Dynamo Language Manual

Contents

Introduction 说明

编程语言的产生是为了表达思想，通常包括逻辑和计算。除了这些目标，Dynamo文本语言（原名DesignScript）已经建立表达设计意图。通常认为，计算式设计是探索性的，Dynamo试图支持如下：我们希望你们找到语言的灵活性、从概念设计、通过设计迭代快速完成你的最终形式。

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.

本手册的结构是给没有编程或建筑几何知识的用户充分展示这两个交叉学科的各种话题。经验更丰富的用户应该跳到他们感兴趣及问题领域的相关章节。每个章节均是独立的，除了在先前章节提供的信息外不需要其他知识。

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.

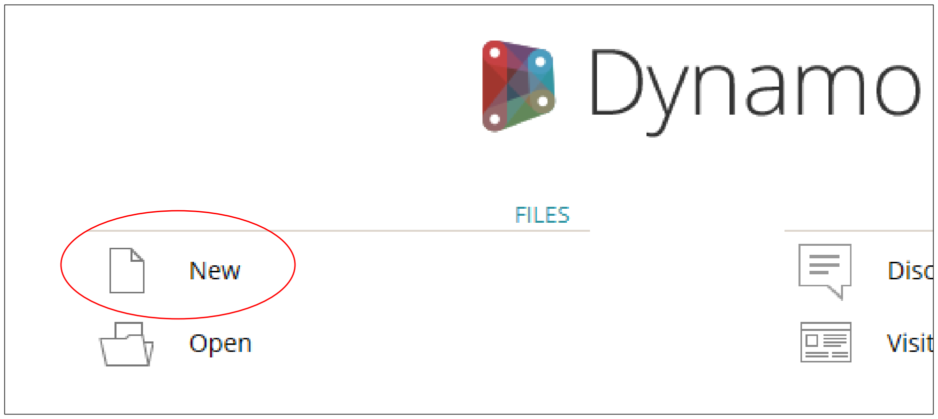
嵌入在Consolas字体里的文本块应可以被粘贴到代码块节点内。代码块的输出应被链接到一个Watch节点，以便看到预期的结果。左边的图块显示程序的正确输出。

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 语言基础

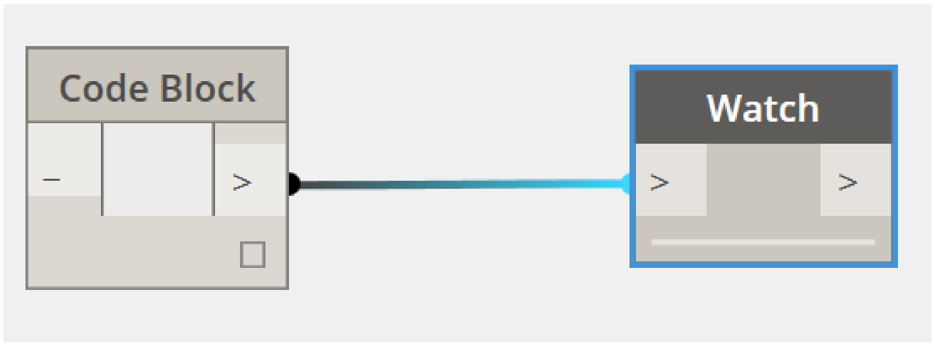
本文探讨Dynamo文本编程语言，使用内部的Dynamo编辑器（有时被称为“Dynamo沙箱”）。打开Dynamo编辑器，在“FILES”菜单选择“New”按钮，新建一个Dynamo脚本。

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:



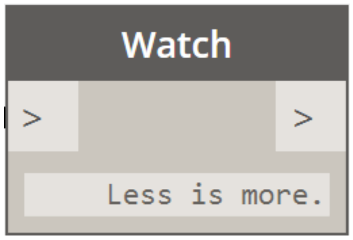
这将打开一个空白的Dynamo图档。双击画布上的任意地方来编写Dynamo文本脚本。这将会打开“代码块”节点。为了更方便看到脚本输出结果，在代码块节点输出端附加一个Watch节点，如下所示：

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:



每个脚本都是一系列的书面命令。其中一些命令创建几何体；另外一些解决数学问题、写入文本文件、或者生成文本字符串。一个简单例子，一行程序生成“Less is more”的引用，看起来像这样：

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:



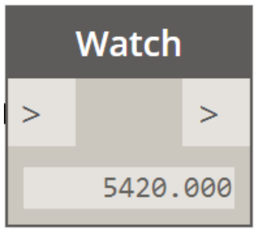
```
"Less is more.";
```

左侧的Watch节点显示脚本输出。

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

该命令新建一个字符串对象。Dynamo中的字符串是由两个引号（"）中之间的字符组成，字符包括空格。代码块不限于生成字符串，一个代码块节点生成数值5420看起来像这样：

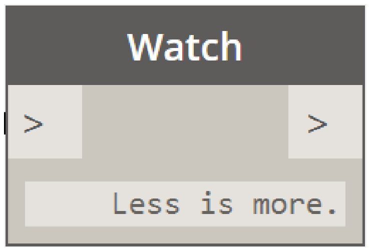
The command generates a new String object. Strings in Dynamo are designated by two quotation marks ("), and the enclosed characters, including spaces, are passed out of the node. Code Block nodes are not limited to generating Strings. A Code Block node to generate the number 5420 looks like this:



```
5420;
```

Dynamo中的命令以分号结束。如果不含，编辑器将会自动增加。另外注意的是，数字和字符组合（空格、制表符和回车等空白字符），一个命令之间的元素无关紧要。这个程序的输出与第一个程序一样。

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."

;
```

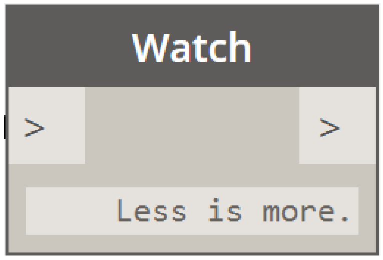
当然，空白字符应用于帮助提高代码的可读性，不管是对自己还是将来的读者。

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

注释是另一种提升代码可读性的工具。在Dynamo中，单行注释代码以双斜杆“//”开头。这将使得节点忽视斜杆后的所有信息，直到回车符（该行的结尾）。多行注释是以斜杆星号（/*）开头，以星号斜杆（*/）结尾。

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, `*/`.

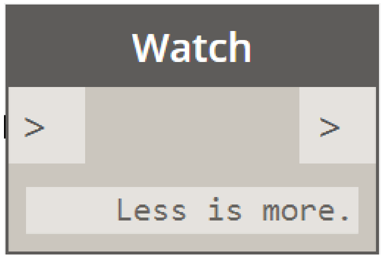
```
// This is a single line comment

/* 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";
```



到目前未知，代码块参数是“文字”值，一个文本字符串或一个数。但是往往更有用的是存储在数据容器中的被称为变量的函数参数，能使代码更具有可读性和消除代码中的冗余命令。变量名字由程序员自己决定，虽然变量名字必须唯一，变量名字以小写或大写字母开始，仅仅包含字母、数字或下划线。变量名字不允许有空白字符。变量名字应该（尽管不是必须的）描述它们说包含的数据。例如，一个变量用来记录一个对象的旋转可被称为"rotation"。为了使用多个单词描述数据，程序员通常使用两个公共约定：用大写字母分割单词，称为camelCase（连续大写字母模拟骆驼的驼峰）;使用下划线分割单词。例如，一个变量来描述一个小圆盘的旋转可命名为smallDiskRotation或small_disk_rotation，这取决于程序的风格。创建一个变量，其名字写在等式的左边，然后写入你要分配给它的值。例如：

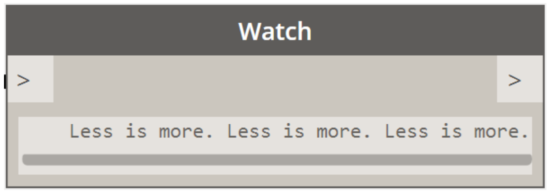
So far the Code Block arguments have been ‘literal’ values, either a text string or a number. However it is often more useful for function arguments to be stored in data containers called variables, which both make code more readable, and eliminate redundant commands in your code. The names of variables 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.";
```



除了显而易见的文本字符串的角色，当以后数据改变时，变量可以帮助减少大量代码的更新。例如：如下引用文本仅需要在一处改变，虽然它在程序中出现了三次。

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.

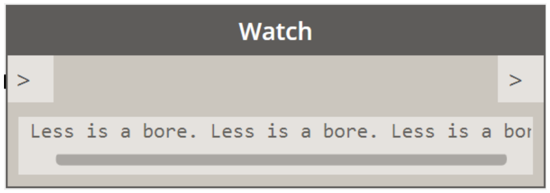


```
Less is more. Less is more. Less is more.
```

```
// My favorite architecture quote

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

在这里，添加一个短语并重复三次，每个短语之间用空格隔开。请注意，使用运算符“+”来连接字符串或变量以形成一个连续的输出。

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.



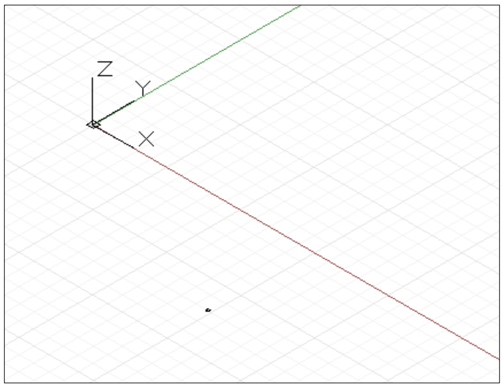
```
Less is a bore. Less is a bore. Less is a bor
```

```
// My NEW favorite architecture quote

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

2: Geometry Basics 几何基础

在Dynamo标准几何库中，点“Point”是最简单的几何对象。所有的几何体均使用称为构造函数的特殊函数来创建，它返回一个特定几何类型的新实例。在Dynamo中，构造函数名称以对象类型名字开头，以点“Point”为例，下面演示其构造方法。使用ByCoordinates构造函数来创建一个指定x、y、z值的笛卡尔坐标的三维点。

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:



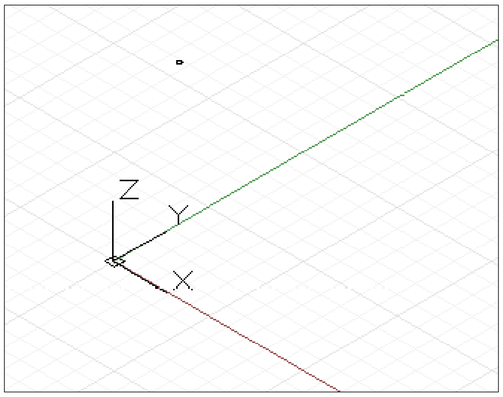
```
// create a point with the following x, y, and z
// coordinates:
x = 10;
y = 2.5;
z = -6;

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

在Dynamo中，构造函数通常以“By”作为前缀，调用这些函数创建一个该类型的新对象。新创建对象覆给了等式左边的变量，其使用等同于原始Point。

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.

多数对象用于许多不同的构造函数，通过给定球体半径，第一和第二旋转角度（单位为“°”或“度”），我们就可以使用构造函数（BySphericalCoordinates）创建球体上的点Point。

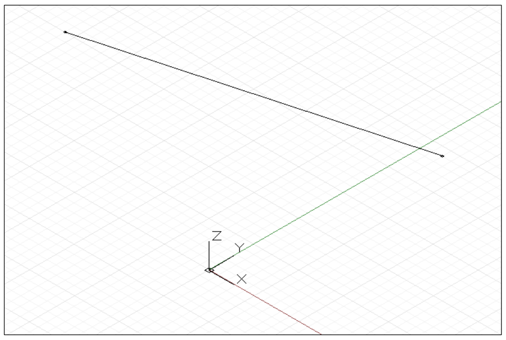
Most objects have many different constructors, and we can use the `BySphericalCoordinates` constructor to create a point lying on a sphere, specified by the sphere's radius, a first rotation angle, and a second rotation angle (specified in degrees):



```
// 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);
```

Point（点）能够用于构建更高维度的几何体，比如线。我们可以使用ByStartPointEndPoint构造函数来创建两点之间的直线对象。

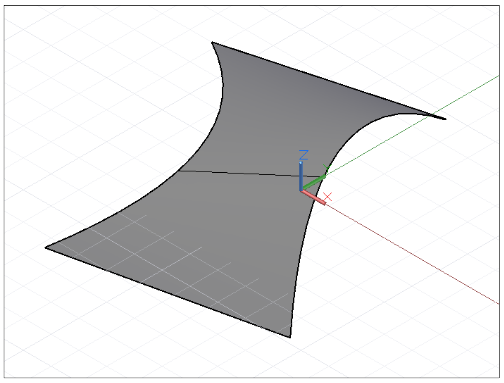
Points can be used to construct higher dimensional geometry such as lines. We can use the `ByStartPointEndPoint` constructor to create a Line object between two points:



```
// 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);
```

同样，Line线可以用于构建更高维度的几何体“面”，例如使用Loft（放样）构造函数，以一些直线或曲线内插为一个表面。

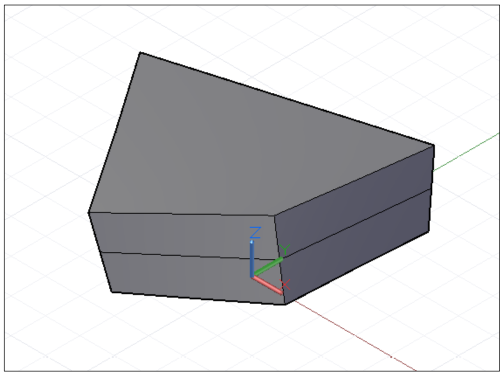
Similarly, lines can be used to create higher dimensional surface geometry, for instance using the `Loft` constructor, which takes a series of lines or curves and interpolates a surface between them.



```
// 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});
```

“面”也可以用于构建更高维度的立体几何，例如通过给定的距离来加厚表面。许多对象有附加功能，称为“方法”，它允许程序员对指定对象执行命令。所有几何体常用方法包括平移和旋转，通过给定的量来分别移动或旋转几何体。“面”有一个Thichken方法，根据输入值来设定“面”的新的厚度。

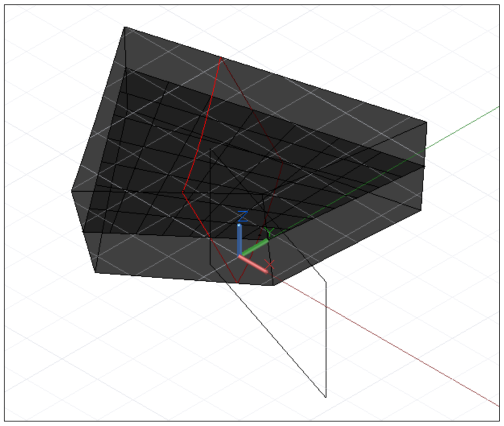
Surfaces too can be used to create higher dimensional *solid* geometry, for instance by thickening the surface by a specified distance. Many objects have functions attached to them, called methods, allowing the programmer to perform commands on that particular object. Methods common to all pieces of 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命令可以从高维对象提取低维几何体。该提取的低维几何体能够形成基本的更高维度几何体，在一个圆柱体（Cyclic）中处理几何创建、提取和重建。在下面的例子中，我们使用生成的实体来创建“面”，使用“面”来创建“曲线”。

`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 基本几何

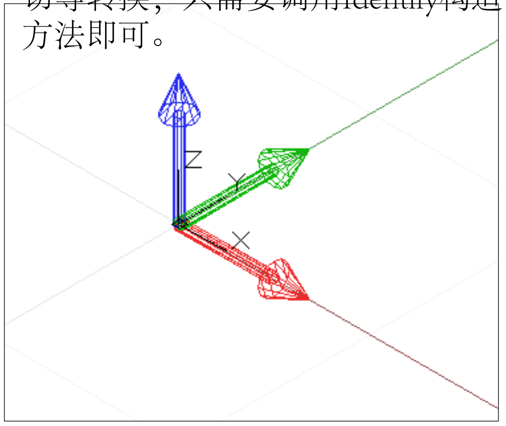
Dynamo能够创建各种复杂几何形体，简单的几何图元形成任何计算式设计的基本骨架：不管是直接表示为最终设计形体，还是以搭积木形式生成更复杂几何体。

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.

虽然没有严格一块的几何体，CoordinateSystem是构建几何体的重要工具。一个Coordinate对象跟踪位置及几何体转换（如旋转、剪切和缩放）
创建一个CoordinateSystem：中心点在（0,0,0），没有旋转、缩放或剪切等转换，只需要调用Identify构造方法即可。

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.

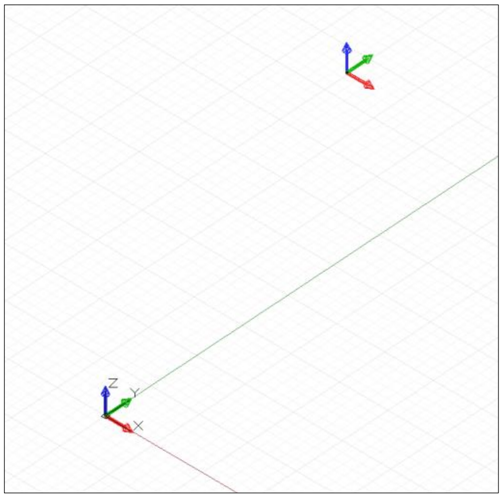
Creating a CoordinateSystem centered at a point with x = 0, y = 0, z = 0, with no rotations, scaling, or sheering transformations, simply requires calling the Identity constructor:



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

cs = CoordinateSystem.Identity();
```

虽然CoordinateSystems几何变换超出本章范围，但另一个构造函数允许你创建一个基于特定点的坐标系，即CoordinateSystem.ByOriginVectors

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);
```

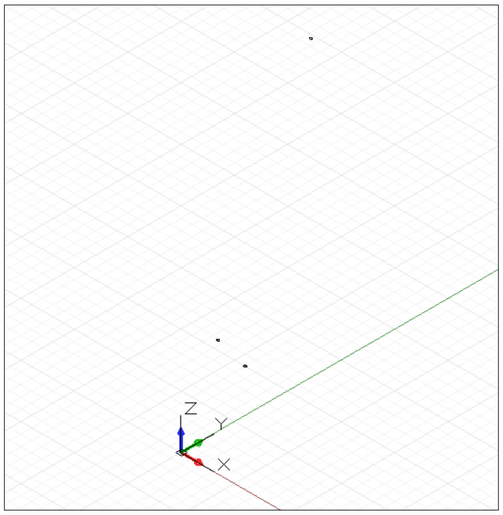
“点”是最简单的几何图元，它代表三维空间中的零维定位。如前所述，在一个特定的坐标系中，有多种方式创建点Point：Point.ByCoordinates使用给定的x、y、z坐标来创建一个点；

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.ByCartesianCoordinates使用给定的x、y、z坐标，在制定的坐标系中来创建一个点；Point.ByCylindricalCoordinates使用半径、旋转角度和高度创建位于圆柱体上的点；Point.BySphericalCoordinates使用半径和两个旋转角度创建位于球体上的点。
下面的例子中展示了在不同坐标系中创建点Point。

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.

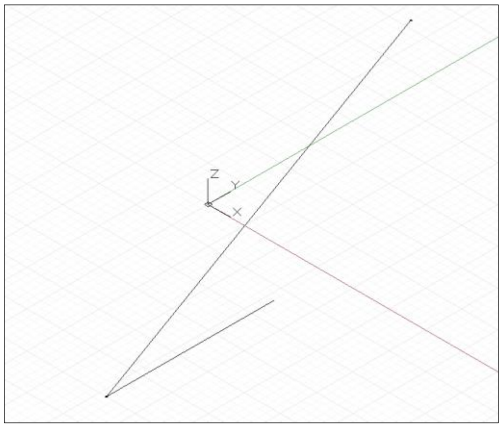
This example shows points created at various coordinate systems:



```
// 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);
```

下一个更高维度的Dynamo基本图元是线段，代表两个端点之间的无限数量的点。通过明确两个边界点，使用Line.ByStartPointEndPoint构造函数可以创建一个直线，或者通过明确一个起始点、方向和该方向上的长度，使用Line.ByStartPointDirectionLength来创建一个直线。

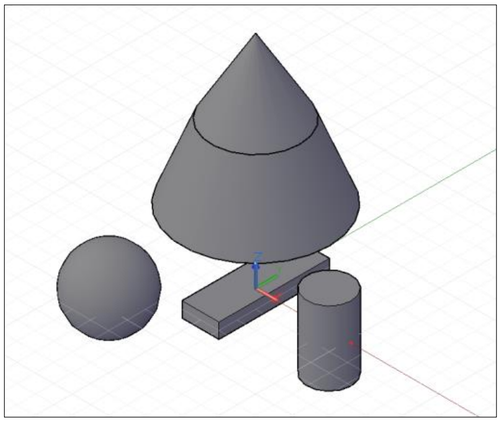
The next higher dimensional Dynamo primitive is a line segment, representing an infinite number of points between two end points. Lines can be created by explicitly stating the two boundary points with the constructor `Line.ByStartPointEndPoint`, or by specifying a start point, direction, and length 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中表示几何图元的基本类型的三维对象：长方体，使用Cuboid.ByLengths创建；圆锥，使用Cone.ByPointsRadius 和Cone.ByPointsRadii创建；圆柱体，使用Cylinder.ByRadiusHeight创建；球体，使用Sphere.ByCenterPointRadius创建。

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`.



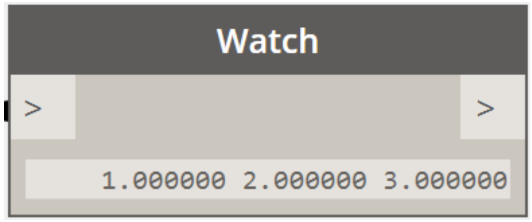
```
// create a cuboid with specified lengths
cs = CoordinateSystem.Identity();

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 数学矢量

在计算式设计中，对象创建时几乎不在其最终位置和形式，而经常使用平移、旋转和基于现有几何体的其他定位。矢量作为一种几何骨架给出几何体的方向和取向，等同于通过三维空间的没有视觉表现的概念化动作。
其最基本的，一个向量代表一个三维空间中的位置，通常认为是一个从（0,0,0）到其位置的一个箭头。可以使用ByCoordinates构造函数，并代入将创建新矢量对象的x，y，z来创建矢量。请注意，矢量对象不是一个几何体对象，且不会出现在Dynamo窗口内。然而，新创建或修改的矢量信息能够打印在控制台中内。

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:



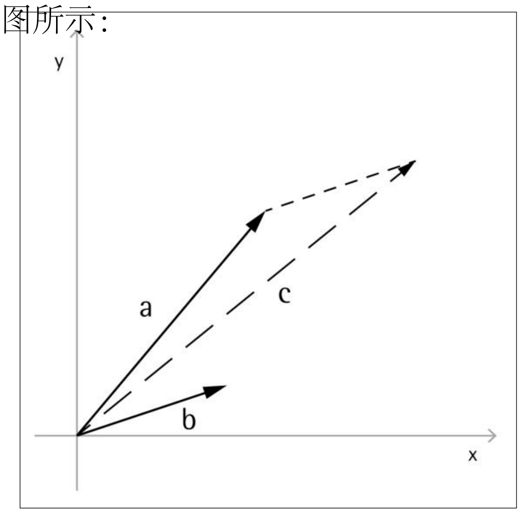
```
1.000000 2.000000 3.000000
```

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

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

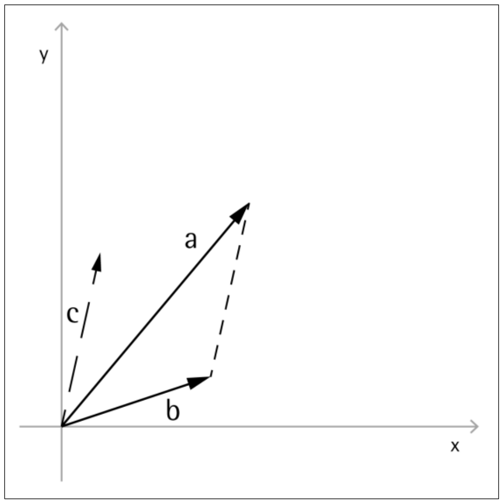
矢量对象定义了一组数学操作，允许在三维空间内进行加、减、乘、以及移动等操作，就像在一维数轴上移动实数。
矢量叠加定义为两个矢量的分量之和，当两个矢量箭布置成“箭头连着箭尾”时，它可以看做是新的矢量结果（第一个箭尾引至第二个箭头）。使用Add方法执行矢量叠加运算，如左图所示：

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.



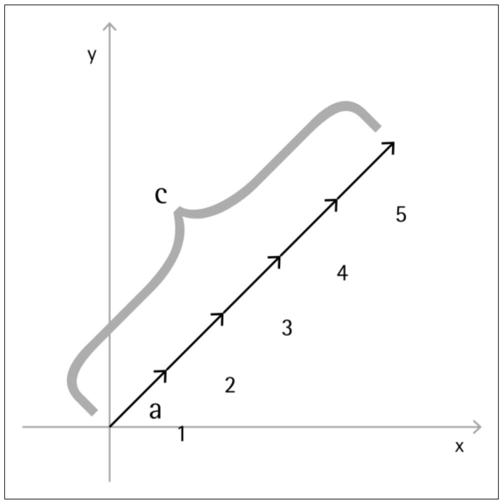
```
a = Vector.ByCoordinates(5, 5, 0);
b = Vector.ByCoordinates(4, 1, 0);

// 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);
```

同样，两个矢量对象可以彼此使用Subtract方法执行减法运算。矢量减可以看做为第一个矢量到第二个矢量的方向（新的箭，箭头在第一个矢量箭头，箭尾在第二矢量箭头）



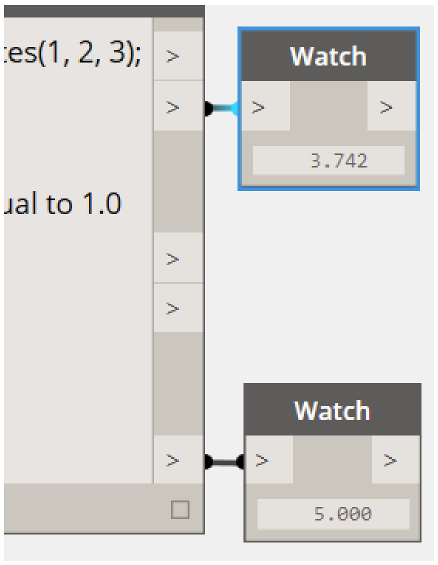
Vector multiplication can be thought of as moving the endpoint of a vector in its own direction by a given scale factor.

矢量数乘可看做为通过给定的缩放因子，沿着矢量方向移动矢量端点。

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

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

缩放矢量，通常需要使其长度精确到某个量。这很容易实现，首先使矢量单位化，也就是说将矢量的长度恰好等于1。

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.



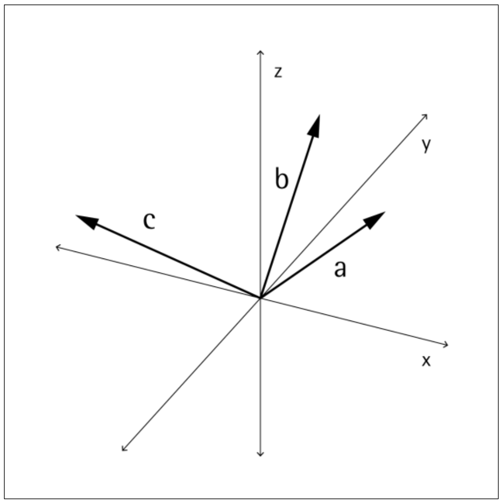
```
a = Vector.ByCoordinates(1, 2, 3);
a_len = a.Length;

// 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仍然指向与（1,2,3,）相同的方向，但是现在它的长度恰好等于5。

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

在数学矢量中的另外两个方法，其与一维数学具有明显的不同，是叉积和点积。叉积是两个矢量正交（90°）生成一个矢量。例如，x轴与y轴的叉积是z轴，而两个给定的矢量不需要正交。叉积运算使用Cross方法。

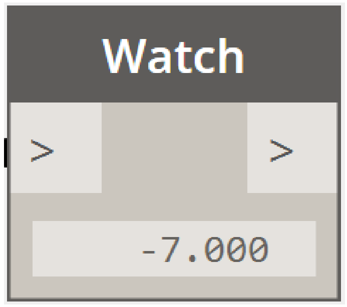
Two additional methods exist in vector math which don’t have clear parallels with 1D math, the cross product and dot product. The cross product is a means of generating a Vector which is orthogonal (at 90 degrees to) to two existing Vectors. For example, the cross product of the x and y axes is the z 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);
```

另外，点积是数学矢量中较为高级功能。两个矢量的点积是一个实数（不是一个矢量对象），且确切来说没有涉及到两个矢量之间的角度。点积的一个有用的特性是：两个矢量的点积为0当且仅当两者垂直。点积运算使用Dot方法。

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.



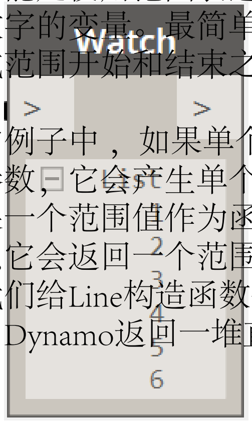
```
a = Vector.ByCoordinates(1, 2, 1);
b = Vector.ByCoordinates(5, -8, 4);

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

5: Range Expressions 范围表达式

几乎所有的设计都涉及到重复元素，并且明确键入每个点的名字和构造函数，以及其他操作，这在脚本中将是非常耗时的。范围表达式给Dynamo程序员提供了一种方法来表达一堆值，即在两侧参数中间设置两个点(..)，这可以生成包含两个极限值之间的数字。
例如：虽然我们看到变量包含单个数字，它可能是使用范围表达式生成包含一堆数字的变量。最简单的范围表达式生成范围开始和结束之间的整数增值。
在之前的例子中，如果单个数字作为函数的参数，它会产生单个结果。同样，如果一个范围值作为函数的参数，那么它会返回一个范围值。
例如，我们给Line构造函数传入一个范围值，Dynamo返回一堆直线。

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.

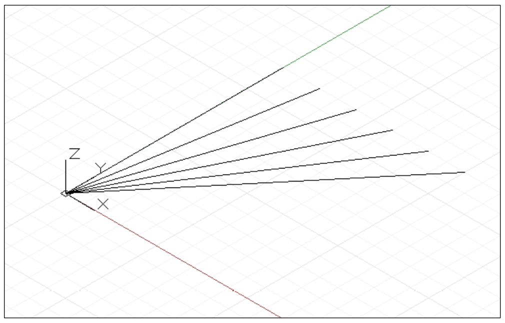


```
List
1
2
3
4
5
6
```

```
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));
```

范围表达式默认以整数增值填充整个数字区域，这对快速生成拓扑结构非常有用，但是不适合实际设计。通过再增加一个省略号(..)给范围表达式，你可以指定范围表达式之间的增值。这里，我们希望得到0与1之间，以0.1为增值的所有数字。

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: