Dynamo Language Manual

Contents

Introduction

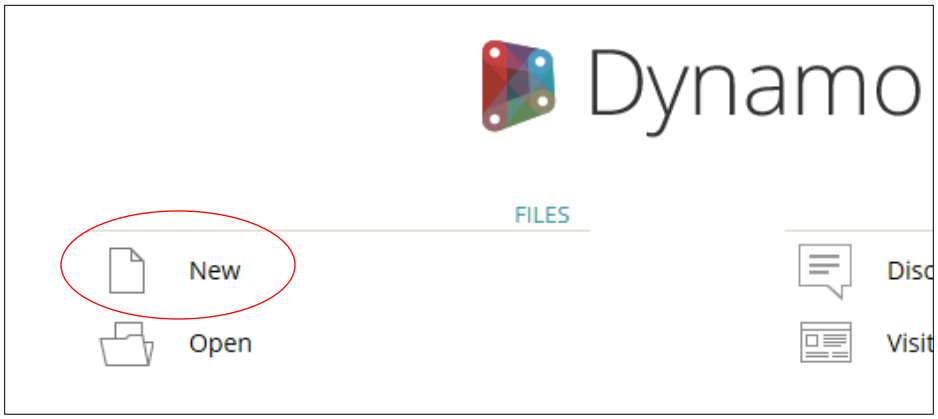
Programming languages are created to express ideas, usually involving logic and calculation. In addition to these objectives, the Dynamo textual language (formerly DesignScript) has been created to express *design intentions*. It is generally recognized that computational designing is exploratory, and Dynamo tries to support this: we hope you find the language flexible and fast enough to take a design from concept, through design iterations, to your final form.

This manual is structured to give a user with no knowledge of either programming or architectural geometry full exposure to a variety of topics in these two intersecting disciplines. Individuals with more experienced backgrounds should jump to the individual sections which are relevant to their interests and problem domain. Each section is self-contained, and doesn't require any knowledge besides the information presented in prior sections.

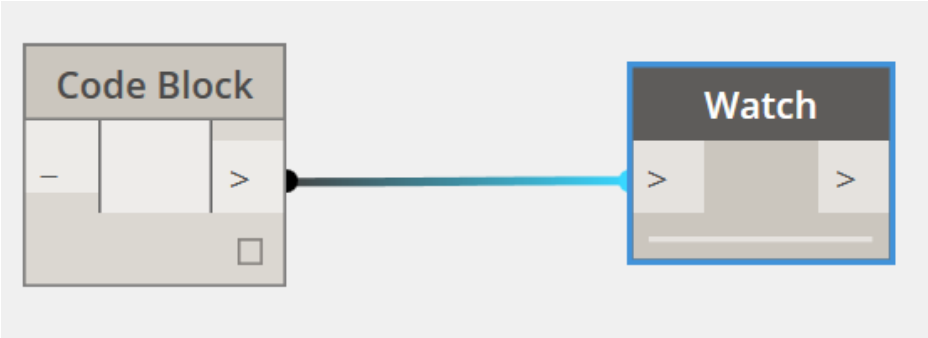
Text blocks inset in the `Consolas` font should be pasted into a Code Block node. The output of the Code Block should be connected into a Watch node to see the intended result. Images are included in the left margin illustrating the correct output of your program.

1: Language Basics

This document discusses the Dynamo textual programming language, used inside of the Dynamo editor (sometimes referred to as “Dynamo Sandbox”). To create a new Dynamo script, open the Dynamo editor, and select the “New” button in the “FILES” group:



This will open a blank Dynamo graph. To write a Dynamo text script, double click anywhere in the canvas. This will bring up a “Code Block” node. In order to easily see the results of our scripts, attach a “Watch” node to the output of your Code Block node, as shown here:

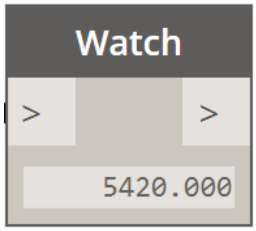


Every script is a series of written commands. Some of these commands create geometry; others solve mathematical problems, write text files, or generate text strings. A simple, one line program which generates the quote “Less is more.” looks like this:


```
"Less is more.";
```

The Watch node on the left shows the output of the script.

The command generates a new String object. Strings in Dynamo are designated by two quotation marks ("), and the enclosed characters, including spaces, are passed out of the node. Code Block nodes are not limited to generating Strings. A Code Block node to generate the number 5420 looks like this:




```
5420;
```

Every command in Dynamo is terminated by a semicolon. If you do not include one, the Editor will add one for you. Also note that the number and combination of spaces, tabs, and carriage returns, called white space, between the elements of a command do not matter. This program produces the exact same output as the first program:


```
                "Less Is More."

;
```

Naturally, the use of white space should be used to help improve the readability of your code, both for yourself and future readers.

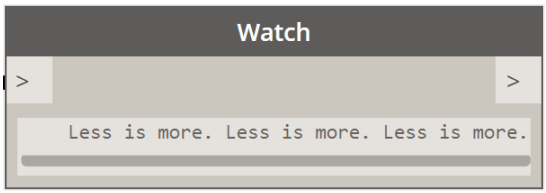
Comments are another tool to help improve the readability of your code. In Dynamo, a single line of code is “commented” with two forward slashes, `//`. This makes the node ignore everything written after the slashes, up to a carriage return (the end of the line). Comments longer than one line begin with a forward slash asterisk, `/*`, and end with an asterisk forward slash, `*/`.


```
// This is a single line comment

/* This is a multiple line comment,
   which continues for multiple
   lines. */

// All of these comments have no effect on
// the execution of the program

// This line prints a quote by Mies van der Rohe
"Less Is More";
```

So far the Code Block arguments have been 'literal' values, either a text string or a number. However it is often more useful for function arguments to be stored in data containers called variables, which both make code more readable, and eliminate redundant commands in your code. The names of variables are up to individual programmers to decide, though each variable name must be unique, start with a lower or uppercase letter, and contain only letters, numbers, or underscores, _. Spaces are not allowed in variable names. Variable names should, though are not required, to describe the data they contain. For instance, a variable to keep track of the rotation of an object could be called `rotation`. To describe data with multiple words, programmers typically use two common conventions: separate the words by capital letters, called camelCase (the successive capital letters mimic the humps of a camel), or to separate individual words with underscores. For instance, a variable to describe the rotation of a small disk might be named `smallDiskRotation` or `small_disk_rotation`, depending on the programmer's stylistic preference. To create a variable, write its name to the left of an equal sign, followed by the value you want to assign to it. For instance:

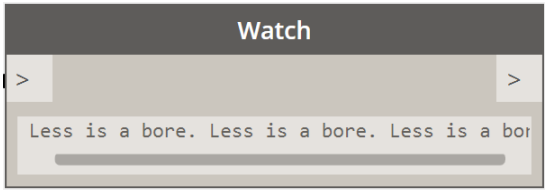

```
quote = "Less is more.";
```

Besides making readily apparent what the role of the text string is, variables can help reduce the amount of code that needs updating if data changes in the future. For instance the text of the following quote only needs to be changed in one place, despite its appearance three times in the program.




```
// My favorite architecture quote

quote = "Less is more.";
quote + " " + quote + " " + quote;
```

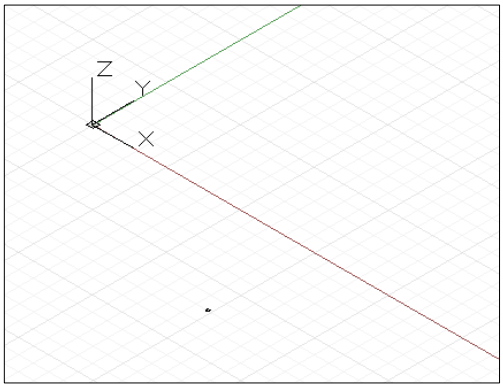
Here we are joining a quote by Mies van der Rohe three times, with spaces between each phrase. Notice the use of the + operator to ‘concatenate’ the strings and variables together to form one continuous output.




```
// My NEW favorite architecture quote

quote = "Less is a bore.";
quote + " " + quote + " " + quote;
```

2: Geometry Basics

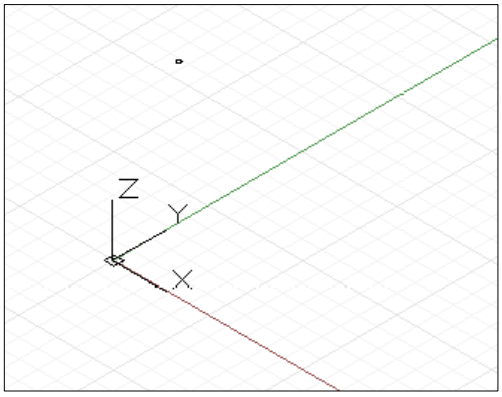
The simplest geometrical object in the Dynamo standard geometry library is a point. All geometry is created using special functions called constructors, which each return a new instance of that particular geometry type. In Dynamo, constructors begin with the name of the object's type, in this case `Point`, followed by the method of construction. To create a three dimensional point specified by x, y, and z Cartesian coordinates, use the `ByCoordinates` constructor:




```
// create a point with the following x, y, and z
// coordinates:
x = 10;
y = 2.5;
z = -6;

p = Point.ByCoordinates(x, y, z);
```

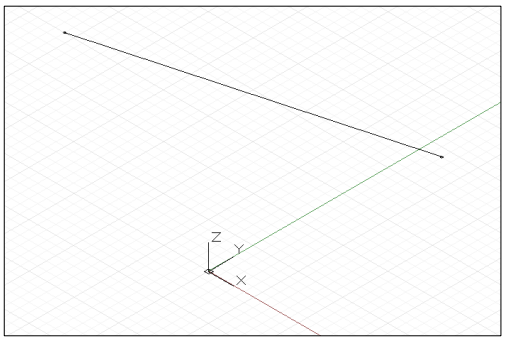
Constructors in Dynamo are typically designated with the "`By`" prefix, and invoking these functions returns a newly created object of that type. This newly created object is stored in the variable named on the left side of the equal sign, and any use of that same original Point.

Most objects have many different constructors, and we can use the `BySphericalCoordinates` constructor to create a point lying on a sphere, specified by the sphere's radius, a first rotation angle, and a second rotation angle (specified in degrees):



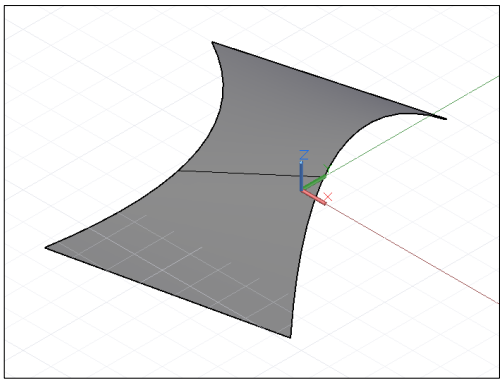

```
// create a point on a sphere with the following radius,
// theta, and phi rotation angles (specified in degrees)
radius = 5;
theta = 75.5;
phi = 120.3;
cs = CoordinateSystem.Identity();

p = Point.BySphericalCoordinates(cs, radius, theta,
    phi);
```

Points can be used to construct higher dimensional geometry such as lines. We can use the `ByStartPointEndPoint` constructor to create a Line object between two points:



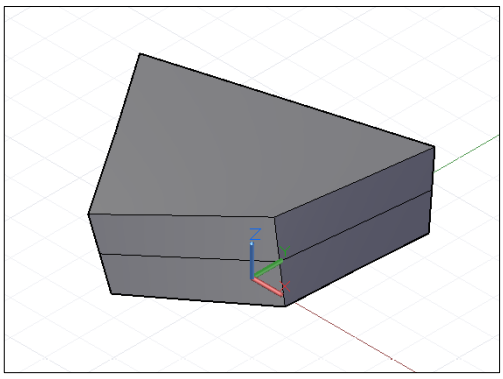

```
// create two points:
p1 = Point.ByCoordinates(3, 10, 2);
p2 = Point.ByCoordinates(-15, 7, 0.5);

// construct a line between p1 and p2
l = Line.ByStartPointEndPoint(p1, p2);
```

Similarly, lines can be used to create higher dimensional surface geometry, for instance using the `Loft` constructor, which takes a series of lines or curves and interpolates a surface between them.



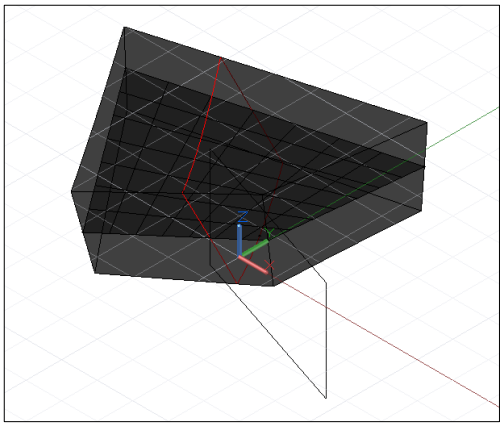
```
// create points:
p1 = Point.ByCoordinates(3, 10, 2);
p2 = Point.ByCoordinates(-15, 7, 0.5);

p3 = Point.ByCoordinates(5, -3, 5);
p4 = Point.ByCoordinates(-5, -6, 2);

p5 = Point.ByCoordinates(9, -10, -2);
p6 = Point.ByCoordinates(-11, -12, -4);

// create lines:
l1 = Line.ByStartPointEndPoint(p1, p2);
l2 = Line.ByStartPointEndPoint(p3, p4);
l3 = Line.ByStartPointEndPoint(p5, p6);

// loft between cross section lines:
surf = Surface.ByLoft({l1, l2, l3});
```

Surfaces too can be used to create higher dimensional *solid* geometry, for instance by thickening the surface by a specified distance. Many objects have functions attached to them, called methods, allowing the programmer to perform commands on that particular object. Methods common to all pieces of geometry include `Translate` and `Rotate`, which respectively translate (move) and rotate the geometry by a specified amount. Surfaces have a `Thicken` method, which take a single input, a number specifying the new thickness of the surface.



```
p1 = Point.ByCoordinates(3, 10, 2);
p2 = Point.ByCoordinates(-15, 7, 0.5);

p3 = Point.ByCoordinates(5, -3, 5);
p4 = Point.ByCoordinates(-5, -6, 2);

l1 = Line.ByStartPointEndPoint(p1, p2);
l2 = Line.ByStartPointEndPoint(p3, p4);

surf = Surface.ByLoft({l1, l2});

// true indicates to thicken both sides of the Surface:
solid = surf.Thicken(4.75, true);
```

`Intersection` commands can extract lower dimensional geometry from higher dimensional objects. This extracted lower dimensional geometry can form the basis for higher dimensional geometry, in a cyclic process of geometrical creation, extraction, and recreation. In this example, we use the generated Solid to create a Surface, and use the Surface to create a Curve.



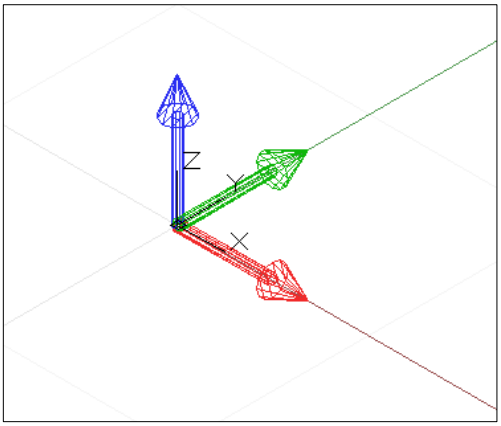
```
p1 = Point.ByCoordinates(3, 10, 2);
p2 = Point.ByCoordinates(-15, 7, 0.5);

p3 = Point.ByCoordinates(5, -3, 5);
p4 = Point.ByCoordinates(-5, -6, 2);

l1 = Line.ByStartPointEndPoint(p1, p2);
l2 = Line.ByStartPointEndPoint(p3, p4);

surf = Surface.ByLoft({l1, l2});

solid = surf.Thicken(4.75, true);

p = Plane.ByOriginNormal(Point.ByCoordinates(2, 0, 0),
    Vector.ByCoordinates(1, 1, 1));

int_surf = solid.Intersect(p);

int_line = int_surf.Intersect(Plane.ByOriginNormal(
    Point.ByCoordinates(0, 0, 0),
    Vector.ByCoordinates(1, 0, 0)));
```

3: Geometric Primitives

While Dynamo is capable of creating a variety of complex geometric forms, simple geometric primitives form the backbone of any computational design: either directly expressed in the final designed form, or used as scaffolding off of which more complex geometry is generated.

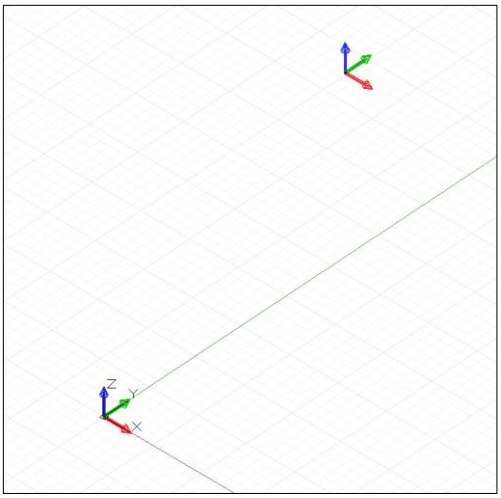
While not strictly a piece of geometry, the CoordinateSystem is an important tool for constructing geometry. A CoordinateSystem object keeps track of both position and geometric transformations such as rotation, sheer, and scaling.

Creating a CoordinateSystem centered at a point with x = 0, y = 0, z = 0, with no rotations, scaling, or sheering transformations, simply requires calling the Identity constructor:




```
// create a CoordinateSystem at x = 0, y = 0, z = 0,
// no rotations, scaling, or sheering transformations

cs = CoordinateSystem.Identity();
```

CoordinateSystems with geometric transformations are beyond the scope of this chapter, though another constructor allows you to create a coordinate system at a specific point, `CoordinateSystem.ByOriginVectors`:

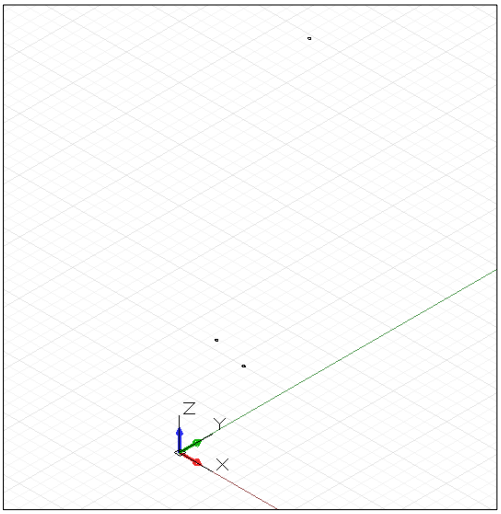


```
// create a CoordinateSystem at a specific location,
// no rotations, scaling, or sheering transformations
x_pos = 3.6;
y_pos = 9.4;
z_pos = 13.0;

origin = Point.ByCoordinates(x_pos, y_pos, z_pos);
identity = CoordinateSystem.Identity();

cs = CoordinateSystem.ByOriginVectors(origin,
    identity.XAxis, identity.YAxis, identity.ZAxis);
```

The simplest geometric primitive is a Point, representing a zero-dimensional location in three-dimensional space. As mentioned earlier there are several different ways to create a point in a particular coordinate system: `Point.ByCoordinates` creates a

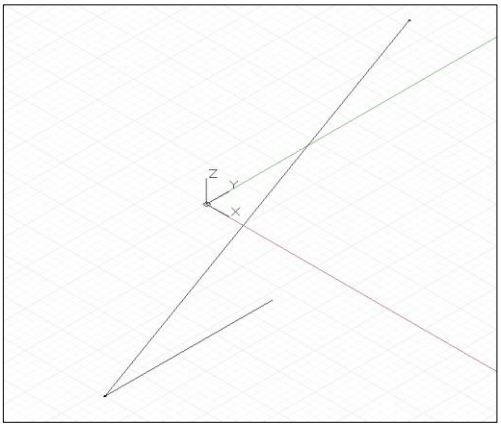
point with specified x, y, and z coordinates; `Point.ByCartesianCoordinates` creates a point with a specified x, y, and z coordinates *in a specific coordinate system*; `Point.ByCylindricalCoordinates` creates a point lying on a cylinder with radius, rotation angle, and height; and `Point.BySphericalCoordinates` creates a point lying on a sphere with radius and two rotation angle.

This example shows points created at various coordinate systems:



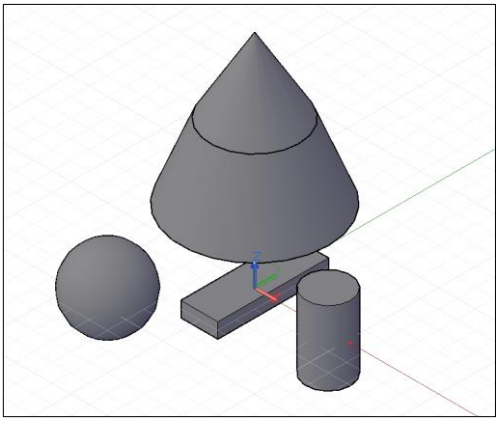

```
// create a point with x, y, and z coordinates
x_pos = 1;
y_pos = 2;
z_pos = 3;

pCoord = Point.ByCoordinates(x_pos, y_pos, z_pos);

// create a point in a specific coordinate system
cs = CoordinateSystem.Identity();
pCoordSystem = Point.ByCartesianCoordinates(cs, x_pos,
    y_pos, z_pos);

// create a point on a cylinder with the following
// radius and height
radius = 5;
height = 15;
theta = 75.5;

pCyl = Point.ByCylindricalCoordinates(cs, radius, theta,
    height);

// create a point on a sphere with radius and two angles

phi = 120.3;

pSphere = Point.BySphericalCoordinates(cs, radius,
    theta, phi);
```

The next higher dimensional Dynamo primitive is a line segment, representing an infinite number of points between two end points. Lines can be created by explicitly stating the two boundary points with the constructor `Line.ByStartPointEndPoint`, or by specifying a start point, direction, and length in that direction, `Line.ByStartPointDirectionLength`.



```
p1 = Point.ByCoordinates(-2, -5, -10);
p2 = Point.ByCoordinates(6, 8, 10);

// a line segment between two points
l2pts = Line.ByStartPointEndPoint(p1, p2);

// a line segment at p1 in direction 1, 1, 1 with
// length 10
lDir = Line.ByStartPointDirectionLength(p1,
    Vector.ByCoordinates(1, 1, 1), 10);
```

Dynamo has objects representing the most basic types of geometric primitives in three dimensions: Cuboids, created with `Cuboid.ByLengths`; Cones, created with `Cone.ByPointsRadius` and `Cone.ByPointsRadii`; Cylinders, created with `Cylinder.ByRadiusHeight`; and Spheres, created with `Sphere.ByCenterPointRadius`.



```
// create a cuboid with specified lengths
cs = CoordinateSystem.Identity();

cub = Cuboid.ByLengths(cs, 5, 15, 2);

// create several cones
p1 = Point.ByCoordinates(0, 0, 10);
p2 = Point.ByCoordinates(0, 0, 20);
p3 = Point.ByCoordinates(0, 0, 30);

cone1 = Cone.ByPointsRadii(p1, p2, 10, 6);
cone2 = Cone.ByPointsRadii(p2, p3, 6, 0);

// make a cylinder
cylCS = cs.Translate(10, 0, 0);

cyl = Cylinder.ByRadiusHeight(cylCS, 3, 10);

// make a sphere
centerP = Point.ByCoordinates(-10, -10, 0);

sph = Sphere.ByCenterPointRadius(centerP, 5);
```

4: Vector Math

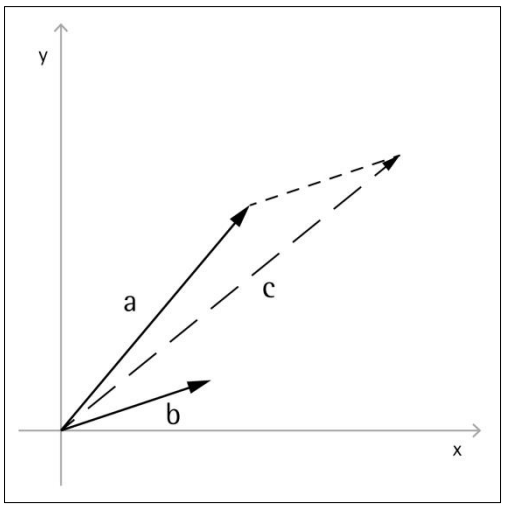
Objects in computational designs are rarely created explicitly in their final position and form, and are most often translated, rotated, and otherwise positioned based off of existing geometry. Vector math serves as a kind-of geometric scaffolding to give direction and orientation to geometry, as well as to conceptualize movements through 3D space without visual representation.

At its most basic, a vector represents a position in 3D space, and is often times thought of as the endpoint of an arrow from the position (0, 0, 0) to that position. Vectors can be created with the `ByCoordinates` constructor, taking the x, y, and z position of the newly created Vector object. Note that Vector objects are not geometric objects, and don't appear in the Dynamo window. However, information about a newly created or modified vector can be printed in the console window:

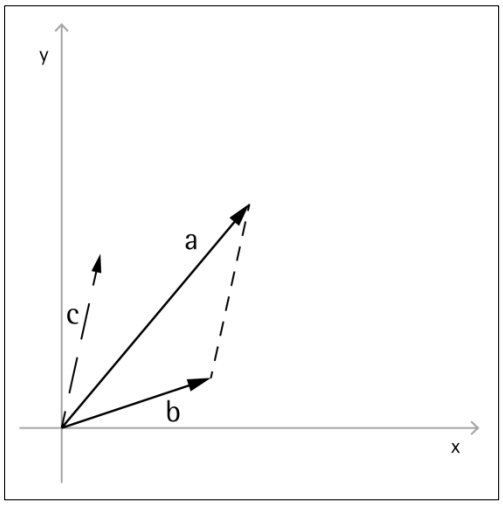

```
// construct a Vector object
v = Vector.ByCoordinates(1, 2, 3);

s = v.X + " " + v.Y + " " + v.Z;
```

A set of mathematical operations are defined on Vector objects, allowing you to add, subtract, multiply, and otherwise move objects in 3D space as you would move real numbers in 1D space on a number line.

Vector addition is defined as the sum of the components of two vectors, and can be thought of as the resulting vector if the two component vector arrows are placed "tip to tail." Vector addition is performed with the `Add` method, and is represented by the diagram on the left.

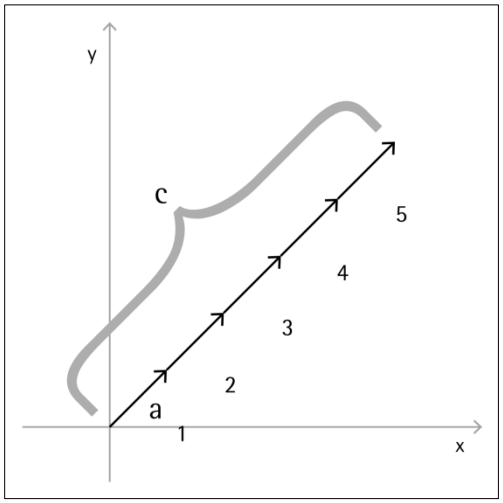



```
a = Vector.ByCoordinates(5, 5, 0);
b = Vector.ByCoordinates(4, 1, 0);

// c has value x = 9, y = 6, z = 0
c = a.Add(b);
```



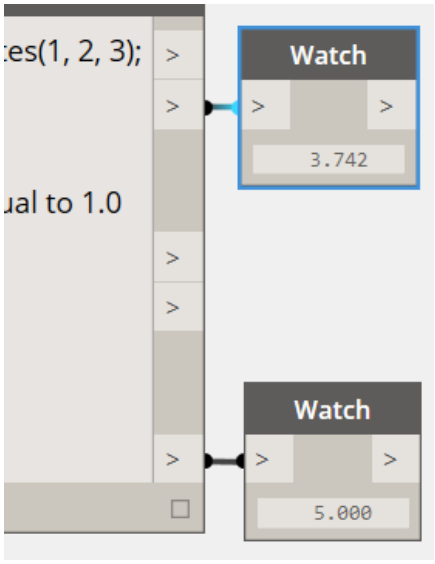
Similarly, two Vector objects can be subtracted from each other with the `Subtract` method. Vector subtraction can be thought of as the direction from first vector to the second vector.

```
a = Vector.ByCoordinates(5, 5, 0);
b = Vector.ByCoordinates(4, 1, 0);

// c has value x = 1, y = 4, z = 0
c = a.Subtract(b);
```



Vector multiplication can be thought of as moving the endpoint of a vector in its own direction by a given scale factor.

```
a = Vector.ByCoordinates(4, 4, 0);

// c has value x = 20, y = 20, z = 0
c = a.Scale(5);
```

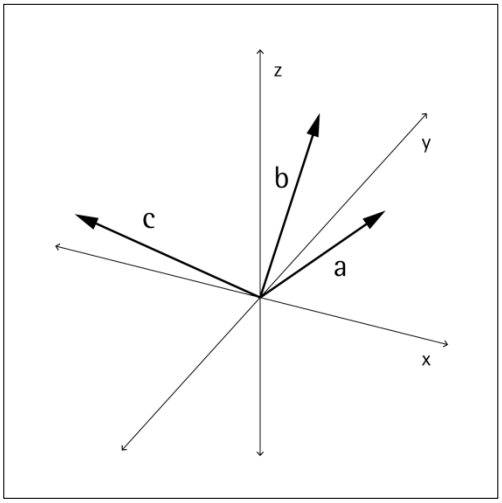


Often it's desired when scaling a vector to have the resulting vector's length *exactly* equal to the scaled amount. This is easily achieved by first normalizing a vector, in other words setting the vector's length exactly equal to one.

```
a = Vector.ByCoordinates(1, 2, 3);
a_len = a.Length;

// set the a's length equal to 1.0
b = a.Normalized();
c = b.Scale(5);

// len is equal to 5
len = c.Length;
```

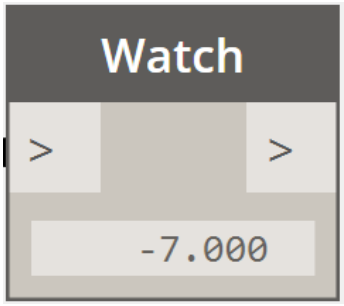
`c` still points in the same direction as `a` (1, 2, 3), though now it has length exactly equal to 5.

Two additional methods exist in vector math which don't have clear parallels with 1D math, the cross product and dot product. The cross product is a means of generating a Vector which is orthogonal (at 90 degrees to) to two existing Vectors. For example, the cross product of the x and y axes is the z axis, though the two input Vectors don't need to be orthogonal to each other. A cross product vector is calculated with the `Cross` method.




```
a = Vector.ByCoordinates(1, 0, 1);
b = Vector.ByCoordinates(0, 1, 1);

// c has value x = -1, y = -1, z = 1
c = a.Cross(b);
```

An additional, though somewhat more advanced function of vector math is the dot product. The dot product between two vectors is a real number (not a Vector object) that relates to, *but is not exactly*, the angle between two vectors. One useful properties of the dot product is that the dot product between two vectors will be 0 if and only if they are perpendicular. The dot product is calculated with the `Dot` method.




```
a = Vector.ByCoordinates(1, 2, 1);
b = Vector.ByCoordinates(5, -8, 4);

// d has value -7
d = a.Dot(b);
```

5: Range Expressions

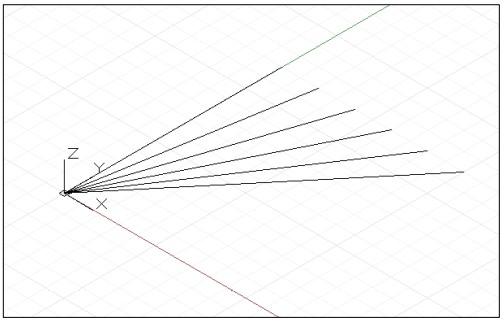
Almost every design involves repetitive elements, and explicitly typing out the names and constructors of every Point, Line, and other primitives in a script would be prohibitively time consuming. Range expressions give a Dynamo programmer the means to express sets of values as parameters on either side of two dots (`..`), generating intermediate numbers between these two extremes.

For instance, while we have seen variables containing a single number, it is possible with range expressions to have variables which contain a set of numbers. The simplest range expression fills in the whole number increments between the range start and end.


```
a = 1..6;
```

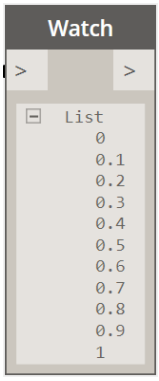
In previous examples, if a single number is passed in as the argument of a function, it would produce a single result. Similarly, if a range of values is passed in as the argument of a function, a range of values is returned.

For instance, if we pass a range of values into the Line constructor, Dynamo returns a range of lines.



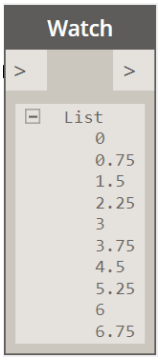

```
x_pos = 1..6;
y_pos = 5;
z_pos = 1;

lines = Line.ByStartPointEndPoint(Point.ByCoordinates(0,
    0, 0), Point.ByCoordinates(x_pos, y_pos, z_pos));
```

By default range expressions fill in the range between numbers incrementing by whole digit numbers, which can be useful for a quick topological sketch, but are less appropriate for actual designs. By adding a second ellipsis (`..`) to the range expression, you can specify the amount the range expression increments between values. Here we want all the numbers between 0 and 1, incrementing by 0.1:



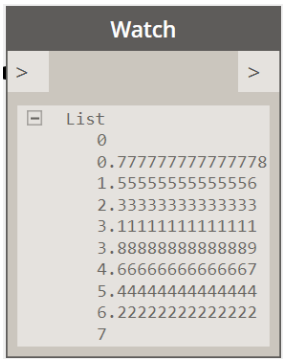

```
a = 0..1..0.1;
```

One problem that can arise when specifying the increment between range expression boundaries is that the numbers generated will not always fall on the final range value. For instance, if we create a range expression between 0 and 7, incrementing by 0.75, the following values are generated:



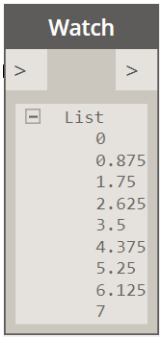

```
a = 0..7..0.75;
```

If a design requires a generated range expression to end precisely on the maximum range expression value, Dynamo can approximate an increment, coming as close as possible while still maintaining an equal distribution of numbers between the range boundaries. This is done with the approximate sign (~) before the third parameter:




```
// DesignScript will increment by 0.777 not 0.75
a = 0..7..~0.75;
```

However, if you want to Dynamo to interpolate between ranges with a discrete number of elements, the # operator allows you to specify this:

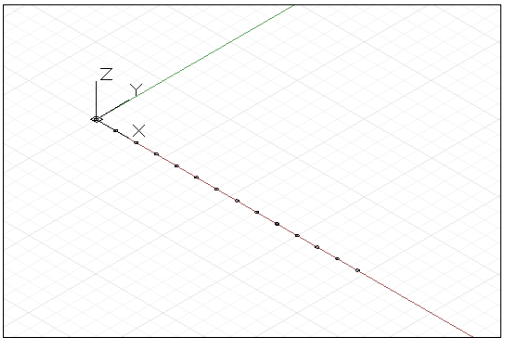



```
// Interpolate between 0 and 7 such that
// “a” will contain 9 elements
a = 0..7..#9;
```

6: Collections

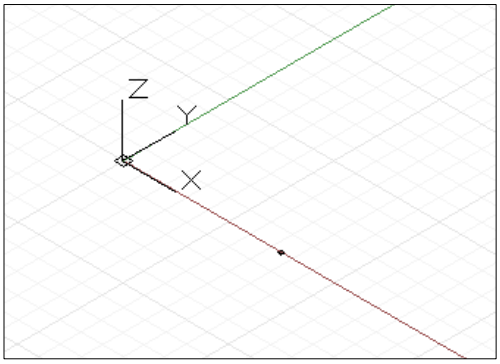
Collections are special types of variables which hold sets of values. For instance, a collection might contain the values 1 to 10, `{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}`, assorted geometry from the result of an Intersection operation, `{Surface, Point, Line, Point}`, or even a set of collections themselves, `{ {1, 2, 3}, {4, 5}, 6}`.

One of the easier ways to generate a collection is with range expressions (see: *Range Expressions*). Range expressions by default generate collections of numbers, though if these collections are passed into functions or constructors, collections of objects are returned.



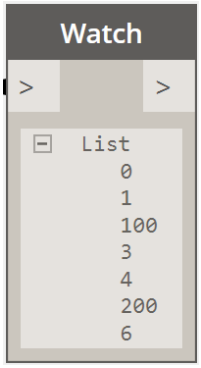

```
// use a range expression to generate a collection of
// numbers
nums = 0..10..0.75;

// use the collection of numbers to generate a
// collection of Points
points = Point.ByCoordinates(nums, 0, 0);
```

When range expressions aren't appropriate, collections can be created empty and manually filled with values. The square bracket operator (`[]`) is used to access members inside of a collection. The square brackets are written after the variable's name, with the number of the individual collection member contained inside. This number is called the collection member's index. For historical reasons, indexing starts at 0, meaning the first element of a collection is accessed with: `collection[0]`, and is often called the "zeroth" number. Subsequent members are accessed by increasing the index by one, for example:



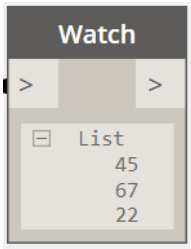

```
// a collection of numbers
nums = 0..10..0.75;

// create a single point with the 6th element of the
// collection
points = Point.ByCoordinates(nums[5], 0, 0);
```

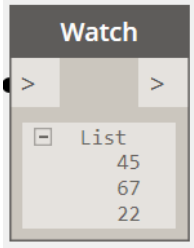
The individual members of a collection can be modified using the same index operator after the collection has been created:




```
// generate a collection of numbers
a = 0..6;

// change several of the elements of a collection
a[2] = 100;
a[5] = 200;
```

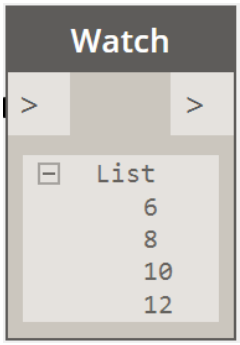
In fact, an entire collection can be created by explicitly setting every member of the collection individually. Explicit collections are created with the curly brace operator (`{}`) wrapping the collection's starting values, or left empty to create an empty collection:





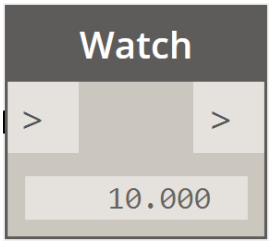

```
// create a collection explicitly
a = { 45, 67, 22 };

// create an empty collection
b = {};

// change several of the elements of a collection
b[0] = 45;
b[1] = 67;
b[2] = 22;
```

Collections can also be used as the indexes to generate new sub collections from a collection. For instance, a collection containing the numbers `{1, 3, 5, 7}`, when used as the index of a collection, would extract the 2nd, 4th, 6th, and 8th elements from a collection (remember that indices start at 0):




```
a = 5..20;

indices = {1, 3, 5, 7};

// create a collection via a collection of indices
b = a[indices];
```

Dynamo contains utility functions to help manage collections. The `Count` function, as the name implies, counts a collection and returns the number of elements it contains.

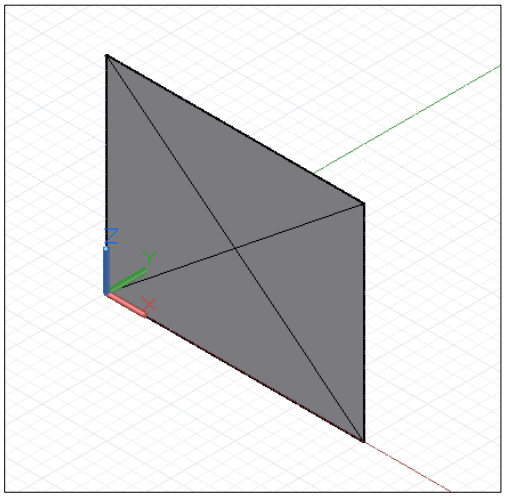



```
// create a collection with 10 elements
a = 1..10;

num_elements = Count(a);
```

7: Functions

Almost all the functionality demonstrated in DesignScript so far is expressed through functions. You can tell a command is a function when it contains a keyword suffixed by a parenthesis containing various inputs. When a function is called in DesignScript, a large amount of code is executed, processing the inputs and returning a result. The constructor function `Point.ByCoordinates(x : double, y : double, z : double)` takes three inputs, processes them, and returns a Point object. Like most programming languages, DesignScript gives programmers the ability to create their own functions. Functions are a crucial part of effective scripts: the process of taking blocks of code with specific functionality, wrapping them in a clear description of inputs and outputs adds both legibility to your code and makes it easier to modify and reuse.

Suppose a programmer had written a script to create a diagonal bracing on a surface:




```
p1 = Point.ByCoordinates(0, 0, 0);
p2 = Point.ByCoordinates(10, 0, 0);

l = Line.ByStartPointEndPoint(p1, p2);

// extrude a line vertically to create a surface
surf = l.Extrude(Vector.ByCoordinates(0, 0,
    1), 8);

// Extract the corner points of the surface
corner_1 = surf.PointAtParameter(0, 0);
corner_2 = surf.PointAtParameter(1, 0);
corner_3 = surf.PointAtParameter(1, 1);
corner_4 = surf.PointAtParameter(0, 1);

// connect opposite corner points to create diagonals
diag_1 = Line.ByStartPointEndPoint(corner_1, corner_3);
diag_2 = Line.ByStartPointEndPoint(corner_2, corner_4);
```

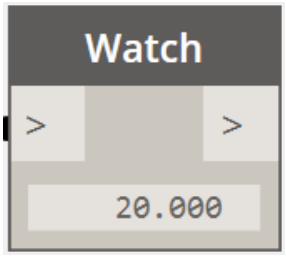
This simple act of creating diagonals over a surface nevertheless takes several lines of code. If we wanted to find the diagonals of hundreds, if not thousands of surfaces, a system of individually extracting corner points and drawing diagonals would be completely impractical. Creating a function to extract the

diagonals from a surface allows a programmer to apply the functionality of several lines of code to any number of base inputs.

Functions are created by writing the `def` keyword, followed by the function name, and a list of function inputs, called arguments, in parenthesis. The code which the function contains is enclosed inside curly braces: `{}`. In DesignScript, functions must return a value, indicated by “assigning” a value to the `return` keyword variable. E.g.

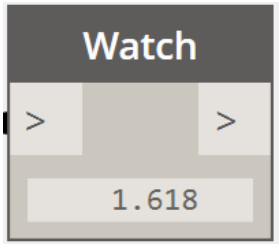
```
def functionName(argument1, argument2, etc, etc, . . .)
{
    // code goes here
    return = returnVariable;
}
```

This function takes a single argument and returns that argument multiplied by 2:




```
def getTimesTwo(arg)
{
    return = arg * 2;
}

times_two = getTimesTwo(10);
```

Functions do not necessarily need to take arguments. A simple function to return the golden ratio looks like this:

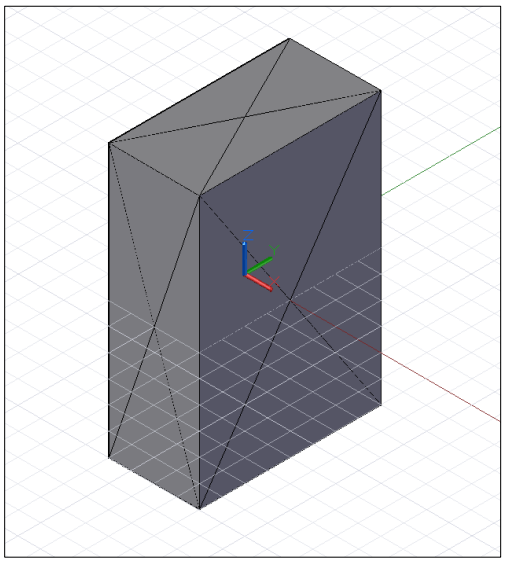



```
def getGoldenRatio()
{
    return = 1.61803399;
}

gr = getGoldenRatio();
```

Before we create a function to wrap our diagonal code, note that functions can only return a single value, yet our diagonal code generates two lines. To get around this issue, we can wrap two objects in curly braces, `{}`, creating a single collection object. For instance, here is a simple function which returns two values:


```
def returnTwoNumbers()
{
    return = {1, 2};
}

two_nums = returnTwoNumbers();
```

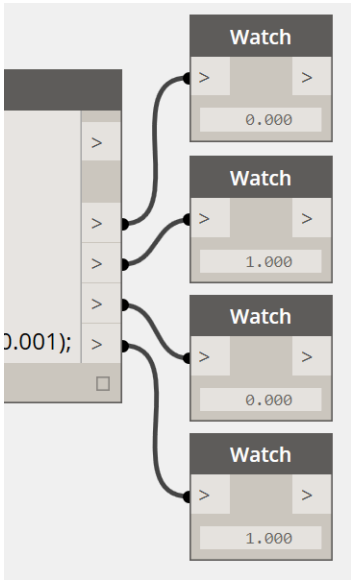
If we wrap the diagonal code in a function, we can create diagonals over a series of surfaces, for instance the faces of a cuboid.



```
def makeDiagonal(surface)
{
    corner_1 = surface.PointAtParameter(0, 0);
    corner_2 = surface.PointAtParameter(1, 0);
    corner_3 = surface.PointAtParameter(1, 1);
    corner_4 = surface.PointAtParameter(0, 1);

    diag_1 = Line.ByStartPointEndPoint(corner_1,
        corner_3);
    diag_2 = Line.ByStartPointEndPoint(corner_2,
        corner_4);

    return = {diag_1, diag_2};
}

c = Cuboid.ByLengths(CoordinateSystem.Identity(),
    10, 20, 30);

diags = makeDiagonal(c.Faces.SurfaceGeometry());
```

8: Math

The Dynamo standard library contains an assortment of mathematical functions to assist writing algorithms and manipulating data. Math functions are prefixed with the `Math` namespace, requiring you to append functions with "`Math.`" in order to use them.

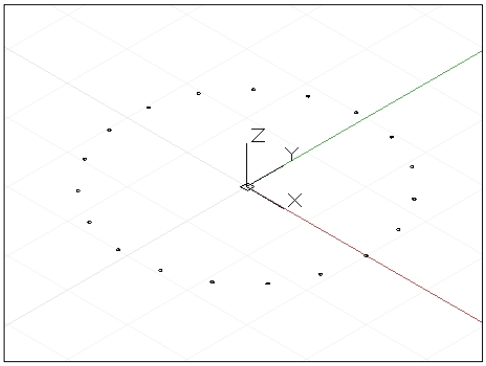


The functions `Floor`, `Ceiling`, and `Round` allow you to convert between floating point numbers and integers with predictable outcomes. All three functions take a single floating point number as input, though `Floor` returns an integer by always rounding down, `Ceiling` returns an integer by always rounding up, and `Round` rounds to the closest integer

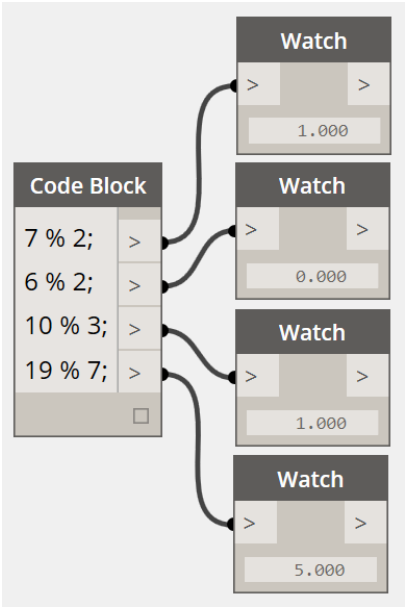
```
val = 0.5;

f = Math.Floor(val);
c = Math.Ceiling(val);
r = Math.Round(val);
r2 = Math.Round(val + 0.001);
```

Dynamo also contains a standard set of trigonometric functions to calculate the sine, cosine, tangent, arcsine, arccosine, and arctangent of angles, with the `Sin`, `Cos`, `Tan`, `Asin`, `Acos`, and `Atan` functions respectively.

While a comprehensive description of trigonometry is beyond the scope of this manual, the sine and cosine functions do frequently occur in computational designs due their ability to trace out positions on a circle with radius 1. By inputting an increasing degree angle, often labeled `theta`, into `Cos` for the `x` position, and `Sin` for the `y` position, the positions on a circle are calculated:




```
num_pts = 20;

// get degree values from 0 to 360
theta = 0..360..#num_pts;

p = Point.ByCoordinates(Math.Cos(theta),
    Math.Sin(theta), 0);
```



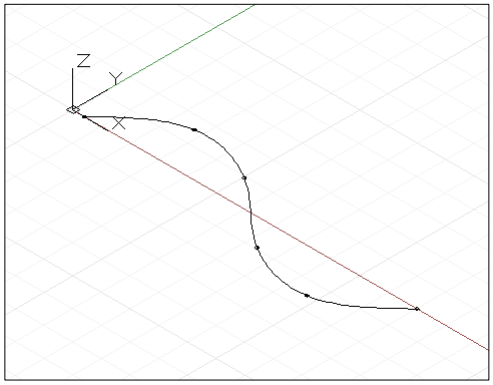
A related math concept not strictly part of the Math standard library is the modulus operator. The modulus operator, indicated by a percent (%) sign, returns the *remainder* from a division between two integer numbers. For instance, 7 divided by 2 is 3 with 1 left over (eg 2 x 3 + 1 = 7). The modulus between 7 and 2 therefore is 1. On the other hand, 2 divides evenly into 6, and therefore the modulus between 6 and 2 is 0. The following example illustrates the result of various modulus operations.

```
7 % 2;
6 % 2;
10 % 3;
19 % 7;
```

9: Curves: Interpreted and Control Points

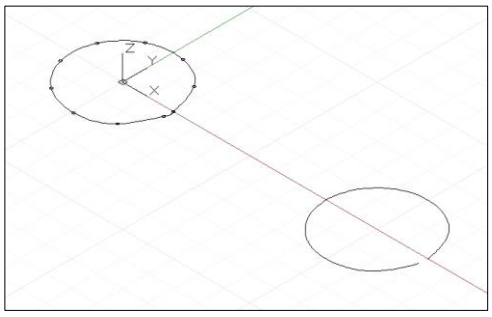
There are two fundamental ways to create free-form curves in Dynamo: specifying a collection of Points and having Dynamo interpret a smooth curve between them, or a more low-level method by specifying the underlying control points of a curve of a certain degree. Interpreted curves are useful when a designer knows exactly the form a line should take, or if the design has specific constraints for where the curve can and cannot pass through. Curves specified via control points are in essence a series of straight line segments which an algorithm smooths into a final curve form. Specifying a curve via control points can be useful for explorations of curve forms with varying degrees of smoothing, or when a smooth continuity between curve segments is required.

To create an interpreted curve, simply pass in a collection of Points to the `NurbsCurve.ByPoints` method.



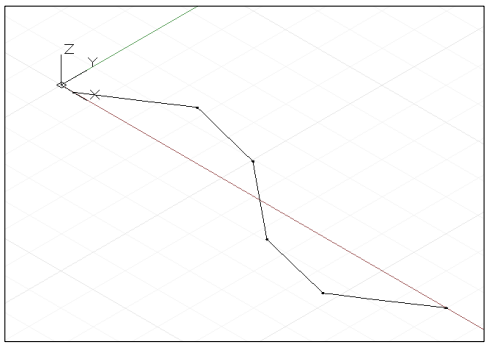

```
num_pts = 6;

s = Math.Sin(0..360..#num_pts) * 4;

pts = Point.ByCoordinates(1..30..#num_pts, s, 0);

int_curve = NurbsCurve.ByPoints(pts);
```

The generated curve intersects each of the input points, beginning and ending at the first and last point in the collection, respectively. An optional periodic parameter can be used to create a periodic curve which is closed. Dynamo will automatically fill in the missing segment, so a duplicate end point (identical to the start point) isn't needed.



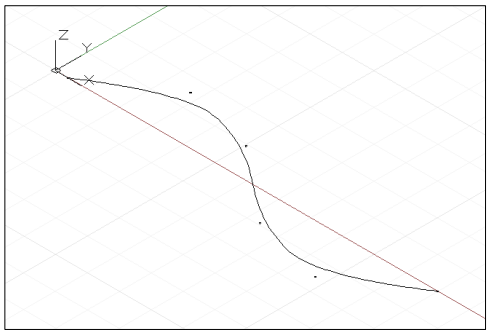

```
pts = Point.ByCoordinates(Math.Cos(0..350..#10),
    Math.Sin(0..350..#10), 0);

// create an closed curve
crv = NurbsCurve.ByPoints(pts, true);

// the same curve, if left open:
crv2 = NurbsCurve.ByPoints(pts.Translate(5, 0, 0),
    false);
```

NurbsCurves are generated in much the same way, with input points represent the endpoints of a straight line segment, and a second parameter specifying the amount and type of smoothing the curve undergoes, called the degree.* A curve with degree 1 has no smoothing; it is a polyline.



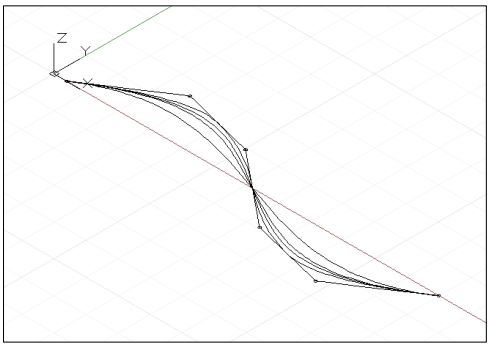

```
num_pts = 6;

pts = Point.ByCoordinates(1..30..#num_pts,
    Math.Sin(0..360..#num_pts) * 4, 0);

// a B-Spline curve with degree 1 is a polyline
ctrl_curve = NurbsCurve.ByControlPoints(pts, 1);
```

A curve with degree 2 is smoothed such that the curve intersects and is tangent to the midpoint of the polyline segments:




```
num_pts = 6;

pts = Point.ByCoordinates(1..30..#num_pts,
    Math.Sin(0..360..#num_pts) * 4, 0);

// a B-Spline curve with degree 2 is smooth
ctrl_curve = NurbsCurve.ByControlPoints(pts, 2);
```

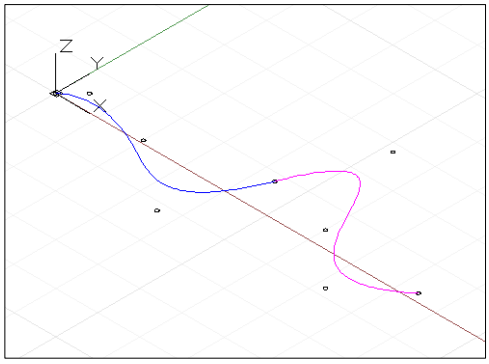
Dynamo supports NURBS (Non-uniform rational B-spline) curves up to degree 20, and the following script illustrates the effect increasing levels of smoothing has on the shape of a curve:




```
num_pts = 6;

pts = Point.ByCoordinates(1..30..#num_pts,
    Math.Sin(0..360..#num_pts) * 4, 0);

def create_curve(pts : Point[], degree : int)
{
       return = NurbsCurve.ByControlPoints(pts,
         degree);
}

ctrl_crvs = create_curve(pts, 1..11);
```

Note that you must have at least one more control point than the degree of the curve.

Another benefit of constructing curves by control vertices is the ability to maintain tangency between individual curve segments. This is done by extracting the direction between the last two control points, and continuing this direction with the first two control points of the following curve. The following example creates two separate NURBS curves which are nevertheless as smooth as one curve:



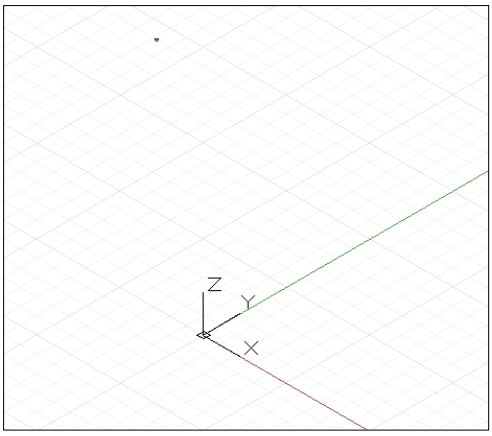

```
pts_1 = {};

pts_1[0] = Point.ByCoordinates(0, 0, 0);
pts_1[1] = Point.ByCoordinates(1, 1, 0);
pts_1[2] = Point.ByCoordinates(5, 0.2, 0);
pts_1[3] = Point.ByCoordinates(9, -3, 0);
pts_1[4] = Point.ByCoordinates(11, 2, 0);

crv_1 = NurbsCurve.ByControlPoints(pts_1, 3);

pts_2 = {};

pts_2[0] = pts_1[4];
end_dir = pts_1[4].Subtract(pts_1[3].AsVector());

pts_2[1] = Point.ByCoordinates(pts_2[0].X + end_dir.X,
    pts_2[0].Y + end_dir.Y, pts_2[0].Z + end_dir.Z);

pts_2[2] = Point.ByCoordinates(15, 1, 0);
pts_2[3] = Point.ByCoordinates(18, -2, 0);
pts_2[4] = Point.ByCoordinates(21, 0.5, 0);

crv_2 = NurbsCurve.ByControlPoints(pts_2, 3);
```

* This is a very simplified description of NURBS curve geometry, for a more accurate and detailed discussion see *Pottmann, et al*, 2007, in the references.

10: Translation, Rotation, and Other Transformations

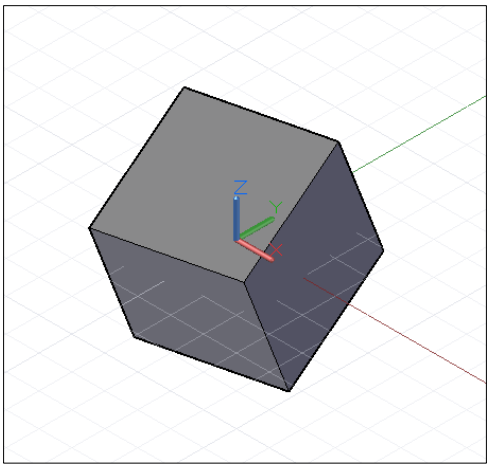
Certain geometry objects can be created by explicitly stating x, y, and z coordinates in three-dimensional space. More often, however, geometry is moved into its final position using geometric transformations on the object itself or on its underlying CoordinateSystem.

The simplest geometric transformation is a translation, which moves an object a specified number of units in the x, y, and z directions.



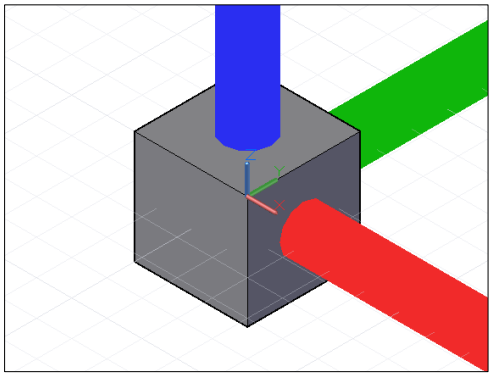

```
// create a point at x = 1, y = 2, z = 3
p = Point.ByCoordinates(1, 2, 3);

// translate the point 10 units in the x direction,
// -20 in y, and 50 in z
// p2's new position is x = 11, y = -18, z = 53
p2 = p.Translate(10, -20, 50);
```

While all objects in Dynamo can be translated by appending the `.Translate` method to the end of the object's name, more complex transformations require transforming the object from one underlying CoordinateSystem to a new CoordinateSystem. For instance, to rotate an object 45 degrees around the x axis, we would transform the object from its existing CoordinateSystem with no rotation, to a CoordinateSystem which had been rotated 45 degrees around the x axis with the `.Transform` method:



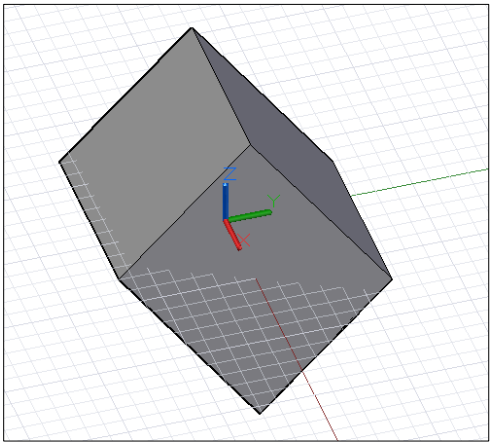

```
cube = Cuboid.ByLengths(CoordinateSystem.Identity(),
    10, 10, 10);

new_cs = CoordinateSystem.Identity();
new_cs2 = new_cs.Rotate(Point.ByCoordinates(0, 0),
    Vector.ByCoordinates(1,0,0.5), 25);

// get the existing coordinate system of the cube
old_cs = CoordinateSystem.Identity();

cube2 = cube.Transform(old_cs, new_cs2);
```

In addition to being translated and rotated, CoordinateSystems can also be created scaled or sheared. A CoordinateSystem can be scaled with the `.Scale` method:



```
cube = Cuboid.ByLengths(CoordinateSystem.Identity(),
    10, 10, 10);

new_cs = CoordinateSystem.Identity();
new_cs2 = new_cs.Scale(20);

old_cs = CoordinateSystem.Identity();

cube = cube.Transform(old_cs, new_cs2);
```

Sheared CoordinateSystems are created by inputting non-orthogonal vectors into the CoordinateSystem constructor.



```
new_cs = CoordinateSystem.ByOriginVectors(
    Point.ByCoordinates(0, 0, 0),
     Vector.ByCoordinates(-1, -1, 1),
     Vector.ByCoordinates(-0.4, 0, 0));

old_cs = CoordinateSystem.Identity();

cube = Cuboid.ByLengths(CoordinateSystem.Identity(),
    5, 5, 5);

new_curves = cube.Transform(old_cs, new_cs);
```

Scaling and shearing are comparatively more complex geometric transformations than rotation and translation, so not every Dynamo object can undergo these transformations. The following table outlines which Dynamo objects can have non-uniformly scaled CoordinateSystems, and sheared CoordinateSystems.

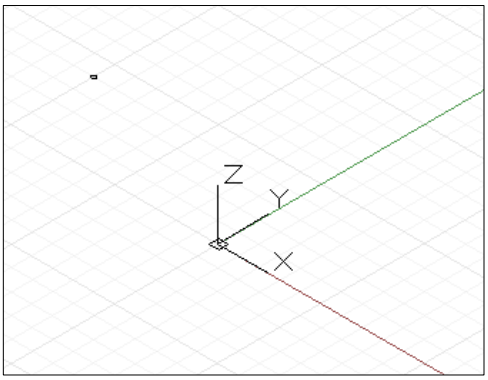
Class	Non-Uniformly Scaled CoordinateSystem	Sheared CoordinateSystem
Arc	No	No
NurbsCurve	Yes	Yes
NurbsSurface	No	No
Circle	No	No
Line	Yes	Yes
Plane	No	No
Point	Yes	Yes
Polygon	No	No
Solid	No	No
Surface	No	No
Text	No	No

11: Conditionals and Boolean Logic

One of the most powerful features of a programming language is the ability to look at the existing objects in a program and vary the program's execution according to these objects' qualities. Programming languages mediate between examinations of an object's qualities and the execution of specific code via a system called Boolean logic.

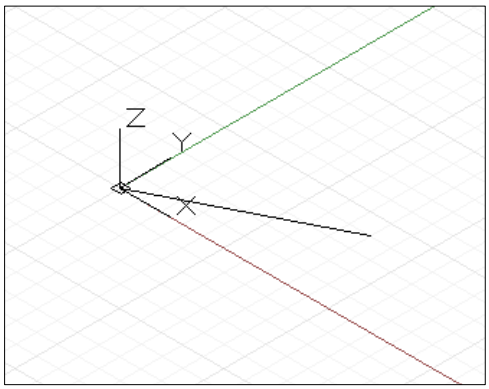
Boolean logic examines whether statements are true or false. Every statement in Boolean logic will be either true or false, there are no other states; no maybe, possible, or perhaps exist. The simplest way to indicate that a Boolean statement is true is with the **`true`** keyword. Similarly, the simplest way to indicate a statement is false is with the **`false`** keyword. The `if` statement allows you to determine if a statement is true of false: if it is true, the first part of the code block executes, if it's false, the second code block executes.

In the following example, the `if` statement contains a **`true`** Boolean statement, so the first block executes and a Point is generated:




```
geometry = [Imperative]
{
    if (true)
    {
        return = Point.ByCoordinates(1, -4, 6);
    }
    else
    {
        return = Line.ByStartPointEndPoint(
            Point.ByCoordinates(0, 0, 0),
            Point.ByCoordinates(10, -4, 6));
    }
}
```

If the contained statement is changed to **`false`**, the second code block executes and a Line is generated:



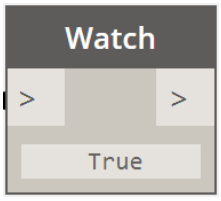

```
geometry = [Imperative]
{
    // change true to false
    if (false)
    {
        return = Point.ByCoordinates(1, -4, 6);
    }
    else
    {
         return = Line.ByStartPointEndPoint(
             Point.ByCoordinates(0, 0, 0),
             Point.ByCoordinates(10, -4, 6));
    }
}
```

Static Boolean statements like these aren't particularly useful; the power of Boolean logic comes from examining the qualities of objects in your script. Boolean logic has six basic operations to evaluate values: less than (<), greater than (>), less than or equal (<=), greater than or equal (>=), equal (==), and not equal (!=). The following chart outlines the Boolean results

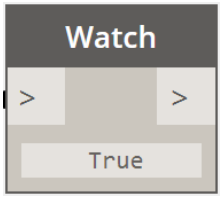
<	Returns true if number on left side is less than number on right side.
>	Returns true if number on left side is greater than number on right side.
<=	Returns true of number on left side is less than or equal to the number on the right side.*
>=	Returns true of number on the left side is greater than or equal to the number on the right side.*
==	Returns true if both numbers are equal*
!=	Returns true if both number are not equal*

* see chapter "Number Types" for limitations of testing equality between two floating point numbers.

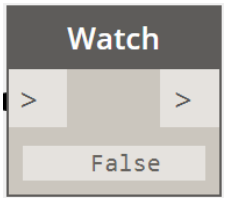
Using one of these six operators on two numbers returns either **`true`** or **`false`**:




```
result = 10 < 30;
```



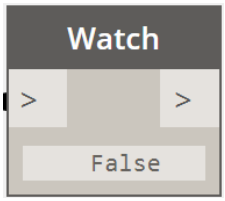

```
result = 15 <= 15;
```



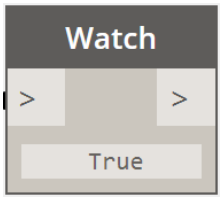

```
result = 99 != 99;
```

Three other Boolean operators exist to compare **`true`** and **`false`** statements: and (`&&`), or (`||`), and not (`!`).

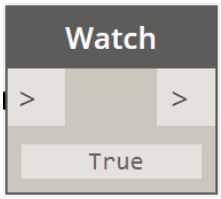
&&	Returns true if the values on both sides are true.
||	Returns true if either of the values on both sides are true.
!	Returns the Boolean opposite




```
result = true && false;
```

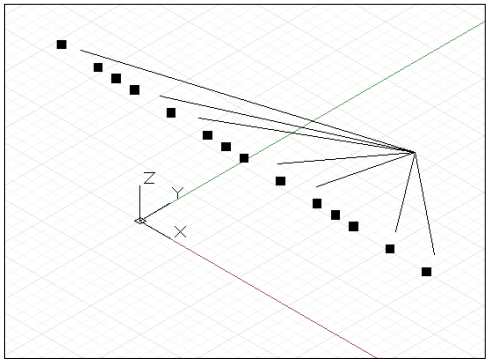



```
result = true || false;
```




```
result = !false;
```

Refactoring the code in the original example demonstrates different code execution paths based on the changing inputs from a range expression:

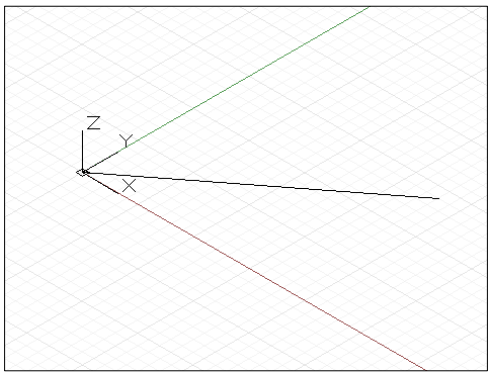



```
def make_geometry(i)
{
    return = [Imperative]
    {
        // test if the input is divisible
        // by either 2 or 3. See "Math"
        if (i % 2 == 0 || i % 3 == 0)
        {
            return = Point.ByCoordinates(i, -4, 10);
        }
        else
        {
            return = Line.ByStartPointEndPoint(
                Point.ByCoordinates(4, 10, 0),
                Point.ByCoordinates(i, -4, 10));
        }
    }
}

g = make_geometry(0..20);
```

12: Looping

Loops are commands to repeat execution over a block of code. The number of times a loop is called can be governed by a collection, where a loop is called with each element of the collection as input, or with a Boolean expression, where the loop is called until the Boolean expression returns **`false`**. Loops can be used to generate collections, search for a solution, or otherwise add repetition without range expressions.

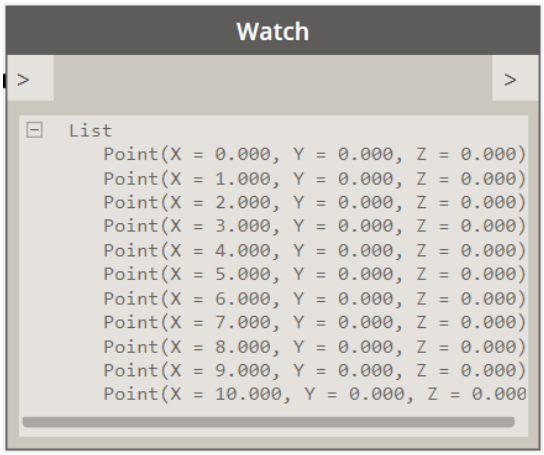
The `while` statement evaluates a Boolean expression, and continues re-executing the contained code block until the Boolean expression is **`true`**. For instance, this script continuously creates and re-creates a line until it has length greater than 10:




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

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

    return = line;
}
```

In associative Dynamo code, if a collection of elements is used as the input to a function which normally takes a single value, the function is called individually for each member of a collection. In imperative Dynamo code a programmer has the option to write code that manually iterates over the collection, extracting individual collection members one at a time.

The `for` statement extracts elements from a collection into a named variable, once for each member of a collection. The syntax for `for` is: `for`(“extracted variable” `in` “input collection”)

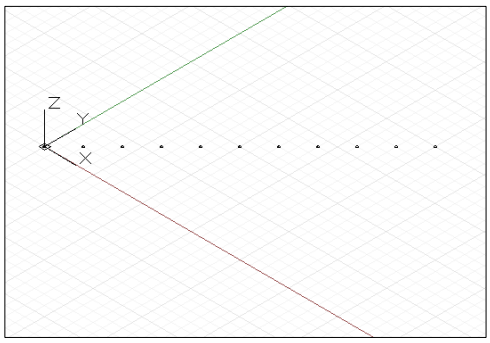



```
geometry = [Imperative]
{
    collection = 0..10;
    points = {};

    for (i in collection)
    {
        points[i] = Point.ByCoordinates(i, 0, 0);
    }

    return = points;
}
```

13: Replication Guides

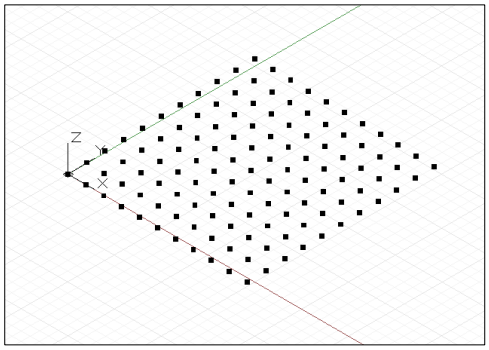
The Dynamo language was created as a domain-specific tool for architects, designers and engineers, and as such has several language features specifically tailored for these disciplines. A common element in these disciplines is the prevalence of objects arrayed repetitive grids, from brick walls and tile floors to façade paneling and column grids. While range expressions offer a convenient means of generating one dimensional collections of elements, replication guides offer a convenient means of generating two and three dimensional collections.

Replication guides take two or three one-dimensional collections, and pair the elements together to generate one, two- or three-dimensional collection. Replication guides are indicated by placing the symbols `<1>`, `<2>`, or `<3>` after a two or three collections on a single line of code. For example, we can use range expressions to generate two one-dimensional collections, and use these collections to generate a collection of points:




```
x_vals = 0..10;
y_vals = 0..10;

p = Point.ByCoordinates(x_vals, y_vals, 0);
```

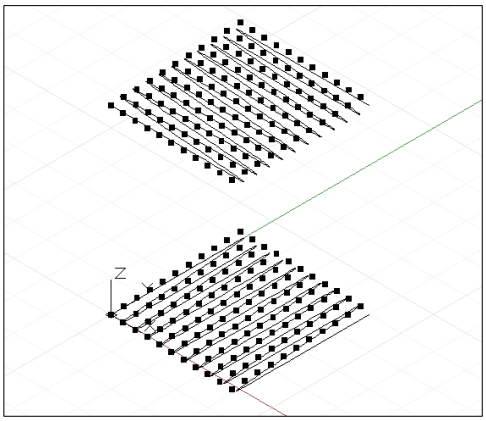
In this example, the first element of `x_vals` is paired with the first element of `y_vals`, the second with the second, and so on for the entire length of the collection. This generates points with values (0, 0, 0), (1, 1, 0), (2, 2, 0), (3, 3, 0), etc…. If we apply a replication guide to this same line of code, we can have Dynamo generate a two-dimensional grid from the two one dimensional collections:




```
x_vals = 0..10;
y_vals = 0..10;

// apply replication guides to the two collections
p = Point.ByCoordinates(x_vals<1>, y_vals<2>, 0);
```

By applying replication guides to `x_vals` and `y_vals`, Dynamo generates every possible combination of values between the two collections, first pairing the 1st element `x_vals` with all the elements in `y_vals`, then pairing the 2nd element of `x_vals` with all the elements in `y_vals`, and so on for every element `of x_vals`.

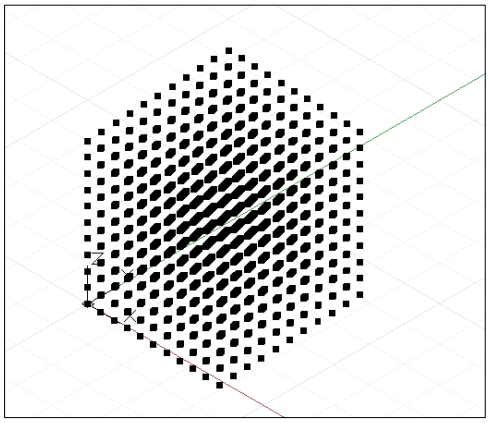
The order of the replication guide numbers (`<1>`, `<2>`, and/or `<3>`) determines the order of the underlying collection. In the following example, the same two one-dimensional collections are used to form two two-dimensional collections, though with the order of `<1>` and `<2>` swapped.




```
x_vals = 0..10;
y_vals = 0..10;

p1 = Point.ByCoordinates(x_vals<1>, y_vals<2>, 0);

// apply the replication guides with a swapped order
// and set the points 14 units higher
p2 = Point.ByCoordinates(x_vals<2>, y_vals<1>, 14);

curve1 = NurbsCurve.ByPoints(Flatten(p1));
curve2 = NurbsCurve.ByPoints(Flatten(p2));
```

`curve1` and `curve2` trace out the generated order of elements in both arrays; notice that they are rotated 90 degrees to each other. p1 was created by extracting elements of `x_vals` and pairing them with `y_vals`, while p2 was created by extracting elements of `y_vals` and pairing them with `x_vals`.

Replication guides also work in three dimensions, by pairing a third collection with a third replication symbol, `<3>`.



```
x_vals = 0..10;
y_vals = 0..10;
z_vals = 0..10;

// generate a 3D matrix of points
p = Point.ByCoordinates(x_vals<1>,y_vals<2>,z_vals<3>);
```

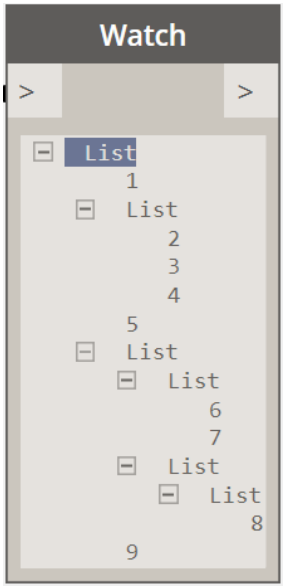
This generates every possible combination of values from combining the elements from `x_vals`, `y_vals`, and `z_vals`.

14: Collection Rank and Jagged Collections

The rank of a collection is defined as the greatest depth of elements inside of a collection. A collection of single values has a rank of 1, while a collection of collections of single values has a rank of 2. Rank can loosely be defined as the number of square bracket ([ ]) operators needed to access the deepest member of a collection. Collections of rank 1 only need a single square bracket to access the deepest member, while collections of rank three require three subsequent brackets. The following table outlines collections of ranks 1-3, though collections can exist up to any rank depth.

Rank	Collection	Access 1st Element
1	`{1, 2, 3, 4, 5}`	`collection[0]`
2	`{ {1, 2}, {3, 4}, {5, 6} }`	`collection[0][0]`
3	`{ { {1, 2}, {3, 4} },` `  { {5, 6}, {7, 8} } }`	`collection[0][0][0]`
...		

Higher ranked collections generated by range expressions and replication guides are always homogeneous, in other words every object of a collection is at the same depth (it is accessed with the same number of [ ] operators). However, not all Dynamo collections contain elements at the same depth. These collections are called jagged, after the fact that the depth rises up and down over the length of the collection. The following code generates a jagged collection:




```
j = {};

j[0] = 1;
j[1] = {2, 3, 4};
j[2] = 5;
j[3] = { {6, 7}, { {8} } };
j[4] = 9;
```

fail when it attempts to perform operations not supported on a collection.

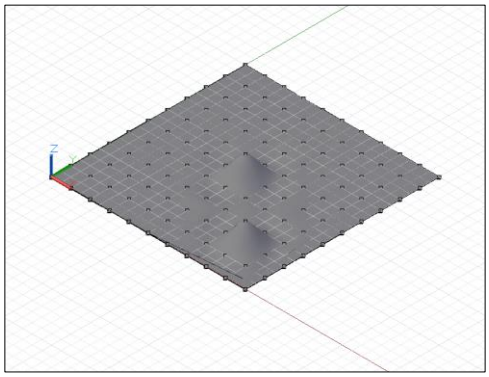
The following example shows how to access all the elements of this jagged collection:


```
// generate a jagged collection
j = {1, {2, 3, 4}, 5, {{6, 7}, {{8}}}, 9};

s = j[0] + " " + j[1][0] + " " + j[1][1] + " " +
    j[1][2] + " " + j[2] + " " +
    j[3][0][0]  + " " + j[3][0][1] + " " +
    j[3][1][0][0] + " " + j[4];
```

15: Surfaces: Interpreted, Control Points, Loft, Revolve

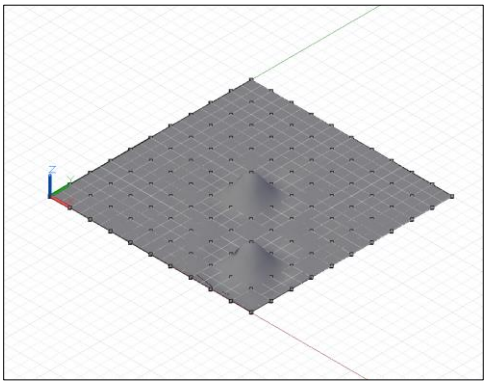
The two-dimensional analog to a NurbsCurve is the NurbsSurface, and like the freeform NurbsCurve, NurbsSurfaces can be constructed with two basic methods: inputting a set of base points and having Dynamo interpret between them, and explicitly specifying the control points of the surface. Also like freeform curves, interpreted surfaces are useful when a designer knows precisely the shape a surface needs to take, or if a design requires the surface to pass through constraint points. On the other hand, Surfaces created by control points can be more useful for exploratory designs across various smoothing levels.

To create an interpreted surface, simply generate a two-dimensional collection of points approximating the shape of a surface. The collection must be rectangular, that is, not jagged. The method `NurbsSurface.ByPoints` constructs a surface from these points.



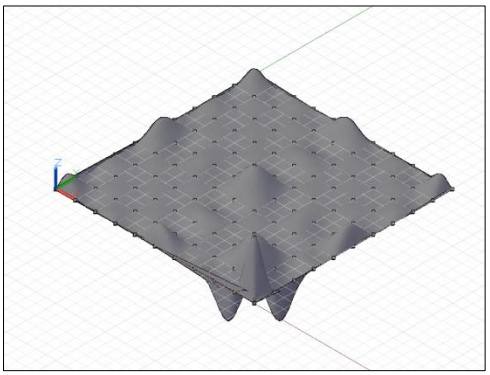
```
// python_points_1 is a set of Points generated with
// a Python script found in Appendix 1

surf = NurbsSurface.ByPoints(python_points_1);
```

Freeform NurbsSurfaces can also be created by specifying underlying control points of a surface. Like NurbsCurves, the control points can be thought of as representing a quadrilateral mesh with straight segments, which, depending on the degree of the surface, is smoothed into the final surface form. To create a NurbsSurface by control points, include two additional parameters to `NurbsSurface.ByPoints`, indicating the degrees of the underlying curves in both directions of the surface.



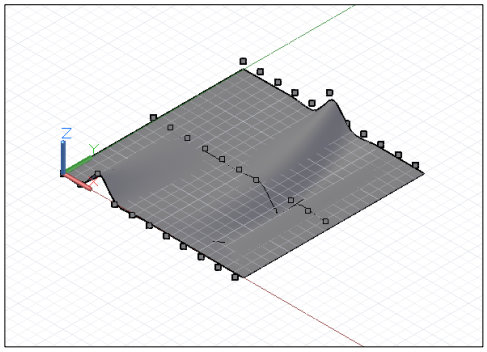
```
// python_points_1 is a set of Points generated with
// a Python script found in Appendix 1

// create a surface of degree 2 with smooth segments
surf = NurbsSurface.ByPoints(python_points_1, 2, 2);
```

We can increase the degree of the NurbsSurface to change the resulting surface geometry:



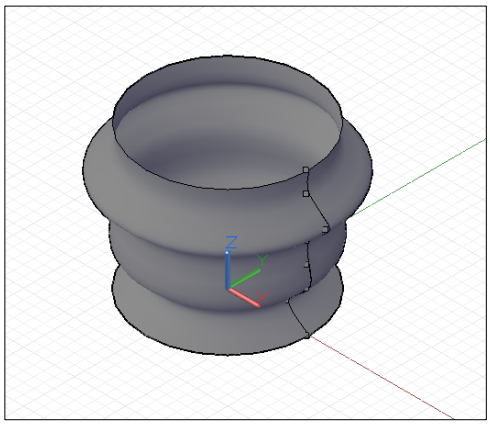
```
// python_points_1 is a set of Points generated with
// a Python script found in Appendix 1

// create a surface of degree 6
surf = NurbsSurface.ByPoints(python_points_1, 6, 6);
```

Just as Surfaces can be created by interpreting between a set of input points, they can be created by interpreting between a set of base curves. This is called lofting. A lofted curve is created using the `Surface.LoftFromCrossSections` constructor, with a collection of input curves as the only parameter.



```
// python_points_2, 3, and 4 are generated with
// Python scripts found in Appendix 1

c1 = NurbsCurve.ByPoints(python_points_2);
c2 = NurbsCurve.ByPoints(python_points_3);
c3 = NurbsCurve.ByPoints(python_points_4);

loft = Surface.LoftFromCrossSections({c1, c2, c3});
```

Surfaces of revolution are an additional type of surface created by sweeping a base curve around a central axis. If interpreted surfaces are the two-dimensional analog to interpreted curves, then surfaces of revolution are the two-dimensional analog to circles and arcs. Surfaces of revolution are specified by a base curve, representing the “edge” of the surface; an axis origin, the base point of the surface; an axis direction, the central “core” direction; a sweep start angle; and a sweep end angle. These are used as the input to the `Surface.Revolve` constructor.



```
pts = {};
pts[0] = Point.ByCoordinates(4, 0, 0);
pts[1] = Point.ByCoordinates(3, 0, 1);
pts[2] = Point.ByCoordinates(4, 0, 2);
pts[3] = Point.ByCoordinates(4, 0, 3);
pts[4] = Point.ByCoordinates(4, 0, 4);
pts[5] = Point.ByCoordinates(5, 0, 5);
pts[6] = Point.ByCoordinates(4, 0, 6);
pts[7] = Point.ByCoordinates(4, 0, 7);

crv = NurbsCurve.ByPoints(pts);

axis_origin = Point.ByCoordinates(0, 0, 0);
axis = Vector.ByCoordinates(0, 0, 1);

surf = Surface.ByRevolve(crv, axis_origin, axis, 0,
    360);
```

16: Geometric Parameterization

In computational designs, curves and surfaces are frequently used as the underlying scaffold to construct subsequent geometry. In order for this early geometry to be used as a foundation for later geometry, the script must be able to extract qualities such as position and orientation across the entire area of the object. Both curves and surfaces support this extraction, and it is called parameterization.

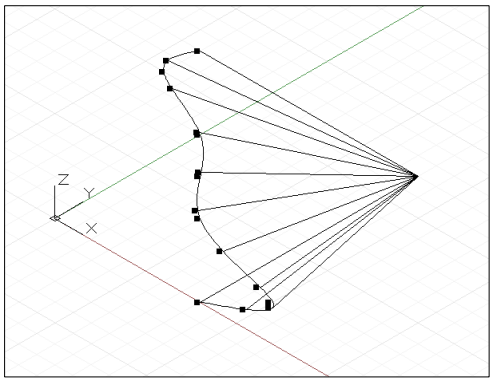
All of the points on a curve can be thought of as having a unique parameter ranging from 0 to 1. If we were to create a NurbsCurve based off of several control or interpreted points, the first point would have the parameter 0, and the last point would have the parameter 1. It's impossible to know in advance what the exact parameter is any intermediate point is, which may sound like a severe limitation though is mitigated by a series of utility functions. Surfaces have a similar parameterization as curves, though with two parameters instead of one, called *u* and *v*. If we were to create a surface with the following points:

```
pts = { {p1, p2, p3},
        {p4, p5, p6},
        {p7, p8, p9} };
```

`p1` would have parameter *u* = 0 *v* = 0, while `p9` would have parameters *u* = 1 *v* = 1.

Parameterization isn't particularly useful when determining points used to generate curves, its main use is to determine the locations if intermediate points generated by NurbsCurve and NurbsSurface constructors.

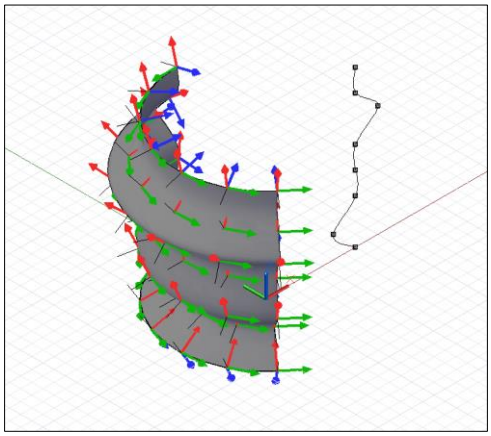
Curves have a method `PointAtParameter`, which takes a single double argument between 0 and 1, and returns the Point object at that parameter. For instance, this script finds the Points at parameters 0, .1, .2, .3, .4, .5, .6, .7, .8, .9, and 1:




```
pts = {};
pts[0] = Point.ByCoordinates(4, 0, 0);
pts[1] = Point.ByCoordinates(6, 0, 1);
pts[2] = Point.ByCoordinates(4, 0, 2);
pts[3] = Point.ByCoordinates(4, 0, 3);
pts[4] = Point.ByCoordinates(4, 0, 4);
pts[5] = Point.ByCoordinates(3, 0, 5);
pts[6] = Point.ByCoordinates(4, 0, 6);

crv = NurbsCurve.ByPoints(pts);

pts_at_param = crv.PointAtParameter(0..1..#11);

// draw Lines to help visualize the points
lines = Line.ByStartPointEndPoint(pts_at_param,
    Point.ByCoordinates(4, 6, 0));
```

Similarly, Surfaces have a method PointAtParameter which takes two arguments, the *u* and *v* parameter of the generated Point.

While extracting individual points on a curve and surface can be useful, scripts often require knowing the particular geometric characteristics at a parameter, such as what direction the Curve or Surface is facing. The methods `CoordinateSystemAtParameterAlongCurve` and `CoordinateSystemAtParameter` find not only the position but an oriented CoordinateSystem at the parameter of a Curve and Surface respectively. For instance, the following script extracts oriented CoordinateSystems along a revolved Surface, and uses the orientation of the CoordinateSystems to generate lines which are sticking off normal to the surface:



```
pts = {};
pts[0] = Point.ByCoordinates(4, 0, 0);
pts[1] = Point.ByCoordinates(3, 0, 1);
pts[2] = Point.ByCoordinates(4, 0, 2);
pts[3] = Point.ByCoordinates(4, 0, 3);
pts[4] = Point.ByCoordinates(4, 0, 4);
pts[5] = Point.ByCoordinates(5, 0, 5);
pts[6] = Point.ByCoordinates(4, 0, 6);
pts[7] = Point.ByCoordinates(4, 0, 7);

crv = NurbsCurve.ByPoints(pts);

axis_origin = Point.ByCoordinates(0, 0, 0);
axis = Vector.ByCoordinates(0, 0, 1);

surf = Surface.ByRevolve(crv, axis_origin, axis, 90,
    140);

cs_array = surf.CoordinateSystemAtParameter(
    (0..1..#7)<1>, (0..1..#7)<2>);

def make_line(cs : CoordinateSystem) {
    lines_start = cs.Origin;
    lines_end = cs.Origin.Translate(cs.ZAxis, -0.75);

    return = Line.ByStartPointEndPoint(lines_start,
        lines_end);
}

lines = make_line(Flatten(cs_array));
```

As mentioned earlier, parameterization is not always uniform across the length of a Curve or a Surface, meaning that the parameter 0.5 doesn't always correspond to the midpoint, and 0.25 doesn't always correspond to the point one quarter along a curve or surface. To get around this limitation, Curves have an additional set of parameterization commands which allow you to find a point at specific lengths along a Curve.

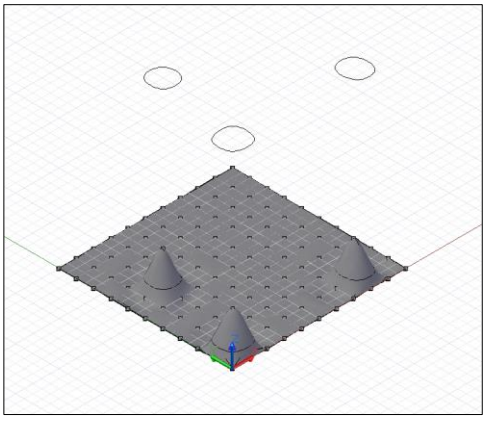
17: Intersection and Trim

Many of the examples so far have focused on the construction of higher dimensional geometry from lower dimensional objects. Intersection methods allow this higher dimensional geometry to generate lower dimensional objects, while the trim and select trim commands allow script to heavily modify geometric forms after they've been created.

The `Intersect` method is defined on all pieces of geometry in Dynamo, meaning that in theory any piece of geometry can be intersected with any other piece of geometry. Naturally some intersections are meaningless, such as intersections involving Points, as the resulting object will always be the input Point itself. The other possible combinations of intersections between objects are outlined in the following chart. The following chart outlines the result of various intersection operations:

	Intersect:			
With:	**Surface**	**Curve**	**Plane**	**Solid**
Surface	Curve	Point	Point, Curve	Surface
Curve	Point	Point	Point	Curve
Plane	Curve	Point	Curve	Curve
Solid	Surface	Curve	Curve	Solid

The following very simple example demonstrates the intersection of a plane with a NurbsSurface. The intersection generates a NurbsCurve array, which can be used like any other NurbsCurve.

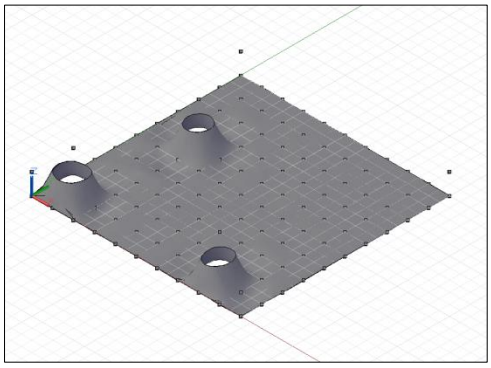


```
// python_points_5 is a set of Points generated with
// a Python script found in Appendix 1

surf = NurbsSurface.ByPoints(python_points_5, 3, 3);

WCS = CoordinateSystem.Identity();

pl = Plane.ByOriginNormal(WCS.Origin.Translate(0, 0,
    0.5), WCS.ZAxis);

// intersect surface, generating three closed curves
crvs = surf.Intersect(pl);

crvs_moved = crvs.Translate(0, 0, 10);
```

The `Trim` method is very similar to the Intersect method, in that it is defined for almost every piece of geometry. Unlike intersect, there are far more limitations on `Trim` than on `Intersect`

	Trim Using:				
On:	**Point**	**Curve**	**Plane**	**Surface**	**Solid**
Curve	Yes	No	No	No	No
Polygon	NA	No	Yes	No	No
Surface	NA	Yes	Yes	Yes	Yes
Solid	NA	NA	Yes	Yes	Yes

Something to note about `Trim` methods is the requirement of a “select” point, a point which determines which geometry to discard, and which pieces to keep. Dynamo finds the closest side of the trimmed geometry to the select point, and this side becomes the side to discard.

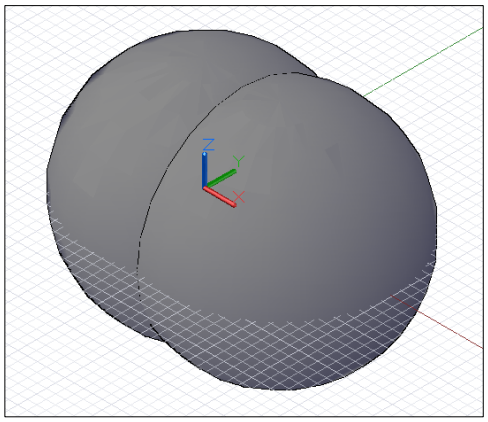


```
// python_points_5 is a set of Points generated with
// a Python script found in Appendix 1

surf = NurbsSurface.ByPoints(python_points_5, 3, 3);

tool_pts = Point.ByCoordinates((-10..20..10)<1>,
    (-10..20..10)<2>, 1);

tool = NurbsSurface.ByPoints(tool_pts);

pick_point = Point.ByCoordinates(8, 1, 3);

// trim with the tool surface, and keep the surface
// closest to pick_point
result = surf.Trim(tool, pick_point);
```

18: Geometric Booleans

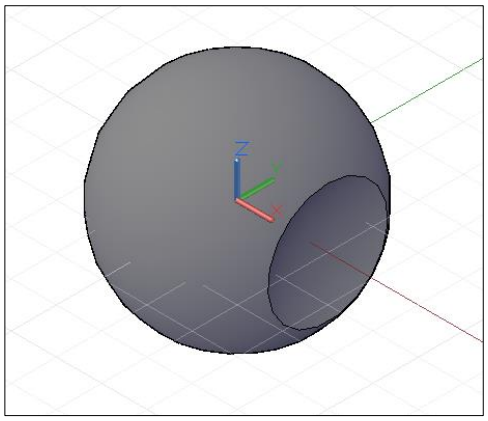
`Intersect`, `Trim`, and `SelectTrim` are primarily used on lower-dimensional geometry such as Points, Curves, and Surfaces. Solid geometry on the other hand, has an additional set of methods for modifying form after their construction, both by subtracting material in a manner similar to `Trim` and combining elements together to form a larger whole.

The `Union` method takes two solid objects and creates a single solid object out of the space covered by both objects. The overlapping space between objects is combined into the final form. This example combines a Sphere and a Cuboid into a single solid Sphere-Cube shape:



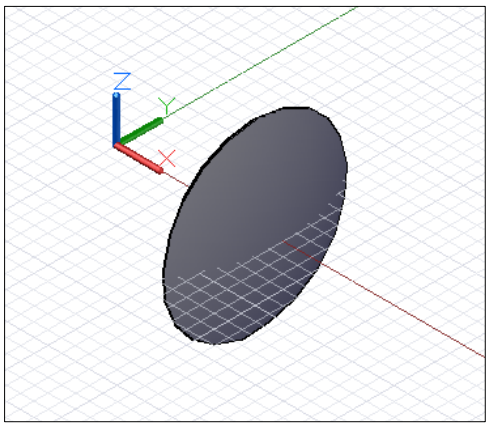
```
s1 = Sphere.ByCenterPointRadius(
    CoordinateSystem.Identity().Origin, 6);

s2 = Sphere.ByCenterPointRadius(
    CoordinateSystem.Identity().Origin.Translate(4, 0,
    0), 6);

combined = s1.Union(s2);
```

The `Difference` method, like Trim, subtracts away the contents of the input tool solid from the base solid. In this example we carve out a small indentation out of a sphere:



```
s = Sphere.ByCenterPointRadius(
    CoordinateSystem.Identity().Origin, 6);

tool = Sphere.ByCenterPointRadius(
    CoordinateSystem.Identity().Origin.Translate(10, 0,
    0), 6);

result = s.Difference(tool);
```

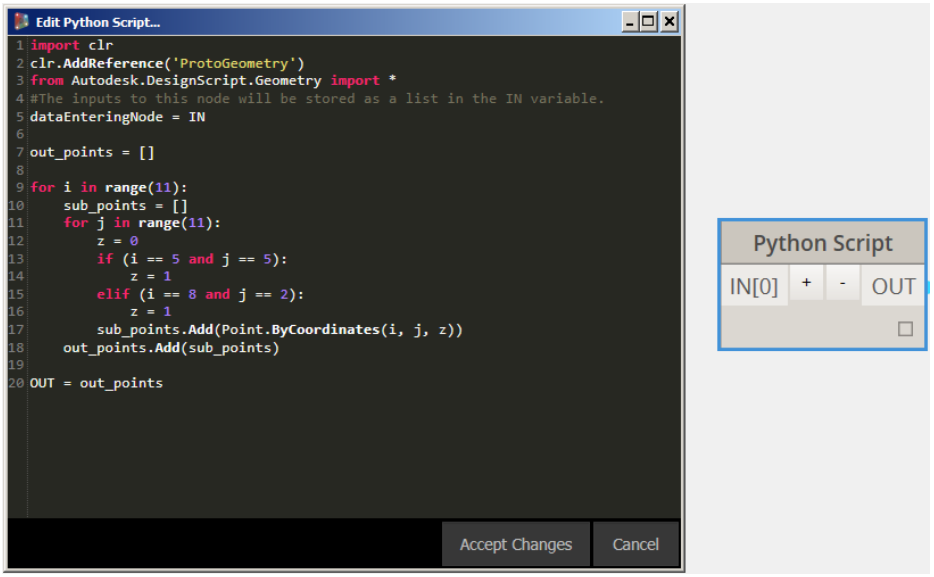
The `Intersect` method returns the overlapping Solid between two solid Inputs. In the following example, `Difference` has been changed to `Intersect`, and the resulting Solid is the missing void initially carved out:




```
s = Sphere.ByCenterPointRadius(
    CoordinateSystem.Identity().Origin, 6);

tool = Sphere.ByCenterPointRadius(
    CoordinateSystem.Identity().Origin.Translate(10, 0,
    0), 6);

result = s.Intersect(tool);
```

A-1: Python Point Generators

The following Python scripts generate point arrays for several examples. They should be pasted into a Python Script node as follows:



python_points_1

```
out_points = []

for i in range(11):
	sub_points = []
	for j in range(11):
		z = 0
		if (i == 5 and j == 5):
			z = 1
		elif (i == 8 and j == 2):
			z = 1
		sub_points.Add(Point.ByCoordinates(i, j, z))
	out_points.Add(sub_points)

OUT = out_points
```

python_points_2

```
out_points = []

for i in range(11):
    z = 0
    if (i == 2):
          z = 1
    out_points.Add(Point.ByCoordinates(i, 0, z))

OUT = out_points
```

python_points_3

```
out_points = []

for i in range(11):
    z = 0
    if (i == 7):
          z = -1
    out_points.Add(Point.ByCoordinates(i, 5, z))

OUT = out_points
```

python_points_4

```
out_points = []

for i in range(11):
    z = 0
    if (i == 5):
          z = 1
    out_points.Add(Point.ByCoordinates(i, 10, z))

OUT = out_points
```

python_points_5

```
out_points = []

for i in range(11):
    sub_points = []
    for j in range(11):
        z = 0
        if (i == 1 and j == 1):
            z = 2
        elif (i == 8 and j == 1):
            z = 2
        elif (i == 2 and j == 6):
            z = 2
        sub_points.Add(Point.ByCoordinates(i, j, z))
    out_points.Add(sub_points)

OUT = out_points
```