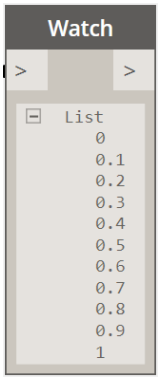
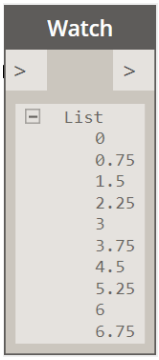



```
a = 0..1..0.1;
```

当指定范围表达的增量时会出现一个问题，即数值往往不会落在最后的范围值。例如，如果我们创建一个0到7的范围表达，增量为0.75，生成的数值如下：

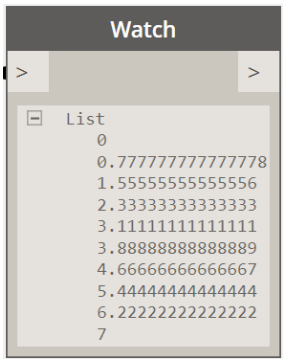
One problem that can arise when specifying the increment between range expression boundaries is that the numbers generated will not always fall on the final range value. For instance, if we create a range expression between 0 and 7, incrementing by 0.75, the following values are generated:




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

如果设计需要产生的范围表达，其结束值正好是最大范围表达式的最大值，Dynam可微调增量值，来尽可能保证范围边界之间的数值均匀分布。这是通过在第三个参数前添加近似符号(~)完成的。

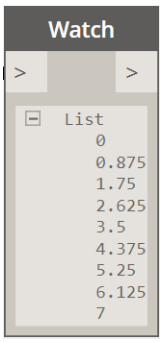
If a design requires a generated range expression to end precisely on the maximum range expression value, Dynamo can approximate an increment, coming as close as possible while still maintaining an equal distribution of numbers between the range boundaries. This is done with the approximate sign (~)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




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

当然，如果你想让Dynamo内插在区域中若干离散的元素，“#”操作符允许你指定：

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;
```

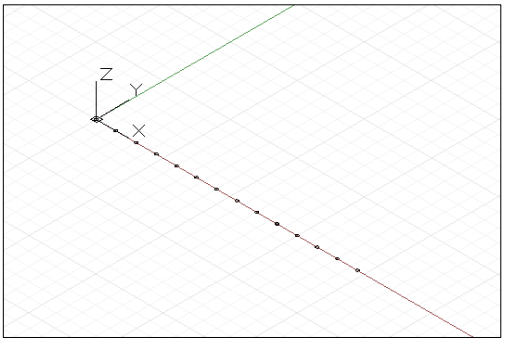
6: Collections 6：集合

集合是特殊类型变量，具有一组值。例如，一个集合可能包含值1到10，{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}；相交操作返回的多种几何对象，{Surface, Point, Line, Point}；甚至包含集合自己{ {1, 2, 3}, {4, 5}, 6}.

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}`.

生成集合的最简单方式是范围表达式（查看：5 范围表达式）。范围表达式默认生成数字集合，但是如果这些对象传入到函数或构造函数，将返回对象集合。

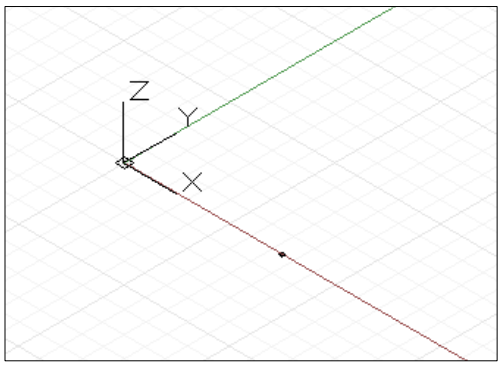
One of the easier ways to generate a collection is with range expressions (see: *Range Expressions*). Range expressions by default generate collections of numbers, though if these collections are passed into functions or constructors, collections of objects are returned.




```
// 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);
```

当范围表达式不是适当的，将创建空的集合，并手动填充值。方括号操作符[]用于范围集合的内部成员方括号书写在变量名后，其里面输入集合成员的序号。序号被称为集合成员索引。由于历史原因，索引从0开始，这意味着集合的首个元素使用collection[0]访问，通常称为“零号”。通过逐渐给索引号增加1，来访问随后成员，例如：

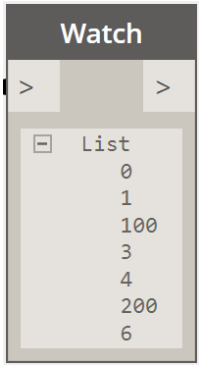
When range expressions aren’t appropriate, collections can be created empty and manually filled with values. The square bracket operator (`[]`) is used to access members inside of a collection. The square brackets are written after the variable’s name, with the number of the individual collection member 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);
```

创建集合后，使用同样的索引操作符可以修改集合的个体成员：

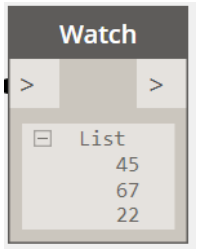
The individual members of a collection can be modified using the same index operator after the collection has been created:

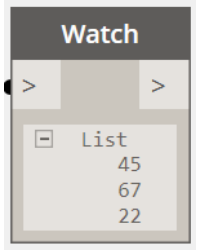



```
// generate a collection of numbers
a = 0..6;

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

实际上，可以显示设置集合的每个成员来创建一个集合。使用花括号操作符{}封装集合的初始值集或者留空创建空集来创建显示集合。

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:

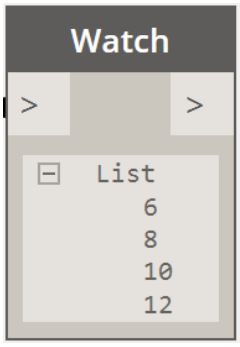





```
// create a collection explicitly
a = { 45, 67, 22 };

// create an empty collection
b = {};

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

集合也可以作为索引集，从一个集合中来生成一个子集合。例如，集合{1,3,5,7}，当被作为一个索引集合，这将从一个集合中提取第2、4、6、8个元素（记住，索引是从0开始的）：

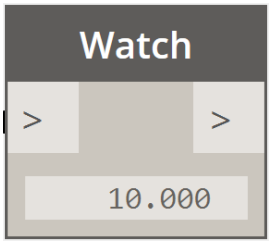
Collections can also be used as the indexes to generate new sub collections from a collection. For instance, a collection containing the numbers `{1, 3, 5, 7}`, when used as the index of a collection, would extract the 2nd, 4th, 6th, and 8th elements from a collection (remember that indices start at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




```
a = 5..20;

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

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

Dynamo提供有用函数来管理集合。函数Count，顾名思义，统计集合并返回它所包含元素的数量。

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);
```

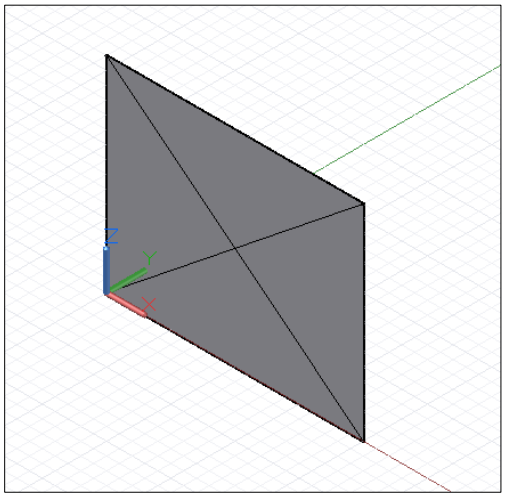
7: Functions 7：函数

到目前未知，在DesignScript里，几乎所有的功能表述是通过函数来表达的。当一个命令含有一个包括各种输入值的圆括号()的后缀的关键字，你可以得出它是一个 函数。在DesignScript里，当函数被调用时，大量代码被执行，并且处理输入值和返回结果。构造函数Point.ByCoordinates(x : double, y : double, z : double)代入三个输入值，处理后返回一个Point对象。正如大多数程序语言，DesignScript给程序员提供创建他们自己的函数的功能。函数是有效脚本的一个关键部分：使用特定功能处理代码块，明确描述其输入与输出来对它（函数）进行包装，以使你的代码更容易修改或重用。

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.

假设，一个程序员编写本在一个表面上创建一个对角拉线：

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);
```

在一个表面上创建对角线的简单行为仍然需几行代码。如果我们想找到数以百计的对角线，不是数以千计的表面，单个提取角点并绘制对角线的系统将完全不切实际。创建一个从面提取对角线的函数将使得程序员把多行代码的功能应用到任意数量的基本输入处（引用函数地方）。

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.

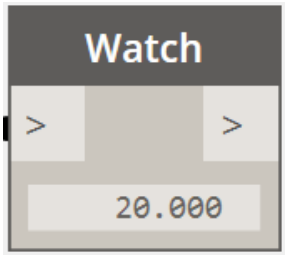
通过在函数名字前添加关键字def，以及在方括号中的输入参数列表来创建函数。函数代码封装在花括号内{}。在DesignScript中，函数必须返回一个值，通过给关键字变量return赋值来实现返回值，例如：

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;
}
```

下面的函数有一个参数，并返回参数的2倍值。

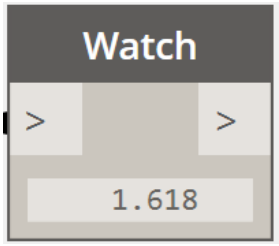
This function takes a single argument and returns that argument multiplied by 2:




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

times_two = getTimesTwo(10);
```

函数不是一定需要带入参数，一个返回黄金分割点的简单函数如下所示：

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();
```

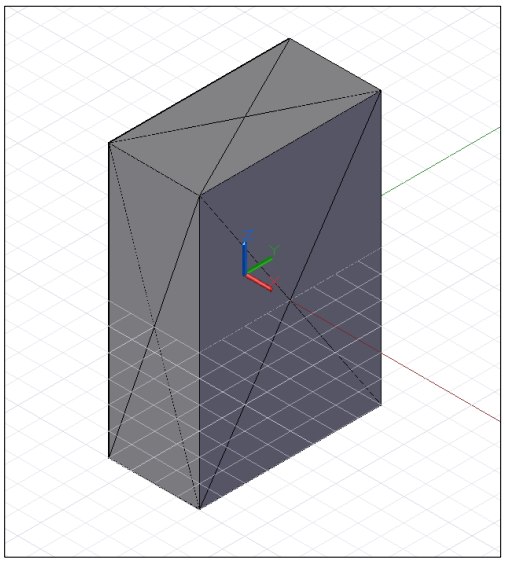
在创建一个封装我们对角线代码的函数前，请注意函数仅仅只能返回单一值，而我们的对角线代码生成两小线。为了解决这个问题，我们可以把两个对象封装在花括号{}内，来创建一个单一集合对象。例如，下面是一个返回两个值的简单函数：

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();
```

如果我们在函数中封装对角线代码，我们可以在一系列表面上创建对角线，例如一个长方体的所有面。

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());
```

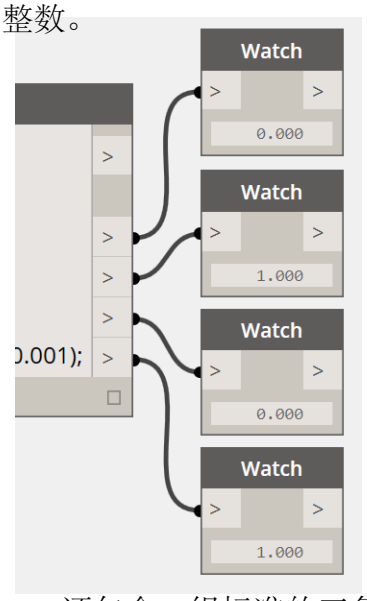
8: Math　　8：数学

Dynamo标准库内包含数学分类函数来辅助编写算法和操纵数据。数学函数以使用命名空间Math作为前缀，为了上使用数学函数，需要在函数前附加“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.

函数Floor，Ceiling和Round允许你使用可预测结果来转换浮点数值和整数值。虽然这三个函数接受一个单一的浮点数作为输入，Floor返回一个总是向下舍入的整数，Ceiling返回一个总是向上舍入的整数，Round返回最接近的整数。

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

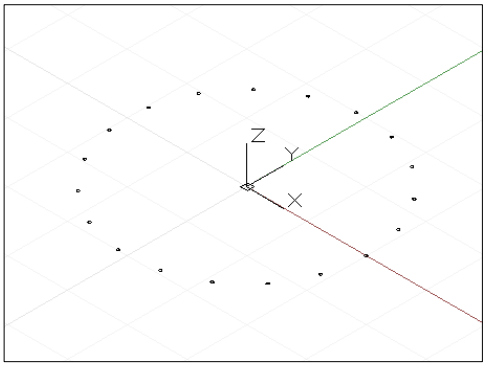



```
val = 0.5;

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

Dynamo还包含一组标准的三角函数，分别使用Sin、Cos、Tan、Asin、Acos和Atan函数来计算正弦、余弦、正切、反正弦、反余弦和反正切。
全面介绍三角函数超出了本书的范围，由于正弦和余弦函数是半径为1的圆上的位置坐标，它们频繁出现在计算式设计中。通过输入一个增量角度，通常标记为 θ ，Cos θ 是x坐标，Sin θ 是y坐标，来计算圆上的坐标值。

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.

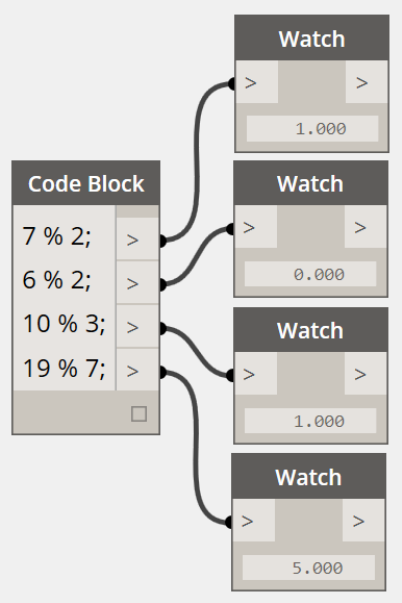
While a comprehensive description of trigonometry is beyond the scope of this manual, the sine and cosine functions do frequently occur in computational designs due their ability to trace out positions on a circle with radius 1. By inputting an increasing degree angle, often labeled `theta`, into `Cos`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

数学标准库中的相关数学规则不太严格一部分是取余操作符。取余操作符，以符号%表示，返回两个整数相除所得的余数。例如：7除以2，商为3，余数为1（等同2×3+1=7）。7取模2后，为1。另一方面，6能够被2均匀等分，因此6月2的模数为0。下面的例子描述了不同取余操作结果的结果：

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.



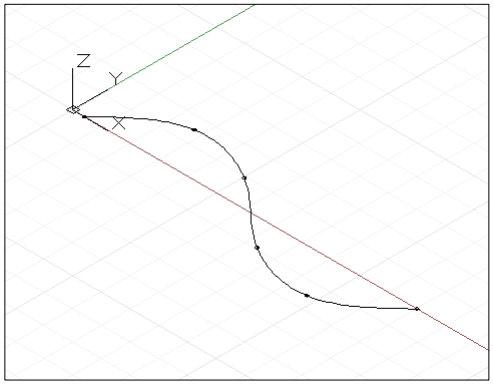

```
7 % 2;
6 % 2;
10 % 3;
19 % 7;
```

9: Curves: Interpreted and Control Points

在Dynamo中有两种方法创建自由曲线：通过指定一个点的集合，Dynamo将会拟合出一条内插点曲线，或者指定一个曲度和控制点上来生成曲线。当设计师知道形体的轮廓或者一条曲线的确切走向时，内插点曲线是非常有用的。通过指定控制点的曲线本质上是由直线段经过一系列算法生成的光滑曲线。通过指定控制点的曲线非常有助于研究不同光滑度的形体，在平滑连接两段曲线时是也是必须的。使用NurbsCurve.ByPoints方法便可生成一条通过集合中所有点的插入点曲线。

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.

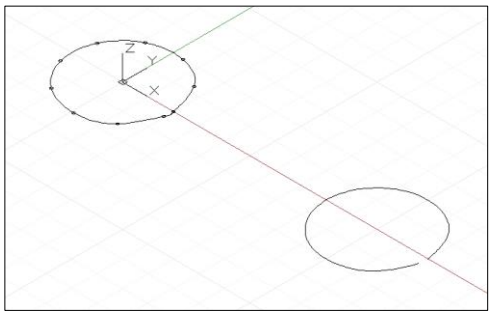
To create an interpreted curve, simply pass in a collection of Points to the `NurbsCurve.ByPoints` method.




```
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);
```

所生成的曲线分别与每个输入点相交，曲线开始于集合中的第一个，结束于最后一个点。一个可选的参数可用于创建封闭曲线。Dynamo将自动补充缺少段，所以不需要一个和开始点重复的结束点。

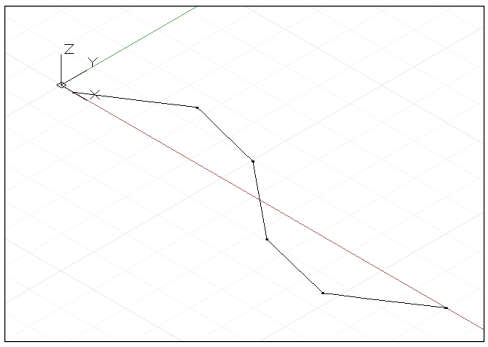
The generated curve intersects each of the input points, beginning and ending at the first and last point in the collection, respectively. An optional periodic parameter can be used to create a periodic curve which is closed. Dynamo will automatically fill in the missing segment, so a duplicate end point (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);
```

生成NurbsCurves多以同一种方法生成，该方法的第一个参数描述曲线的插入点，第二个参数描述曲线的光滑程度，称为曲度。曲度为1的曲线并不平滑，是一条折线。

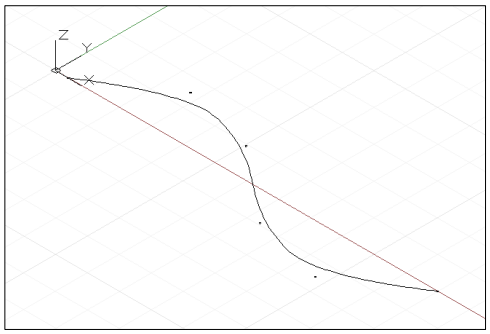
NurbsCurves are generated in much the same way, with input points represent the endpoints of a straight line segment, and a second parameter specifying the amount and type of smoothing the curve undergoes, called the degree.* A curve with degree 1 has no smoothing; it is a polyline.




```
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);
```

曲度为2的曲线是平滑的，并与折线线段的中点相切：

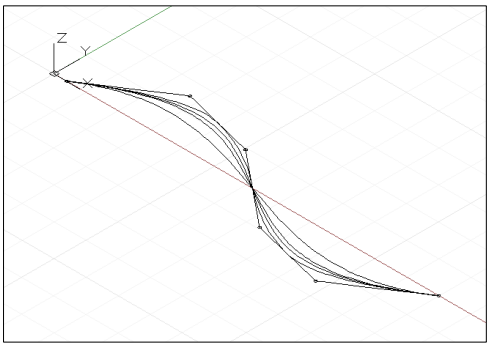
A curve with degree 2 is smoothed such that the curve intersects and is tangent to the midpoint of the polyline segments:




```
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支持曲度最大为20的NURBS曲线（非均匀有理B样条曲线），下面的代码说明不同的曲度对曲线光滑度的影响：

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:



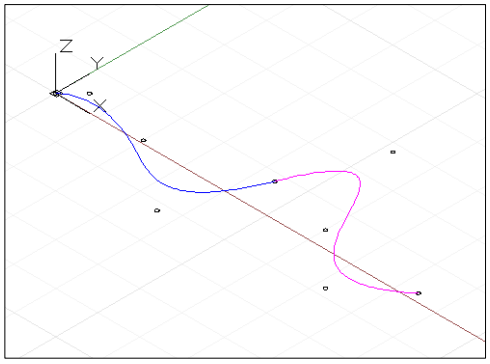

```
num_pts = 6;

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);
```

需要注意的是，控制点的个数必须比曲度大1。

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

通过控制点生成曲线的另一个好处是，保证不同曲线段之间相切的关系。这是通过使前一条曲线最后两点的方向与下条曲线开始两点的方向相同来实现的。下面的示例创建了两条NURBS曲线，但它们光滑得就像一条曲线:

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);
```

这只是NURBS曲线的简单描述，更详细的信息请参见其他书资料。

* 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.

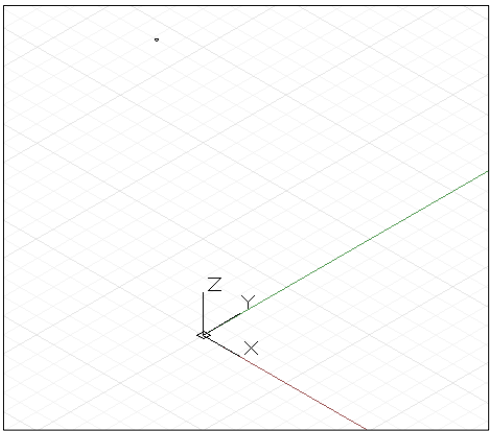
10: Translation, Rotation, and Other Transformations

某些几何对象可以通过指定明确的x、y、z三维坐标来创建。然而，更多时候几何体是通过各种几何变换来生成的。

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.

最简单的几何变换是平移，通过指定的距离和方向来移动物体。

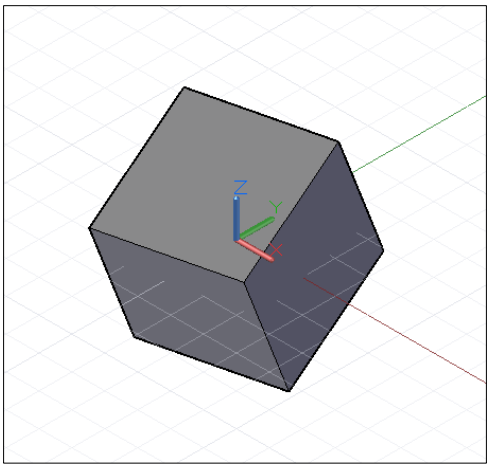
The simplest geometric transformation is a translation, which moves an object a specified number of units in the x, y, and z directions.




```
// 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);
```

尽管在Dynamo 的所有对象可以通过在对象名后附加Translate方法来平移对象，但是更复杂的平移操作需要指定一个现有的坐标系或者新建一个。例如，想将一个物体绕x轴旋转45度时，我们需要使用Transform方法将原本没有旋转的坐标系沿x轴旋转45度：

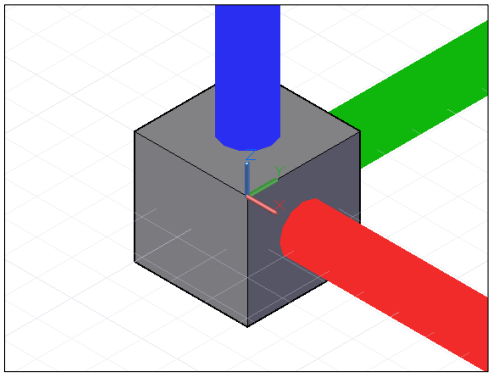
While all objects in Dynamo can be translated by appending the `.Translate` method to the end of the object's name, more complex transformations require transforming the object from one underlying CoordinateSystem to a new CoordinateSystem. For instance, to rotate an object 45 degrees around the x axis, 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);
```

除了平移、旋转，坐标系也可以用于缩放和切变。坐标系可以用Scale方法进行缩放：

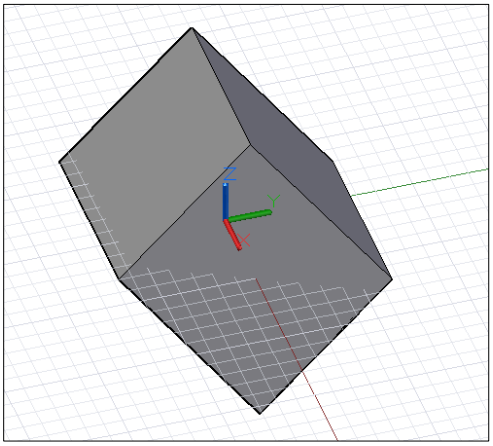
In addition to being translated and rotated, CoordinateSystems can also be created scaled or sheared. A CoordinateSystem can be scaled with the `.Scale` method:



```
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);
```

切变坐标系通过对一个原始坐标系输入一个非正交向量来生成。

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);
```

缩放和切变坐标系相对于旋转和平移较为复杂，所以并非Dynamo中的每个对象可以接受这些变换。下表概述了这两种坐标系对Dynamo中各种对象的使用情况：

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

编程语言的强大功能之一就是可以通过执行代码对已有对象进行判断，根据对象的特性。编程语言是通过一个叫做逻辑运算的系统来对对象进行判断和执行代码的。

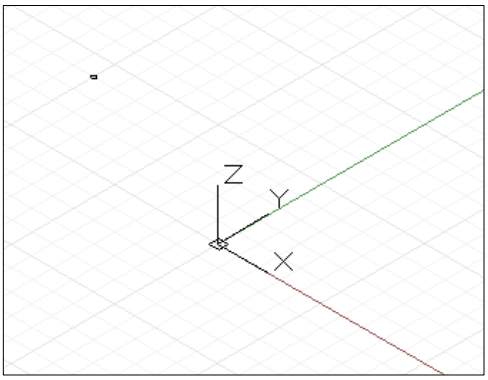
One of the most powerful features of a programming language is the ability to look at the existing objects in a program and vary the program's execution according to these objects' qualities. Programming languages mediate between examinations of an object's qualities and the execution of specific code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

逻辑运算用于判断语句的真假。在逻辑运算中的每个语句的结果只可能是真或假，没有其他结果。表明结果为真的最简单方法是使用关键字true。同样，表明结果为假的最简单方法是使用关键字false。if语句允许你判断语句的真假：如果结果为真则第一段代码被执行，如果是假则第二段代码被执行。

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.

在以下示例中，if条件语句中包含一个为真的语句，所以第一段语句被执行并生成一个点：

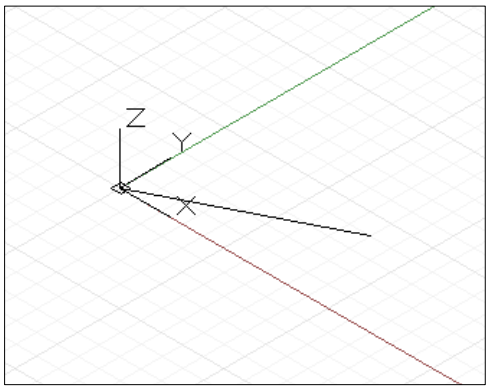
In the following example, the `if` statement contains a **`true`** Boolean statement, so the first block executes and a Point is generated:




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

如果包含的语句改为假，则第二段代码被执行并生成一条线：

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));
    }
}
```

像这些简单的真假语句并不是特别有用；逻辑运算语句的真正作用在于判断脚本中对象的真假。逻辑运算有六个基本操作：小于(<)，大于(>)，小于或等于(< =)，大于或等于(> =)，等于(= =)，不等于(! =)。下面的表格概述了逻辑运算结果

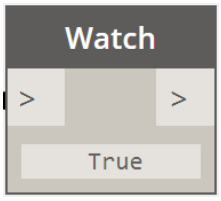
Static Boolean statements like these aren't particularly useful; the power of Boolean logic comes from examining the qualities of objects in your script. Boolean logic has six basic operations to evaluate values: less than (<), greater than (>), less than or equal (<=), greater than or equal (>=), equal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

* 请参阅"数字类型"一节，了解两个浮点数据运算时的限制。

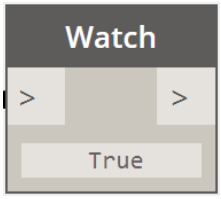
* see chapter "Number Types" for limitations of testing equality between two floating point numbers.

对两个数值使用以上运算操作将返回 true 或 false：

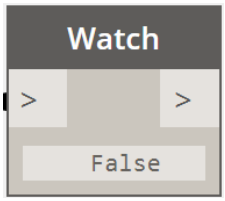


Using one of these six operators on two numbers returns either **`true`** or **`false`**:

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




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



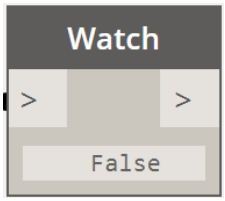

```
result = 99 != 99;
```

其他三个布尔运算符用于比较：和 (&)，或 (|) 和不 (!)。

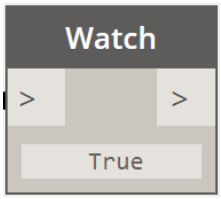
Three other Boolean operators exist to compare **`true`** and **`false`** statements: and (&&), or (||), and not (!).

&&	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

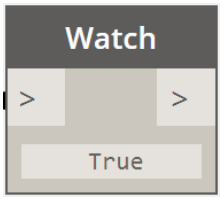
如果两边都为真，则返回 true。
如果任一边为真，则返回 true。
返回相反的布尔值




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



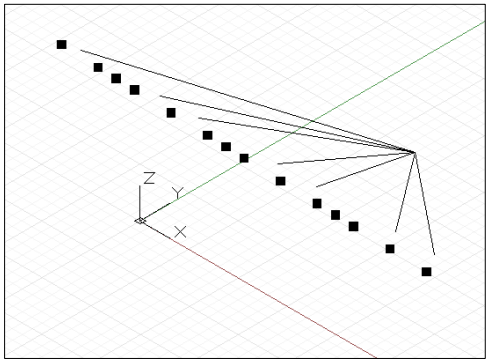

```
result = true || false;
```




```
result = !false;
```

以下代码修改自最初的例子，演示通过输入数列来生成一条路径：

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);
```