMOUR

CURIOUS HUMAN BEING

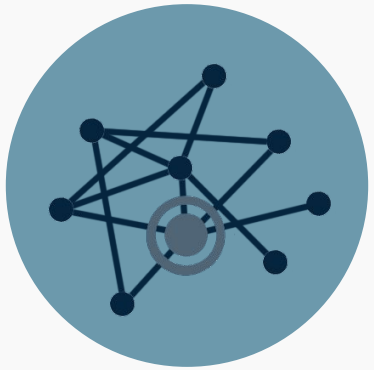
- Industry expertise in Architecture, Landscaping, Industrial Design, and the Construction Sectors
- New job on the horizon - currently a Lone Ranger. Previously at designtech.io
- Focus upon Automation, Computational Workflows and Education
- Passionate about leveraging technology to do laborious and repetitive tasks in order to allow designers to have fun again

INTRODUCTION
KEY LEARNING OBJECTIVES

1.

DesignScript Fundamentals:

Syntax, Object Types, Legibility, UI and IDE



2.

Data Management:

Replication, List@Level, Nesting and Conflicts

3.

Power and Control:

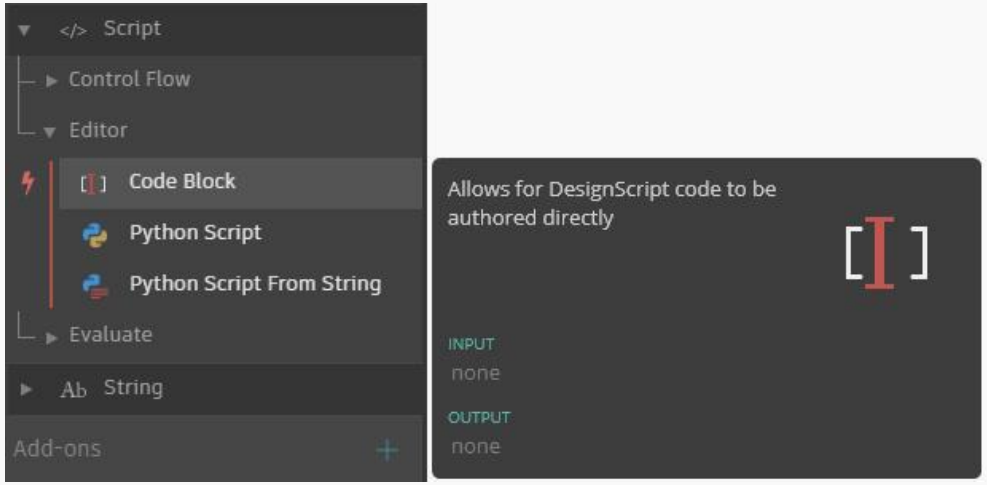
Conditional statements, Operators, Looping and Custom Definitions

What is DesignScript?

The foundational language of Dynamo

Legible scripting language

Intersection of Design and Scripting



Where can I find DesignScript in Dynamo?

Authored inside a special node called a 'Code Block'
Accessible through the Search features or Located in the Library:

- **Script -> Editor -> Code Block**

Note: Code Blocks have their own keyboard shortcut of **double left click.**

DesignScript Fundamentals

[Exploring the Building Blocks of DesignScript]

IDE
AUTO-COMPLETE

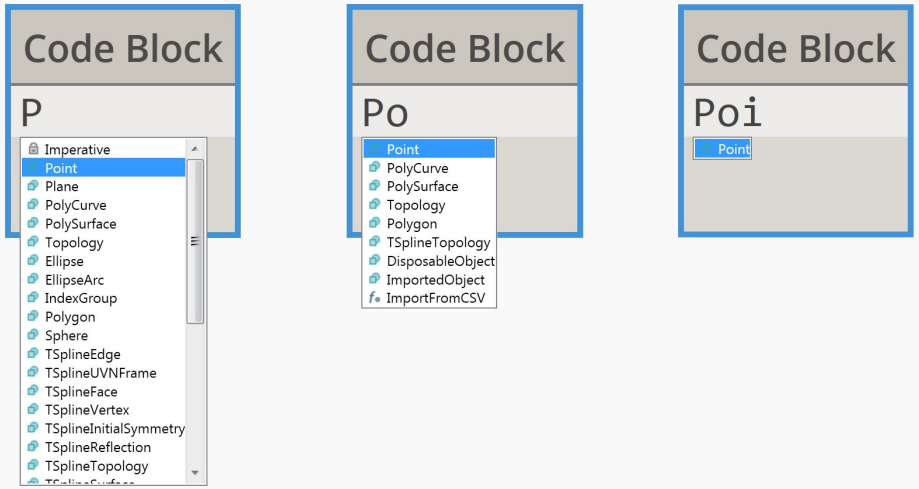
DesignScript is written inside a **Code Block**.

When typing, the integrated **auto-complete** help feature will prompt you to help accelerate your code by suggesting functions and inputs. These suggestions populate in a *scrollable box* underneath your typed DesignScript line of code.

You can *tab* or *click on the function* to finish the current suggestion.

Auto-complete will limit possible options the more that you type as showcased in the images to the right.

Note: **IDE** is an acronym for 'Integrated Development Environment' and Dynamo's one is simple; including **auto-complete** and **dot-notation**. Debugging happens through warnings thrown by the node itself and outputs are checked using **Watch nodes**.



IDE
DOT-NOTATION

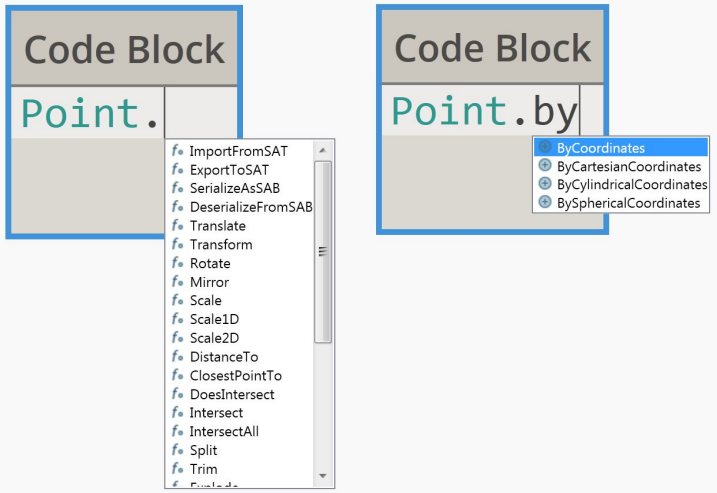
DesignScript contains a **dot-notation** feature that showcases available constructors (creators), methods (actions) and properties (queries) of your chosen Class.

To access **dot-notation** you simply put a dot (period) after your Class. This will bring up a drop-down menu, aligned with your pointer, that you can (using auto-complete) type in your chosen action.

Functions are demarcated by the function marker (**fx**) icon: 

Constructors are demarcated by the plus (**+**) icon: 

Properties do not appear inside of dot-notation.



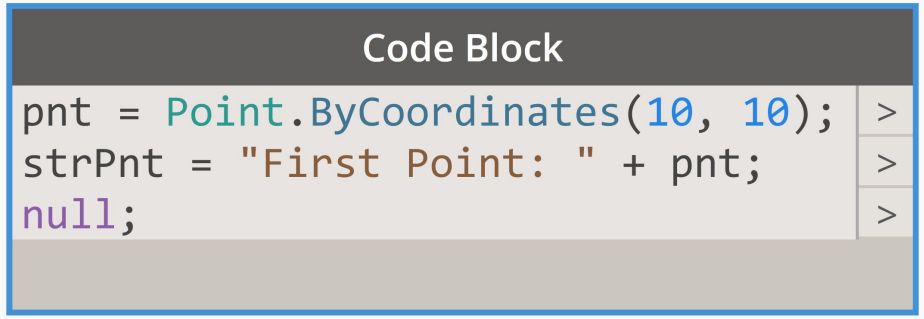
IDE
COLOUR

The **UI** (User Interface) of DesignScript is simply colourised code inside of a **Code Block**. Colour is used to differentiate various features:

Classes are green.
Methods are blue.
Primitive Types are purple.
Numbers are a bolder blue.
Strings are brown.

All other syntax inside of DesignScript is **black**.

Colours can be used as a guide in helping for swift and accurate reading of DesignScript code.



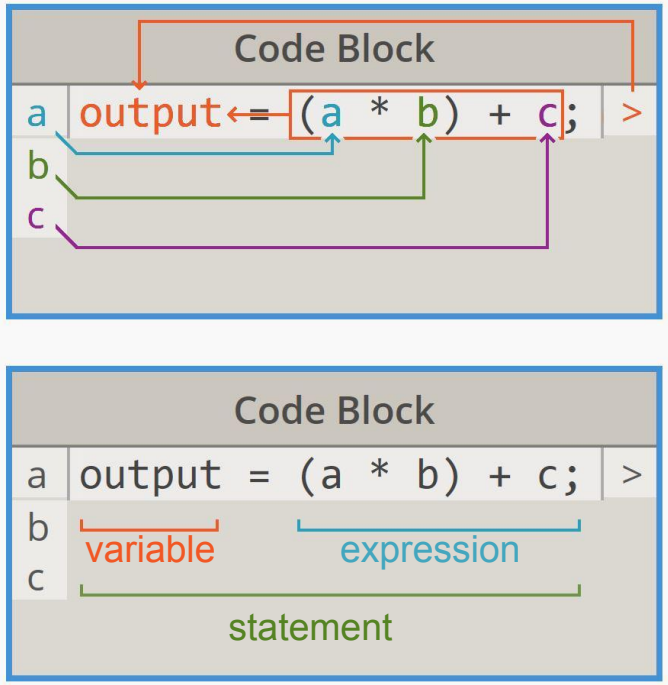
CONCEPT
CODE BLOCKS & STATEMENTS

A **Code Block** will automatically generate Input ports and output ports based upon the code authored within. Output ports are generated by terminating a line of code with a semi-colon (**;**):

output = Our named variable.
a | **b** | **c** = Undefined variables become input ports.
> = Output ports are generated at the end of every line of code.

The value of the expression is assigned right-to-left to the variable.
The expression is evaluated according to the precedence of the operators and brackets.

Learn more at http://designscript.io/DesignScript_user_manual_0.1.pdf

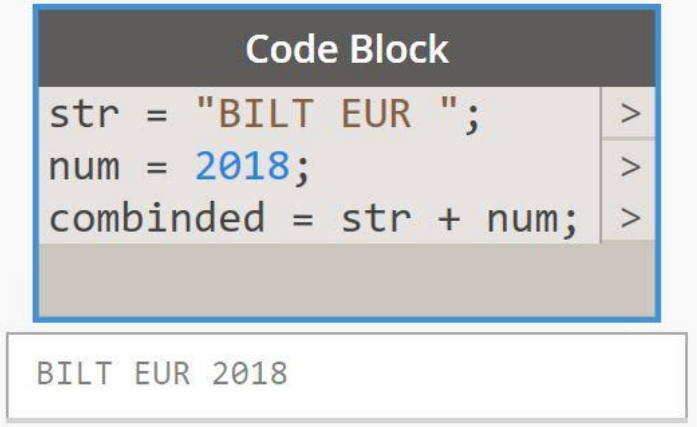


CONCEPT
VARIABLES

Variable - They act like containers that hold information, callable by their name*:

```
str = "BILT EUR ";
num = 2018;
combined = str + num;
```

Assign using the **variableName = data** syntax.
Variables are used to store information in memory to then be referenced and manipulated in DesignScript.
They allow us to label data with a descriptive name to make sense of our code.
They provide clarity when others read your code (If you are diligent at naming!).



Note: You can assign a variable to another variable.
Using concatenation you have to account for the spaces yourself (Refer to example image).
*Variables are not **global**. They are **local** to their own Code Block.

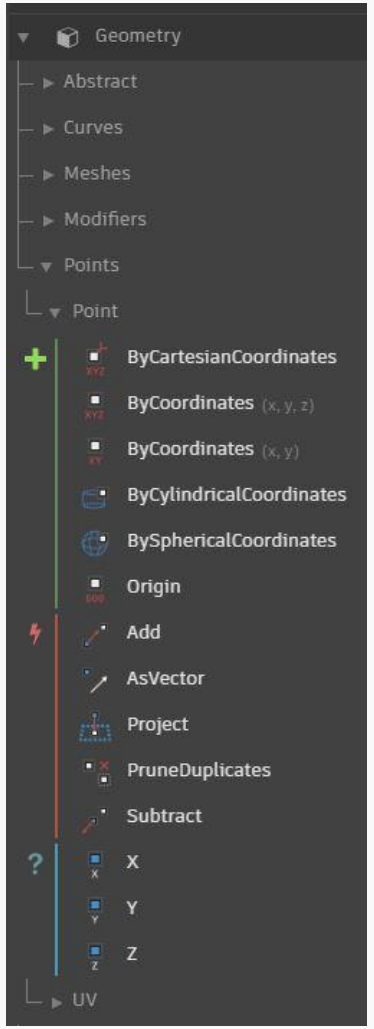
CONCEPT
ORGANISATION

A **namespace** is an abstract container to hold logical groupings of functionality (E.g Objects Class). This is best illustrated with an example:

Geometry (**Namespace**)
- Points (**Class**)
  - Point (**Class**)
    - ByCoordinates (**Constructor** aka Create)
    - Add (**Method** aka Action)
    - X (**Property** aka Query)

They are a way to implement scope (Define boundaries).
Can be understood as a 'Parent : Child' relationship (Inheritance).

Note: In object-oriented programming, a **Class** is a template definition of the methods and variables in a particular kind of object. An **Object** is an instance of a class that contains real values instead of variables.



CONCEPT
COMMENTS

In DesignScript there are two commenting methods; **Single Line** and **Multi-Line**; Single line comments are initialised by a double backslash (//), multi-line comments are initialised by a **back-slash and asterisk** (/*) and closed by an **asterisk and back-slash** (*/):

```
// I'm a Single Line comment
/* I'm a multi
line
Comment */;
```

Single line comments will not 'wrap' to the next line.
Multi-line comments will wrap until closed.
Single line comments do not need to be terminated by a semi-colon. Multi-line comments do.

Code Block

data	`// I'm a single line comment`
moreData	`data;` >
	`/* I'm a multi`
	`line`
	`comment */;`
	`moreData;` >

OBJECT TYPES
BOOLEAN

Boolean - DesignScript for 'true / false'. Boolean values are constant objects and are used to represent truth values:

boolTrue = **true**;
boolFalse = **false**;

In numeric contexts booleans behave like the integers 0 and 1. Boolean values can also be referenced as Yes or No.

Code Block

```
boolTrue = true;
boolFalse = false;
```

OBJECT TYPES
NUMBERS

Int (Integer) - Whole number:

intVar = 10;

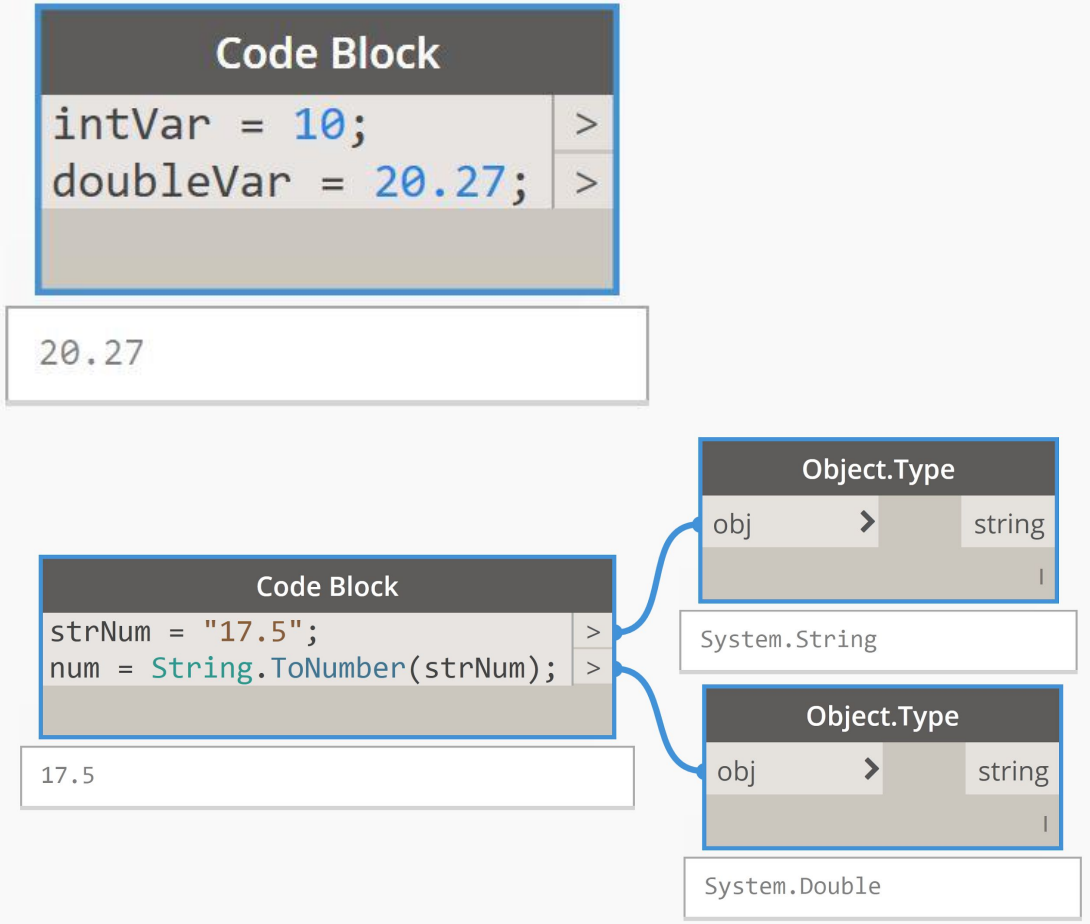
Double (Float / Number) - Decimal placed number:

doubleVar = 20.27;

As both **int** and **double** are reserved keywords we can't use them as variable names.
Code Blocks only return the final result of the line of code (As shown in the image exemplar).

You can cast (Change the type of) a *String* representation of a number **to** a number using:

String.ToNumber(number);



OBJECT TYPES

STRINGS

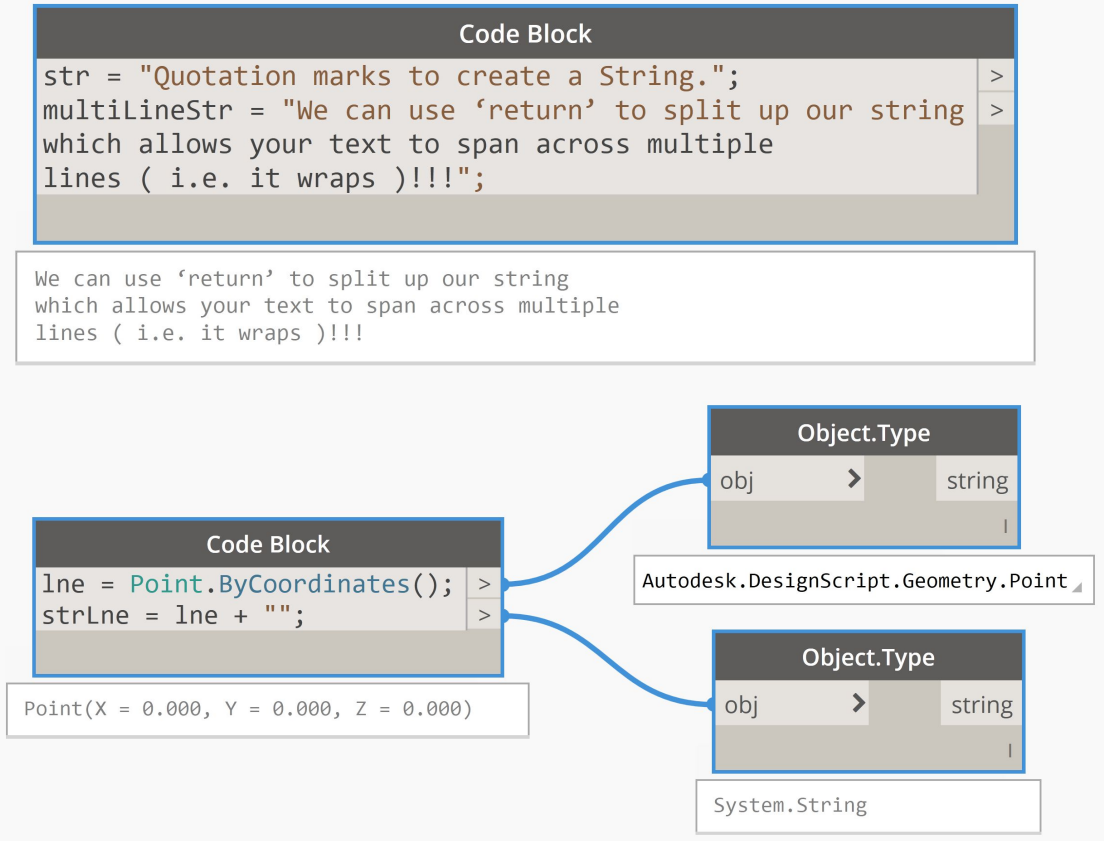
String - DesignScript for 'Text'. Enclose inside of quotation marks:

```
str = "Quotation marks to create a String.";
multiLineStr = "We can use 'return' to split up our string
which allows your text to span across multiple
lines ( i.e. it wraps )!!!";
```

You can cast (Change the type of) any object **to** a String by concatenating empty quotations to that type:

```
lne = Point.ByCoordinates();
strLne = lne + "";
```

Note: There is a graphical inconsistency with the wrapped line text that fails to colour the subsequent lines. The code will correctly execute however.
Not to be confused with **text objects** inside of Revit



EXPLORE
VARIABLES, STRINGS & NUMBERS

Create your own **variable(s)** now by assigning a **string** or **number** value to each of the following inside a fresh **Code Block**, (And replacing my information with your own):

name = **“Name: “**;
myName = **“Sol Amour“**;
country = **“Country: “**;
originCountry = **“New Zealand“**;
age = **“Age: “**;
myAge = **31.5**;

Check the preview bubble (Or use a Watch node) to see the results!

Code Block

```
name = "Name: ";
myName = "Sol Amour";
country = "Country: ";
originCountry = "New Zealand";
age = "Age: ";
myAge = 31.5;
```

```
31.5
```

OBJECT TYPES
LISTS

List - A changeable (mutable) ordered container of elements:

emptyList = [ ];
numberList = [1, 2, 3];
stringList = [“BILT EUR”, “2018”];
mixedList = [1, 2.5, “Learning”, stringList];

Lists are declared by a named variable (var) followed by equals (=) and square brackets ([ ]).

Note: The way I am naming my variables is called Camel Case, where the first word is all lowercase and all subsequent words have their first letter capitalised. This is the preferred naming convention for the Dynamo team.

*Square braces exist in Dynamo 2.0 onwards - Previously **lists** were initialised by curly braces.*

Code Block

```
emptyList = [];
numberList = [1, 2, 3];
stringList = ["BILT EUR", 2018];
mixedList = [1, 2.5, "Learning", stringList];
```

```
List
    0 1
    1 2.5
    2 Learning
  3 List
      0 BILT EUR
      1 2018
@L3 @L2 @L1                                    {5}
```

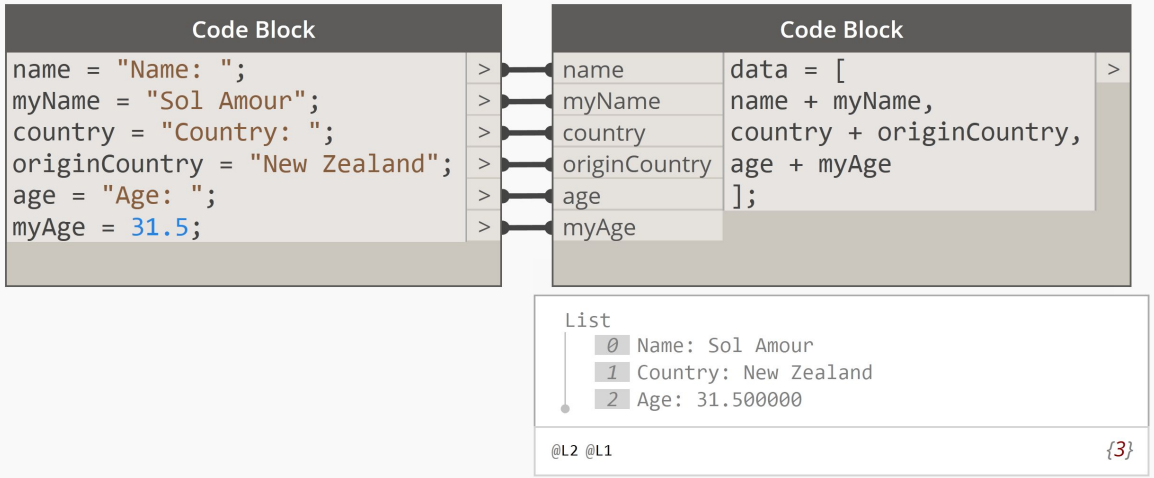
EXPLORE
LISTS

Exploring **Lists** using our previous **Code Block** data, we can pair our data together using *concatenation* (**+**). Write the following inside your existing **Code Block**:

```
data = [
    name + myName,
    country + originCountry,
    age + myAge
    ];
```

Check the preview bubble (Or use a Watch node) to see the results!

Note: I am lavishly using *returns* here, but if you prefer you can write your data all on one single line. Using returns like the above example increases code clarity.



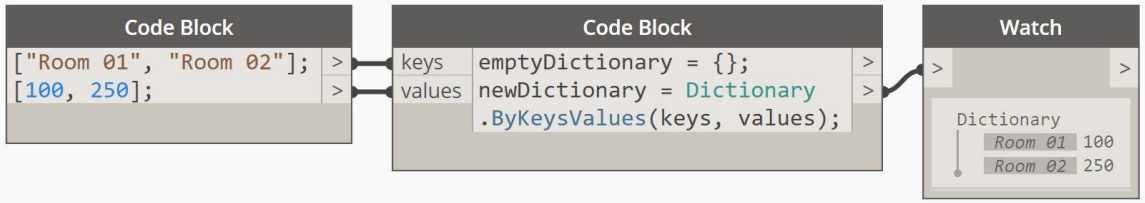
OBJECT TYPES
DICTIONARY

Dictionary (Dictionary.ByKeysValues) - A collection of 'key : value' paired objects:

emptyDictionary = { };
newDictionary = Dictionary.ByKeysValues(keys, values);

Dictionaries are declared by a named variable (var) followed by equals (=) and curly braces ({ }).
Dictionaries are unordered lists. They will 'shuffle'. You get correct values by calling their respective 'keys'.
Values are only returned when you query the 'key' index using square braces or the correct function:

newDictionary["Room 01"];
[returns] **100;**



EXPLORE
DICTIONARIES

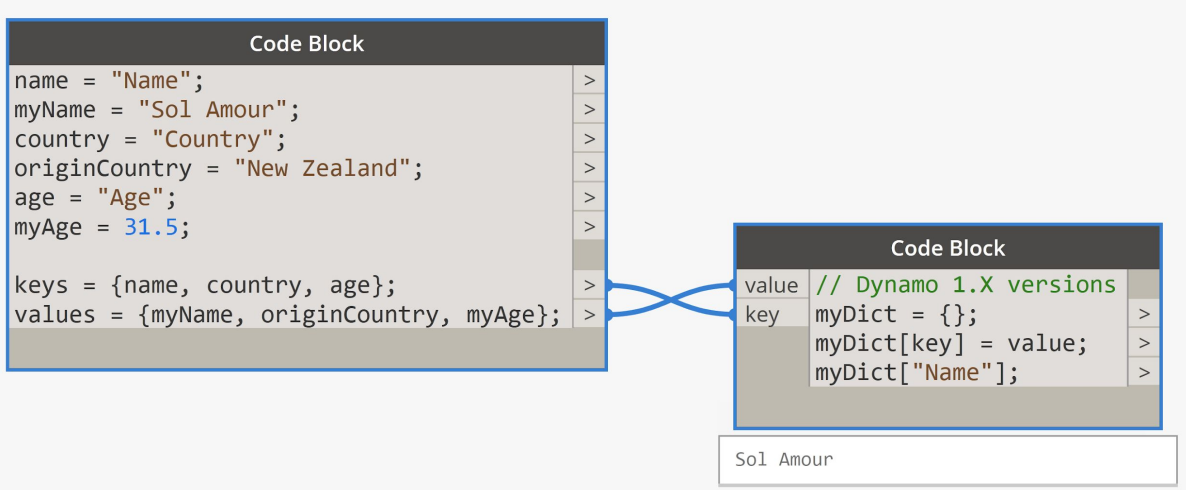
However, a better data container for general demographic information is **Dictionaries**. We need a **list** of 'keys' and a **list** of 'values':

keyList = [name, country, age];
valueList = [myName, originCountry, myAge];

personData = Dictionary.ByKeysValues(keys, values);

Type in the above 2.0 Dictionary syntax and see your results after generating **'key'** and **'value'** lists. Then check the preview bubble (Or use a Watch node) to see the results!

Note: The image to the left is the Dynamo 1.X version when Lists and Dictionaries both shared the Curly Braces syntax.



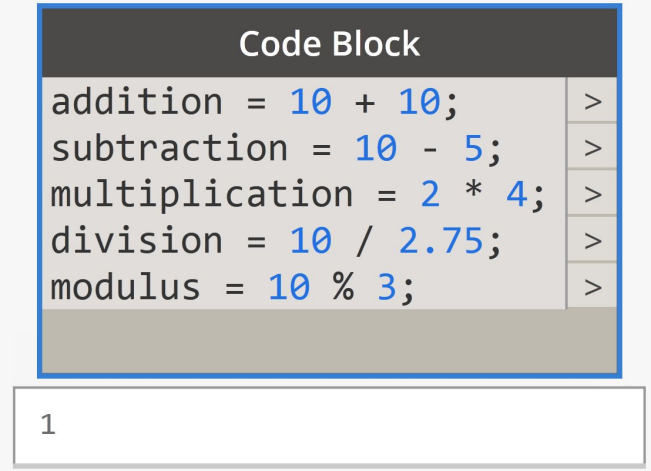
OPERATORS
ARITHMETIC

In DesignScript we have Operators which manipulate a value / variable. **Arithmetic** covers the mathematical operations and are as follows syntactically:

addition = 10 + 10;
subtraction = 10 – 5;
multiplication = 2 * 4;
division = 10 / 2.75;
modulus = 10 % 3;

Modulus is the division remainder - After the first number is divided by the second and rounded down - what is left? (3 goes into 10 three times, with 1 as the remainder)

Note: Addition (+) can also be used for string concatenation.



OPERATORS
COMPARISON

Comparison Operators allows us to check relationships between things (variables) and result in either a **True** or a **False** boolean. Syntactically they are:

```
greaterThan = 10 > 15;
greaterThanOrEqualTo = 10 >= 10;
lessThan = 5 < 10;
lessThanOrEqualTo = 5 <= 3;
equality = 5 == 8;
notEquals = 5 != 10;
```

Code Block

```
greaterThan = 10 > 15;
greaterThanOrEqualTo = 10 >= 10;
lessThan = 5 < 10;
lessThanOrEqualTo = 5 <= 3;
equality = 5 == 8;
notEquals = 5 != 10;
```

```
true
```

OPERATORS
MEMBERSHIP & BOOLEAN

DesignScript has two primary membership operators (Whether or not something is inside a container (list)):

List.Contains(list, object) = Evaluates to true if it finds a thing (variable) inside the container, false if it does not.
!List.Contains(list, object) = Evaluates to true if it does not find a thing (variable) inside the container, false if it does.

Boolean operators will check a value against multiple True / False queries and will evaluate as follows:

and (&&) = Evaluates to true if **all** expressions are True, false if it does not.
or (||) = Evaluates to true if **any** expression is True, False if it does not.

Code Block

```
lst = [1, 2, 2.4, 5];
contains = List.Contains(lst, 2);
doesNotContain = !List.Contains(lst, 2.4);
```

Code Block

```
range = 0..5;
andOperator = range > 1 && range < 4;
orOperator = range == 2 || range >= 4;
```

EXPLORE
ARITHMETIC, COMPARISON & BOOLEAN

A common comparison use case is an **'If'** statement. Syntactically this is represented inside of DesignScript as follows:

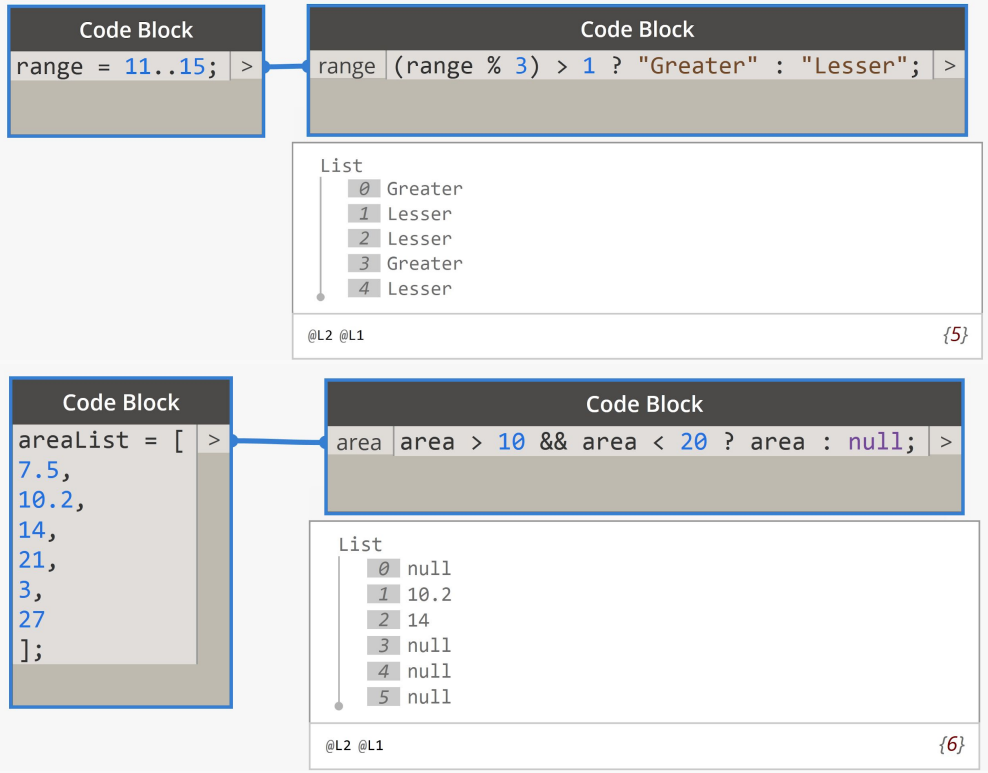
item comparison item ? doThisIfTrue : doThisIfFalse;

range = 11..15;
(range % 3) > 1 ? "Greater" : "Lesser";

We can use this syntax to build complex *Masks* for the *List.FilterByBoolMask* function that pass through compliant objects:

area > 10 && area < 20 ? area : null;

Note: Parenthesis are only needed with complex equations as BEDMAS algebraic order of operations apply. We use parenthesis in the above example but not below. Both will function without but for clarity can be used.



SYNTAX
RESERVED KEYWORDS

Some words in DesignScript are **reserved** – which means you cannot use them inside the Dynamo integration of DesignScript.

All **Classes** are also reserved and cannot be used as keywords (E.g. 'List', 'Point', 'Area' etc.).

break	class	constructor	continue	def
else	elseif	extends	false	for
from	if	import	in	null
return	static	true	while	

Data Management

[Managing data through powerful DesignScript features]

CONCEPT
USING FUNCTIONS AS ARGUMENTS

DesignScript allows for the use of functions as arguments. For example you can call two *Point.ByCoordinates()* functions inside of a *Line.ByStartPointEndPoint()*. To call a function multiple times inside a singular Code Block you would want to write each function on it's own line:

```
startPoint = Point.ByCoordinates( );
endPoint = Point.ByCoordinates(10, 0, 0);
newLine = Line.ByStartPointEndPoint( startPoint, endPoint );
```

However, if the function will not be called more than once, you can write your code more concisely by calling the function directly inside of another function:

```
newLine = Line.ByStartPointEndPoint( Point.ByCoordinates( ),
Point.ByCoordinates( 10, 0, 0 ) );
```

Code Block

```
startPoint = Point.ByCoordinates();
endPoint = Point.ByCoordinates(10, 0, 0);
newLine = Line.ByStartPointEndPoint(startPoint, endPoint);
```

```
Line(StartPoint = Point(X = 0.000, Y = 0.000, Z = 0.000), EndPoint = Point(X =
```

Code Block

```
newLine = Line.ByStartPointEndPoint(Point.ByCoordinates(),
Point.ByCoordinates(10, 0, 0));
```

```
Line(StartPoint = Point(X = 0.000, Y = 0.000, Z = 0.000), EndPoint = Point(X =
```

CONCEPT
STATIC VS. INSTANCE METHODS

DesignScript allows for two different ways to call a function: **static** and **instance.** Static methods are called in full (verbose) and are the preferred way (safest) to call a function in DesignScript. Creating *Surface Points* statically is initialized as follows:

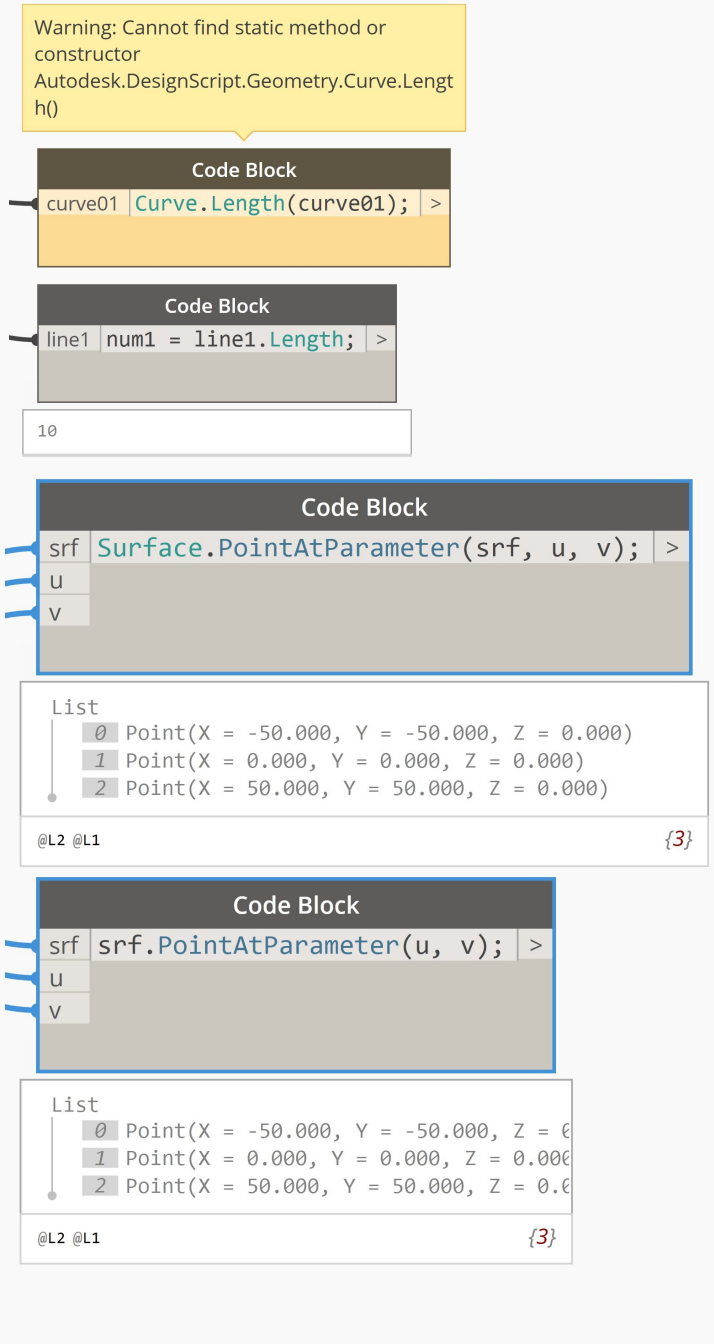
Surface.PointAtParameter(srf, u, v);

Creating an instance of *Surface Points* is called as follows:

srf.PointAtParameter(u, v);

Static methods include all arguments inside of the parenthesis.
Instance methods skip use the first argument to use before dot-notation and then omit it from within the parenthesis.

Note: Some functions (Typically properties) do not have a static method or constructor and only use their **instance** method. When in doubt, use **Node2Code** to see how a function converts a Nodal implementation to DesignScript.



CONCEPT
CLASS CONFLICTS

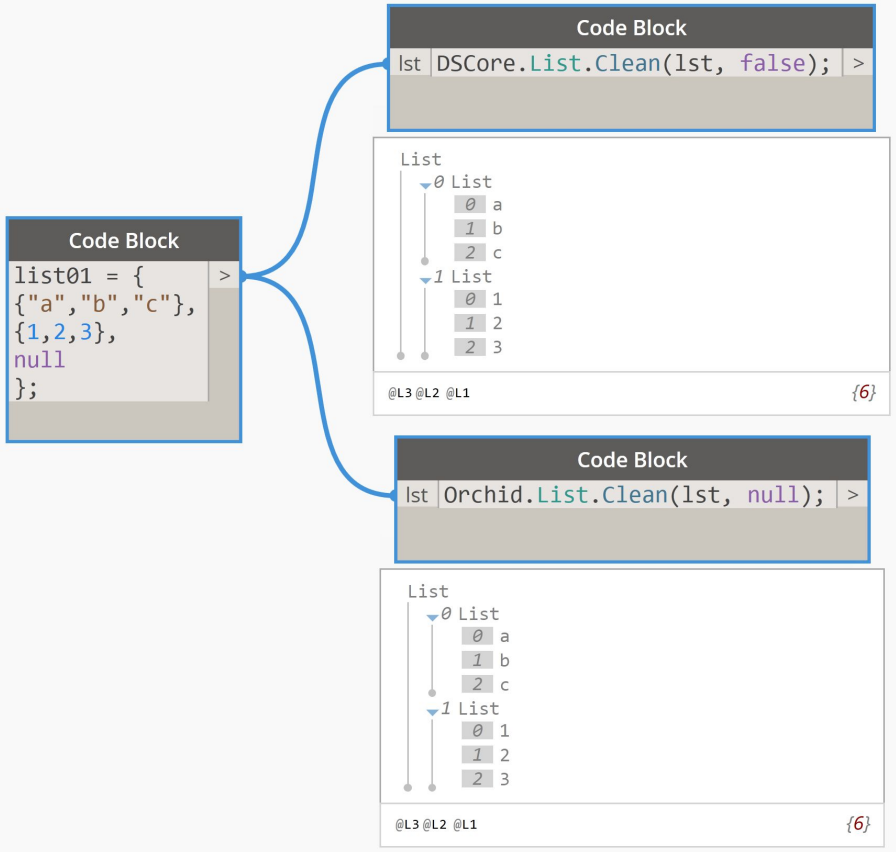
When you have custom packages installed (Zero-touch) that contain a **Class Namespace** that is the same as an out-of-the-box version then you may have to specify the full class path to correctly initialise your chosen function. If you are using a custom package that contains its own *'List'* class, then using the *Clean* method inside of Dynamo could return a warning:

List.Clean(list01);

In this instance, we will have to specify the full Class path (Based on the .dll it comes from):

DSCore.List.Clean(list01);

DSCore will remain black rather than colour.
All tab-finished dot notation will finalise **DSCore.DateTime** as default (So you'll have to manually delete that portion).

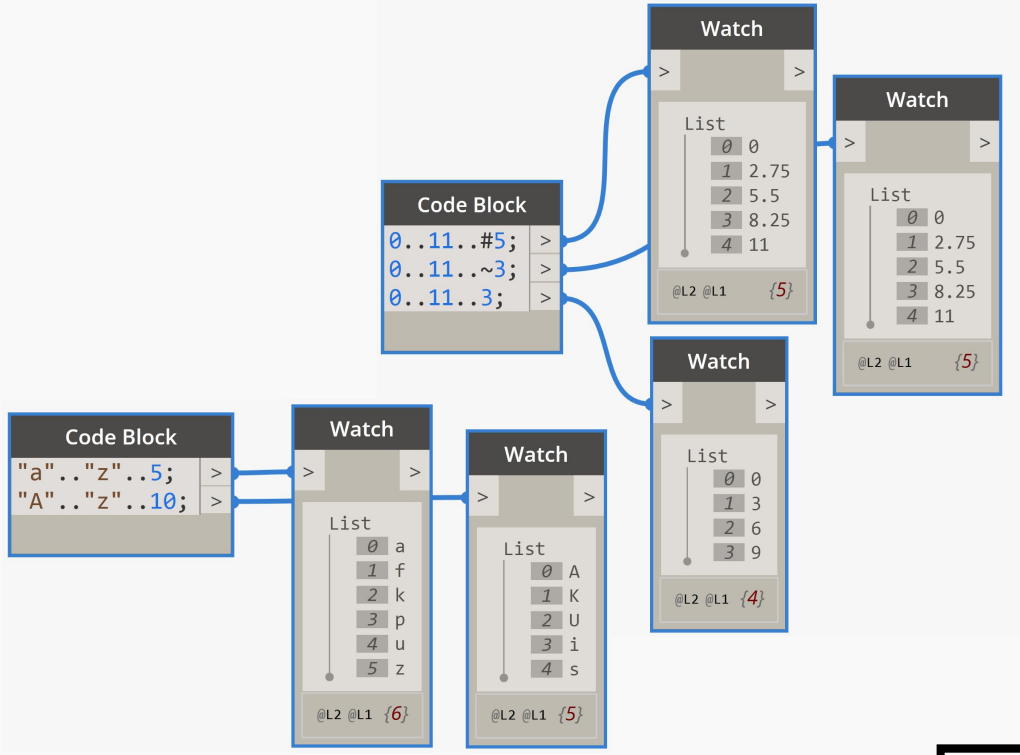


FEATURE
RANGE

A *range* in DesignScript is a series of numbers from a **start** to an **end** point with a designated **step** (Spacing typology) and is initialised in the following ways:

start..end..step;
start..end..#amount;
start..end..~approximate;

Ranges can be either **numeric** or **alphabetic**.
Alphabetic ranges vary depending on capitalisation.
The *start..end..#amount* range (E.g *0..1..#20*) is harder to reproduce in nodal form due to rounding - the DesignScript variant is swifter.
A range using the # (amount) spacing will always return the **start** value at index 0 and the **end** value at index -1 (last index).
If you omit the *step* there is a default value of **1** (*0..10* is the same as *0..10..1*).



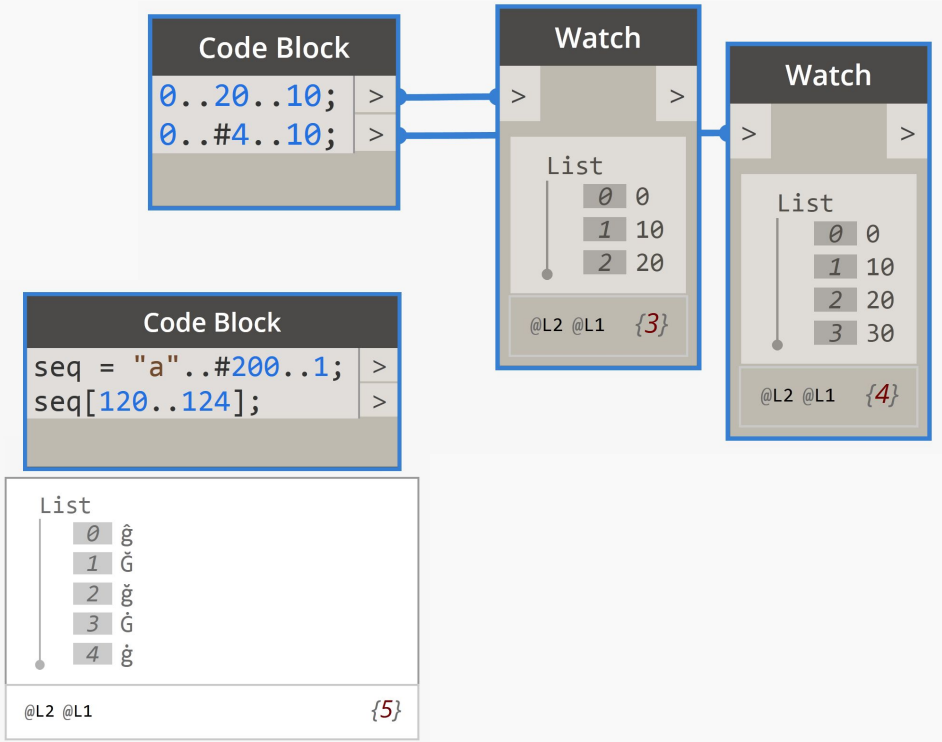
FEATURE
SEQUENCE

A *sequence* in DesignScript is a series of numbers from a **start** value, to a chosen **amount** and incremented by a **step**. Sequences are initialised in the following ways:

start..amount..step;
start..#amount..step;

Sequences can be either **numeric** or **alphabetic**.
Alphabetic sequences vary depending on capitalisation.
Special characters can be *found* in sequences.

Note: For sequences, we prefix the middle variable inside the initilisation syntax.
Sequences are not compatible with the tilde (~) approximation.



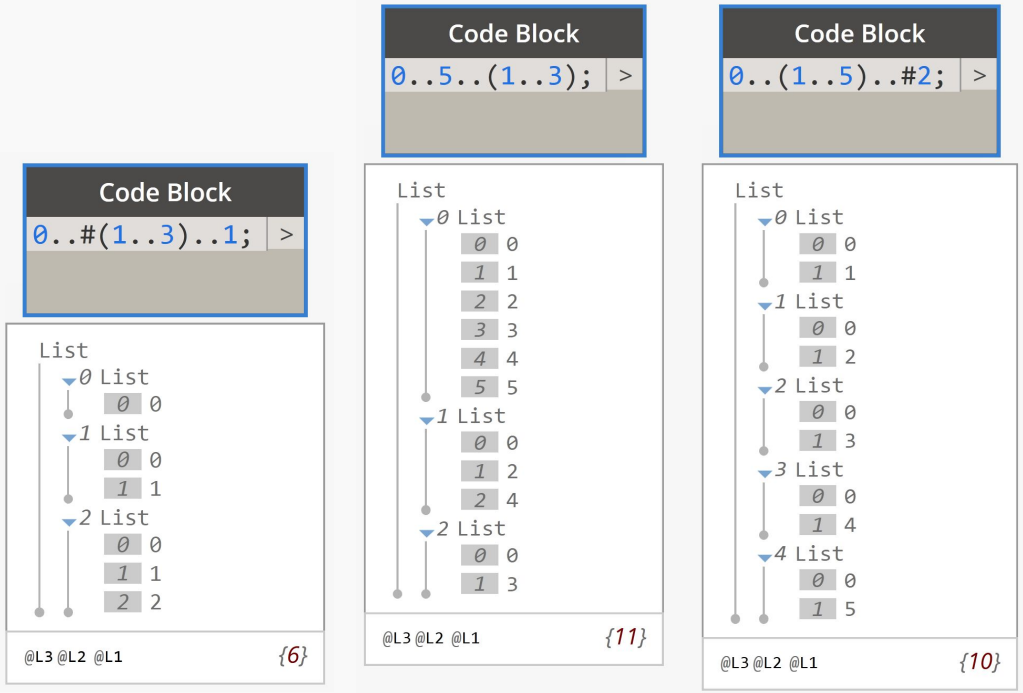
EXPLORE
NESTED RANGE & SEQUENCES

Let's explore the *range* or *sequence* features through nesting. You can put a **nested** range or sequences at any point of our DesignScript syntax with the exception of the **start** value:

```
0..5..(1..3);
0..(1..5)..#2;

0..#(1..3)..1;
```

Have a play with the *alphabetic* versions of ranges and sequences and check the preview bubble (Or use a Watch node) to see the results!



FEATURE
GET ITEM AT INDEX

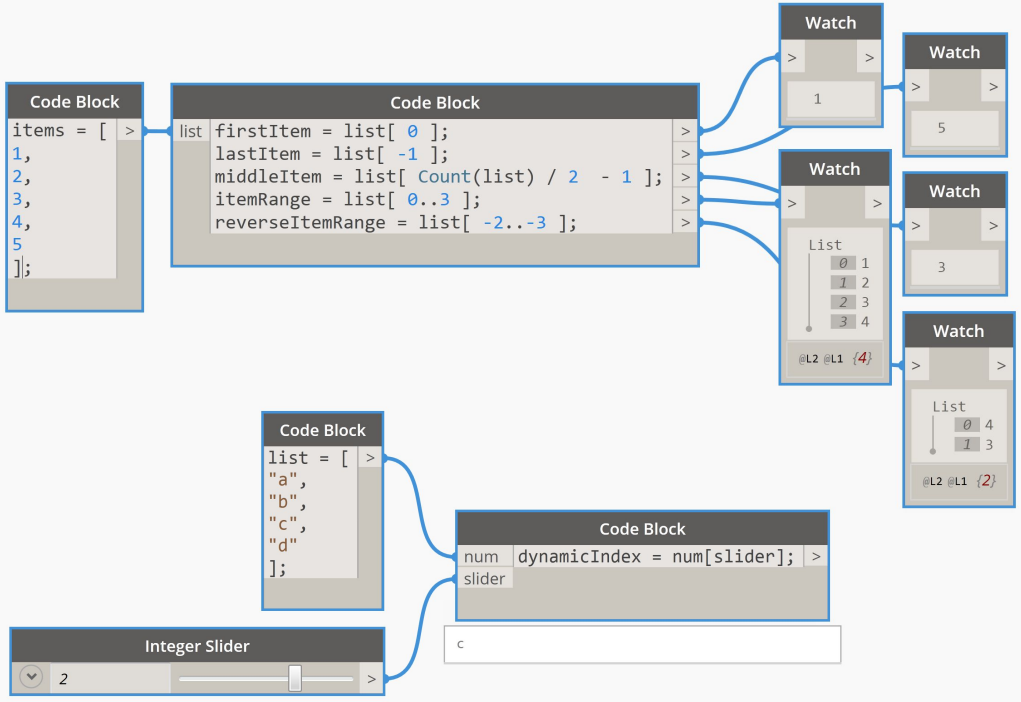
DesignScript has a textual version of the node **List.GetItemAtIndex(index)** and a shorthand version which is initialised by square brackets (**[ ]**) after a **list** with a number inside, such as **list[number]**:

```
firstItem = list[ 0 ];
lastItem = list[ -1 ];
*middleItem = list[ Count(list) / 2  - 1 ];
itemRange = list[ 0..3 ];
reverseItemRange = list[ -2..-3 ];
```

If you put a variable inside, you can attach an *integer slider* to the newly generated input port to dynamically get items at index.

```
dynamicIndex = list[ slider ];
```

Note: *Middle Item will only work if there is an odd number of items in your list.



EXPLORE
GET ITEM AT INDEX

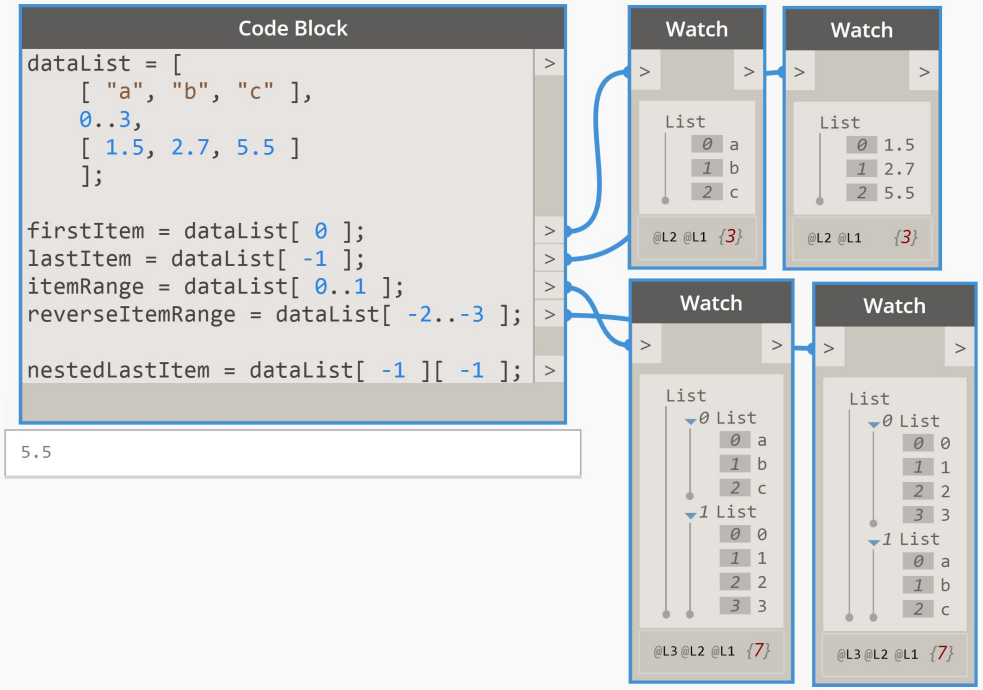
Let's explore the **Get Item At Index** shorthand features in DesignScript:

```
dataList = [
[ "a", "b", "c" ],
0..3,
[ 1.5, 2.7, 5.5 ]
];

firstItem = dataList[ 0 ];
lastItem = dataList[ -1 ];
itemRange = dataList[ 0..3 ];
reverseItemRange = dataList[ -2..-3 ];
```

To get items out of nested lists, you double-up (Or triple and so on) the Get Item At Index syntax:

```
nestedLastItem = dataList[ -1 ][ -1 ];
```



FEATURE
REPLICATION: BASICS

DesignScript uses **Replication** instead of Lacing which allows you to feed in a collection of objects instead of a single value to any input and specify how that data interacts.

Replication is initiated by an Integer inside angle brackets (**<#>**). These are called *replication guides*:

Point.ByCoordinates(x<1>, y<2>, z<3>);

Replication is *hierarchical,* not proximical. The numbers inside of the greater-than and less-than replication signs have a hierarchical relationship to each other when called inside the same function:

x<1>, y<2>, z<3> == x<1>, y<34>, z<120>

Note: The number hierarchy also stipulates what collection has dominance (i.e. governs the replication relationship).

Code Block

```
nums = 0..5000..#10;
noReplication = Point.ByCoordinates(nums, nums, 0);
replication = Point.ByCoordinates(nums<1>, nums<2>, 0);
```

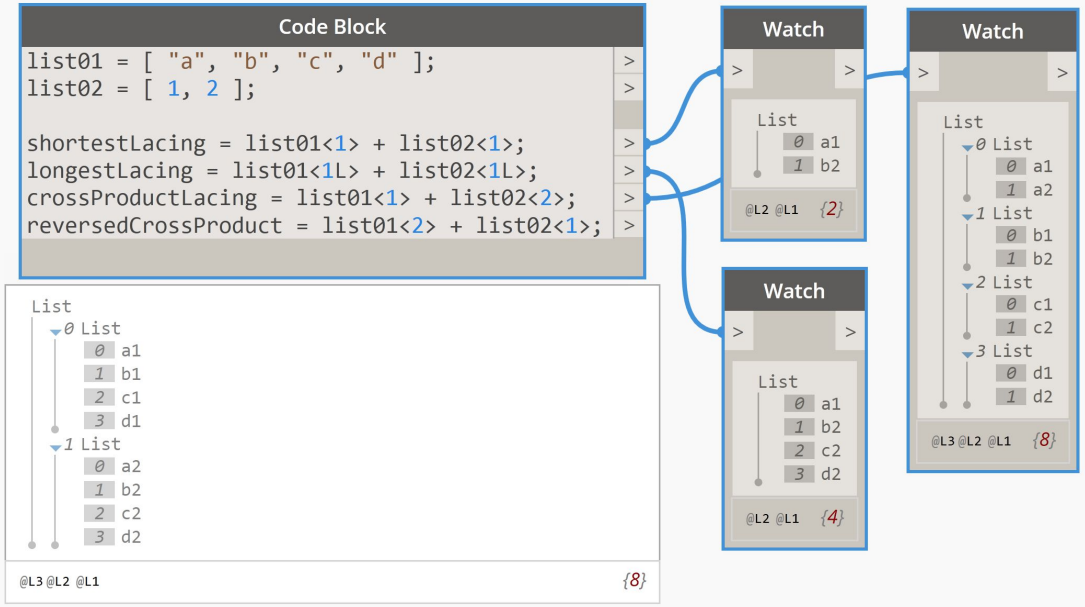
```
List
  0 List
    0 Point(X = 0.000, Y = 0.000, Z = 0.000)
    1 Point(X = 0.000, Y = 555.556, Z = 0.000)
    2 Point(X = 0.000, Y = 1111.111, Z = 0.000)
    3 Point(X = 0.000, Y = 1666.667, Z = 0.000)
    4 Point(X = 0.000, Y = 2222.222, Z = 0.000)
    5 Point(X = 0.000, Y = 2777.778, Z = 0.000)
    6 Point(X = 0.000, Y = 3333.333, Z = 0.000)
    7 Point(X = 0.000, Y = 3888.889, Z = 0.000)
    8 Point(X = 0.000, Y = 4444.444, Z = 0.000)
    9 Point(X = 0.000, Y = 5000.000, Z = 0.000)
  1 List
    0 Point(X = 555.556, Y = 0.000, Z = 0.000)
    1 Point(X = 555.556, Y = 555.556, Z = 0.000)
    2 Point(X = 555.556, Y = 1111.111, Z = 0.000)
    3 Point(X = 555.556, Y = 1666.667, Z = 0.000)
@L3 @L2 @L1                                        {100}
```

FEATURE
REPLICATION: LACING TYPES

Replication can cover all of the possibilities of data matching that *lacing* can in a Node based workflow. Replication is used to *control data matching* otherwise Dynamo approximates the matching using **auto lacing**:

list01 = [“a”, “b”, “c”, “d”];
list02 = [1, 2,];

shortestLacing = list01<1> + list02<1>;
longestLacing = list01<1L> + list02<1L>;
crossProductLacing = list01<1> + list02<2>;
reversedCrossProduct = list01<2> + list02<1>;



Note: If you have three lists you are **replicating** across, you’ll need to use three tiers of replication number to specify dominance (I.e. list01<1>, list02<2>, list03<3>) etc. If we switch the number dominance inside the replication guides, the data changes as shown in the preview bubble of the code block.

FEATURE
REPLICATION: LIST LEVELS

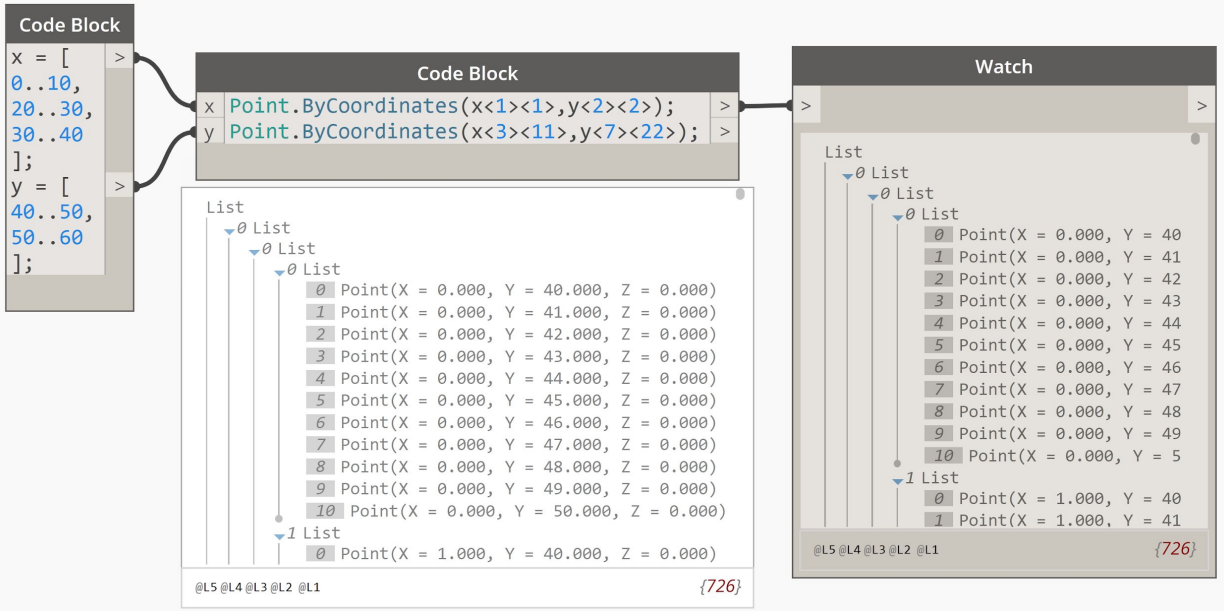
Replication can also be *chained* together to deal with multi-level data matching. Each *replication guide* is paired with its counterpart - starting from the outer list:

function(argument<1L><1> , argument<1L><2>);

Each replication guide that follows an argument indicates a level. Each *replication set* only matters hierarchically in relation to other members of that set.

Point.ByCoordinates(x<1><1>, y<2><2>); ==
Point.ByCoordinates(x<3><11> , y<7><22>);

Rank dominance of each multi-level data matching matters only in relation to each paired *replication set*.



EXPLORE
REPLICATION

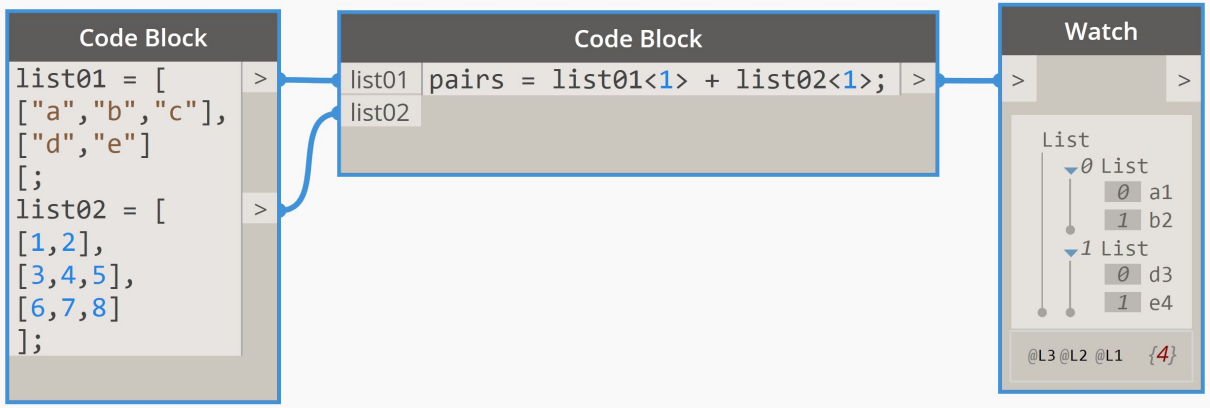
Let's explore *replication* with a simple use case - pairing alphabetic and numeric characters in non-homogenous lists:

list01 = [["a", "b" , "c"] , ["d", "e"]];
list02 = [[1, 2] , [3, 4, 5], [6, 7, 8]];

Explore the following in a new *Code Block*:

Shortest lacing: **list01<1> + list02<1>;**
Longest lacing: **list01<1L> + list02<1L>;**
Cross Product: **list01<1> + list02<2>;**
Multi-Tier: **list01<#><#> + list02<#><#>;**

Note: **list01<1> + list02<1>** is the same as **list01 + list02** is the same as **list01<2> + list02<2>** and so on (Any equal number across the same paired replication sets will function this way).



FEATURE
LIST @ LEVEL: CONCEPT

DesignScript uses **List@Level** to specify the level of list you want to work with right at the argument. List@Level is *NOT* the Rank of a list, but the *location* of a list. List@Level uses the following syntax:

List@Level: **@L#**
List@Level (Keeping List Structure): **@@L#**

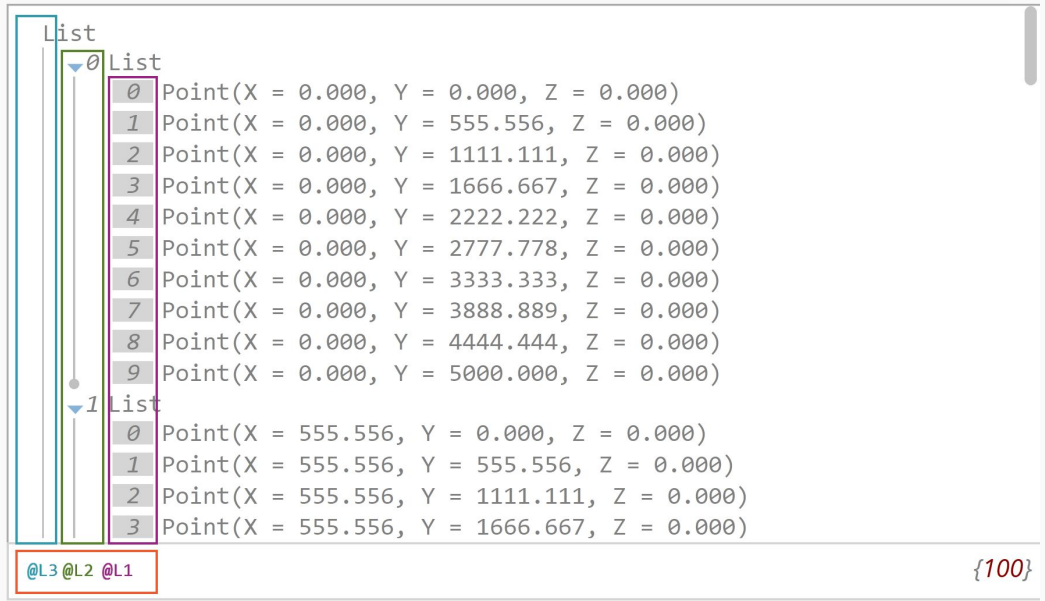
Point.ByCoordinates(x@L2, y@L2);

List@Level is in essence a for loop running across the chosen level.
List@Level can be used in place of *replication* (lacing) in most cases.
List@Level pairs data from a level to another chosen level (**level indicators** match List@Level syntax) .
The most useful level option is the same as the nodal default: **@L2**

Note: In the 1.X versions of Dynamo, the syntax for List@Level was: @**-#** using a negative (**-**) prefix in lieu of an uppercase **'L'**: E.G @**-2**.

Code Block

```
nums = 0..5000..#10;
noReplication = Point.ByCoordinates(nums, nums, 0);
replication = Point.ByCoordinates(nums<1>, nums<2>, 0);
```



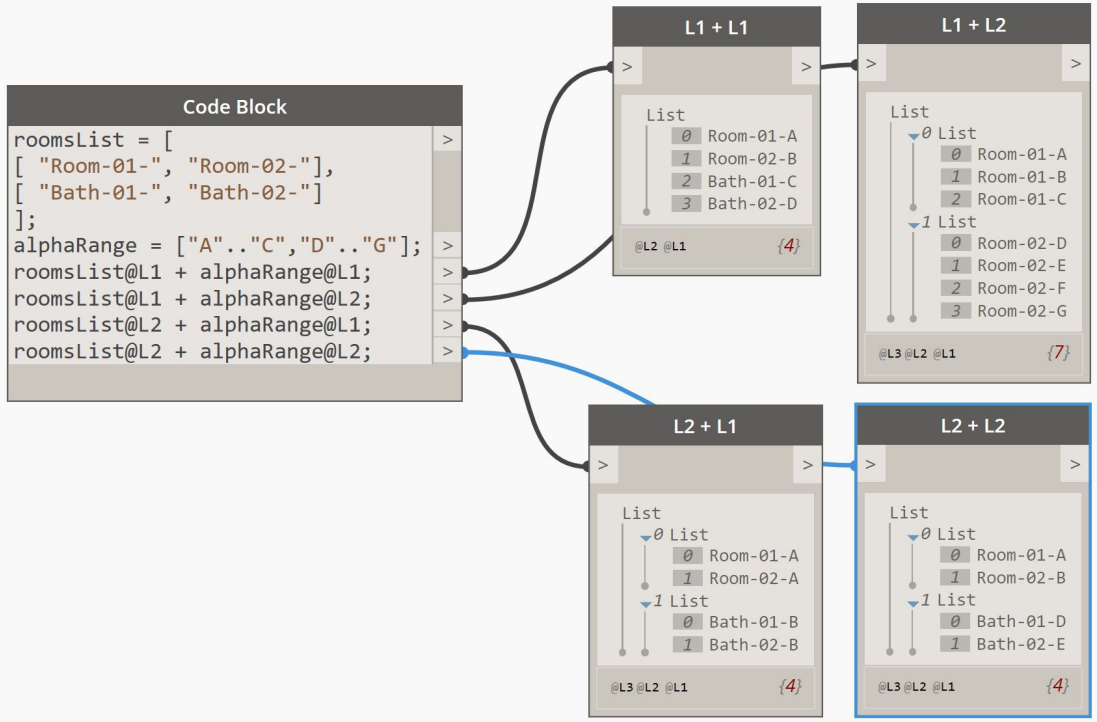
FEATURE
LIST @ LEVEL

List@Level can allow us to pair data in a logical format. In the visual example here we firstly pair the **roomsList** at *location level 1* with the **alphaRange** at *location level 1:*

roomsList@L1 + alphaRange@L1;

Notice how changing the **Level** that you are operating on changes the output values.

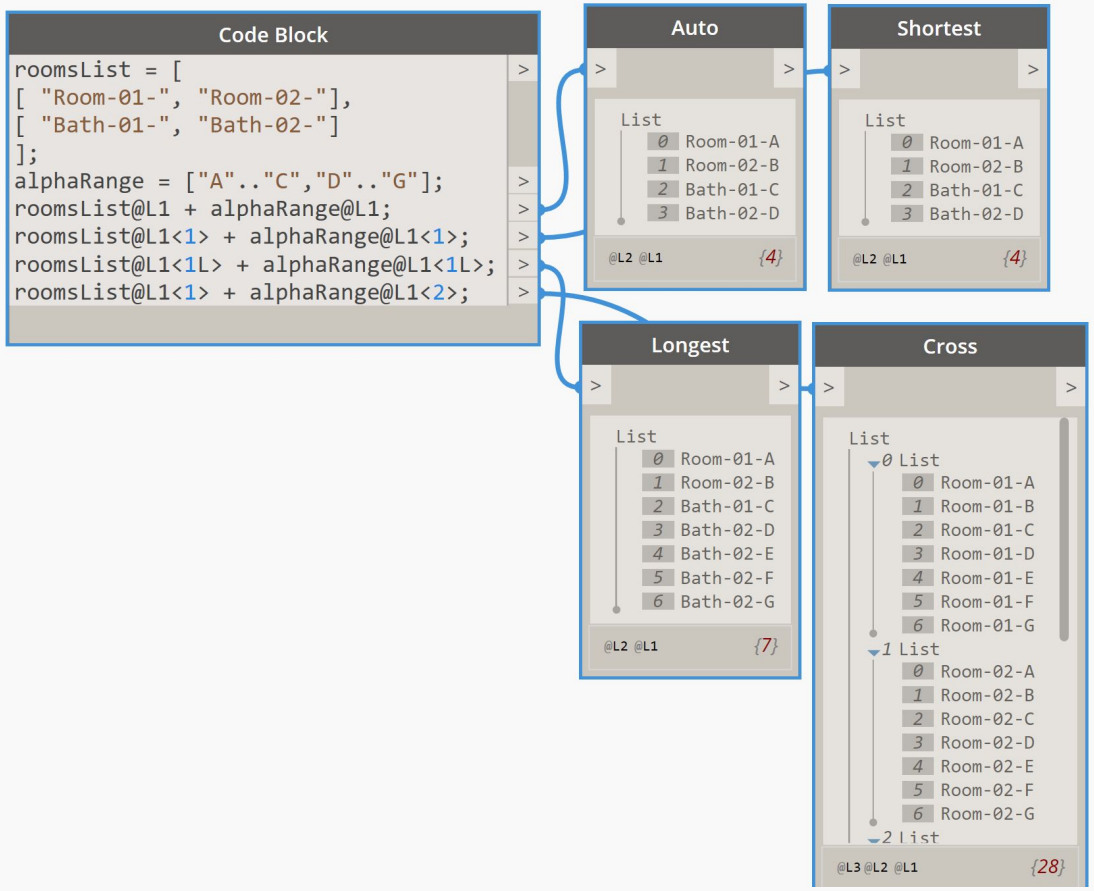
Note: All of these are being matched with **auto lacing**.



FEATURE
LIST @ LEVEL & REPLICATION

List@Level can allow us to pair data in a logical format. This logic can be manipulated further by appending *replication guides* to the data. In the visual example, as previous, we pair the **roomsList** with the **alphaRange** at various *location levels* to achieve different lacing formats:

autoLacing = roomsList@L1 + alphaRange@L1
shortestLacing = roomsList@L1<1> + alphaRange@L1<1>;
longestLacing = roomsList@L1<1L> + alphaRange@L1<1L>;
crossProductLacing = roomsList@L1<1> + alphaRange@L1<2>;



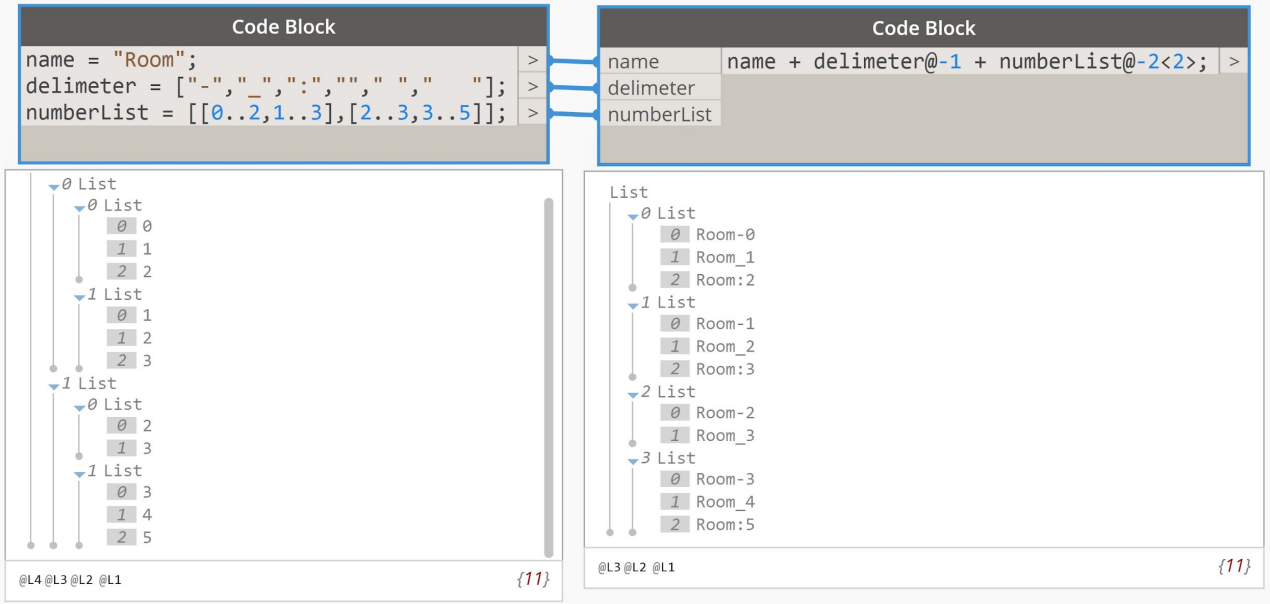
EXPLORE
LIST @ LEVEL

Let's explore **List@Level** by using 3x lists of data to pair (With various data counts).

name = **"Room"**;
delimeter = [**"-"**, **"_"** , **":"** , **""** , **" "** , **" "**];
numberList = [[**0..2**, **1..3**], [**2..3**, **3..5**]];

Pair them using any combination of **List@Level** and **Replication Guides** - or try omitting the replication guides altogether.

Remember you can also use the special **'L'** for *longest lacing* and check the preview bubble (Or use a Watch node) to see the results!



Power and Control

[Using conditions, flow control and looping]

CONCEPT
ASSOCIATIVE VS. IMPERATIVE

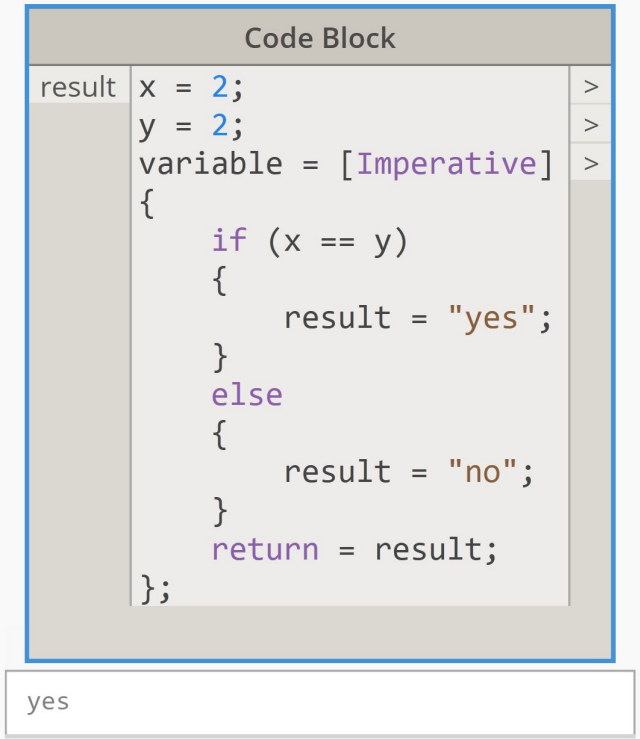
Associative programming uses the concept of *graph dependencies* to establish *'flow control.'* Associative is the default mode inside of Code Blocks.

Imperative programming is characterized by explicit *flow control* using *'For'* and *'While'* loops (for iteration) and *if/elseif/else* statements (for conditional statements). To initialise Imperative code you use the following syntax:

[Imperative] { code };

Imperative code is executed line by line - Associative code executes based on relationships. In basic terms, you use **Imperative** for conditional statements, looping and function passing.

Note; These two modes differ from **1.X** to **2.X** versions. Changes to 'upstream' variables are automatically propagated to 'downstream' variables in 1.X but you can no longer utilise the same variable name multiple times inside a **Code Block** in 2.X.



ISSUES

IMPERATIVE CODE

[Bug] **Imperative code** doesn't currently allow for *Namespace lookup* and will automatically generate input ports when a full path is given.

A workaround solution (Until this bug is fixed) is to define the function **outside** the Imperative Block (Refer to images).

The following Github Issues thread has context for the above issue: https://github.com/DynamoDS/Dynamo/issues/8796

Code Block

```
pnt = Autodesk.Point.ByCoordinates;
lne = Autodesk.Line.ByStartPointEndPoint;

geometry = [Imperative]
{
    x = 1;
    start = pnt(0,0,0);
    end = pnt(x,x,x);
    line = lne(start,end);

    while (line.Length < 10)
    {
        x = x + 1;
        end = pnt(x,x,x);
        line = lne(start,end);
    }
    return = line;
};
```

Code Block

Autodesk

```
geometry = [Imperative]
{
    x = 1;
    start = Autodesk.Point.ByCoordinates(0,0,0);
    end = Autodesk.Point.ByCoordinates(x, x, x);
    line = Autodesk.Line.ByStartPointEndPoint(start,end);

    while (line.Length < 10)
    {
        x = x + 1;
        end = Autodesk.Point.ByCoordinates(x,x,x);
        line = Autodesk.Line.ByStartPointEndPoint(start,end);
    }
    return = line;
};
```

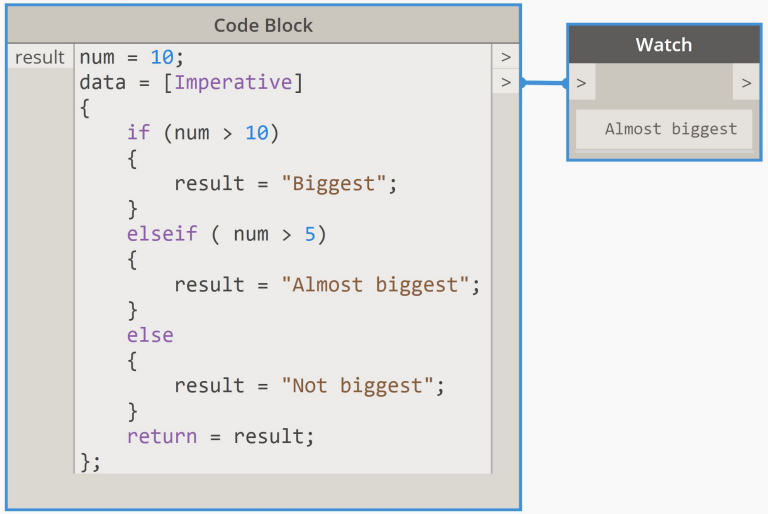
null

CONCEPT
CONDITIONAL STATEMENTS

DesignScript allows for **conditional statements**. These statements always evaluate to either *True* or *False (Booleans)*. *"If this is true, then that happens, otherwise something else happens"* - The resulting action of the statement is driven by the boolean value:

if(condition) { return something };
elseif (condition) { return something };
else { return something };

Conditional statements can be wrapped into *while loops* to continue execution until a certain condition is met before terminating.
Conditional statements can be used as a *'gate'* to allow a section of code to execute or not. Use **'If'** to specify a block of code to be executed if a specified condition is true. Use **'else'** to specify a block of code to be executed if the same condition is false. Use **'elseif'** to specify a new condition to test if the first condition is false.

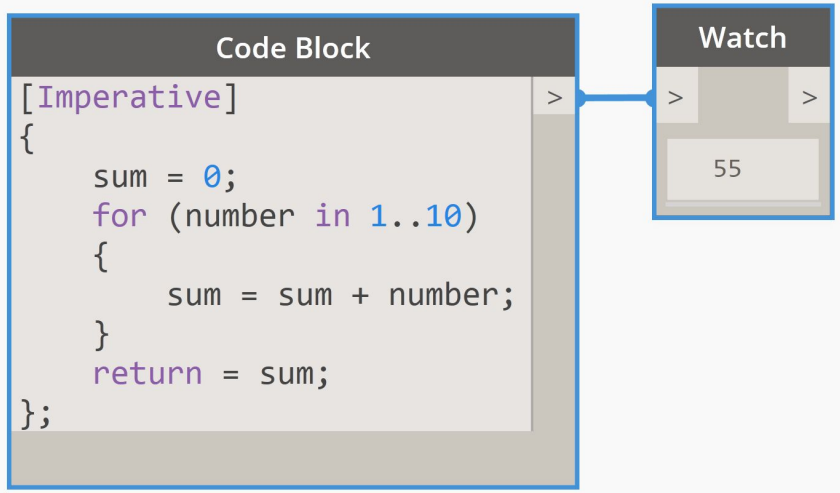


Note: Indentation is NOT required in DesignScript, but is advisable for clarity.

CONCEPT
FOR LOOP

A **For** loop is used to iterate over elements of a sequence (Run one at a time sequentially). It is typically used when you want to repeat a piece of code across all items of a list. In simple terms it works as: *"For all elements in a list, do this"*. Syntactically it is written in DesignScript as follows:

```
results = [ ];  // Create empty list to catch results
for ( thing in list )
{
        Do this to thing;
        Add new thing to results;
}
return = results;
```



Note: To use any looping, as mentioned above, you need to initialise an **Imperative block.** A For loop will utilise reserved keywords liberally to iterate, stop and continue looping processes.

EXPLORE
FOR LOOP

Let's explore a **For loop** using the Modulus operator to create a *Mask*. If a number is divisible by 3 (Has no remainder), then pass out a *true*, else pass out a *false*:

We are using a particular method of populating our *result* list - by specifying the exact index in that list we want to add a new item. Syntactically in DesignScript this is written as follows:

list[index] = itemToPopulate;

We declare a results list to populate by specifying an empty list using square braces (**[ ]**).

Code Block

```
num = 5;

loop = [Imperative]
{
    result = [];
    for (i in 0..num)
    {
        if (i % 3 == 0)
        {
            result[i] = true;
        }
        else
        {
            result[i] = false;
        }
    }
    return = result;
};
```

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

CONCEPT
WHILE LOOP

A **While** loop will tell the computer to do run a section of code while a **condition** evaluates to True (Running the same code again and again) and will terminate and return a result as soon as that condition evaluates to False. A **while** loop consists of a block of code and a condition. Syntactically it is written in DesignScript as follows:

```
results = [ ];  // Create empty list to catch results
while ( condition exists )
{
        Do this to thing;
        Add new thing to results;
}
return = newList;
```

Note: Be VERY careful when using **while** loops that you don't run into *infinite loops*. When you create an *infinite loop* where a condition can never be met - it will run the same thing over and over for eternity (In essence you'll crash your Dynamo session and grey-screen) .

Code Block

```
minArea = 100;
maxArea = 150;

[Imperative]
{
    randNum = Math.Rand();
    newArea = randNum * 200;

    while (newArea > maxArea || newArea < minArea)
    {
        newRandNum = Math.Rand();
        newArea = newRandNum * 200;
    }

    return = Math.Round(newArea);
};
```

147

EXPLORE
WHILE LOOP

Lets explore our Minimum and Maximum gated Area **While** loop. Write the following loop in a new **Code Block**.

We explore the use of conditionals **(if | elseif | else)** and a while loop that is constrained by an OR (**||**) conditional statement.

In this situation we're passing out three possible results; If our Area matches our minimum acceptable value we return a string of *'Minimum'*, if our Area matches our maximum acceptable value we return a string of *'Maximum'* and if it sits anywhere else in between our demarcated range we return the actual Area value.

Code Block

```
minArea = 100;
maxArea = 110;

[Imperative]
{
    randNum = Math.Rand();
    newArea = randNum * 200;

    while (newArea > maxArea || newArea < minArea)
    {
        newRandNum = Math.Rand();
        newArea = Math.Round(newRandNum * 200);
    }

    if (newArea == minArea)
    {
        return = "Minimum";
    }
    elseif (newArea == maxArea)
    {
        return = "Maximum";
    }
    else
    {
        return = newArea;
    }
};
```

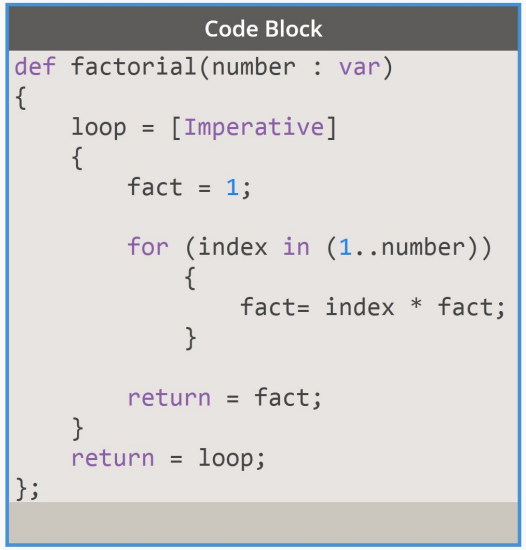
```
104
```

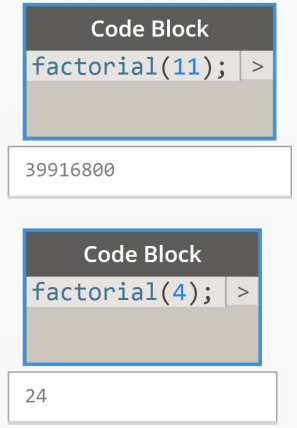
CONCEPT
CUSTOM DEFINITION: BASICS

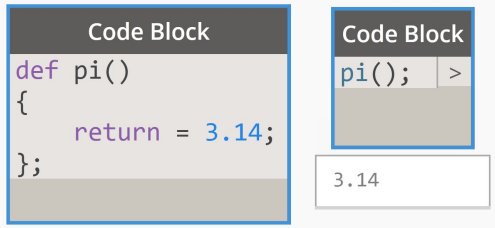
DesignScript allows you to create **custom definitions**. These definitions are the equivalent of *custom nodes* in Dynamo that wrap up many other nodes. In simple terms, they will call a *function* that contains *arguments* that are manipulated by code inside the definition and then the definition *returns* as a result.

```
def myCustomDefinition( argument1, argument2, argumentN )
    {
      code goes here;
      return = result;
    };
```

Custom functions will show up in the auto-complete feature when you begin typing.
Custom functions use parentheses to demarcate arguments and curly braces to contain all definition code.
Custom functions can have no arguments and return a static value.
To return multiple values you need to return a list.
Definitions can be called inside new **Code Blocks.**







CONCEPT
CUSTOM DEFINITION: INPUTS

Custom Definitions typically must specify the *type* of argument and the *dimensionality.* The type of argument is defined after a colon (**:**) and comprise of the following:

int | double | bool | string | var | object

The dimensionality of arguments are defined by square braces and use the following syntax:

```
singleItem = def myDef( var );
list = def myDef( var : [ ] );
listOfLists = def myDef( var : [ ][ ] );
arbitraryRank = def myDef( var : [ ]..[ ] );
```

Var is a generic / arbitrary data type that will accept anything (Including lists comprised of various data types).

Code Block

```
def timesFour( num : int[]..[])
{
    return = num * 4;
};
```

Code Block

```
range = 0..10;
timesFour(range);
```

```
List
  0 0
  1 4
  2 8
  3 12
  4 16
  5 20
  6 24
  7 28
  8 32
  9 36
  10 40
```

@L2 @L1 {11}

Code Block

```
timesFour(2);
```

```
8
```

Warning: One or more of the input types are not matching. Couldn't find a version of timesFour that takes arguments of type (string)

Code Block

```
timesFour("hello");
```

```
null
```

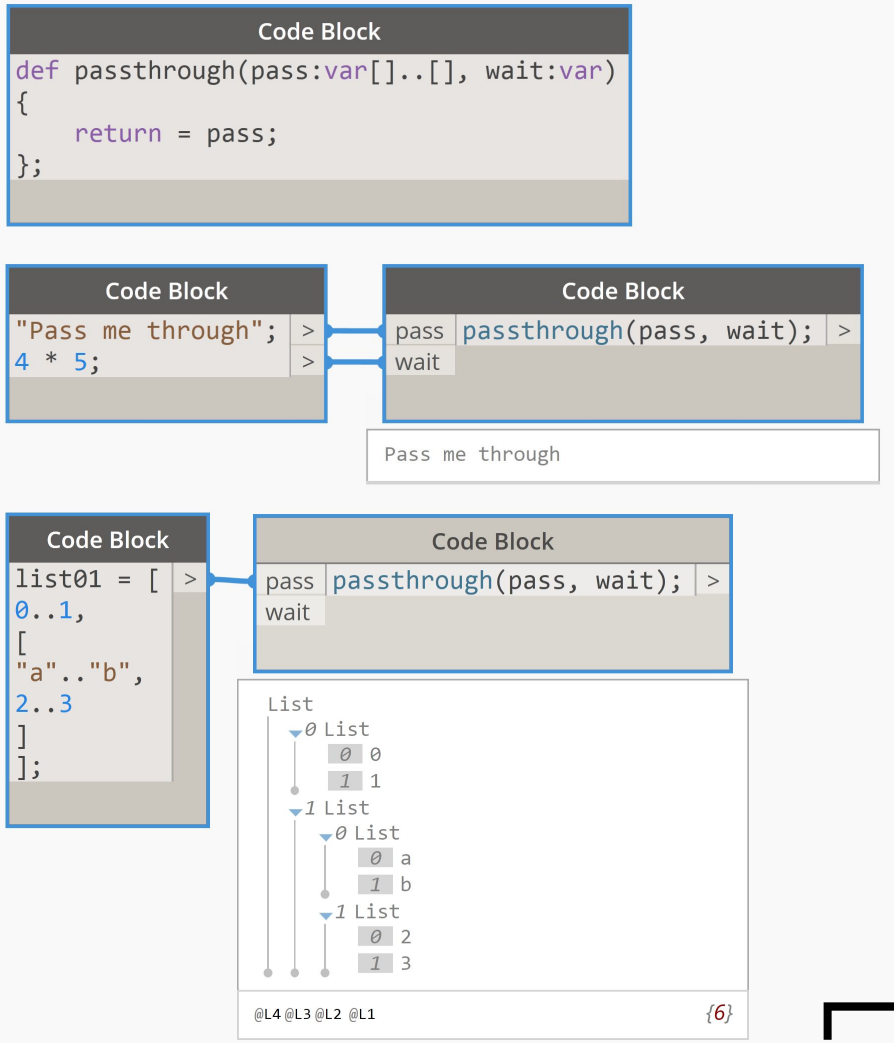
EXPLORE
CUSTOM DEFINITION

Let's explore **typology** and **dimensionality** inside a *Custom Definition.* Choose a *typology* from the following list (**int** | **double** | **bool** | **string** | **var** | **object**) and explore all *dimensionality* options (**var** | **var[ ]** | **var[ ][ ]** | **var[ ]..[ ]**):

```
def passthrough( pass : var[ ]..[ ], wait : var )
{
   return = pass;
};
```

What happens when you change the dimensionality? Try feeding in single objects, lists and multi-tier lists and check the preview bubble (Or use a Watch node) to see the results!

Thanks to **Andrea's Dieckmann** for the *Passthrough* inspiration.



ISSUES
CUSTOM DEFINITIONS

[Bug] Definitions are **global** until the Dynamo session is closed. This means that deleted functions can still appear in auto-complete. That definition can be called in any workspace until the Dynamo session ends.

[Bug] Overloading functions (Creating definitions that are of the same name with different arguments) behave weirdly and should be avoided.

[WIP] Do NOT use **List@Level** or **replication** in Imperative code blocks as the DesignScript engine will not execute that code.

Wrapping it all Together

[Let's explore a complex example!]

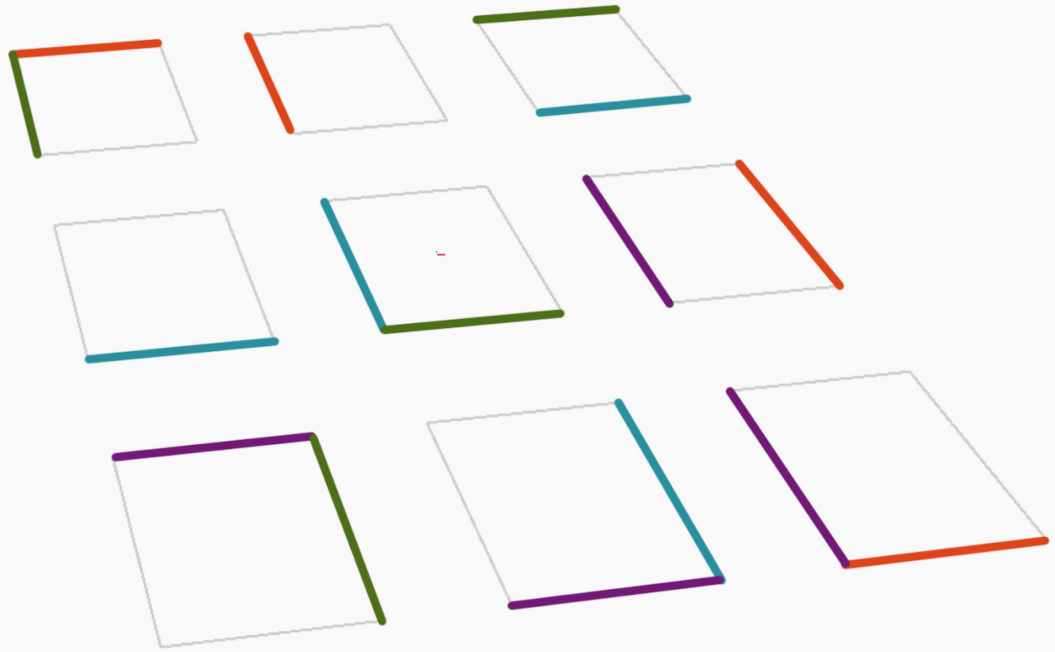
EXPLORE
GROUPED CURVES: PROBLEM

Let's explore the creation of a **Grouped Curves definition.** This definition will take a list of unordered curves (Must be touching) and group them together into grouped loops.

In this example we will take a series of *9 Exploded Rectangles* and shuffle the resultant curves. This gives us *36 individual curves* of which to group correctly back into their respective rectangles.

A few dispersed curves are showcased in the image to the right (If Shuffled List was Chopped into sets of 4x).

A big shout-out to **Vikram Subbaiah** for allowing me to use his *DesignScript code*: https://forum.dynamobim.com/t/grouping-of-curves-with-design-script/15122



EXPLORE

GROUPED CURVES: RECTANGLES

Type the following DesignScript code inside a new **Code Block** to generate our Rectangles:

```
location = ( -2000..2000..#3 );
csSpread = CoordinateSystem.ByOrigin( location<1>, location<2> );
rec = Rectangle.ByWidthLength( csSpread, 1250, 1250 );
allCurves = Flatten( rec.Explode() );
shuffledList = List.Shuffle( allCurves );
grey = Color.ByARGB( 255, 205, 205, 205 );
orange = Color.ByARGB( 255, 230, 146, 0 );
shuffledCurves = shuffledList[ 0..3 ];
restOfCurves = shuffledList[ 4..Count(shuffledList) - 1 ];
greyDisplay = Display.ByGeometryColor( restOfCurves, grey );
orangeDisplay = Display.ByGeometryColor( shuffledCurves, orange );
```

Code Block

```
// Creating a Location spread ( Numbers for X / Y )
location = (-2000..2000..#3);
// Creating a Coordinate System distribution grid
csSpread = CoordinateSystem.ByOrigin(location<1>, location<2>);
// Generating Rectangles from the Coordinate System
rec = Rectangle.ByWidthLength(csSpread, 1250, 1250);

// Exploding the rectangles to get all curves and Flattening
allCurves = Flatten(rec.Explode());
// Shuffling our exploded list to jumble all curves
shuffledList = List.Shuffle(allCurves);

// Colours
grey = Color.ByARGB(255, 205, 205, 205);
orange = Color.ByARGB(255, 230, 146, 0 );

// First 4x curves
shuffledCurves = shuffledList[0..3];
// All curves except first 4x
restOfCurves = shuffledList[4..Count(shuffledList) - 1];
// Colour display of Curves
greyDisplay = Display.ByGeometryColor(restOfCurves, grey);
orangeDisplay = Display.ByGeometryColor(shuffledCurves, orange);
```

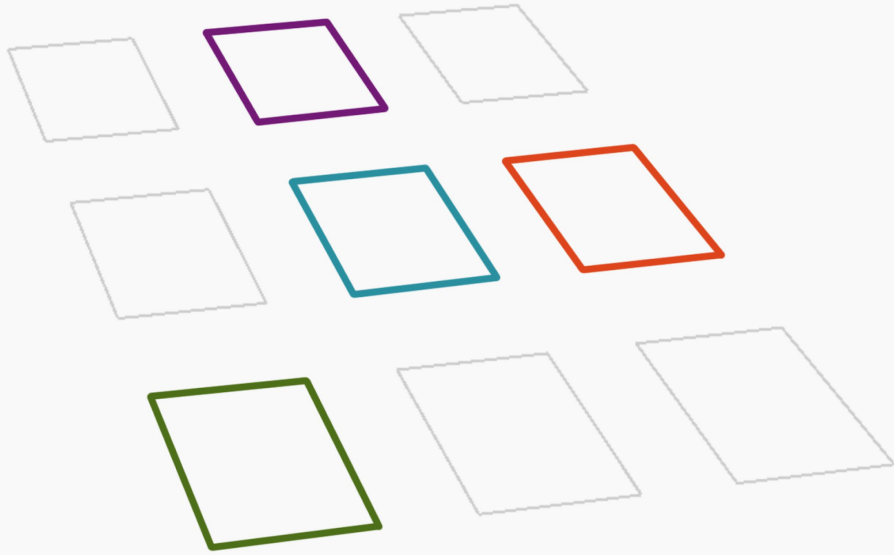
Note: We have built in visualisation into this script. Simply add a new code block with any variable inside and plug in the **greyDisplay** and **orangeDisplay** output ports to visualise.

EXPLORE
GROUPED CURVES: THEORY

The **Grouped Curves** definitions will query the Start and End points of each individual curve and find the *indices* of any other curve that has a coincident start or end point.

If the curves pass this test - then they are grouped together into a new sub-list.

A singular list of curves will be split into 9x sublists containing 4x curves that each make up a Rectangle when joined back into a PolyCurve.



EXPLORE

GROUPED CURVES: DEFINITION #1

We need to generate 2x custom definitions in order to successfully group our curves. The first definition defines the Group Indices and is written as follows:

```
def grpIndx( ind : var[ ]..[ ] )
  {
    indiceOccurances = List.SetIntersection( ind<1>, ind<2> );
    indiceCount = List.Count( inindiceOccurances<1><2> ) > 0;
    indiceFilter = List.FilterByBoolMask( ind, indiceCount<1> );
    indiceFirstItems = List.FirstItem( indiceFilter<1> );
    sublistUniqueIndices = List.UniqueItems( List.Flatten
    ( indiceFirstItems<1>, - 1 )<1> );
    sortedUniqueIndices = List.UniqueItems( List.Sort
    ( sublistUniqueIndices<1> ) );

    return = sortedUniqueIndices;
  };
```

Code Block

```
def grpIndx(ind:var[]..[])
{
    // Collect multiple-occurances of Indices only with Set-Intersection
    indiceOccurances = List.SetIntersection(ind<1>, ind<2>);
    // Create a Mask that is 'True' if the Count is greater than zero
    indiceCount = List.Count(indiceOccurances<1><2>) > 0;
    // Filter our Indices list
    indiceFilter = List.FilterByBoolMask(ind, indiceCount<1>);
    // Get our First item of every sub-list
    indiceFirstItems = List.FirstItem(indiceFilter<1>);
    // Get the Unique Indices from a Flat indices list in each sub-list
    sublistUniqueIndices = List.UniqueItems(List.Flatten
    (indiceFirstItems<1>, -1)<1>);
    // Get the Unique Indices from sorted sub-lists
    sortedUniqueIndices = List.UniqueItems(List.Sort(sublistUniqueIndices<1>));
    return = sortedUniqueIndices;
};
```

EXPLORE

GROUPED CURVES: DEFINITION #2

```
def grpCrvs(crv:var[]..[])
{
	geometryDistance = (crv.StartPoint)<1>.DistanceTo(crv<2>) == 0
	|| (crv.EndPoint)<1>.DistanceTo(crv<2>) == 0;
	allIndices = List.AllIndicesOf(geometryDistance<1>, true);
	checkBool = true;
	curveLoop = [Imperative]
	{
		while (checkBool)
		{
			indiceCount1 = List.Count(allIndices);
			allIndices = grpIndx(allIndices);
			indiceCount2 = List.Count(allIndices);
			checkBool = indiceCount2 != indiceCount1;
		}
	    return = allIndices;
	}
	groupedCurves = List.GetItemAtIndex(crv, curveLoop);
	return = groupedCurves;
};
```

Code Block

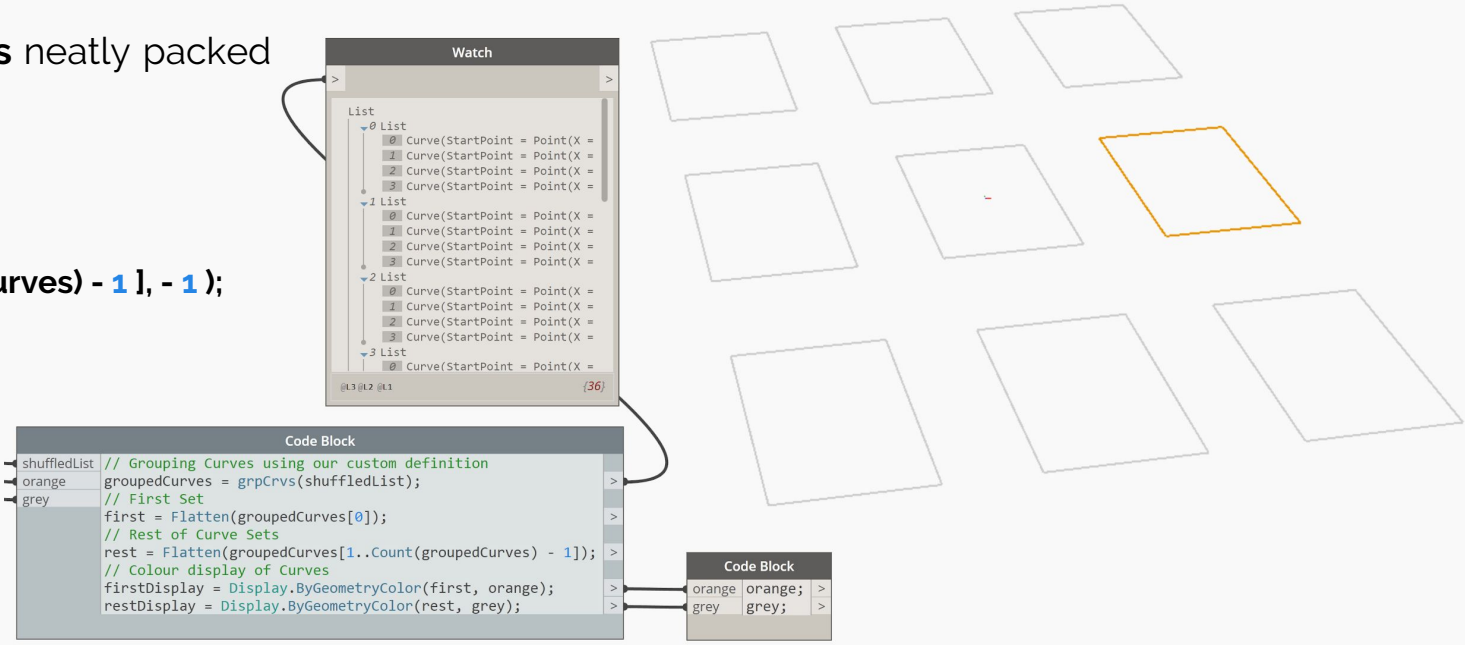
```
def grpCrvs(crv:var[]..[])
{
    // Check if the Distance from each Curve StartPoint is zero OR
    // each Curve EndPoint is zero and create a Mask
    geometryDistance = (crv.StartPoint)<1>.DistanceTo(crv<2>) == 0
    || (crv.EndPoint)<1>.DistanceTo(crv<2>) == 0;
    // Get all Indices of 'true'
    allIndices = List.AllIndicesOf(geometryDistance<1>, true);
    // Set a check boolean variable to 'true' as default
    checkBool = true;
    // Create an Imperative Loop attached to the variable 'crvLoop'
    curveLoop = [Imperative]
    {
        // Create a While loop that will run until our checkBool is set
        // to 'false'
        while (checkBool)
        {
            // Count all indices
            indiceCount1 = List.Count(allIndices);
            // Group indices using our second Custom Function 'grpIndx'
            allIndices = grpIndx(allIndices);
            // Count our gouped indices list
            indiceCount2 = List.Count(allIndices);
            // If our counted lists do not match, then equal True to
            // continue our While loop
            checkBool = indiceCount2 != indiceCount1;
        }
        // From our While loop, return our 'allIndices' which resets the
        // While loop for the next curve
        return = allIndices;
    }
    // Then we want to group our curves based on indices thrown out from
    // our loop. We achieve this with 'GetItemAtIndex'
    groupedCurves = List.GetItemAtIndex(crv, curveLoop);
    // Finally we return our grouped curves from the entire definition
    return = groupedCurves;
};
```

EXPLORE
GROUPED CURVES: COMPLETE

Wrapping it all together, we get our **Grouped Curves** neatly packed into sub-lists of curve loops.

```
groupedCurves = grpCrvs( shuffledList );
first = List.Flatten( groupedCurves[ 0 ], - 1 ) ;
rest = List.Flatten(groupedCurves[ 1..Count(groupedCurves) - 1 ], - 1 );
firstDisplay = Display.ByGeometryColor( first, orange );
restDisplay = Display.ByGeometryColor( rest, grey );
```

Note: You can wrap this code up into the same Code Block as the previous **Rectangles** code block to stop the creation of more input ports. As all code executed inside the same Code Block will show up in the background preview - we will turn off *Preview* mode and use other Code Blocks to only show a desired result.



Any Questions ?

REFERENCES
STANDING ON THE SHOULDERS OF GIANTS

- http://primer.dynamobim.org/en/07_Code-Block/7_Code-Blocks-and-Design-Script.html
- http://dynamobim.org/wp-content/links/DesignScriptGuide.pdf
- http://dynamobim.org/wp-content/links/DesignScriptDocumentation.pdf
- https://github.com/DynamoDS/DesignScript/blob/master/LanguageSpec.md
- http://designscript.io/
- http://designscript.io/DesignScript_user_manual_0.1.pdf

REMINDER:

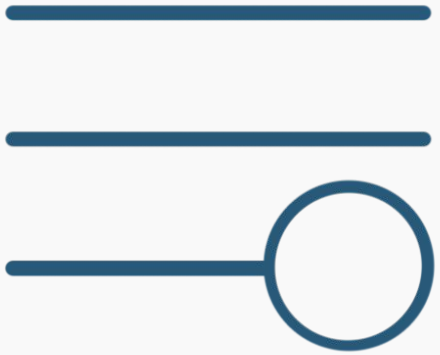
Speaker Feedback is appreciated

Fill in the Survey on the Mobile App

GR CENTRE

LJUBLJANA

11 – 13 OCT 2018



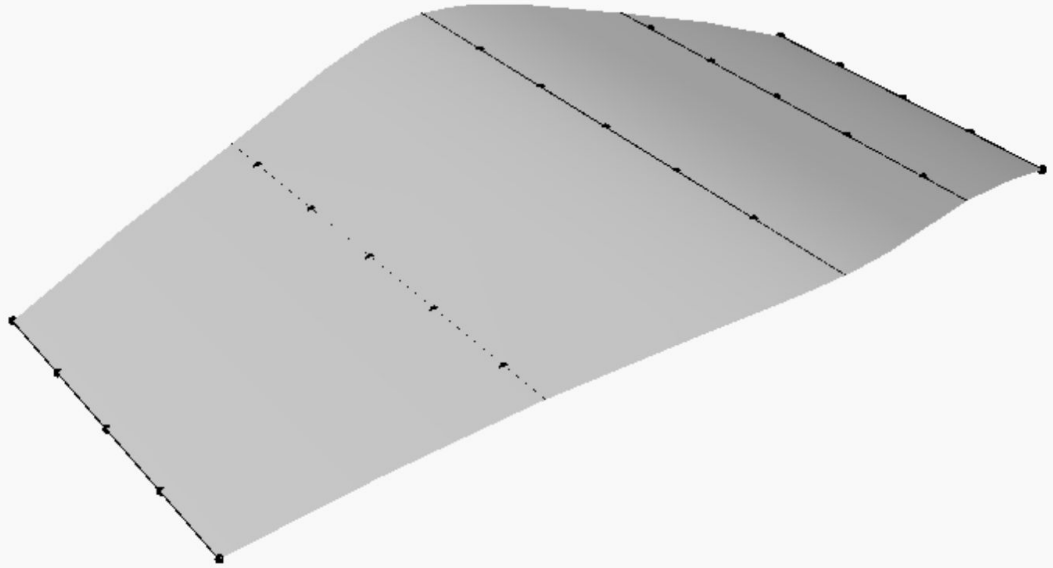
Addendum

[Another example if time permits]

EXPLORE
STEPPED LIST: SURFACE

Let's explore the creation of a **Stepped List definition**. In order to do so, we want to generate a Surface over which to populate points. Type the following inside a new **Code Block**:

```
//Creating a base surface to apply our custom definition to
x = -50000..50000..#5;
y = -15000..15000..#5;
z = [ 0, 7500, 12000, 7500, 0 ];
pnts = Point.ByCoordinates( x<1>, y<2>, z<1> );
crv = PolyCurve.ByPoints( pnts );
num = [ 0, 4000, 8000, 4000, 0 ];
extS = Curve.ExtendStart( crv, num );
extE = Curve.ExtendEnd( extS, num );
srf = Surface.ByLoft( extE );
```



This will result in a Surface (**srf**) as showcased in the image.

EXPLORE
STEPPED LIST: DEFINITION

To create the **Stepped List Definition** we will input the following code in a fresh **Code Block**:

```
def steppedList( firstNumber : var[ ]..[ ], secondNumber : var[ ]..[ ], step : var[ ] )
{
modifiedStep = ( secondNumber - firstNumber ) * step;

return = firstNumber + modifiedStep;
};
```

Shoutout: Unknown author on the dynamobim.org forums from 2014.

Code Block

```
/*
Author: Unknown
Location: forum.dynamobim.com

A shout-out to the unknown author of this custom definition (Found on the forums
in 2014)
*/
def steppedList(firstNumber : var[]..[], secondNumber : var[]..[], step : var[])
{
modifiedStep = (secondNumber - firstNumber) * step;

return = firstNumber + modifiedStep;
};
```

EXPLORE
STEPPED LIST: APPLICATION

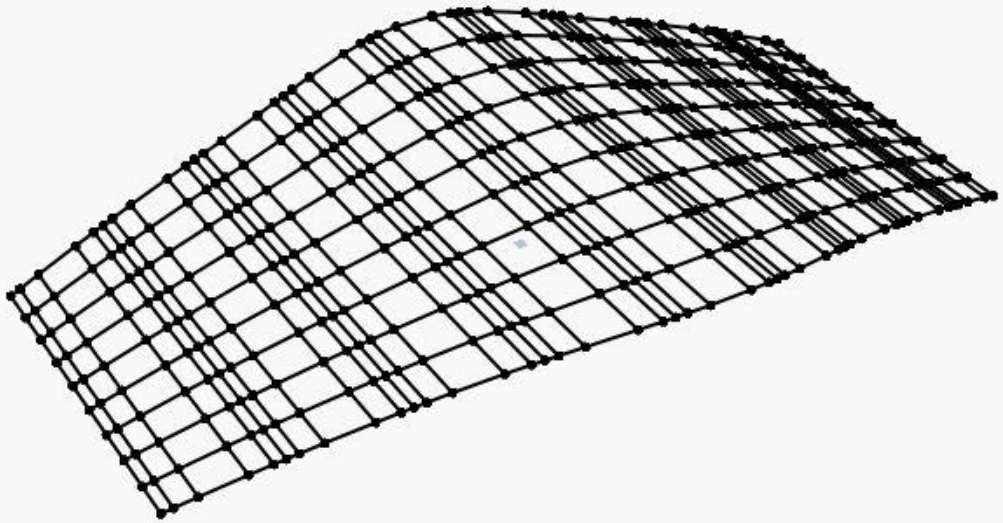
```
//Number range from '0' to '1', equally spaced 'divNumber' times
baseDivision = 0..1..#divNumber;
//All numbers in range except the last
uDivStart = 0..( divNumber - 2 );
//All numbers in range except the first
uDivEnd = 1..( divNumber - 1 );
//Running our custom definition
uDivision = steppedList( baseDivision[ uDivStart ]<1>,
baseDivision[ uDivEnd ]<1>, step );
//Flattend 'uDivision' definition results
resultUDiv = DSCore.List.Flatten( uDivision, - 1);
vDiv = 0..1..#6;
// Surface UV Points
srfPnts = Surface.PointAtParameter( srf<1>, vDiv<2>, resultUDiv<3> );
//Lateral PolyCurves
lateralCurves = PolyCurve.ByPoints( srfPnts );
//Transverse PolyCurves
transverseCurves = PolyCurve.ByPoints( DSCore.List.Transpose( srfPnts ) );
```

Code Block

```
srf //Inputs
    divNumber = 10;
    step = [0, 0.1, 0.3, 0.55, 0.7, 0.9];

    //Number range from '0' to '1', equally spaced 'divNumber' times
    baseDivision = 0..1..#divNumber;
    //All numbers in range except the last
    uDivStart = 0..(divNumber - 2);
    //All numbers in range except the first
    uDivEnd = 1..(divNumber - 1);
    //Running our custom definition
    uDivision = steppedList(baseDivision[uDivStart]<1>,
    baseDivision[uDivEnd]<1>, step);
    //Flattend 'uDivision' definition results
    resultUDiv = DSCore.List.Flatten(uDivision, - 1);
    vDiv = 0..1..#6;
    // Surface UV Points
    srfPnts = Surface.PointAtParameter(srf<1>, vDiv<2>, resultUDiv<3>);
    //Lateral PolyCurves
    lateralCurves = PolyCurve.ByPoints(srfPnts);
    //Transverse PolyCurves
    transverseCurves = PolyCurve.ByPoints(DSCore.List.Transpose(srfPnts));
```

EXPLORE
STEPPED LIST: RESULTS



Code Block

```
def steppedList( firstNumber : var[]..[], secondNumber : var[]..[], step : var[])
{
modifiedStep = ( secondNumber - firstNumber ) * step;
return = firstNumber + modifiedStep;
};
```

Code Block

```
divNumber = 10;
step = [0, 0.1, 0.3, 0.7, 0.9];
```

Code Block

```
//Creating a base surface to apply our custom defintion to
x = -5000..5000..#5;
y = -1500..1500..#5;
z = [0, 750, 1200, 750, 0];
pnts = Point.ByCoordinates(x<1>,y<2>,z<1>);
crv = PolyCurve.ByPoints(pnts);
num = [0, 400, 800, 400, 0];
extS = Curve.ExtendStart(crv, num);
extE = Curve.ExtendEnd(extS, num);
srf = Surface.ByLoft(extE);
```

Code Block

Inputs: `divNumber`, `step`, `srf`

```
//Number range from '0' to '1', equally spaced 'divNumber' times
baseDivision = 0..1..#divNumber;
//All numbers in range except the last
uDivStart = 0..divNumber-2;
//All numbers in range except the first
uDivEnd = 1..divNumber-1;
//Running our custom definition
uDivision = steppedList(baseDivision[uDivStart]<1>,
baseDivision[uDivEnd]<1>, step);
//Flattend 'uDivision' definition results
resultUDiv = List.Flatten(uDivision,-1);
vDiv = 0..1..#10;
// Surface UV Points
srfPnts = Surface.PointAtParameter(srf<1>, vDiv<2>, resultUDiv<3>);
//Lateral PolyCurves
lateralCurves = PolyCurve.ByPoints(srfPnts);
//Transverse PolyCurves
transverseCurves = PolyCurve.ByPoints(List.Transpose(srfPnts));
```