## Session 2.2

# DesignScript: A robust dive into the language underlying Dynamo Sol Amour, Curious Human Being

**Class Description**

This lab will explore and cover DesignScript, the language that underlies Dynamo. Starting with the bare basics, it will ease into this textual language from the approach of a complete beginner, exploring not only the core building blocks but integral features and concepts that are imperative for a user to understand when shifting from a node based workflow to a code based one.

We will cover DesignScript fundamentals, covering syntax, object types, legibility, the UI and IDE. Diving deeper into object types, we will explore the building blocks of DesignScript and features that enable clever use of them. We will touch upon auto-complete and dot notation, explore lists and dictionaries and understand the nuances inherent within.

We explore Data Management tools that DesignScript has through core language features such as conditional statements and operators will allow us to leverage masks and filtration techniques, as well as looking at data generation in the form of number ranges and sequences, before exploring the limitations of Node2Code and when and how to use such a feature before exploring more advanced features integral to high-level DesignScript use with regards to data management such as nesting, replication and List@Level.

We finish with a deep dive into the power of DesignScript over nodal based workflows: Conditionals and looping, where we understand how to loop functionality across data lists one by one and wrap up powerful workflows into custom definitions.

**About the Speaker:**

Sol holds both a Bachelor of Architectural Studies and a Masters of Architecture from Victoria University of Wellington, New Zealand. He has experience in a diverse range of fields ranging from construction through landscaping, industrial design and architecture. He has focused his career on education and a technical understanding of complex problems, tempered by carefully considered holistic design. Most of his career has been set within multidisciplinary firms, working on large scale commercial projects, where his focus has been upon communication, clarity and effectiveness between all parties involved - all while delivering outstanding bespoke design solutions.

His passions are exploring diverse fields inclusive of automation, parametric design, form finding, generative and iterative design, and software augmentation and creation.

## DesignScript Fundamentals

In the DesignScript fundamentals section we will cover the following: Syntax, Object Types, Legibility, UI and IDE.

### Integrated Development Environment (IDE)

DesignScript is authored (written) inside a **Code Block**. When typing, the integrated **auto-complete** help features 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. You can also *tab* or *click on the function* to finish the current line and auto-complete will limit possible options the more that you type as showcased in the images to the right.

DesignScript contains a **dot-notation** feature that showcases available constructors (creators), methods (sctions 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 but properties do not appear inside of dot-notation.

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.

### Concepts

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.

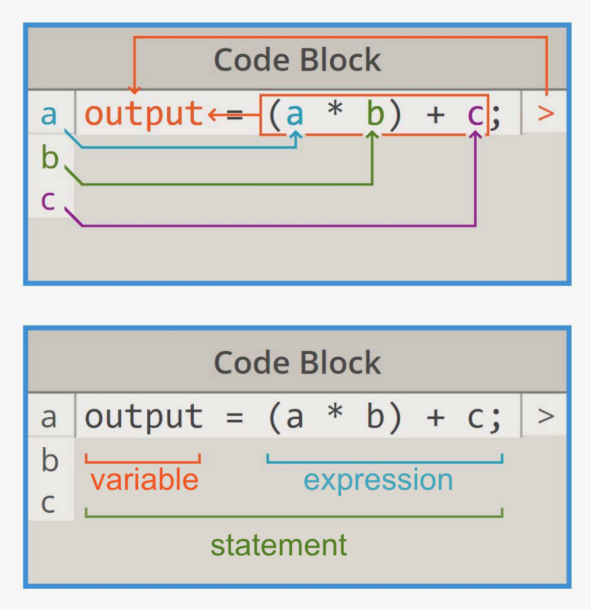


Figure 1: Code Block

A **Variable** is what you name stuff and use to call to it when you need it:

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

Assign using the '*variableName = object*' 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 and they act like containers that hold information, callable by their name. They also provide clarity when others read your code (If you are diligent at naming!).

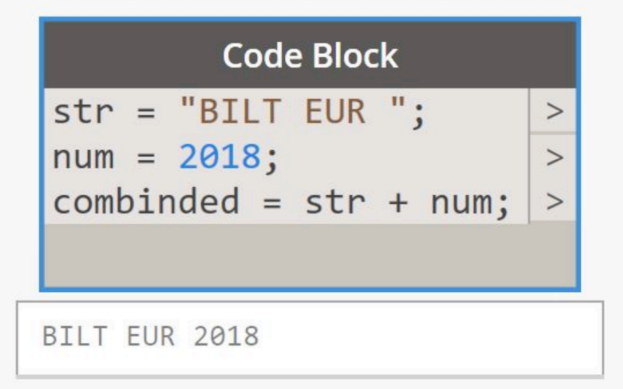


Figure 2: Variables

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).

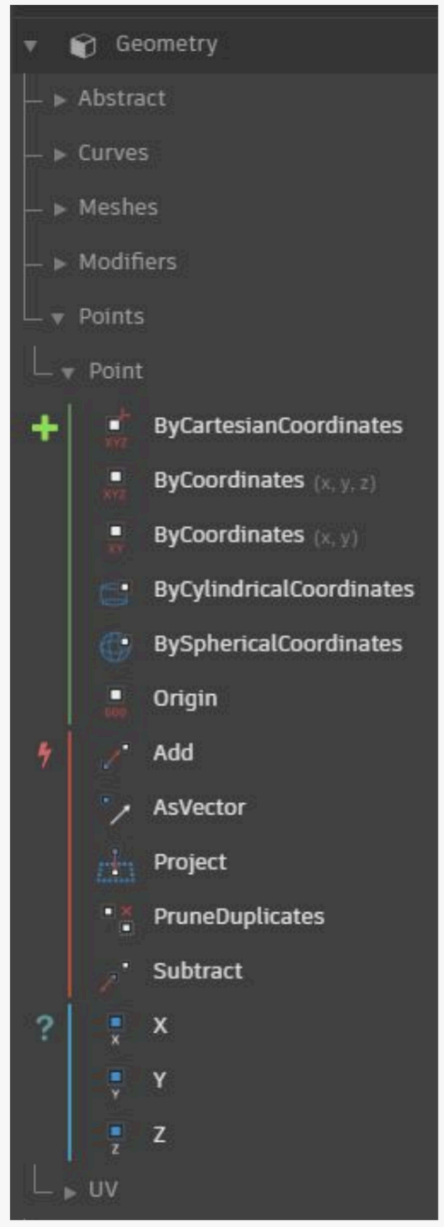


Figure 3: Namespace

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.

## Object Types

**Booleans** are used in 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 and can also be referenced as Yes or No (In contexts such as Revit).

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 Figure 4 below).

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

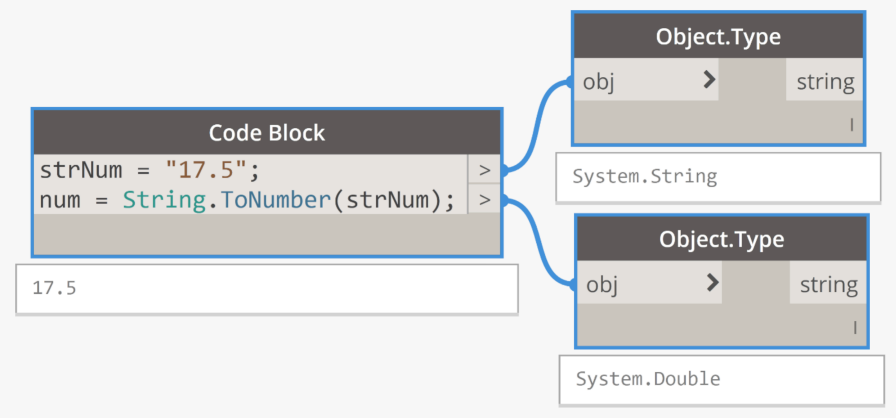


Figure 4: Casting to Number

A **String** is DesignScript for 'Text'. Enclose text inside of quotation marks to create a string:

```
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 )!!!";
```

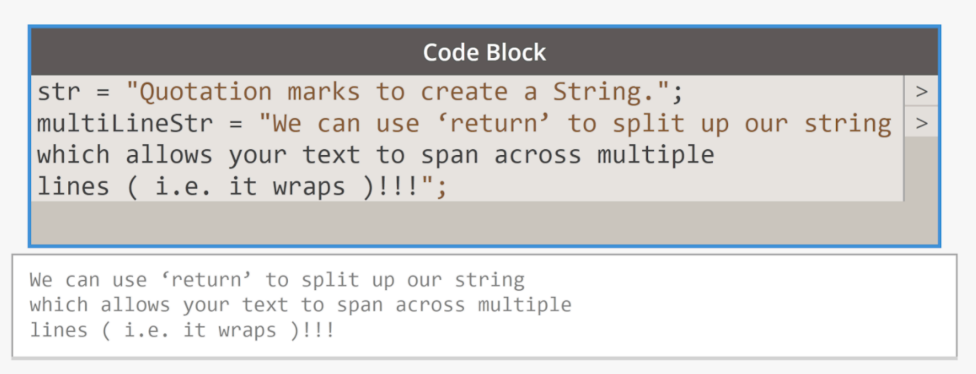


Figure 5: Strings

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

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

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

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;
```

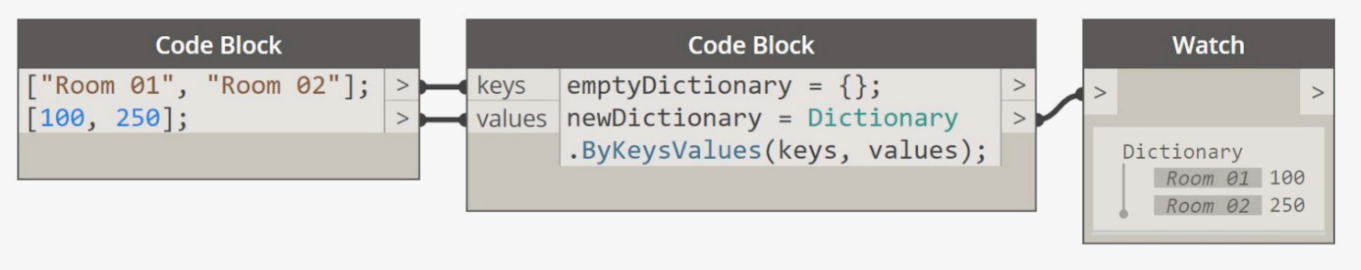


Figure 6: Dictionary

## Operators

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.

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;
```

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.

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. ).

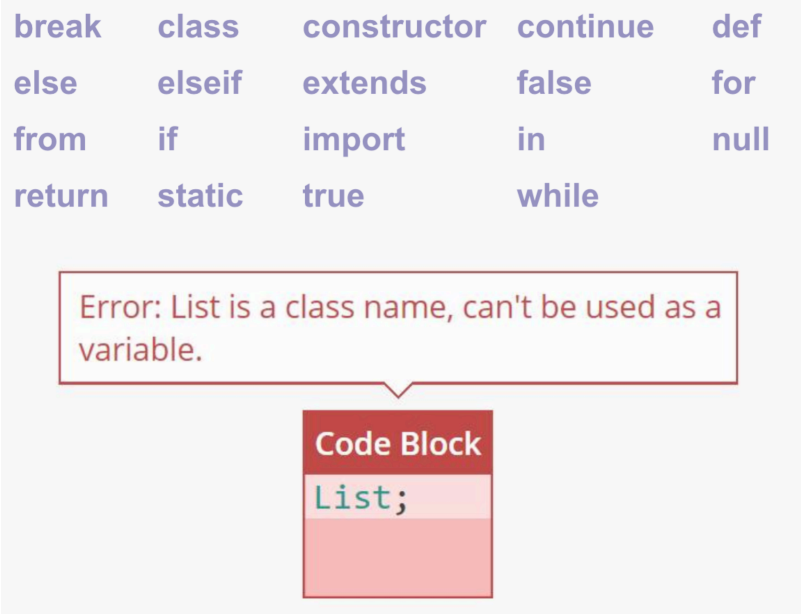


Figure 7: Reserved Keywords

## Data Management

In the Data Management section we will cover the following: Replication, List@Level, Nesting and Conflicts.

### Concepts

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 =
```

**Figure 8: Nesting**

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 before the dot-notation and 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.

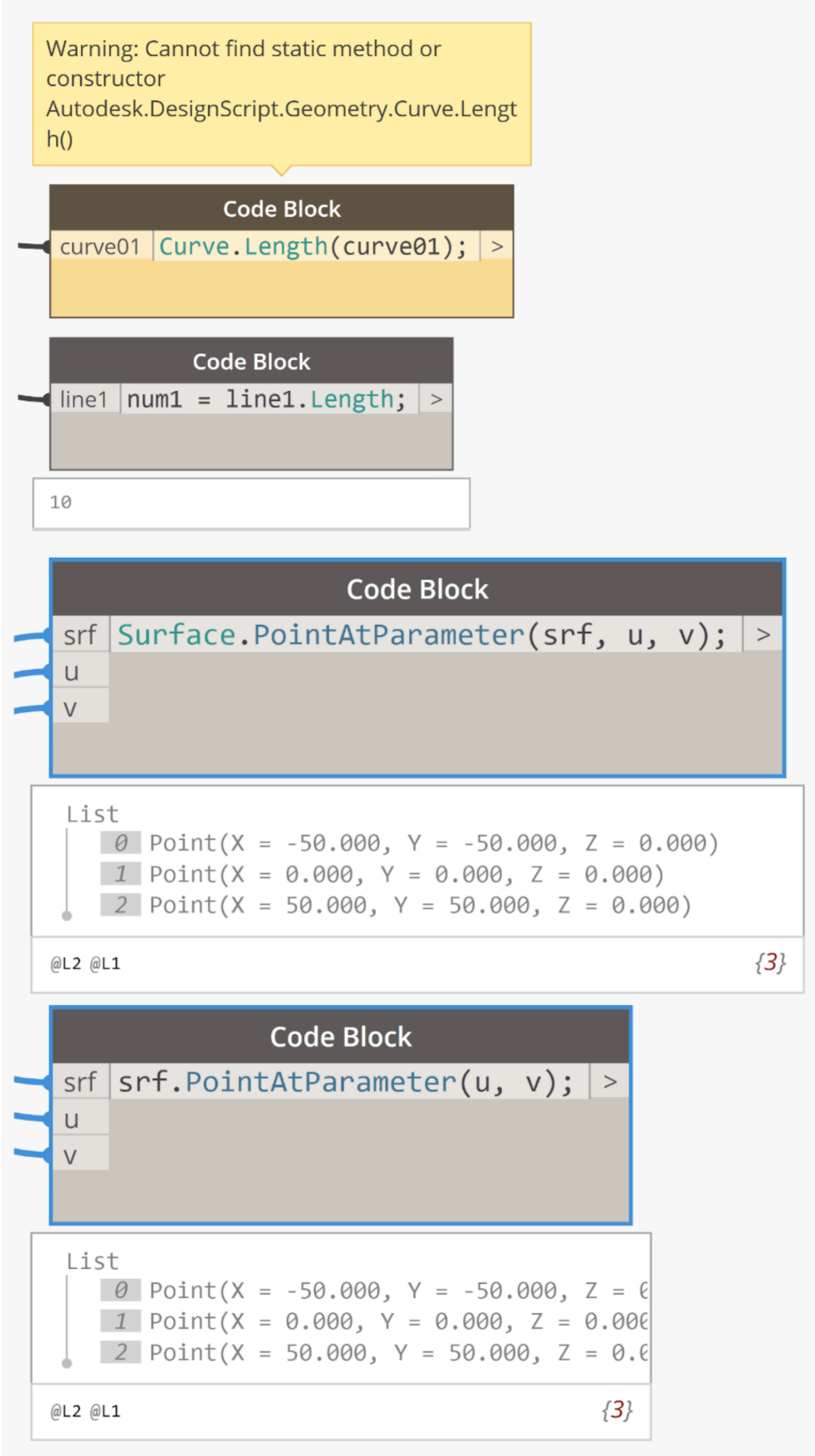


Figure 9: Static vs. Instance Methods

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 and all tab-finished dot notation will finalise **DSCore.DateTime** as default (So you'll have to manually delete that portion).

## Features

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** and alphabetic ranges vary depending on capitalisation. The *start..end..#amount* range (E.g *0..1..#20*) is harder to reproduce in nodal form - 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*).

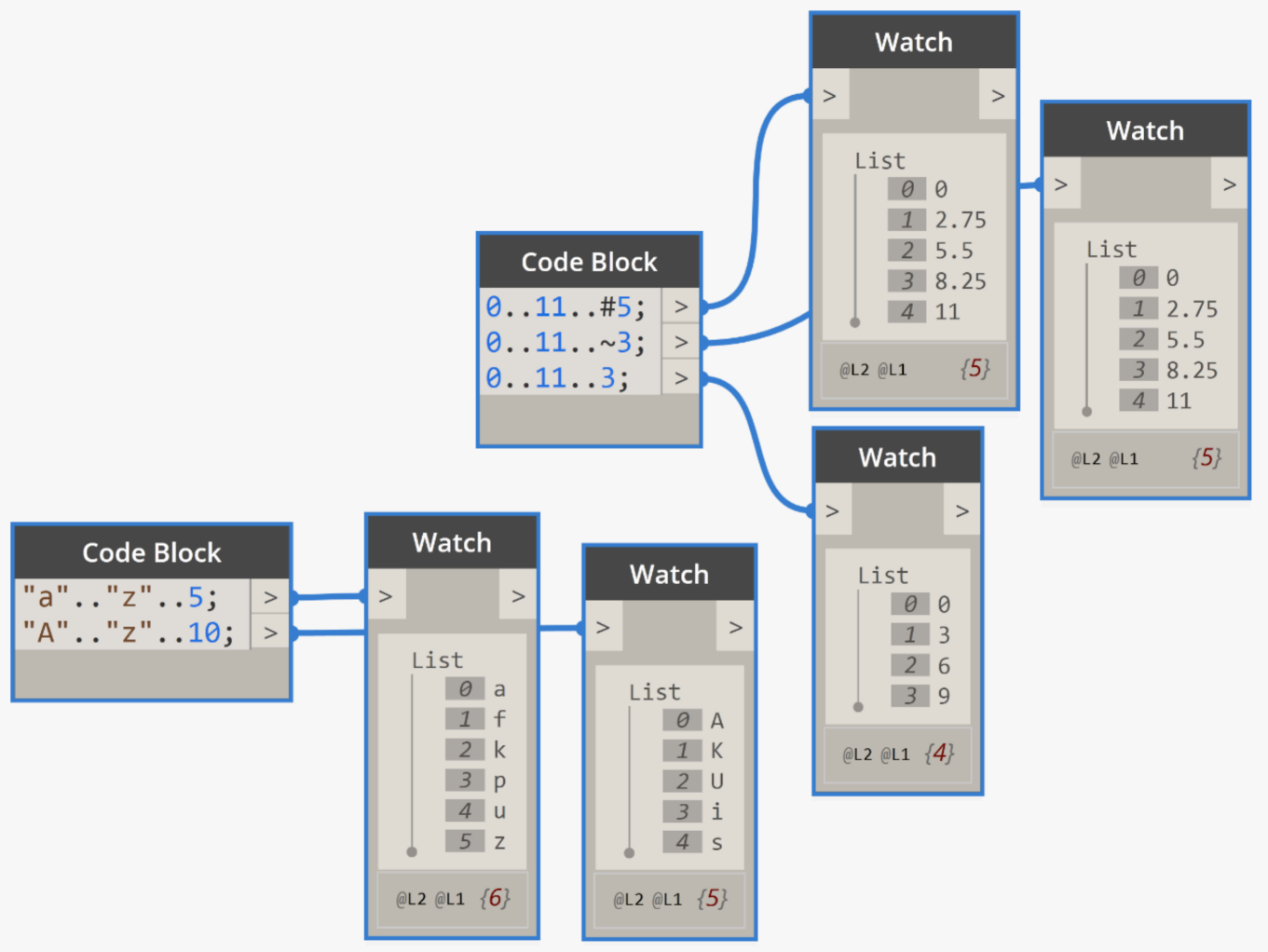


Figure 10: Ranges

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.

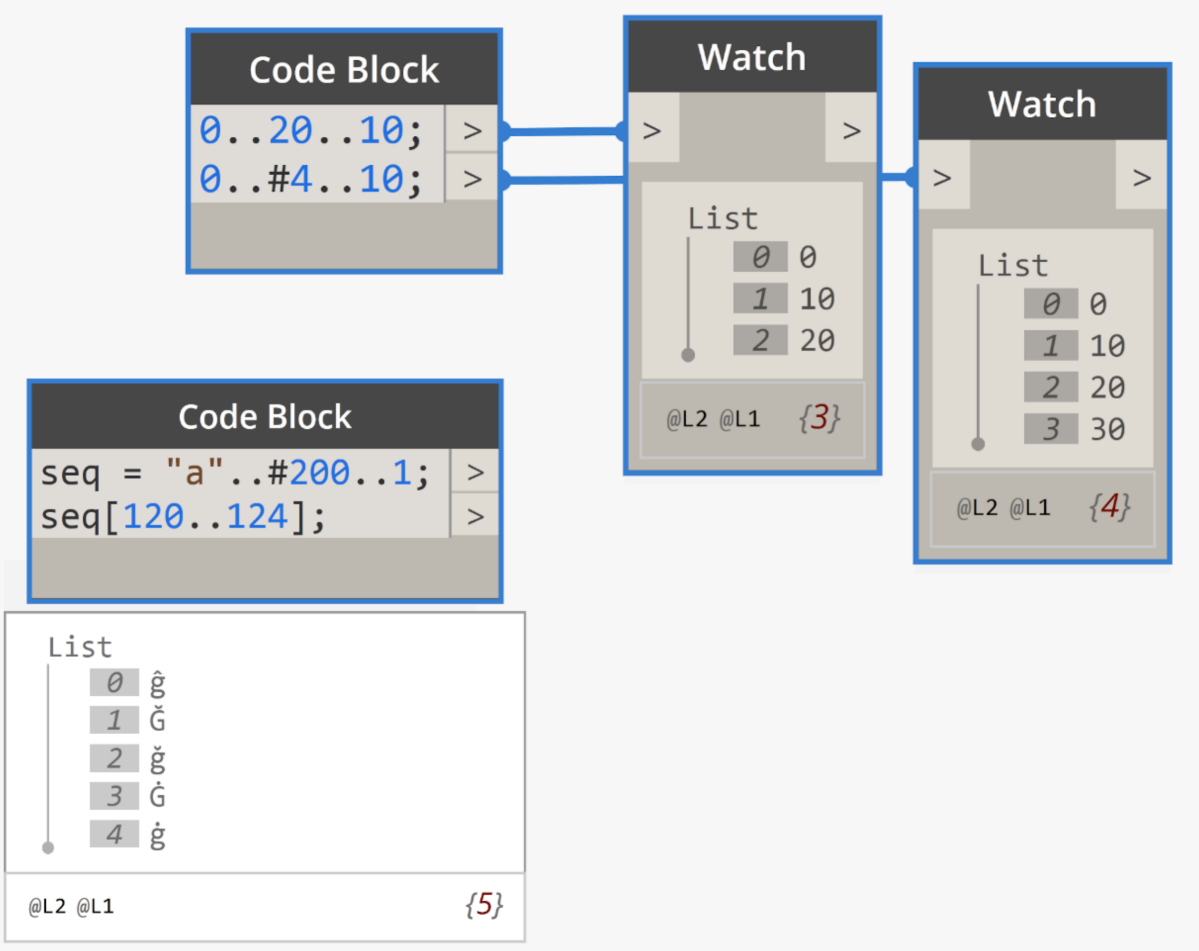


Figure 11: Sequences

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.

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 a Less-than sign followed by a Number and then a Greater-than sign ( <#> ) which 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 )

**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.

**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*.

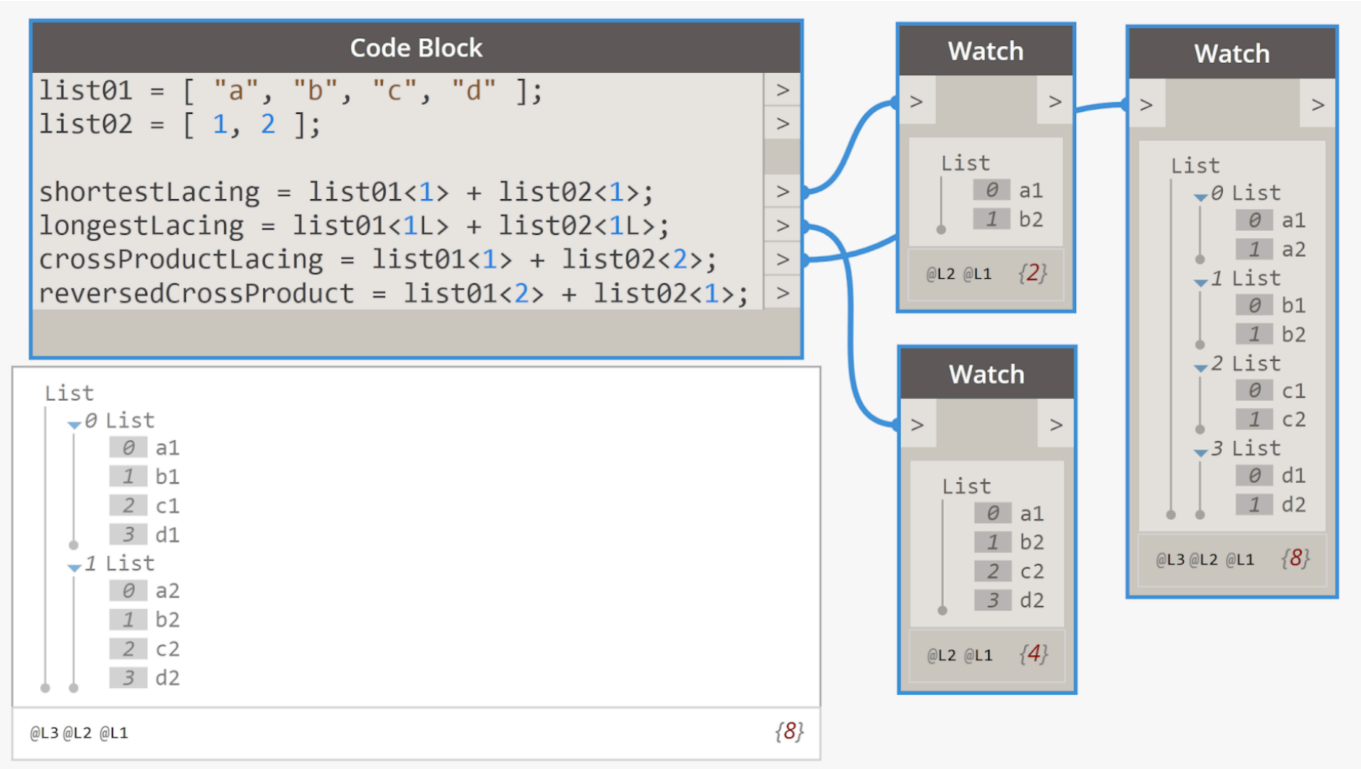


Figure 12: Replication

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);
```

```
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}
```

Figure 13: List @ Level levels

**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**.

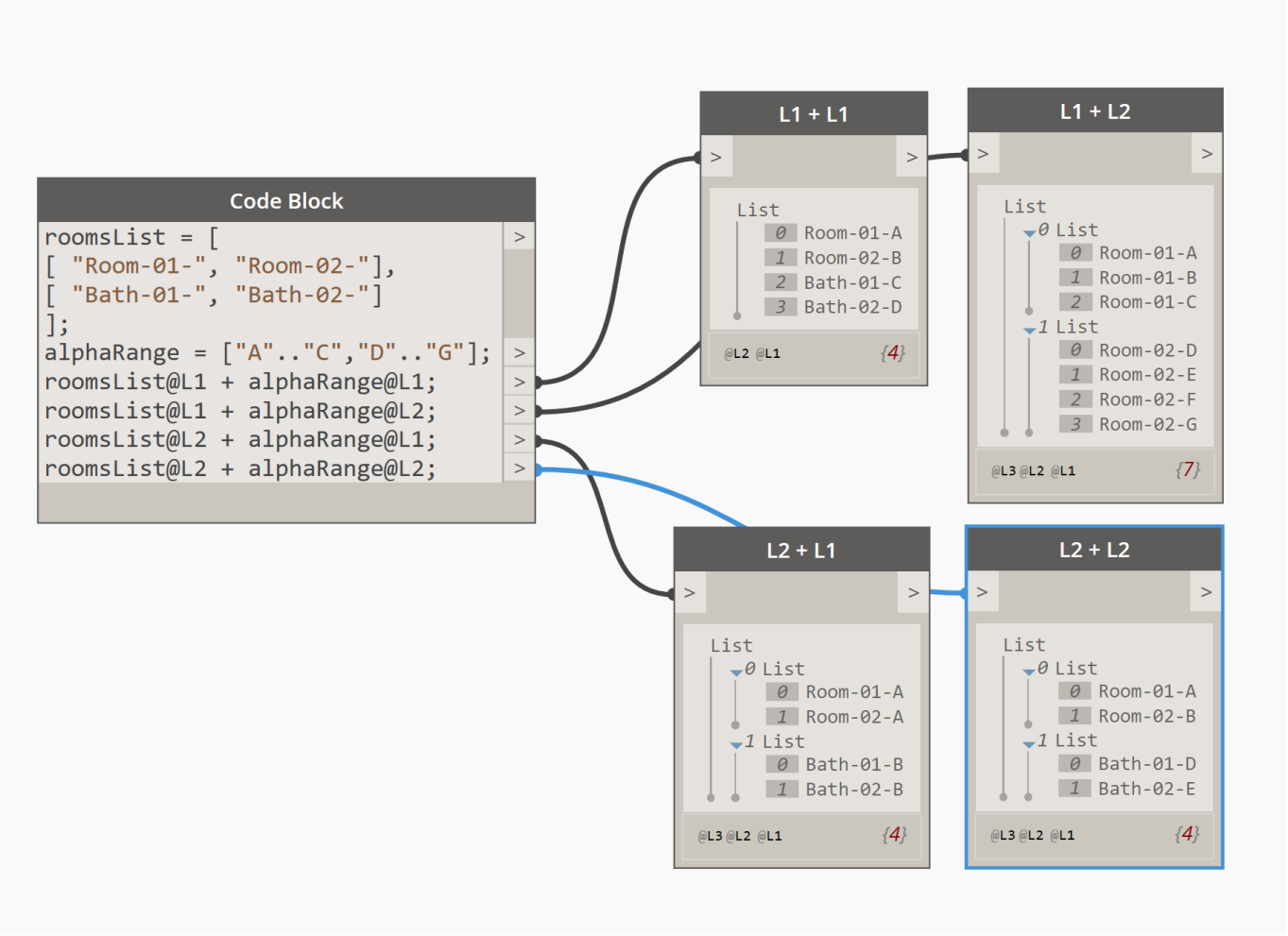


Figure 14: List @ Level example

**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>;
```

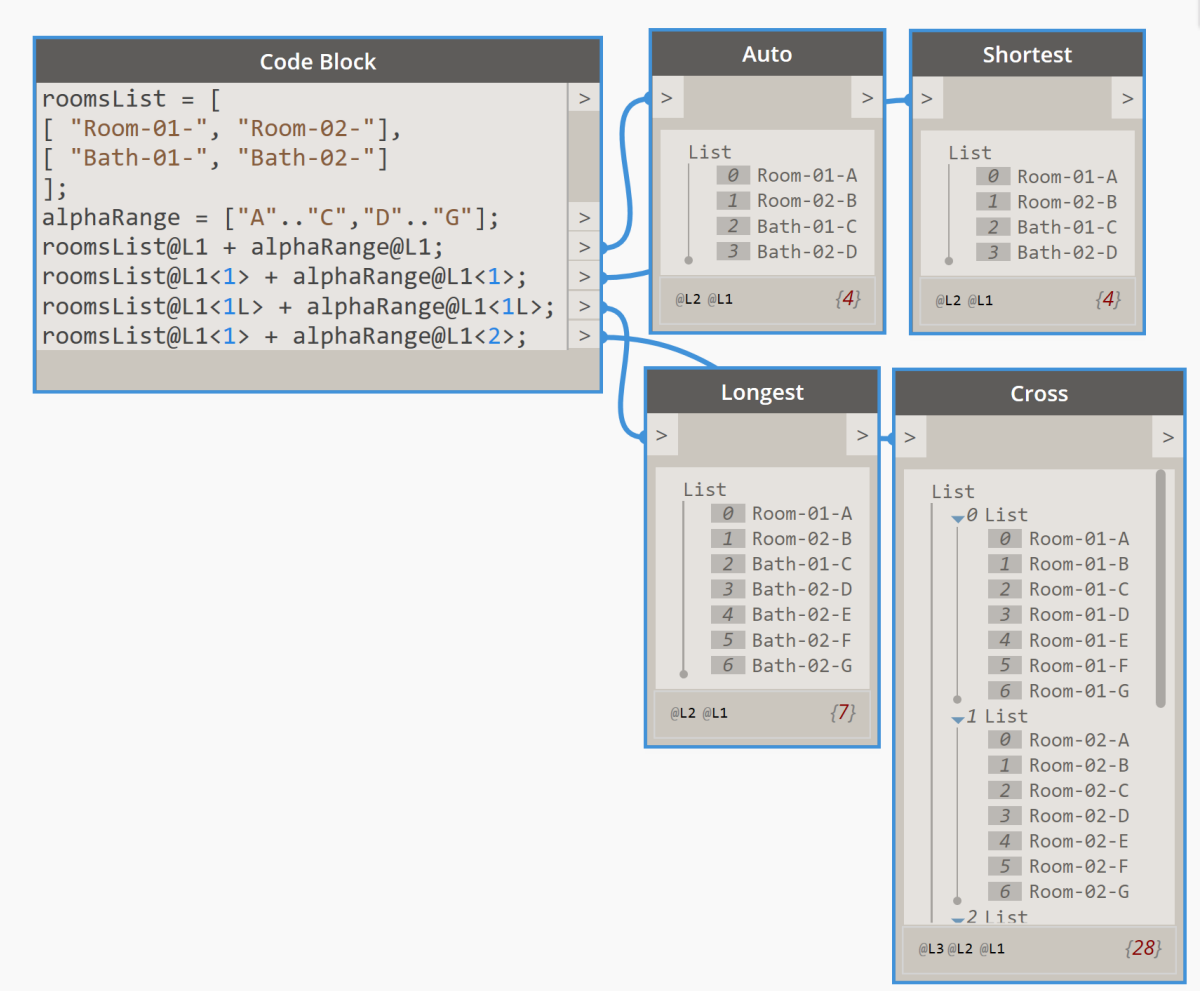


Figure 15: Replication Typologies

## Power and Control

In the Power and Control section we will cover the following: Conditional statements, Operators, Looping and Custom Definitions.

### Concepts

**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 };
variable = [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.

**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

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 a 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 be used as a *'gate'* to allow a section of code to execute or not. Conditional statements always start with an **'if'** statement and are terminated by an **'else'** statement in a single loop; If you want to check against multiple conditions you can nest as many **'elseif'** statements together as required before the terminating **'else'** statement.
Indentation is NOT required in DesignScript, but is advisable for clarity.

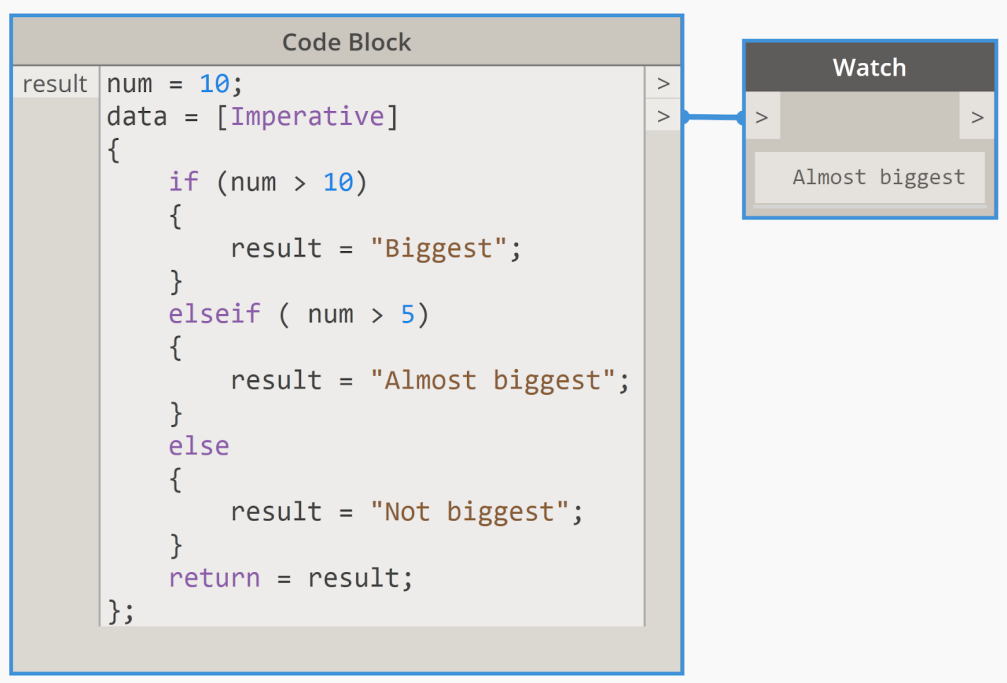


Figure 16: Conditional Statements

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 "*n*" number of times. 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 list;
}
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.

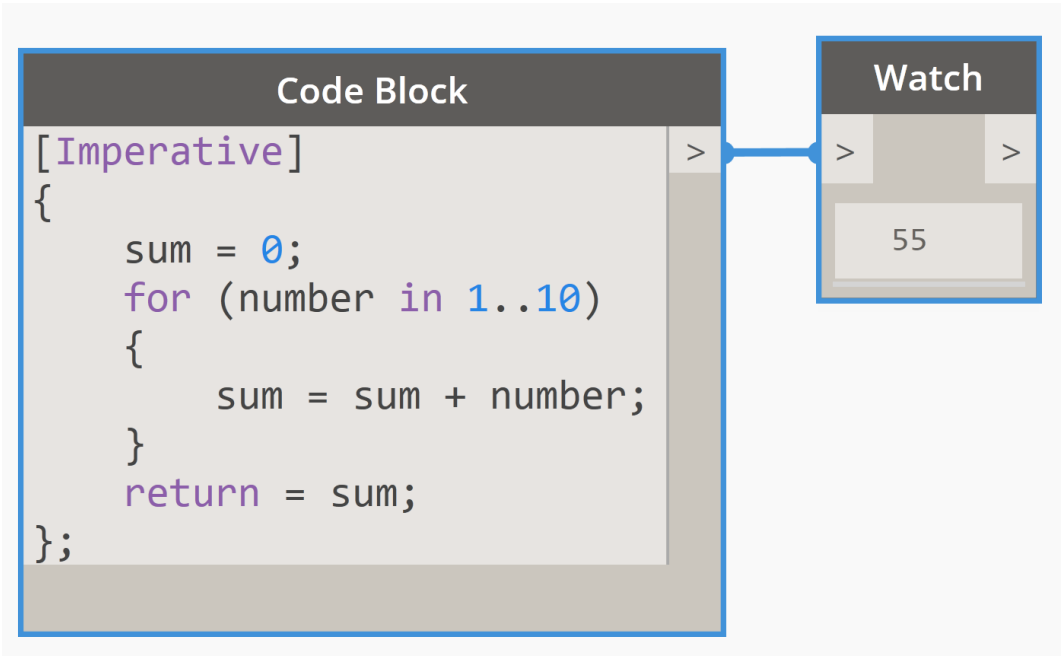


Figure 17: For Loop

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* that has no way of ending - 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

Figure 18: While Loop

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.**

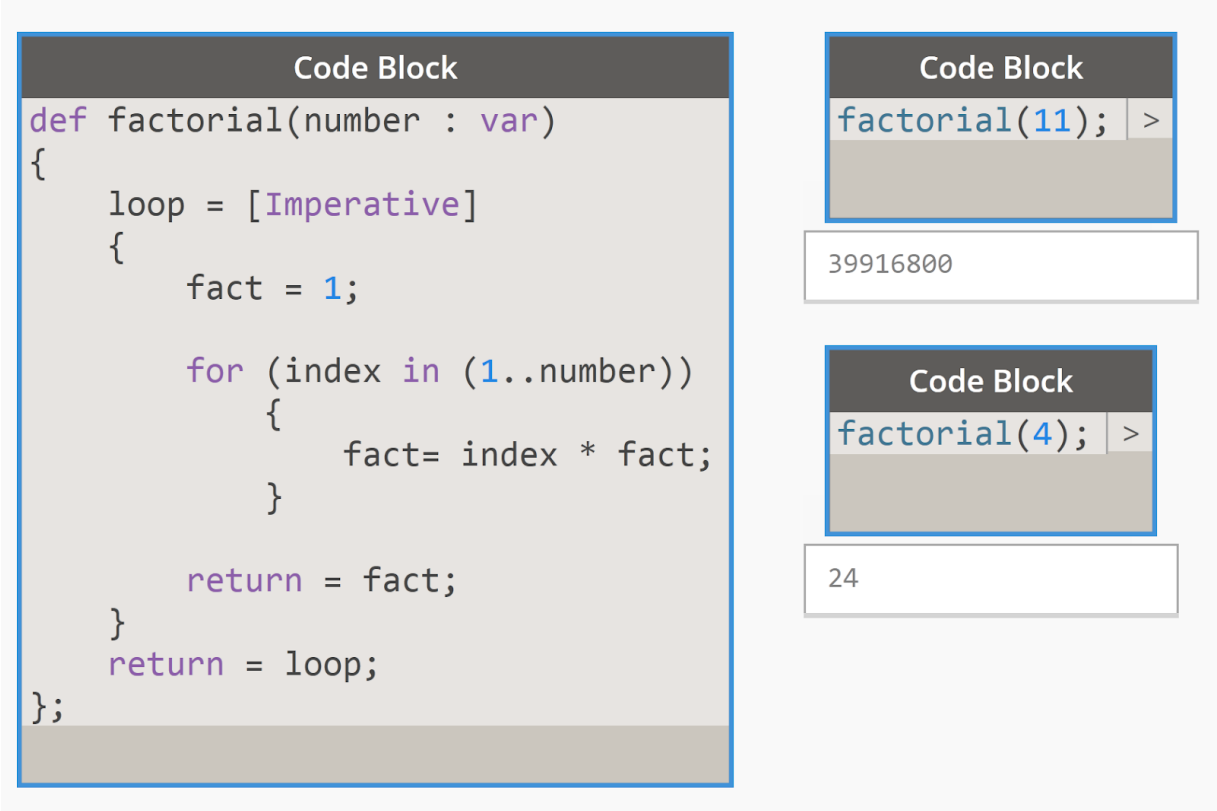


Figure 19: Factorial Definition

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

**int** | **double** | **bool** | **string** | **var**

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).

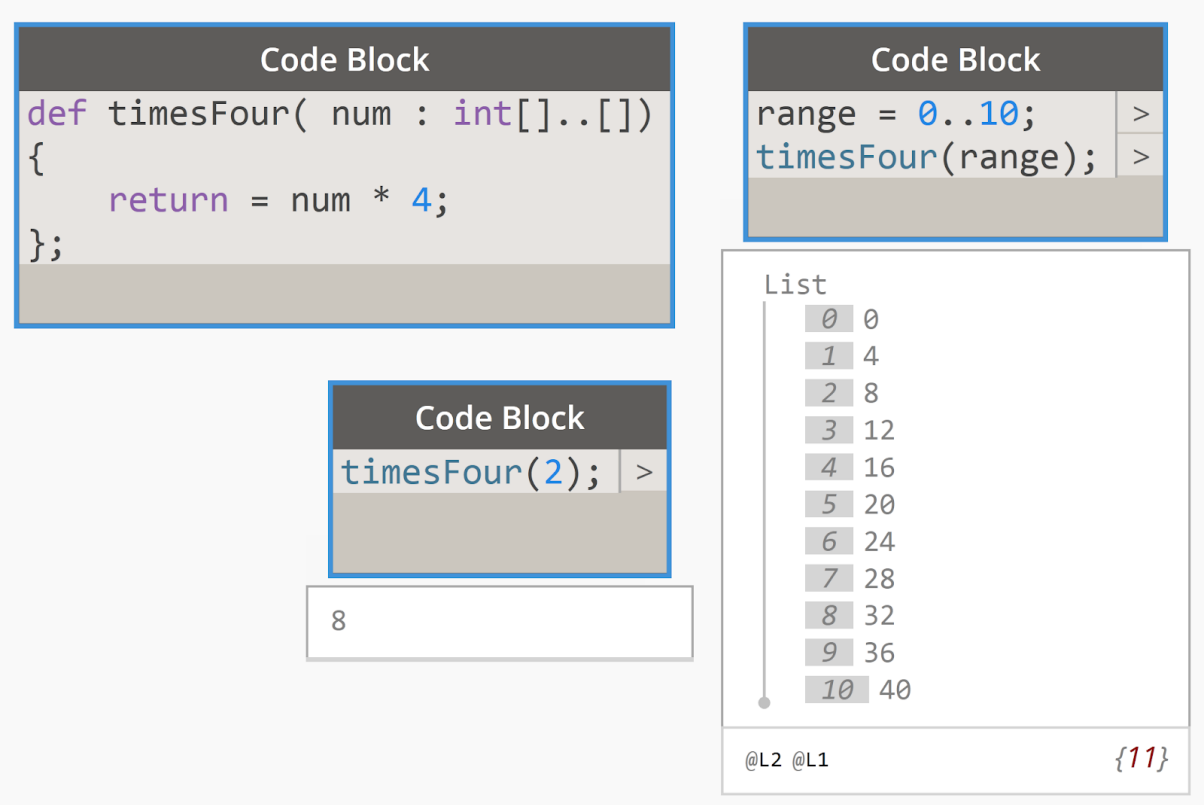


Figure 20: Dimensionality and Typology

Definitions are **global** until the Dynamo session is closed. This means that deleted functions can still appear in auto-complete. Overloading functions (Creating definitions that are of the same name with different arguments) behave weirdly and should be avoided. Do NOT use **List@Level** or **replication** in Imperative code blocks as the DesignScript engine will not execute that code.