DesignScript Language Manual

Contact

Manual Author:

Patrick Tierney, patrick.tierney@autodesk.com

Language Team:

Luke Church: luke.church@autodesk.com

DesignScript Test Development: design.script.td@autodesk.com

Contents

Introduction

Programming languages are created to express ideas usually involving logic and calculation. In addition to these objectives, DesignScript has been created to express design intentions. These intensions might include: the definition of different design and performance parameters and constraints, the use of these parameters to generate the constructive geometry of a building design at different levels of detail, how this geometry may be expressed in materials, leading to the physical form of the building, and the analysis for such design representations including, cost, structural efficiency and energy performance.

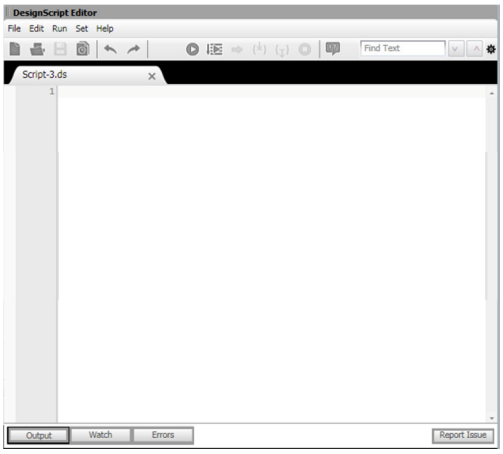
It is generally recognized that this type of designing is essentially exploratory, and therefore DesignScript is itself a language which is intended to support exploratory programming. This means that we may start writing a program without a defined goal in mind, or perhaps with some initial concept or direction, but this may evolve during the course of the programmatic experimentation.

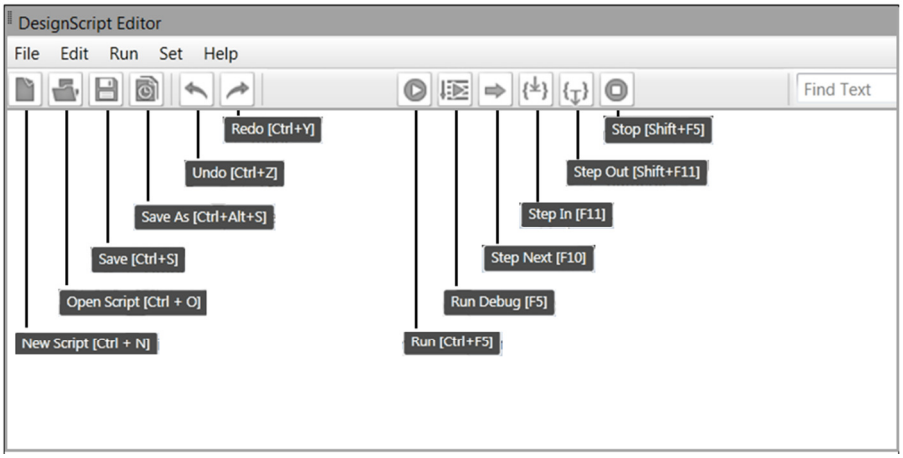
DesignScript is intended to be a very flexible and adaptable language so as to support this type of exploratory programming. The language was created to expresses designs in an intuitive form, both for novices and advanced users alike; this goal was not as much of an inherent contradiction as it sounds: much of the complexity in advanced architectural geometry scripts is due to the difficulty of shoehorning designs into existing languages created for an entirely different purposes. To facilitate an intuitive expression of design intentions by novices and experts alike, the language of DesignScript is tailored to common architectural and design patterns: the repetition of elements in one and two dimensions, the existence of unique elements in a homogenous field, inherent interconnectivity between designed elements, and the need to "glue" a wide variety of processes from a variety of disciplines together in a single system. DesignScript is applicable to a variety of problem domains and disciplines, and while it is most often used to create geometry inside of a host application such as AutoCAD, it can be used as a self-standing language in a variety of geometry hosts, running locally or distributed over dozens or thousands of servers.

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.

1: Language Basics

DesignScript is a written computer programming language, and as such each program starts as a pure text file, appended with the extension ".ds". This file can be written and edited in any number of text editors, which can be especially useful if you have previous familiarity with a specific application, or with the DesignScript Editor if you want to take advantage of more advanced software development features where execution and debugging facilities are integrated with the program editor. A DesignScript program is executed by either opening a pre-written text file in the DesignScript editor or typing it in the Editor's text box, and then running it by pressing the play button.





Every program is a series of written commands. Some of these commands create geometry; others solve mathematical problems, write text files, or print text into the output text box, called the console. A simple, one line program which writes the quote "Less is more." looks like this:

```
s = Print("Less is more.");
```

```
• Less is more.
```

The grey box on the left shows the console output of the program.

The command **`Print`** is called a function, and the parentheses indicate that the function takes a number of arguments, data the function needs to complete its task. The **`Print`** function takes a single, string-type argument, and prints that same argument to the console. Strings in DesignScript are designated by two quotation marks (**`"`**), and the enclosed characters, including spaces, are passed to the function. Many functions can take

arguments of different types, including **Print**. For instance, a program to print the number 5420 looks like this:

- 5420

```
s = Print (5420);
```

Every command in DesignScript is terminated by a semicolon. If you do not include one, the Editor will display an error and won't execute the program. 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.

```
   s
           = Print (
     "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 DesignScript, a single line of code is "commented" with two forward slashes, **//**. This makes the Editor 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, ***/**.

- Less is more.

```
// 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
s = Print("Less is more.");
```

So far the arguments to the Print function 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."
- Less is more

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

s = Print(quote);
```

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.

- quote = "Less is more."
- Less is more. Less is more. Less is more.

```
// My favorite architecture quote

quote = "Less is more.";

s = Print(quote + " " + quote + " " + quote);
```

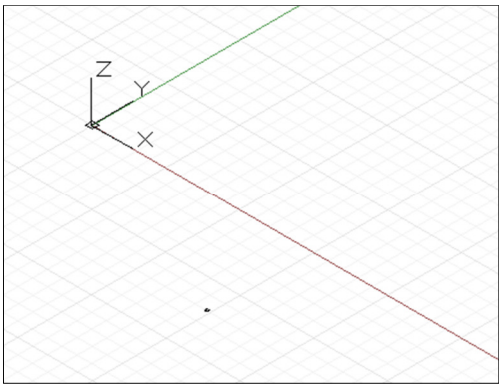
Here we are using in the Print function to write a quote by Mies van der Rohe three times, with spaces between each phrase. The command joins literal strings `" "`, with references to a strings via a variable. Notice also the use of the + sign to 'concatenate' the strings and variables together to form one continuous output.

- quote = "Less is bore."
- Less is bore. Less is bore. Less is bore.

```
// My NEW favorite architecture quote

quote = "Less is bore.";

s = Print(quote + " " + quote + " " + quote);
```

2: Geometry Basics

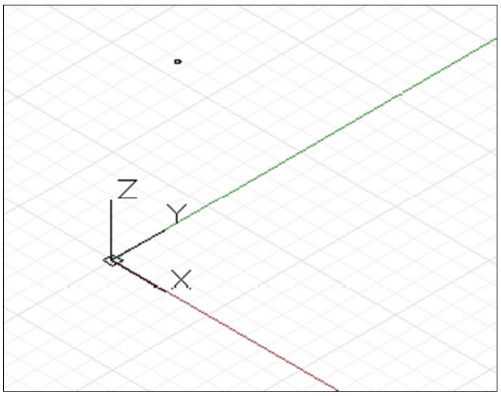
The simplest geometrical object in the DesignScript standard geometry library is a point. All geometry is created using special functions called constructors, which each return a new instance of that particular geometry type. In DesignScript, constructors begin with the name of the object's type, in 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:




```
import("ProtoGeometry.dll");

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

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

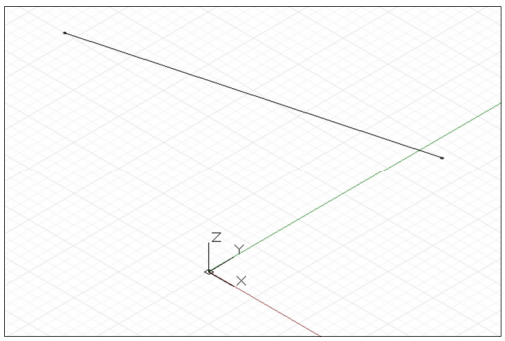
Constructors in DesignScript are typically designated with the "`By`" prefix, and invoking these functions returns a newly created object of that type. This newly created object is stored in the variable named on the left side of the equal sign, and any use of that same original Point.

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):



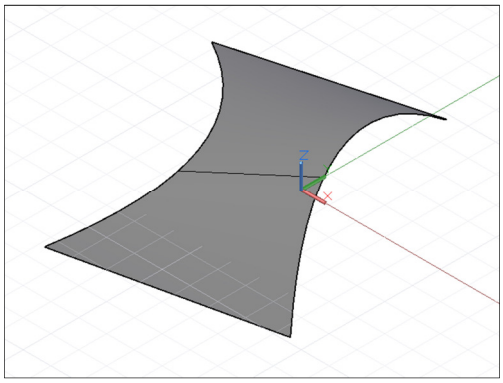

```
import("ProtoGeometry.dll");

// 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:



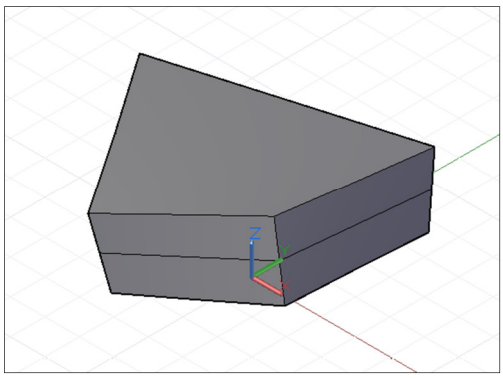

```
import("ProtoGeometry.dll");

// 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.



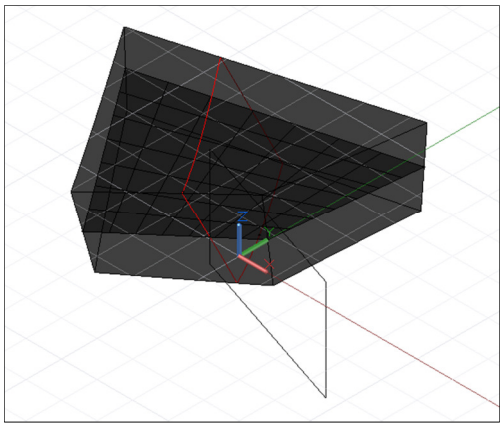

```
import("ProtoGeometry.dll");

// 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.LoftFromCrossSections({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.



```
import("ProtoGeometry.dll");

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.LoftFromCrossSections({l1, l2});

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.



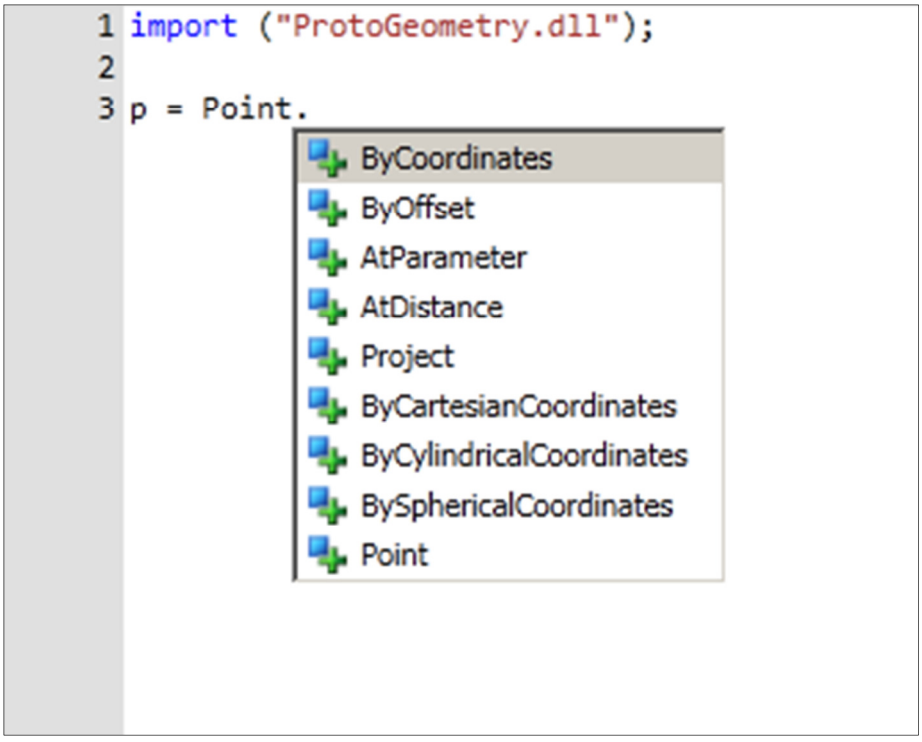
```
import("ProtoGeometry.dll");

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.LoftFromCrossSections({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: IDE Basics: Autocomplete, Stepping, Bugs, Error Messages

The DesignScript editor window provides a convenient package for users to write and edit scripts, execute scripts' commands, and track down the source of problems when execution goes wrong. Systems like the DesignScript editor are called Integrated Development Environments (IDEs), and understanding a programming language's IDE is crucial to effectively write code.

One of the most useful IDE tools for programmers new to a language is autocomplete, which searches through a list of programming language terms and suggests names which would lead to a valid syntax in your program. Autocomplete can be both a powerful tool for avoiding typos in your document, as well as a means of exploring the available programming commands of a language and its libraries.

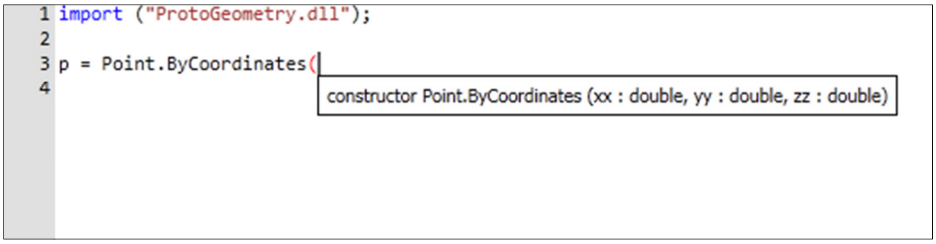
Suppose you want to create a Point in DesignScript, but don't know the proper syntax to create one. Rather than consulting the DesignScript reference documentation, it is instead possible to simply write `Point.`, indicating you want to search through the commands associated with the Point type, and a list of methods associated with Point will be presented in a popup window. We can tell by the names of functions in this list that the methods starting with "`By...`" are constructor methods, in other words they can be used to create new instances of Point objects.



As you continue typing, the list will progressively narrow to contain only the letters typed so far. A single method name at the top of the list will be

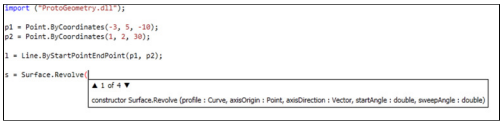
highlighted. Pressing enter or return will fill in this name into your document. Alternatively, it is possible to simply click on the desired method name, and it will be entered into your document.

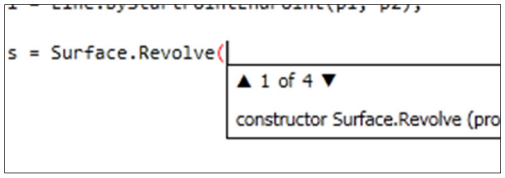
Simply knowing the proper name of a method isn't sufficient to using it properly, as each method requires a specific set of inputs in order to function. Typing an open parenthesis symbol "`(`" brings up a list of required arguments. In this example, we can see that the Point constructor `ByCoordinates` takes three arguments: x, y, and z positions in 3D space. Each argument is separated by a comma, with a description of the argument on the left side of the colon, and the argument type on the right side.



Some methods have more than one valid argument list, in which case they become "overloaded" methods. When a method is overloaded, up and down arrows appear in the upper left corner of the autocompletet window, and pressing the up and down arrows on the keyboard will cycle through a list of available argument combinations.

For instance, when typing the following script:






```
import("ProtoGeometry.dll");

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

l = Line.ByStartPointEndPoint(p1, p2);

s = Surface.Revolve(l, Point.ByCoordinates(0, 0, 0),
    Vector.ByCoordinates(0, 0, 1), 0, 360);
```

DesignScript will present a list of four availably argument combinations after typing in "`s = Surface.Revolve(`".

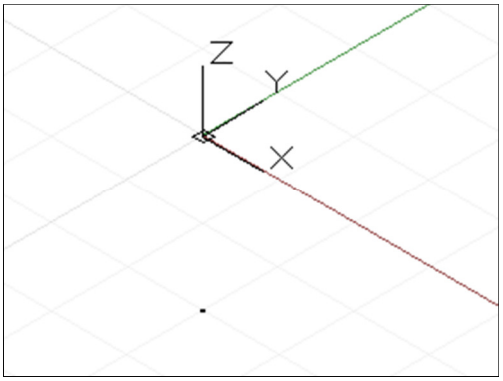
When a DesignScript script is executed by pressing the Run button, the compiler attempts to execute the entire script, stopping only when it reaches the end of the document or encounters mal-formed code, called a "bug". Often it is useful to run script commands one at a time both from a

geometric and language perspective: the construction of complex geometric forms can more easily be observed when watching geometric primitives combine into more sophisticated shapes, and bugs are often more easily found when stepping through code, observing precisely where a program's data goes awry. This form of script execution is called "stepping", and the act of tracking down mal-formed code bugs is called "debugging".

To step through a program, press the "Run (Debug)" play button in the DesignScript IDE. Subsequent lines of a program are executed by repeatedly pressing the "Step" button. If we step through the preceding script, the current line of execution is highlighted in green, generating geometry in the main window. Stepping through the program generates two points, a line from these points, and finally a hyperboloid by rotating this line.

When the DesignScript compiler encounters code which it doesn't recognize, error messages are printed to the Errors tab of the Editor window. After printing an initial error message, execution continues despite encountering a problem, often leading to a cascading series of error messages, as mal-formed objects are fed into functions, leading to more mal-formed objects and more error messages. This is one of the reasons why stepping through code one line at a time is often required to find the exact location of a bug.

If we modify the preceding example to contain a bug:




```
import("ProtoGeometry.dll");

p1 = Point.ByCoordinates(-5, 5, -10);

// this line contains a bug, but we don't know that yet
p2 = Point.ByCoordinates(2, 6, 10, 0);

l = Line.ByStartPointEndPoint(p1, p2);

s = Surface.Revolve(l, Point.ByCoordinates(0, 0, 0),
    Vector.ByCoordinates(0, 0, 1), 0, 360);
```

Stepping though the program, in combination with the printed error messages, can help us locate the source of the problem. Executing the program with the Run button generates the following output:

```
!> Function 'ByCoordinates' not found

!> Value cannot be null.
   Parameter name: endPt

!> the profile is null
   Parameter name: profile
```

Which by their own are quite cryptic. If we run the script in Debug mode, we get an initial error as before: "`Function 'ByCoordinates' not found`," which does little to help find the bug. As we step through the code, we can see that the after the line "`p2 = Point.ByCoordinates(2, 6, 10, 0);`," an error is raised. This tells us that the `Point.ByCoordinates` isn't executing correctly. Indeed, the DesignScript language documentation and autocomplete say that `Point.ByCoordinates` takes three arguments, not four. This identifies the location of our bug.

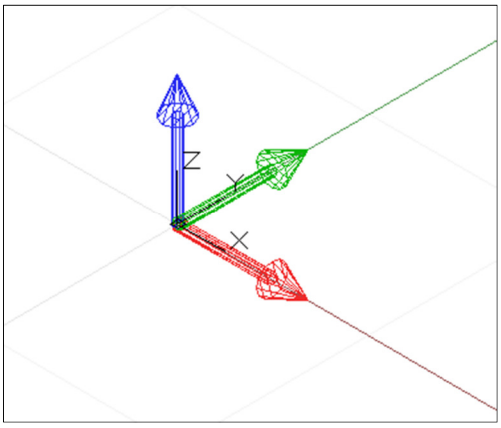
The stepping functionality of the DesignScript IDE allows a programmer to execute their code in a slow, methodical manner, after each step checking to see if the produced output matches a proposed hypothesis, as well as the programmer's own mental model of what the script is doing.

4: Geometric Primitives

While DesignScript is capable of creating a variety of complex geometric forms, simple geometry 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.

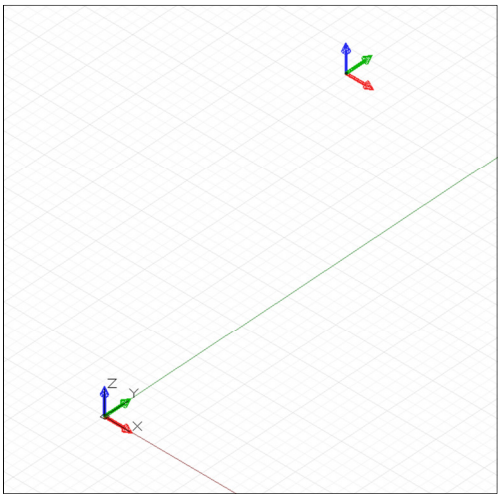
While not strictly a piece of geometry, the CoordinateSystem is an important tool for constructing geometry. A CoordinateSystem object keeps track of both position and geometric transformations such as rotation, sheer, and scaling.

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 or the WCS (World Coordinate System) property:




```
import("ProtoGeometry.dll");

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

cs = CoordinateSystem.Identity();

// create an identical CoordinateSystem with a slightly
// simpler syntax
cs2 = CoordinateSystem.WCS;
```

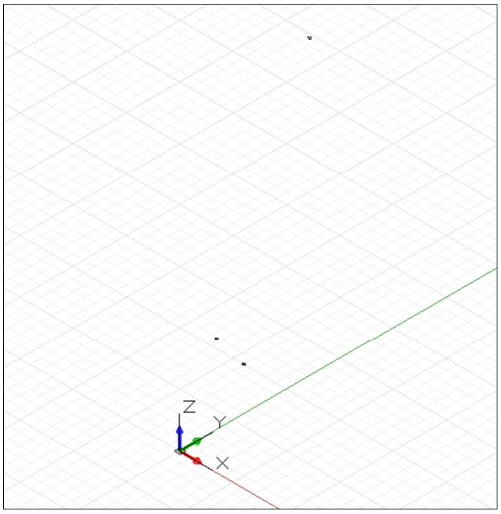
CoordinateSystems with geometric transformations are beyond the scope of this chapter, though another constructor allows you to create a coordinate system at a specific point, `CoordinateSystem.ByOriginVectors`:




```
import("ProtoGeometry.dll");

// 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.

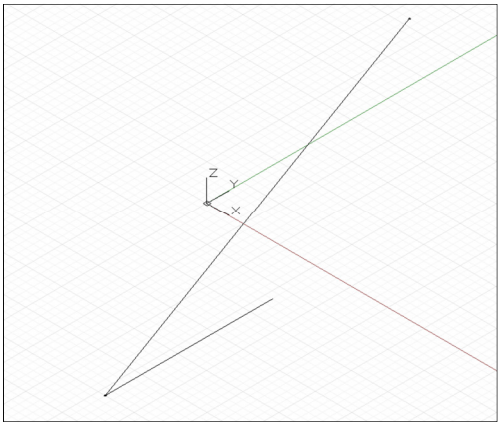
This example shows points created at various coordinate systems:




```
import("ProtoGeometry.dll");

// 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);
```

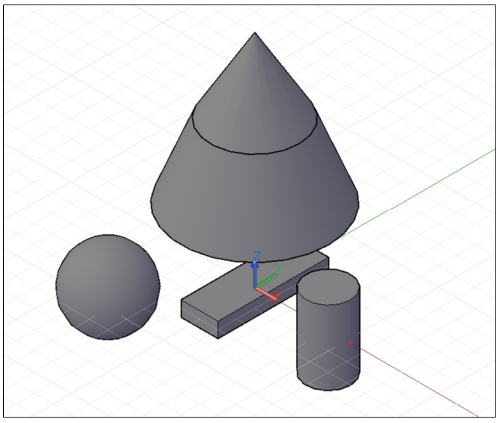
The next higher dimensional DesignScript primitive is a line segment, representing an infinite number of points between two end points. Lines can be created by explicitly stating the two boundary points with the constructor `Line.ByStartPointEndPoint`, or by specifying a start

point, direction, and length in that direction, `Line.ByStartPointDirectionLength`.




```
import("ProtoGeometry.dll");

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);
```

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



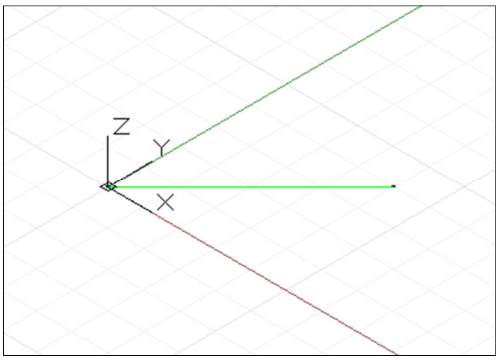
```
import("ProtoGeometry.dll");

// 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);

l = Line.ByStartPointEndPoint(p2, p3);

cone1 = Cone.ByStartPointEndPointRadius(p1, p2, 10, 6);
cone2 = Cone.ByCenterLineRadius(l, 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);
```

5: Colors

The tools of computational design allow scripts to generate an incredibly large amount of geometry with minimal effort. Unless steps are taken in a script to organize and label this geometry, the resulting objects can become an indiscernible mess.

The simplest way to label an object is to set its color. This is done by assigning a new Color object to the object's Color attribute. A Color object can be created either explicitly from a set of predefined colors, or with four integer values, ranging from 0 to 255, representing the object's alpha, red, green, and blue color components. A value of 0 means the Color has no amount of that color component, while 255 means it has 100% of that color component. For a Color's alpha, 0 means the object will be completely transparent, while 255 means the object will be completely opaque. Setting a Color by explicit RGB values can be very useful as you can select these objects based on their color in the AutoCAD host.

For instance, we can create a Line, and color it green by setting the Red = 0, Green = 255, and Blue = 0




```
import("ProtoGeometry.dll");

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

l = Line.ByStartPointEndPoint(p1, p2);

l.Color = Color.ByARGB(255, 0, 255, 0);
```

A green line appears in the output window. If we wanted to select all the green lines in the AutoCAD window, we can use the "`qselect`" command. Type `qselect` in the AutoCAD command line (not the DesignScript editor), select "Select Color..." in the Value menu, and enter the values Red=0, Green=255, Blue=0 in the "True Color" tab. All the lines labeled with blue are now selected.

The following colors are predefined in DesignScript:

Black	White	Red	Green
Blue	Yellow	Cyan	Magenta
Purple	Periwinkle	Orange	

6: 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.

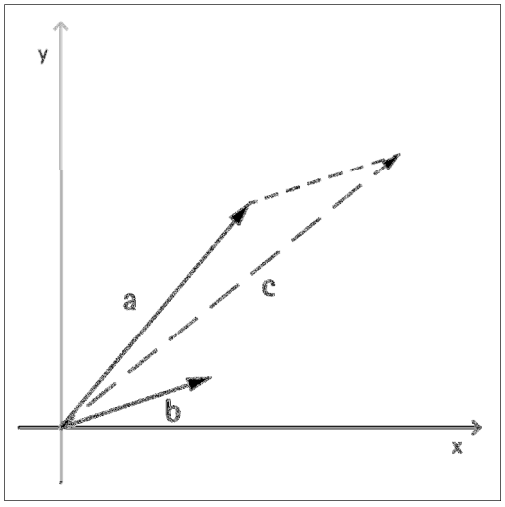
At its most basic, a vector represents a position in 3D space, and is often times thought of as the endpoint of an arrow from the position (0, 0, 0) to that position. Vectors can be created with the `ByCoordinates` constructor, taking the x, y, and z position of the newly created Vector object. Note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

```
import("ProtoGeometry.dll");

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

s = Print(v.X);
s = Print(v.Y);
s = Print(v.Z);
```

- 1.0
- 2.0
- 3.0

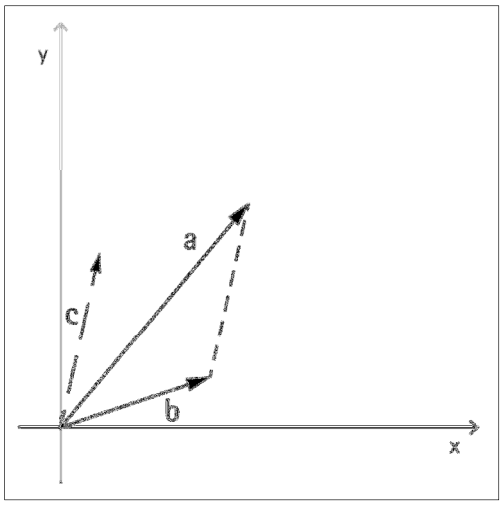
A set of mathematical operations are defined on Vector objects, allowing you to add, subtract, multiply, and otherwise move objects in 3D space as you would move real numbers in 1D space on a number line.

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 `Vector.op_Addition` method, and is represented by the diagram on the left.



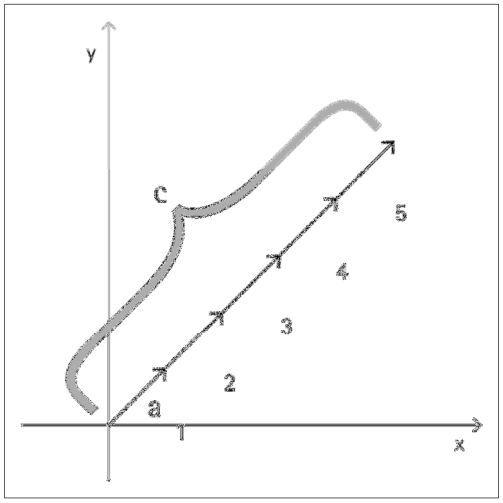

```
import("ProtoGeometry.dll");

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

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

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




```
import("ProtoGeometry.dll");

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

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

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




```
import("ProtoGeometry.dll");

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

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

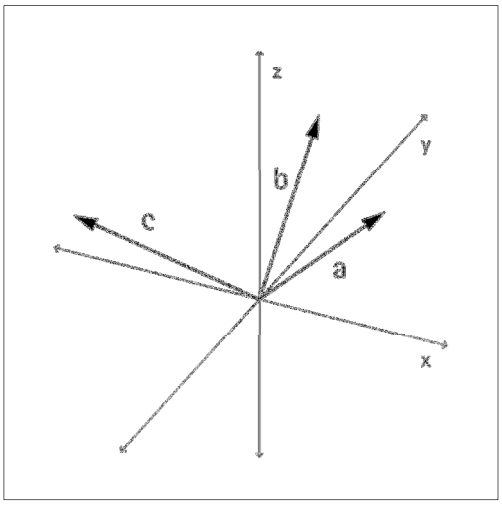
Often it's desired when scaling a vector to have the resulting vector's length *exactly* equal to the scaled amount. This is easily achieved by first normalizing a vector, in other words setting the vector's length exactly equal to one.

- a_len = 3.75
- len = 5.0

```
import("ProtoGeometry.dll");

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

// set the a's length equal to 1.0
b = a.Normalize();
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.




```
import("ProtoGeometry.dll");

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.

- -7.00000

```
import("ProtoGeometry.dll");

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

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

s = Print(d);
```

7: 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 DesignScript programmer the means to express sets of values as parameters on either side of two dots (`..`), generating intermediate numbers between these two extremes.

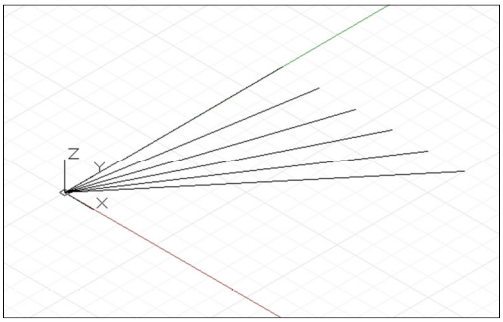
For instance, while we have seen variables containing a single number, it is possible with range expressions to have variables which contain a set of numbers. The simplest range expression fills in the whole number increments between the range start and end.

```
a = 10;
b = 1..6;

s = Print(a);
s = Print(b);
```

- 10
- {1, 2, 3, 4, 5, 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, DesignScript returns a range of lines.




```
import("ProtoGeometry.dll");

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;

s = Print(a);
```

- {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}

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;

s = Print(a);
```

- {0.0, 0.75, 1.5, 2.25, 3.0, 3.75, 4.5, 5.25, 6.0, 6.75}

If a design requires a generated range expression to end precisely on the maximum range expression value, DesignScript 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;

s = Print(a);
```

- {0.0, 0.777778, 1.555556, 2.333333, 3.111111, 3.888889, 4.666667, 5.444444, 6.222222, 7.0}

However, if you want to DesignScript 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;

s = Print(a);
```

- {0.0, 0.875, 1.75, 2.625, 3.5, 4.375, 5.25, 6.125, 7.0}

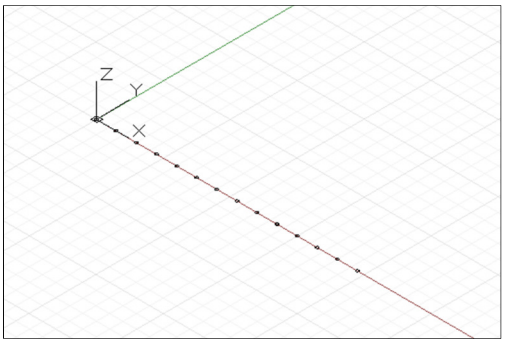
8: Collections

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.

```
import("ProtoGeometry.dll");

// use a range expression to generate a collection of
// numbers
nums = 0..10..0.75;
s = Print(nums);

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

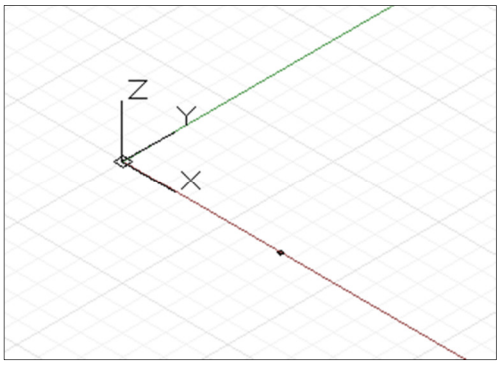
- `{0.0, 0.75, 1.5, 2.25, 3.0, 3.75, 4.5, 5.25, 6.0, 6.75, 7.5, 8.25, 9.0, 9.75}`



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:

```
import("ProtoGeometry.dll");

// a collection of numbers
nums = 0..10..0.75;
s = Print(nums);
// create a single point with the 6th element of the
// collection
points = Point.ByCoordinates(nums[5], 0, 0);
```

- `{0.0, 0.75, 1.5, 2.25, 3.0, 3.75, 4.5, 5.25, 6.0, 6.75, 7.5, 8.25, 9.0, 9.75}`



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

- {0, 1, 100, 3, 4, 200, 6}

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

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

s = Print(a);
```

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:

- {45, 67, 22}
- {45, 67, 22}

```
// 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;

s = Print(a);
s = Print(b);
```

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):

- {5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
- {6, 8, 10, 12}

```
a = 5..20;

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

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

s = Print(a);
s = Print(b);
```

DesignScript 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.

- 10

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

num_elements = Count(a);

s = Print(num_elements);
```

9: Number Types

There are two fundamental types of numbers in DesignScript: integers, representing discrete whole number values, and floating point numbers, representing values with decimal points. The DesignScript compiler automatically creates a number type based on the starting value of the number: if it contains a decimal point (`.`), the compiler creates a floating point number. Otherwise, it creates an integer.

```
// create an integer with value exactly 1
i = 1;

// create a floating point number with approximate
// value 1
f = 1.0;

s = Print(i);
s = Print(f);
```

- 1
- 1.000000

Integers represent discrete, absolute values; an integer with value 1 is unequivocally 1, and will never change. However, floating point values, due to the limitations of computers, should be thought of as *approximates* for the values specified. While it is easy for a human to look at the numbers `1.0` and `1.00000000000000000001` and tell that they are different (by `0.00000000000000000001`), computers simply cannot resolve differences this small. To a computer, `1.0` and `1.00000000000000000001` *are* the same number. While this doesn't pose a problem for almost all designs (the number `0.00000000000000000001` is imperceptible regardless of if the units are meters, kilometers, or millimeters), it is something to be aware of. As a rule of thumb, DesignScript can only keep track of 6 decimal places of precision. If you try creating numbers outside this limit, DesignScript will round to the nearest number. For example:

```
// create a floating point number approximately 1.0
f1 = 1.0;

// attempt to create a floating point number beyond the
// precision of DesignScript. This number will be
// rounded. f1 and f2 become the same number
f2 = 1.00000000000000000001;

s = Print(f1);
s = Print(f2);
```

- 1.000000
- 1.000000

The discrete precision of integers makes them appropriate for counting and indexing members of collections; it should be obvious that it makes no sense to access the 4.78^{th} element of a collection, or generate 2.3 line intersection operations. Nevertheless, DesignScript will try to make its best guess as to what was the intended index by rounding to the closest integer:

```
import("Math.dll");

// create an index
index = 1.6;

// create a collection
collection = 1..10;

// DesignScript will round to the closest integer, in
// this case 2
estimated_value = collection[index];

s = Print(estimated_value);

// Floor converts a floating point value to an integer
// by always rounding down to a lower integer
value = collection[Math.Floor(index)];

s = Print(value);
```

- 3
- 2

Floating point numbers on the other hand, are used for most geometric types and operations, to denote position, distance, direction, angles, etc.

Because integers are limited to whole numbers only, two integers can be tested for equality against other integers in a predictable manner. Floating point numbers, on the other hand, can be tested for equality, but with sometimes unexpected results.

```
// create two seemingly different numbers
a = 1.0;
b = 0.9999999;

// test if the two numbers are equal
s = Print(a == b ? "Are equal" : "Are not equal");
```

- Are equal

With integers, on the other hand, we get the expected result:

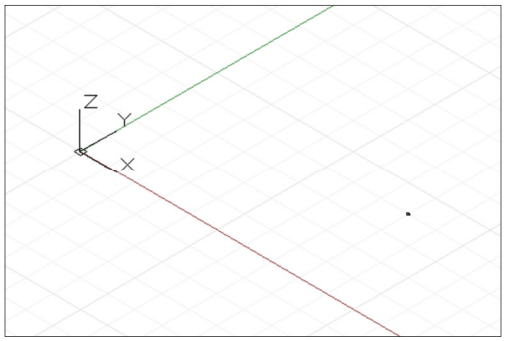
```
// create two seemingly different numbers
a = 100000000;
b = 99999999;

// test if the two numbers are equal
s = Print(a == b ? "Are equal" : "Are not equal");
```

- Are not equal

10: Associativity

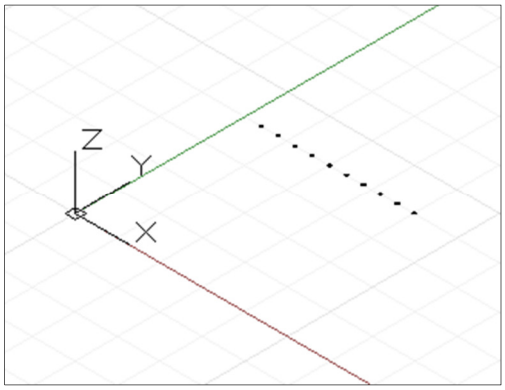
DesignScript scripts are sets of commands, all of which take inputs and return outputs. Variables act as a glue between DesignScript commands, acting as the inputs to functions, and saving the outputs from functions. The connection between functions and variables is significantly stronger in DesignScript than other programming languages: once a variable is used in a function, any changes to that variable later in the program will cause the original function to update. This is called associativity.

For example, if we create a variable **x**, and use that variable to create a Point **p**, but then later change **x**, **p** will change too.




```
import("ProtoGeometry.dll");

// create a variable with value 10
x = 10;

// create a Point at x = 10, y = 10, z = 0
p = Point.ByCoordinates(x, 10, 0);

// update x to be 20
// this will implicitly cause an update to p, calling
// p = Point.ByCoordinates(x, 10, 0) automatically
x = 20;
```

Associativity can accommodate a variety of changes to a variable. For instance, a variable go from being a single value to a collection, or vice versa.




```
import("ProtoGeometry.dll");

x = 10;

p = Point.ByCoordinates(x, 10, 0);

// updating x to be a collection of 10 numbers causes
// p to become a collection of 10 Points
x = 1..10;
```

A DesignScript programmer should be aware that associative changes can have unexpected repercussions on a program, for instance if a variable changes into an incompatible type used by a dependent constructor or function the script will break.

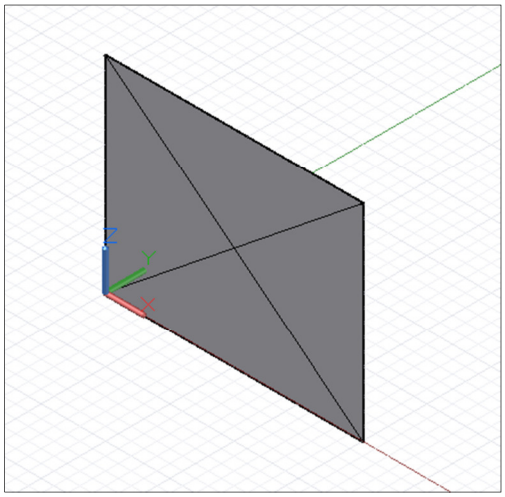
- ERROR

```
import("ProtoGeometry.dll");

x = 10;

p = Point.ByCoordinates(x, 10, 0);

// updating x to be a Point causes the script to fail
x = Point.ByCoordinates(1, 2, 3);
```

11: 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:



```
import("ProtoGeometry.dll");

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.ExtrudeAsSurface(8, Vector.ByCoordinates(0, 0,
    1));

// 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);
s = Print(times_two);
```

- 20

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();
s = Print(gr);
```

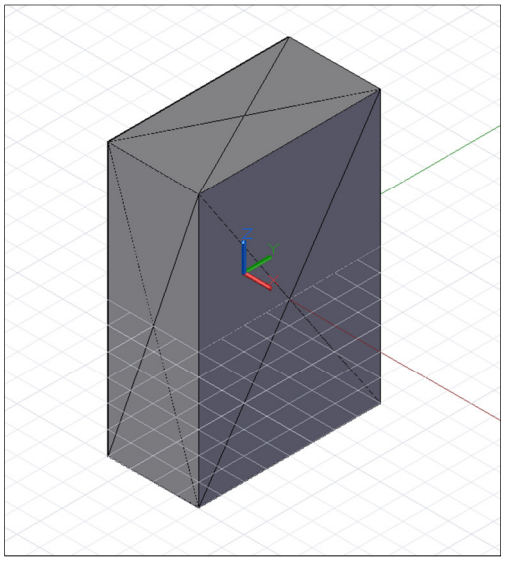
- 1.618034

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:

- {1, 2}

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

two_nums = returnTwoNumbers();
s = Print(two_nums);
```

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.



```
import("ProtoGeometry.dll");

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);
```

12: Math

The DesignScript standard library contains an assortment of mathematical functions to assist writing algorithms and manipulating data. To use the standard math library, you must import the Math.dll library with the following line of code:

```
import ("Math.dll");
```

Math functions are prefixed with the `Math` namespace, requiring you to append functions with "`Math.`" in order to use them.

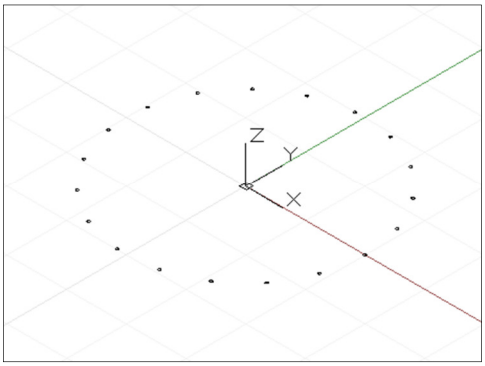
The functions `Floor`, `Ceiling`, and `Round` allow you to convert between floating point numbers and integers with predictable outcomes. All three functions take a single floating point number as input, though `Floor` returns an integer by always rounding down, `Ceiling` returns an integer by always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

```
import ("Math.dll");

val = 0.5;

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

- f = 0
- c = 1
- r = 0
- r2 = 1

DesignScript 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:




```
import("ProtoGeometry.dll");
import("Math.dll");

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.

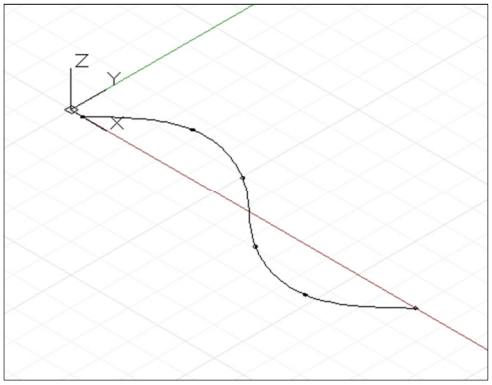
```
s = Print(7 % 2);
s = Print(6 % 2);
s = Print(10 % 3);
s = Print(19 % 7);
```

- 1
- 0
- 1
- 5

13: Curves: Interpreted and Control Points

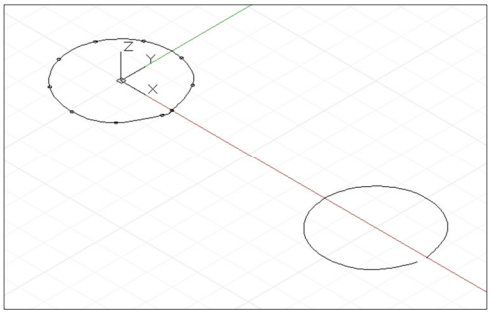
There are two fundamental ways to create free-form curves in DesignScript: specifying a collection of Points and having DesignScript interpret a smooth curve between them, or a more low-level method by specifying the underlying control points of a curve of a certain degree. Interpreted curves are useful 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 `BSplineCurve.ByPoints` method.



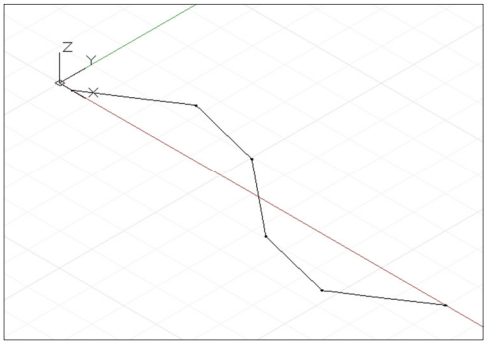

```
import("ProtoGeometry.dll");
import("Math.dll");

num_pts = 6;

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

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

int_curve = BSplineCurve.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. DesignScript will automatically fill in the missing segment, so a duplicate end point (identical to the start point) isn't needed.



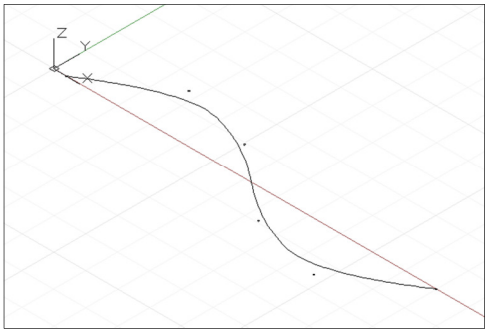

```
import("ProtoGeometry.dll");
import("Math.dll");

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

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

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

BSplineCurves 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.



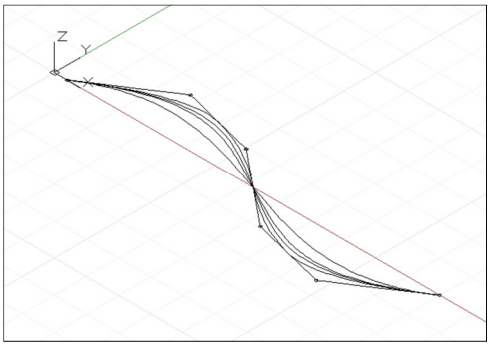

```
import("ProtoGeometry.dll");
import("Math.dll");

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 = BSplineCurve.ByControlVertices(pts, 1);
```

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



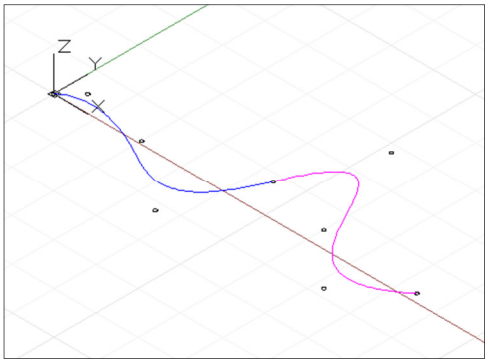

```
import("ProtoGeometry.dll");
import("Math.dll");

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 = BSplineCurve.ByControlVertices(pts, 2);
```

DesignScript supports B-Spline curves up to degree 11, and the following script illustrates the effect increasing levels of smoothing has on the shape of a curve:




```
import("ProtoGeometry.dll");
import("Math.dll");

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 = BSplineCurve.ByControlVertices(pts,
         degree);
}

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

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 BSpline curves which are nevertheless as smooth as one curve:



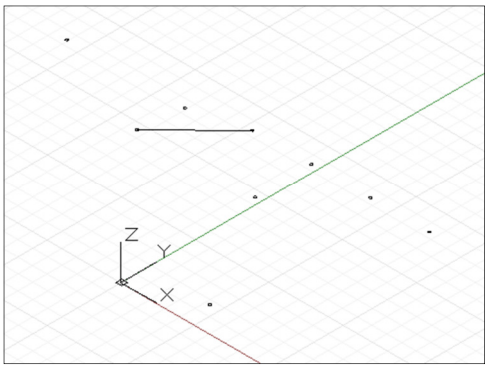

```
import("ProtoGeometry.dll");

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 = BSplineCurve.ByControlVertices(pts_1, 3);
crv_1.Color = Color.ByARGB(255, 0, 0, 255);

pts_2 = {};

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

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 = BSplineCurve.ByControlVertices(pts_2, 3);
crv_2.Color = Color.ByARGB(255, 255, 0, 255);
```

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

14: IDE: Step In, Step Out, Watching, Breakpoints

When bugs appear in a DesignScript script, the DesignScript IDE has several tools to assist tracking them down. Stepping, as previously mentioned, allows a programmer to slow down the execution of a script, examining the side effects of each line of code one line at a time.

Breakpoints allow a programmer to similarly slow down the execution of a program, though only stopping at specific lines of code. This can be useful in longer programs where a programmer has an understanding of a majority of the code base, but needs to slow down execution in a few selective moments. For instance, much of your code might be so familiar you don't question whether or not it contains a bug. Other parts of a script, on the other hand, may be sufficiently new or complex to warrant a slower execution when looking for a bug. For instance, the following example contains several lines of straight-forward code to generate geometry in an array, as well as a more complex piece of code to generate a subset from this collection. As it happens, this more complex piece of code contains a bug, and setting a breakpoint at or right before this line of code allows a programmer to skip the tedious task of stepping through the earlier geometry code.

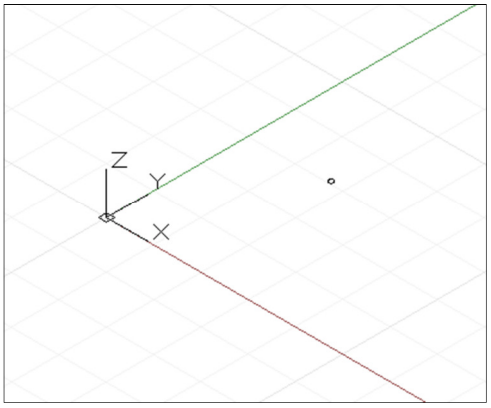



```
import("ProtoGeometry.dll");

geometry = {};

geometry[0] = Point.ByCoordinates(4.5, 8.9, 110.0);
geometry[1] = Point.ByCoordinates(34.0, 77.0, 90.3);
geometry[2] = Line.ByStartPointEndPoint (geometry[0],
    geometry[1]);
geometry[3] = Point.ByCoordinates(-1, -45, 200.1);
geometry[4] = geometry[1].Translate(50, 0, 0);
geometry[5] = Point.ByCoordinates(54, 21, 0.24);

// The programmer suspects a bug is here.
// Setting a breakpoint at or before this line can
// assist finding the bug
subset = geometry[0..8];

subset_move = subset.Translate(100, 0, 0);
```

Another debugging tool to help programmers sift through the large amount of data contained in a script is the ability to watch specific variables during code execution. By default, the DesignScript IDE displays information about the contents of every variable's value for each line of code, and in complex scripts with many co-dependences between variables

this generates a substantial amount of output. While this does present a comprehensive view of everything that's changing in a script, it is often more useful to only view the changes to a specific variable. To watch a specific variable, switch to the Watch tab in the bottom of the DesignScript IDE and click on an empty cell in Name column. Now, when you step through the code, the value of watched variables will updated in the Value column.

Programmers use functions to organize code, and large programs contain numerous functions, often nested inside each other. The default behavior of the DesignScript IDE is to report the values returned from a function, and continue executing past the function. If a programmer is curious about the code contained inside of a function, he or she must tell the compiler to enter into a function and begin stepping at the start of the function's code. This is called "stepping in" to a function. For instance, if we take the following code example:




```
import("ProtoGeometry.dll");
import("Math.dll");

def complexFunction()
{
    num = 2.34057;
    v1 = Math.Log(num);
    v2 = Math.Sqrt(num);
    return = { v1, v2 };
}

result = complexFunction();

point = Point.ByCoordinates(result[0], result[1], 0);
```

As we can see, most of complexity of this script is contained inside of `complexFunction`. If we were to step through the script by simply pressing the next button repeatedly, we wouldn't be able to view the inner workings of this function. By pressing the "Step In" button when the debugger has highlighted `result = complexFunction()`, the compiler jumps to the top of the script and execution continues inside this function. Note that you can continue stepping into the functions contained inside of functions to an unlimited level of depth.

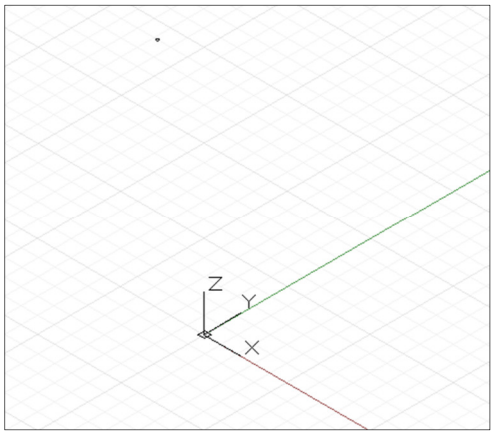
Functions can often times become quite lengthy, and a programmer may find that before reaching the end of a function, they have examined all relevant lines of code; in this case manually stepping through the rest of

the function would be both unnecessary and time consuming. In these situations, a programmer can immediately exit from a function, resuming execution where the function returns a value. This is called “Stepping Out” of a function. To step out of a function, simply press the Step Out button while in Debug mode.

15: Translation, Rotation, and Other Transformations

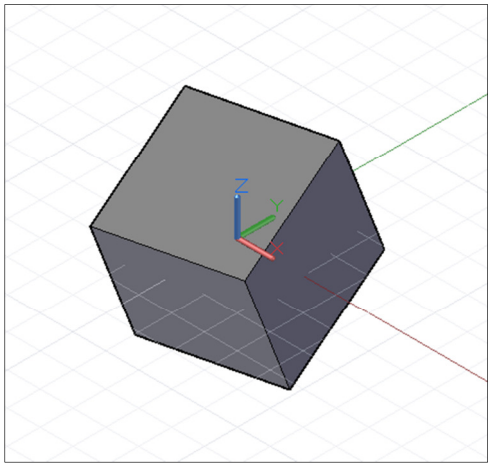
Certain geometry objects can be created by explicitly stating x, y, and z coordinates in three-dimensional space. More often, however, geometry is moved into its final position using geometric transformations on the object itself or on its underlying CoordinateSystem.

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



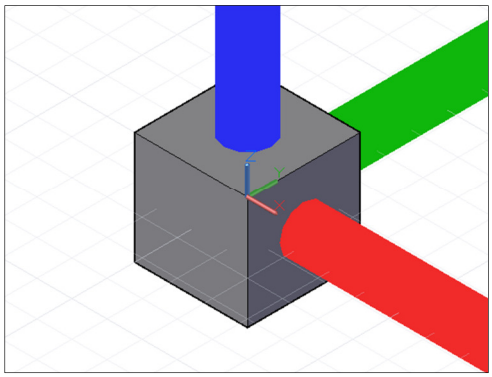

```
import("ProtoGeometry.dll");

// 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
// p's now position is x = 11, y = -18, z = 53
p = p.Translate(10, -20, 50);
```

While all objects in DesignScript 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:



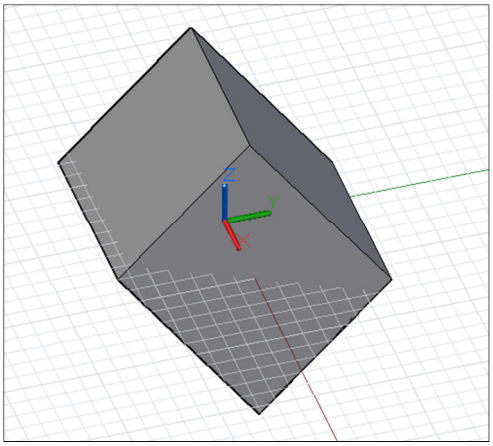

```
import("ProtoGeometry.dll");

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

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

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

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

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



```
import("ProtoGeometry.dll");

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

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

old_cs = CoordinateSystem.Identity();

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

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



```
import("ProtoGeometry.dll");

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

old_cs = CoordinateSystem.Identity();

cube = Cuboid.ByLengths(CoordinateSystem.WCS, 5, 5, 5);
sub_d = SubDivisionMesh.FromGeometry(cube, 10);

sub_d = sub_d.Transform(old_cs, new_cs);

cube.Visible = false;
```

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

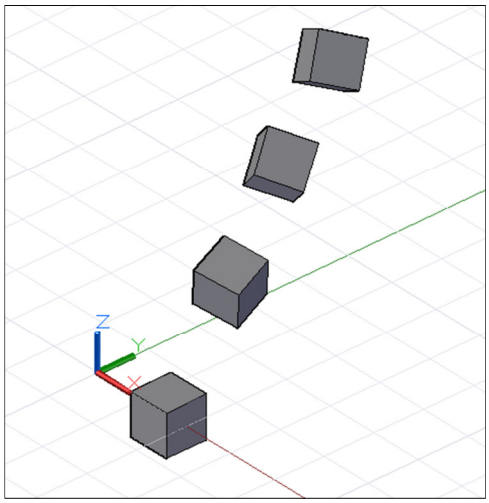
Class	Non-Uniformly Scaled CoordinateSystem	Sheared CoordinateSystem
Arc	Yes	Yes
BSplineCurve	Yes	Yes
BSplineSurface	Yes	Yes
Circle	Yes	Yes
Line	Yes	Yes
Plane	No	No
Point	Yes	Yes
Polygon	Yes	Yes
Solid	Yes	Yes
SubDivisionMesh	Yes	Yes
Surface	Yes	Yes
Text	No	No

16: Imperative and Associative Blocks

By default, DesignScript is an associative language (see: *Associativity*), which is suited for design exploration and iterative sketching. However, there are some instances when associative programming leads to complex algorithms and difficult to read code. In some of these instances, the complexity can be simplified by a more traditional and straight-forward programming style called Imperative programming invoked by declaring an Imperative key word or 'directive.' These blocks allow code to execute in a purely sequential manner: variables modified later in a program have no effect on the code which preceded it.

Imperative blocks are declared with the `[Imperative]` command, wrapping code in curly braces (`{}`). Like functions, Imperative blocks are required to return a value with the `return =` command.

The following example shows how to use an Imperative block to create a collection of Cuboids by reusing a CoordinateSystem from a previous operation:

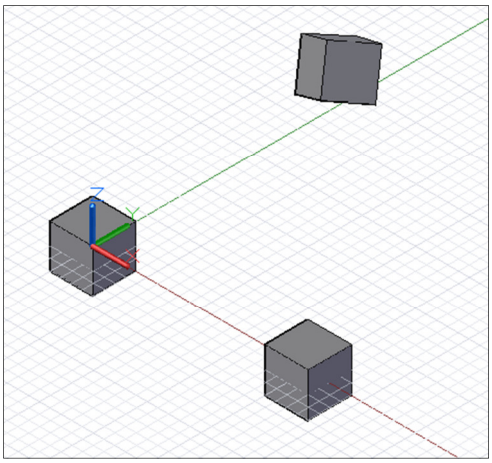



```
import("ProtoGeometry.dll");

cuboids = [Imperative]
{
    c = {};
    cs = CoordinateSystem.Identity();
    cs = cs.Translate(20, 0, 0);
    c[0] = Cuboid.ByLengths(cs, 10, 10, 10);
    cs = cs.Rotate(25, Vector.ByCoordinates(1, 1, 1));
    cs = cs.Translate(0, 20, 20);
    c[1] = Cuboid.ByLengths(cs, 10, 10, 10);
    cs = cs.Rotate(35, Vector.ByCoordinates(1, 1, 1));
    cs = cs.Translate(0, 20, 20);
    c[2] = Cuboid.ByLengths(cs, 10, 10, 10);
    cs = cs.Rotate(15, Vector.ByCoordinates(1, 1, 1));
    cs = cs.Translate(0, 20, 20);
    c[3] = Cuboid.ByLengths(cs, 10, 10, 10);

    return = c;
}
```

Despite the fact that all three Cuboids were created using the same CoordinateSystem variable with the exact same line of code, `Cuboid.ByLengths(cs, 10, 10, 10)`, all three Cuboids are different. If this example had been created in an Associative section of DesignScript,

all three Cuboids would be identical, with each update to the `cs` variable changing the dependent Cuboids.

DesignScript also allows a programmer to switch from an Imperative block back to an Associative block, if this leads to more legible code and cleaner algorithms. Similar to Imperative blocks, Associative blocks are designated with the `[Associative]` key word, also wrapping code in curly braces (`{}`), and also requiring the programmer to return a value with the `return =` command.



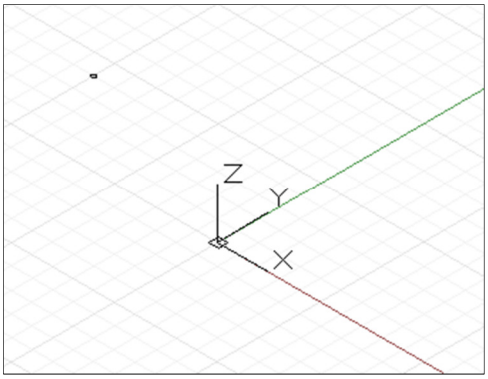
```
import("ProtoGeometry.dll");
import("Math.dll");

cuboids = [Imperative]
{
    c = {};
    cs = CoordinateSystem.Identity();
    c[0] = Cuboid.ByLengths(cs, 10, 10, 10);
    cs = cs.Translate(50, 0, 0);
    c[1] = Cuboid.ByLengths(cs, 10, 10, 10);

    rot = [Associative] {
        arr = 0..10;
        sqrts = Math.Sqrt(arr);
        return = Average(sqrts) * 40;
    }

    cs = cs.Rotate(rot, Vector.ByCoordinates(1, 1, 1));
    cs = cs.Translate(0, 50, 0);
    c[2] = Cuboid.ByLengths(cs, 10, 10, 10);

    return = c;
}
```

17: 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.

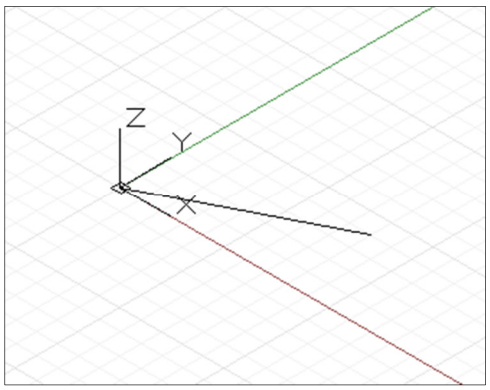
Boolean logic examines whether statements are true or false. Every statement in Boolean logic will be either true or false, there are no other states; no maybe, possible, or perhaps exist. The simplest way to indicate that a Boolean statement is true is with the **`true`** keyword. Similarly, the simplest 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:




```
import("ProtoGeometry.dll");

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:




```
import("ProtoGeometry.dll");

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`**:

- `true`

```
result = 10 < 30;

s = Print(result);
```

- `true`

```
result = 15 <= 15;

s = Print(result);
```

- `false`

```
result = 99 != 99;

s = Print(result);
```

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

- `false`

```
result = true && false;

s = Print(result);
```

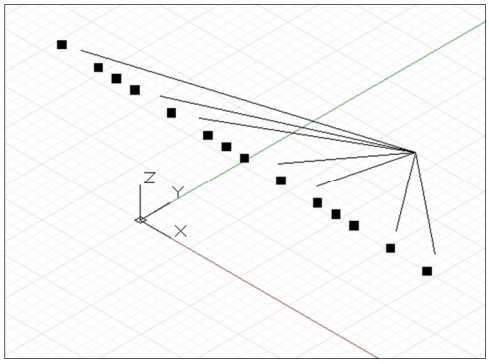
- `true`

```
result = true || false;

s = Print(result);
```

- `true`

```
result = !false;

s = Print(result);
```

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

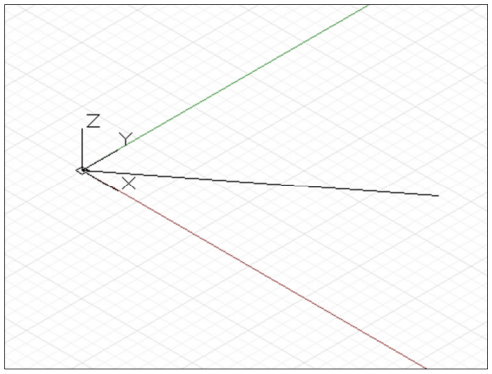



```
import("ProtoGeometry.dll");

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);
```

18: 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:




```
import("ProtoGeometry.dll");

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 DesignScript 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 DesignScript 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"**`)`**

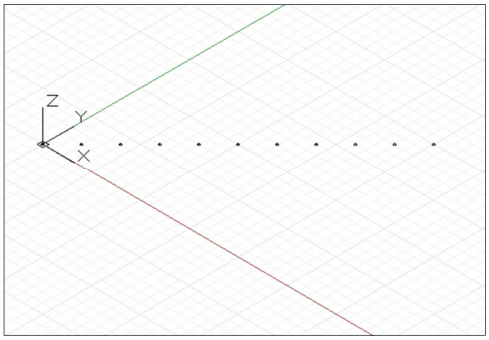
- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10

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

    for (i in collection)
    {
        s = Print(i);
    }
}
```

19: Replication Guides

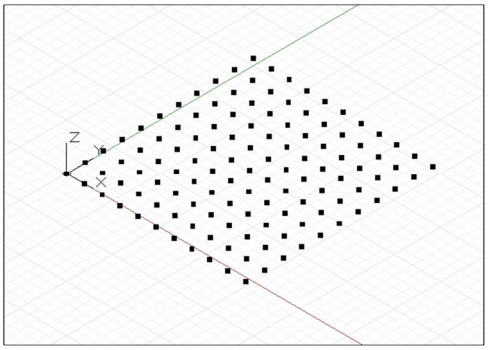
The DesignScript language was created as a domain-specific tool for architects, designers and engineers, and as such has several language features specifically tailored for these disciplines. A common element in these disciplines is the prevalence of objects arrayed repetitive grids, from brick walls 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:




```
import("ProtoGeometry.dll");

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 DesignScript generate a two-dimensional grid from the two one dimensional collections:

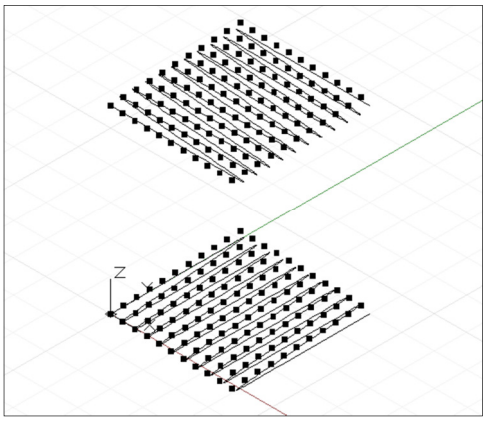



```
import("ProtoGeometry.dll");

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`, DesignScript 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**.

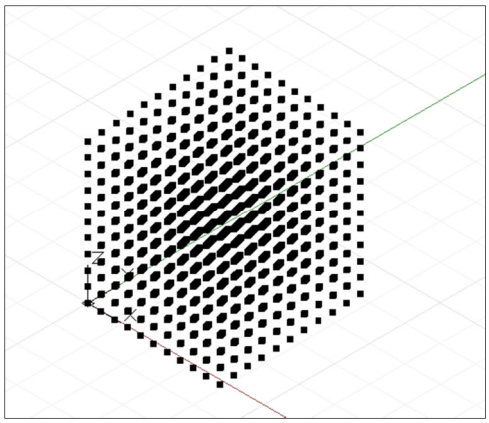
The order of the replication guide numbers (**<1>**, **<2>**, and/or **<3>**) determines the order of the underlying collection. In the following example, the same two one-dimensional collections are used to form two two-dimensional collections, though with the order of **<1>** and **<2>** swapped.



```
import("ProtoGeometry.dll");

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 = BSplineCurve.ByPoints(Flatten(p1));
curve2 = BSplineCurve.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>**.



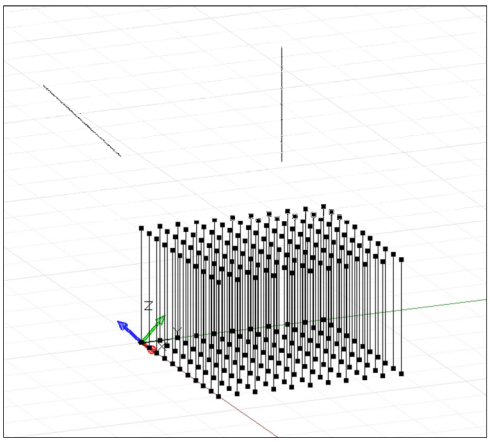
```
import("ProtoGeometry.dll");

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>);
```

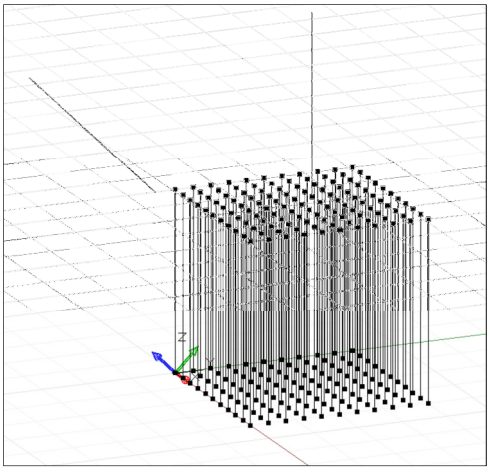
This generates every possible combination of values from combining the elements from **x_vals**, **y_vals**, and **z_vals**.

20: Modifier Stack and Blocks

Range expression, replication guides, and imperative loops offer a means to quickly and automatically generated collections of numbers in one, two, and three dimensions. However, design and aesthetic constraints may require some of these elements to deviate from their initial form. DesignScript allows a script to make changes to elements in an array by "stacking" transformations onto individual objects, all while maintaining the associative connection between elements.

For instance, we can use replication guides to generate two grids of Points, and use these to generate a grid of Lines. If we wanted one of these Lines to be moved and transformed, we can append this individual transformation to a pre-existing collection:

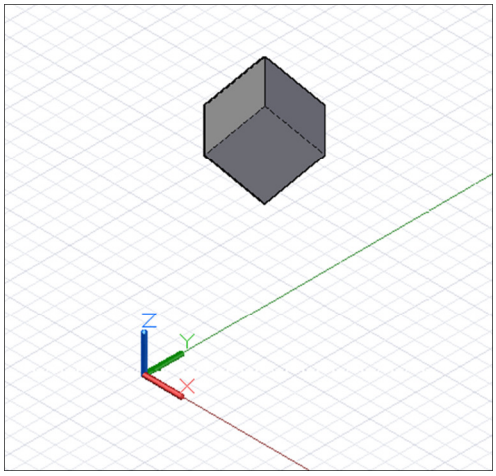




```
import("ProtoGeometry.dll");

height = 5;

p1 = Point.ByCoordinates((0..10)<1>, (0..10)<2>, 0);
p2 = Point.ByCoordinates((0..10)<1>, (0..10)<2>,
    height);

l = Line.ByStartPointEndPoint(p1, p2);

l[4][6] = l[4][6].Translate(0, 0, 10);

new_cs = CoordinateSystem.Identity();
new_cs = new_cs.Rotate(45, Vector.ByCoordinates(1,0,0));

l[4][6] = l[4][6].Transform(
    l[4][6].ContextCoordinateSystem, new_cs);

height = 10;
```

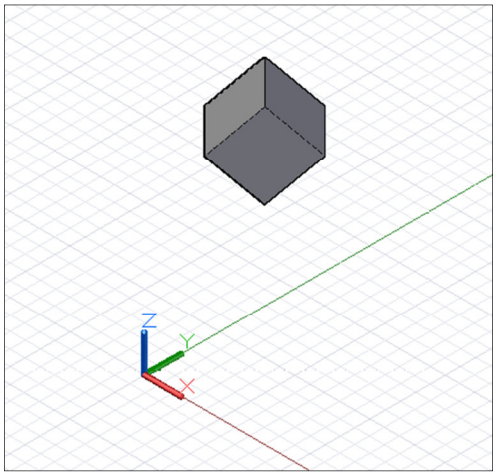
As we can see by stepping through this code, when **`height`** changes, **`p2`** updates, updating **`l`**. However, the modification to **`l[4][6]`** is preserved.

A related modifier syntax is the modifier block, allowing an object to be modified immediately after creation without the need for temporary variables. For instance, if a script required a solid to be translated and transformed immediately after creation, one way to structure the script is the following:



```
import("ProtoGeometry.dll");

c = Cuboid.ByLengths(CoordinateSystem.WCS, 10, 10, 10);

c = c.Translate(30, 0, 0);

c = c.Transform(CoordinateSystem.WCS,
    CoordinateSystem.BySphericalCoordinates(
    CoordinateSystem.WCS, 20, 45, 45));
```

While this script is both valid and fairly succinct, the repeated used of the temporary variable **c** is repetitive without describing anything more about the objects in the script. For this reason, DesignScript has an alternate syntax for object construction called the "modifier block." Modifier blocks are indicated by wrapping both the constructor and subsequent commands inside curly braces (**{ }**), terminated by a semicolon (**;**). Assign the resulting variable with an equal sign (**=**) as you would with a normal constructor. Besides removing redundant naming, the use of the modifier block makes the operations performed on the object more legible and succinct. The following script creates the same object but with a Modifier Block:



```
import("ProtoGeometry.dll");

p = {
    Cuboid.ByLengths(CoordinateSystem.WCS, 10, 10, 10);

    Translate(30, 0, 0);

    Transform(CoordinateSystem.WCS,
        CoordinateSystem.BySphericalCoordinates(
        CoordinateSystem.WCS, 20, 45, 45));
};
```

21: 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 DesignScript 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:

- `{1, {2, 3, 4}, 5,`
  `{ {6, 7}, { {8} } }, 9}`

```
j = {};

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

s = Print(j);
```

However, jagged collections can be problematic when writing code, as not every element is accessed with the same number of square brackets. Code which iterates over the length of a collection will sometimes access a single element, and other times access a collection. If code hasn't been written to check what type of object is returned from a collection, the code may fail when it attempts to perform operations not supported on a collection.

- null

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

j_sum = [Imperative]
{
    sum = 0;
    for (i in 0..(Count(j) - 1))
    {
        // access the "i"th element of j,
        // and add it to sum
        sum = sum + j[i];
    }
    return = sum;
}

s = Print(j_sum);
```

The preceding example fails when **i** is equal to **1**, and **j[i]** returns a collection. Trying to add a collection to an integer is not possible, and generates an error. The following example shows how to access all the elements of this jagged collection:

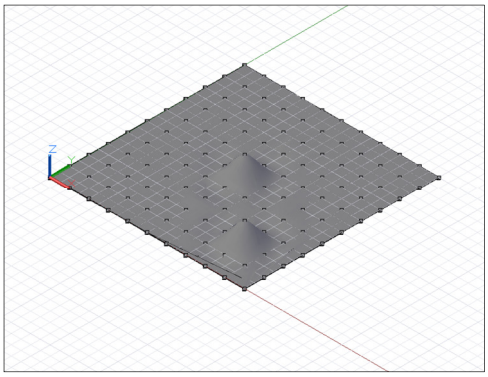
- {1, {2, 3, 4}, 5,
  { {6, 7}, { {8} } }, 9}
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9

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

s = Print(j);

s = Print( j[0] );
s = Print( j[1][0] );
s = Print( j[1][1] );
s = Print( j[1][2] );
s = Print( j[2] );
s = Print( j[3][0][0] );
s = Print( j[3][0][1] );
s = Print( j[3][1][0][0] );
s = Print( j[4] );
```

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

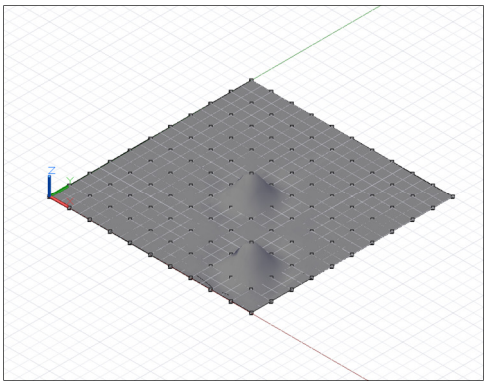
The two-dimensional analog to a BSplineCurve is the BSplineSurface, and like the freeform BSplineCurve, BSplineSurfaces can be constructed with two basic methods: inputting a set of base points and having DesignScript interpret between them, and explicitly specifying the control points of the surface. 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 `BSplineSurface.ByPoints` constructs a surface from these points.



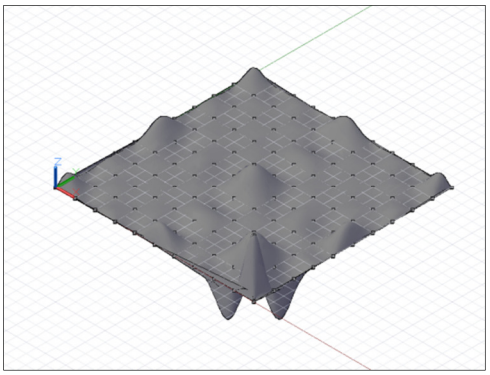
```
import("ProtoGeometry.dll");

// generate a grid of points
pts = Point.ByCoordinates((0..10)<1>, (0..10)<2>, 0);

// create several "bumps"
pts[5][5] = pts[5][5].Translate(0, 0, 1);
pts[8][2] = pts[8][2].Translate(0, 0, 1);

surf = BSplineSurface.ByPoints(pts);
```

Freeform BSplineSurfaces can also be created by specifying underlying control points of a surface. Like BSplineCurves, 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 BSplineSurface by control points, include two additional parameters to `BSplineSurface.ByPoints`, indicating the degrees of the underlying curves in both directions of the surface.



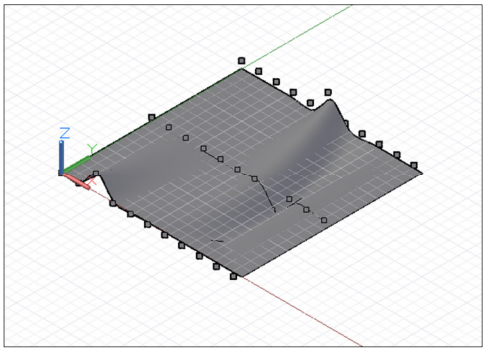
```
import("ProtoGeometry.dll");

pts = Point.ByCoordinates((0..10)<1>, (0..10)<2>, 0);

pts[5][5] = pts[5][5].Translate(0, 0, 1);
pts[8][2] = pts[8][2].Translate(0, 0, 1);

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

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



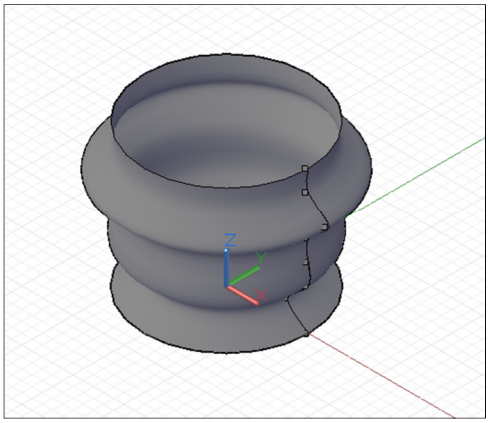
```
import("ProtoGeometry.dll");

pts = Point.ByCoordinates((0..10)<1>, (0..10)<2>, 0);

pts[5][5] = pts[5][5].Translate(0, 0, 1);
pts[8][2] = pts[8][2].Translate(0, 0, 1);

// create a surface of degree 6
surf = BSplineSurface.ByPoints(pts, 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.



```
import("ProtoGeometry.dll");

p1 = Point.ByCoordinates(0..10, 0, 0);
p1[2] = p1[2].Translate(0, 0, 1);
c1 = BSplineCurve.ByPoints(p1);

p2 = Point.ByCoordinates(0..10, 5, 0);
p2[7] = p2[7].Translate(0, 0, -1);
c2 = BSplineCurve.ByPoints(p2);

p3 = Point.ByCoordinates(0..10, 10, 0);
p3[5] = p3[5].Translate(0, 0, 1);
c3 = BSplineCurve.ByPoints(p3);

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.



```
import("ProtoGeometry.dll");

pts = Point.ByCoordinates(4, 0, 0..7);
pts[1] = pts[1].Translate(-1, 0, 0);
pts[5] = pts[5].Translate(1, 0, 0);

crv = BSplineCurve.ByPoints(pts);

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

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

23: 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.

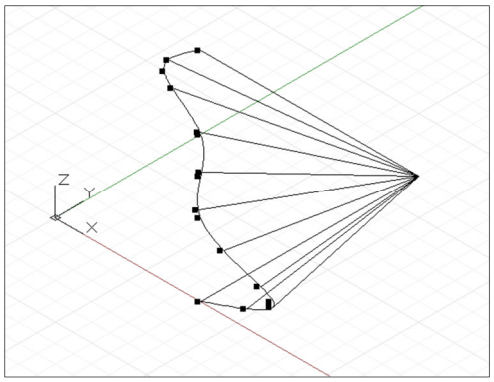
All of the points on a curve can be thought of as having a unique parameter ranging from 0 to 1. If we were to create a BSplineCurve based off of several control or interpreted points, the first point would have the parameter 0, and the last point would have the parameter 1. It's impossible to know in 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 BSplineCurve and BSplineSurface constructors.

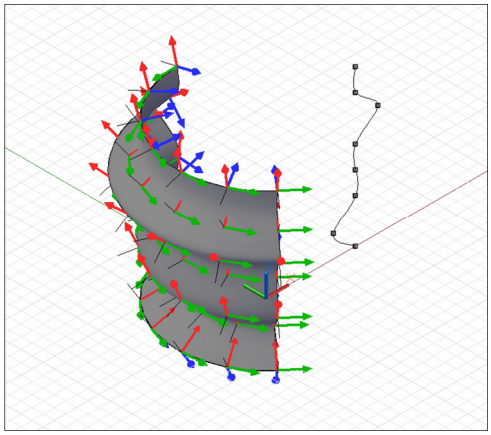
Curves have a method `PointAtParameter`, which takes a single double argument between 0 and 1, and returns the Point object at that parameter. For instance, this script finds the Points at parameters 0, .1, .2, .3, .4, .5, .6, .7, .8, .9, and 1:




```
import("ProtoGeometry.dll");

pts = Point.ByCoordinates(4, 0, 0..6);
pts[1] = pts[1].Translate(2, 0 ,0);
pts[5] = pts[5].Translate(-1, 0 ,0);

crv = BSplineCurve.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 `CoordinateSystemAtParameters` 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:



```
import("ProtoGeometry.dll");

pts = Point.ByCoordinates(4, 0, 0..7);
pts[1] = pts[1].Translate(-1, 0, 0);
pts[5] = pts[5].Translate(1, 0, 0);

crv = BSplineCurve.ByPoints(pts);

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

surf = Surface.Revolve(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.

`PointAtDistance` and `CoordinateSystemAtDistanceAlongCurve` respectively return a Point and CoordinateSystem at a specific length along a Curve. Computational designs often times require splitting a curve into an equal number of curve segments. Rather than manually measuring the length of a curve and creating points at specific distances, DesignScript offers several methods to assist with this process.
`PointsAtEqualArcLength` and `CoordinateSystemsAtEqualArcLength` return a set of Points and CoordinateSystems equally distributed along a curve. For instance, if these methods were called with the input of 3, they would return an object at the start of the curve, end of the curve, and at the curve's midpoint (as measured by distance).

24: Intersection, Trim, and Select 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 DesignScript, 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. Note both that there are fundamental limitations between intersections (such as the inability to intersect Curves and Solids), as well limitations on intersections based on how an object was constructed or its topology.

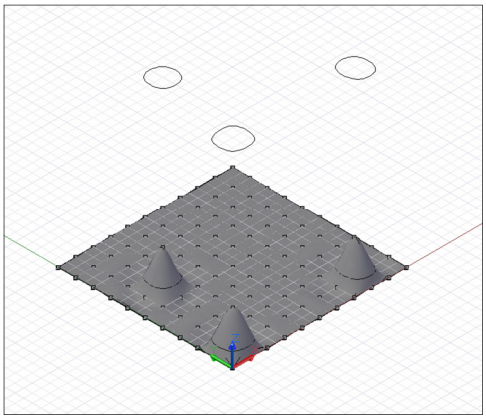
	Intersect:			
With:	**Surface**	**Curve**	**Plane**	**Solid**
Surface	Curve	Point (1,2,3)	Point, Curve (3)	Surface
Curve	Point (1,2,3)	Point	Point	Curve
Plane	Curve	Point (3)	Curve	Curve
Solid	Surface	Curve	Curve	Solid

1. *BSplineSurface.Intersect: Intersection of a closed surface with curves may fail, if the closed surface was created from curves using the ByPoints methods, but should succeed if the surface is created from curves created using the ByControlVertices method.*

2. Curve.Intersect: If either the input curve is created with the BSplineCurve.ByPoints method or the surface to intersect is closed then the intersections will fail. However if the input curve is a periodic closed curve create with the BSplineCurve.ByControlVertices method, then the intersections will succeed.

3. Curve.Intersect: Where the other intersection input is a Plane or a Surface and the curve to be intersected is only partially on the Surface or Plane, the current limitation is it only returns one point

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

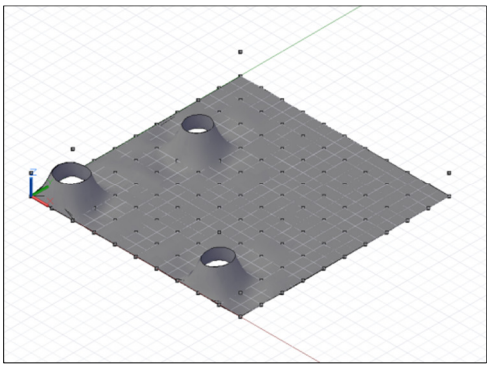


```
import("ProtoGeometry.dll");

WCS = CoordinateSystem.Identity();

p = Point.ByCoordinates((0..10)<1>, (0..10)<2>, 0);
p[1][1] = p[1][1].Translate(0, 0, 2);
p[8][1] = p[8][1].Translate(0, 0, 2);
p[2][6] = p[2][6].Translate(0, 0, 2);

surf = BSplineSurface.ByPoints(p, 3, 3);

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`**

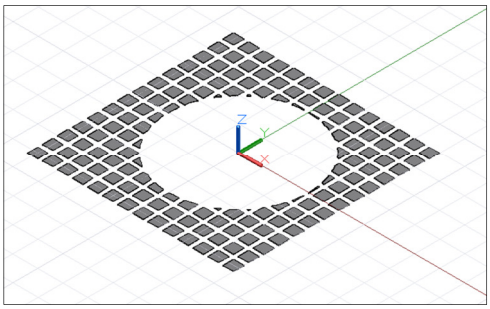
	Trim Using:				
On:	**Point**	**Curve**	**Plane**	**Surface**	**Solid**
Curve	Yes	No	No	No	No
Polygon	NA	No	Yes	No	No
Surface	NA	Yes	Yes	Yes	Yes
Solid	NA	NA	Yes	Yes	Yes

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



```
import("ProtoGeometry.dll");

p = Point.ByCoordinates((0..10)<1>, (0..10)<2>, 0);
p[1][1] = p[1][1].Translate(0, 0, 2);
p[8][1] = p[8][1].Translate(0, 0, 2);
p[2][6] = p[2][6].Translate(0, 0, 2);

surf = BSplineSurface.ByPoints(p, 3, 3);

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

tool = BSplineSurface.ByPoints(tool_pts);

pick_point = Point.ByCoordinates(5, 5, 0);

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

tool.Visible = false;
surf.Visible = false;
```

`SelectTrim` is a method unique to DesignScrit which combines the act of trimming with the act of deleting whole pieces of geometry which are contained inside of a of a selection tool. For instance, in the following example, some of the base surfaces fall completely inside of the cutting tool, without intersecting the tool boundary. If we were to use a simple Trim command, these interior objects would remain untouched. By using `SelectTrim`, we get a proper void in a field of objects:

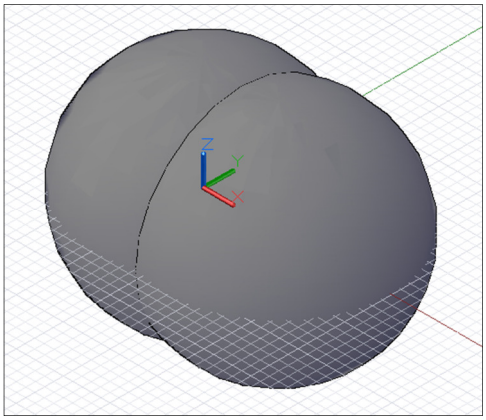


```
import("ProtoGeometry.dll");

def makeSurf(p)
{
    p0 = p.Translate(-0.5, -0.5, 0);
    p1 = p.Translate(0.5, -0.5, 0);
    p2 = p.Translate(0.5, 0.5, 0);
    p3 = p.Translate(-0.5, 0.5, 0);

    pts = { {p0, p1}, {p3, p2} };

    return = BSplineSurface.ByPoints(pts);
}

surf = makeSurf(Point.ByCoordinates((-10..10..1.5)<1>,
    (-10..10..1.5)<2>, 0));

tool = Cylinder.ByRadiusHeight(
    CoordinateSystem.Identity(), 7.0, 3);

trimmed_surfs = Surface.SelectTrim(surf, tool, false);

surf.Visible = false;
tool.Visible = false;
```

25: Geometric Booleans

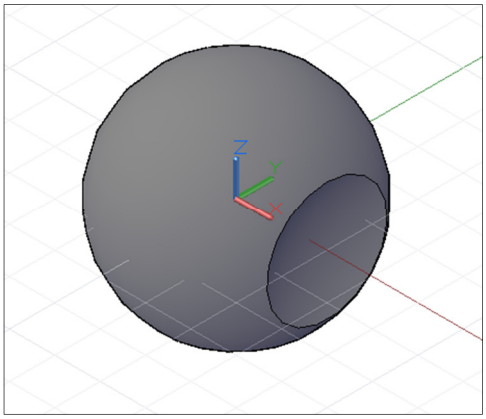
`Intersect`, `Trim`, and `SelectTrim` are primarily used on lower-dimensional geometry such as Points, Curves, and Surfaces. Solid geometry on the other hand, has an additional set of methods for modifying form after their construction, both by subtracting material in a manner similar to `Trim` and combining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



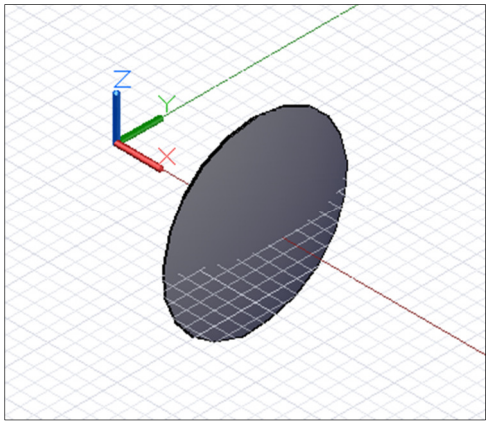
```
import("ProtoGeometry.dll");

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

s2 = Sphere.ByCenterPointRadius(
    CoordinateSystem.WCS.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:



```
import("ProtoGeometry.dll");

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

tool = Sphere.ByCenterPointRadius(
    CoordinateSystem.WCS.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:

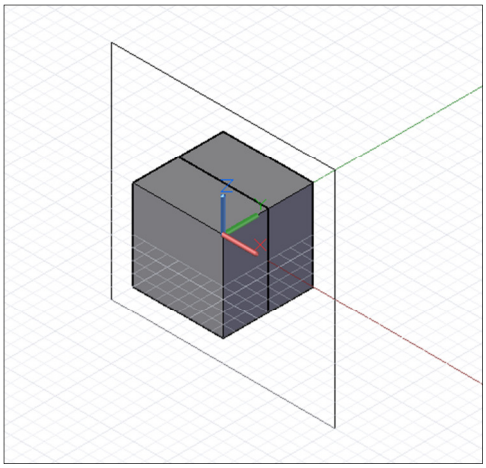



```
import("ProtoGeometry.dll");

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

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

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

26: Non Manifold Geometry

The basic `Union`, `Difference`, and `Intersect` operations on Solid geometry return a homogeneous mass with a simple outer boundary. If we think of the `Union` method, there is another possibility for the resulting Solid: one of the overlapping solid masses is discarded, but the interior surfaces remain. In DesignScript, Solid geometry which has interior partitions, dividing the Solid into interior cells is called non-manifold.

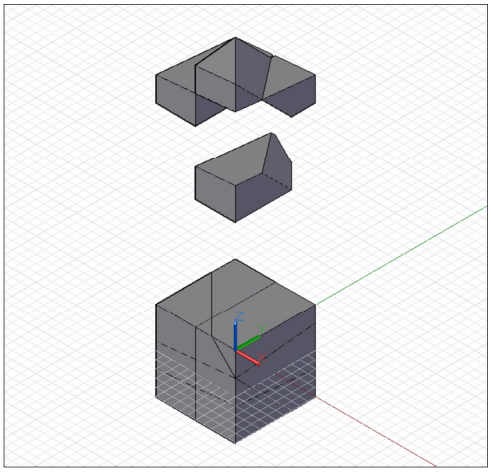
We can create a simple non-manifold Solid by slicing a Cuboid with a plane, generating an interior partition at the plane of intersection, and two Cells on either side.



```
import("ProtoGeometry.dll");

c = Cuboid.ByLengths(CoordinateSystem.WCS, 4, 4, 4);

p = Plane.ByOriginNormal(CoordinateSystem.WCS.Origin,
    CoordinateSystem.WCS.YAxis);

// the false flag tells DesignScript to create non-
// manifold geometry, instead of a regular Solid.
non_manifold_solid = c.Slice(p, false);
```

Unlike a regular piece of solid geometry, a non-manifold solid has an attached data structure representing the underlying topology of the object. The fundamental objects contained inside a non-manifold solid are Cell, Face, Edge, and Vertex. Each object is linked to both its parent and children objects, as well as the neighboring objects in the geometry's structure. For instance, every non-manifold solid contains a list of Cells. Each Cell contains a list of Faces, as well as a list of adjacent Cells. Each Face contains both the Cells an either side of the Face, as well the perimeter Edges. Each Edge contains two Vertices, as well as every Face which uses this Edge.

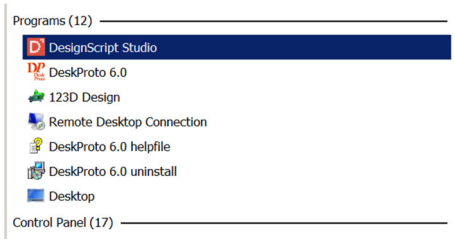
This data structure, in combination with a programming language, allows scripts to walk through the entire contents of a solid, and generate meaningful designs based on relative location and topological adjacencies between objects. For instance, in the following example, the script is able to extract adjacent Cells to a given Cells despite the complex and multi-faceted partitioning of the interior Cuboid.



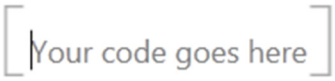
```
import("ProtoGeometry.dll");

WCS = CoordinateSystem.WCS;
c = Cuboid.ByLengths(WCS, 5, 5, 5);

nms = c.Slice(Plane.ByOriginNormal(WCS.Origin,
    WCS.XAxis), false);

nms2 = nms.Slice(
    Plane.ByOriginNormal(WCS.Origin.Translate(
        1, 0, 0), Vector.ByCoordinates(1, 1, 1)),
        false);

nms3 = nms2.Slice(Plane.ByOriginNormal(
    WCS.Origin.Translate(0, 0, 1),
    Vector.ByCoordinates(-0.1, -0.1, 1)), false);

nms4 = nms3.Slice(Plane.ByOriginNormal(
    WCS.Origin.Translate(0, 0, -1),
    Vector.ByCoordinates(-0.1, 0.1, 1)), false);

central_cell = nms4.Cells[6].SolidGeometry.Translate(
    0, 0, 12);

cells =
    nms4.Cells[6].AdjacentCells.SolidGeometry.Translate(
        0, 0, 16);
```

DSS-1: DesignScript Studio Introduction

DesignScript Studio (DSS) is an Alpha version of DesignScript giving a glimpse into Autodesk's newest visual programming tools. However, as Alpha software, be aware that not all language and geometry features may work.



DSS is a stand-alone application; it doesn't need to run inside AutoCAD. Launch it as you would any other application, typically by searching for "DesignScript Studio" in the Start menu.

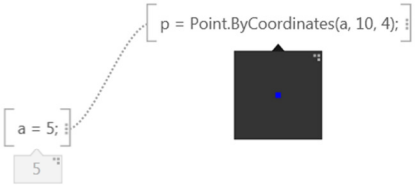


The main IDE of DSS similar to the AutoCAD IDE, though scripts are composed on a 2D canvas. To begin writing a script, double click in the white space. A text cursor will appear.

Let's create a variable **a** storing the value **5**. Type:

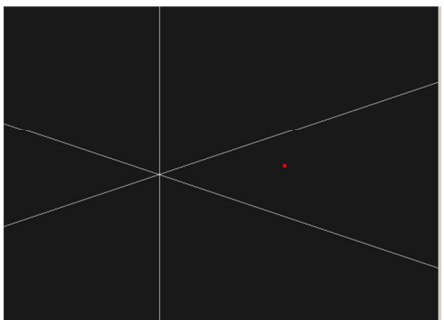
```
a = 5;
```

into the text box, and click into the canvas white space. Make sure you click out of the box into the white space of the canvas; simply pressing return will create a new line. After clicking out of the text box, DesignScript parses your script and generates a preview, in this case "5."



Now that we've created a variable named **a** we can use it anywhere in the DSS canvas. Let's create a point using **a**. Double click into the canvas white space to bring up another text box, and type:

```
p = Point.ByCoordinates(a, 10, 4);
```



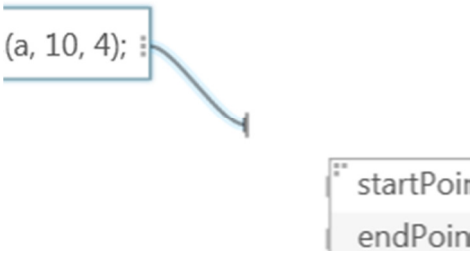
Notice that DSS has picked up the fact that **a** is used to construct **p**, and has indicated that with a dotted line connecting both pieces of code. The black box beneath the point text gives a visual preview of the newly created point, and a window will pop up offering a model space view of the point.

We can also create geometry graphically using a node, without having to type any DesignScript commands. Find the ByStartPointEndPoint node under Line, under ProtoGeometry.dll in the Library box, and double click on it. A **`Line`** node will be placed in the DSS canvas.

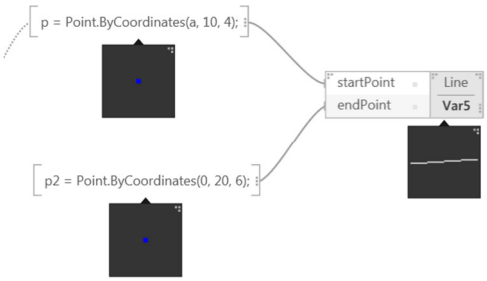
`Line` nodes require two points, so create another **`Point`** by double clicking into the canvas and typing the following command:

```
p2 = Point.ByCoordinates(0, 20, 6);
```

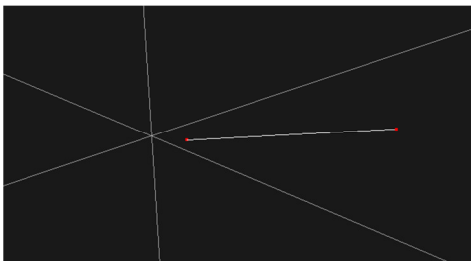
And click out into the canvas.



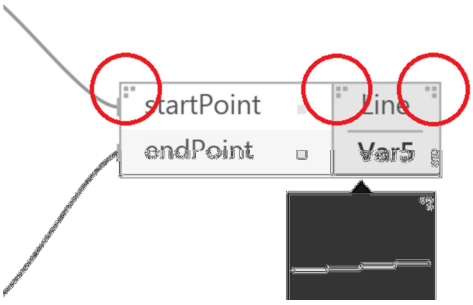
One of the more powerful features of DesignScript Studio is the ability to connect objects created in code with objects created as nodes. To connect nodes visually, hover your mouse over the three dots on the right side of a code block or node, click, and drag. You will see a wire protruding from the node. If you drag the wire to the left hand side of another node, a connection is formed, linking the output of a line of code or a node to the input parameters of a function or node.



If we connect **`p`** and **`p2`** to the Line node, a **`Line`** is created, and appears in model space.

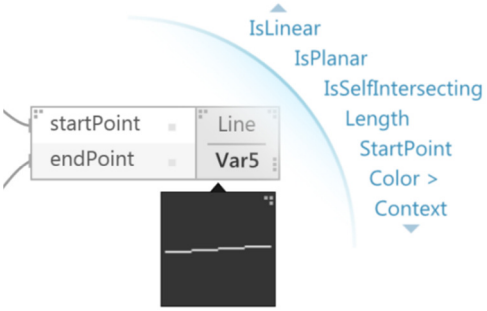


DSS-2: Contextual Menu

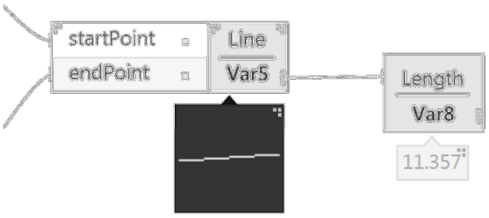


DesignScript Studio has a contextual menu allowing you to add functions and methods to your visual scripts, similar to autocomplete in the DesignScript IDE.

Availability of the contextual menu is indicated by three dots in the corner of a node.

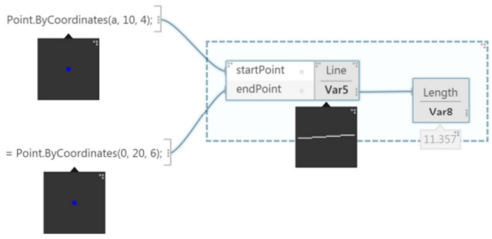


Hovering over these dots brings up a contextual menu with a list of commands. Similar to autocomplete, selecting a command in this contextual menu will add a node to the canvas of the same name.

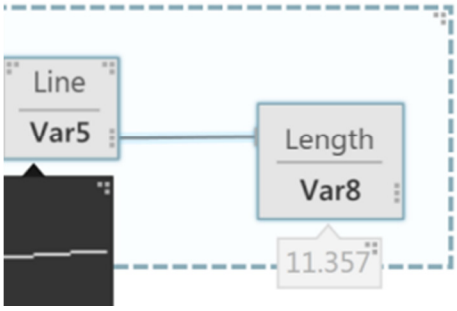


Using the graph from the previous example, we can select Length from the contextual menu to add a Length node. Note that you may have to use the scroll wheel on your mouse to find the Length method. After Length is selected, a node is added, and it is automatically connected to the Line node originating the command. It's value, 11.357, is displayed under the node.

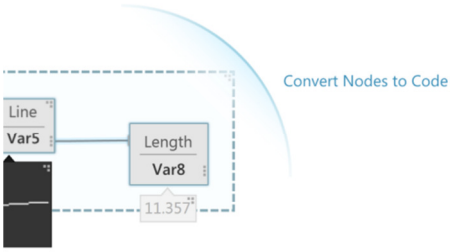
DSS-3: Node to Code



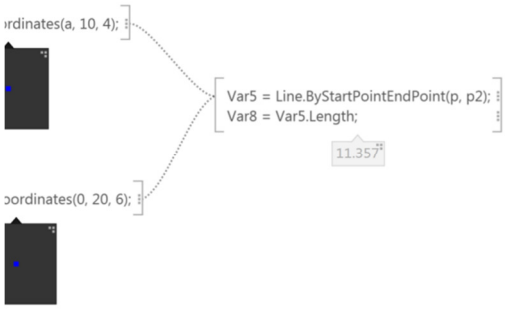
As your node-based scripts grow in size, you may find that it is hard to understand, edit, and develop in a graphical environment. DesignScript Studio allows you to convert all or segments of your visual graphs to traditional text-based code.



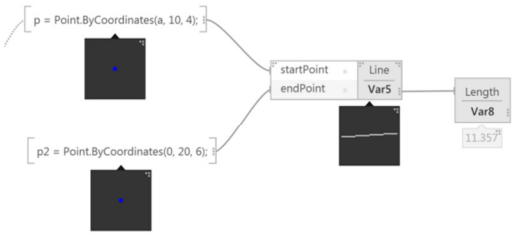
Taking our existing Line graph as an example, if we select the Line and Length nodes, a bounding box appears, with a three-dot contextual menu marker in the upper right hand corner.



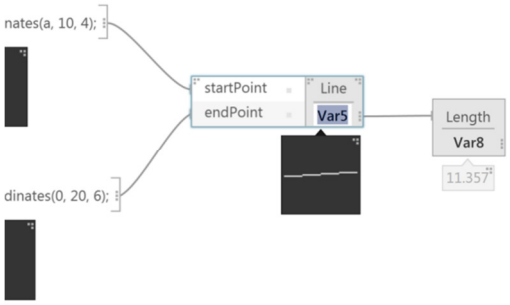
Hovering over this contextual menu marker will bring up the "Convert Nodes to Code" command.



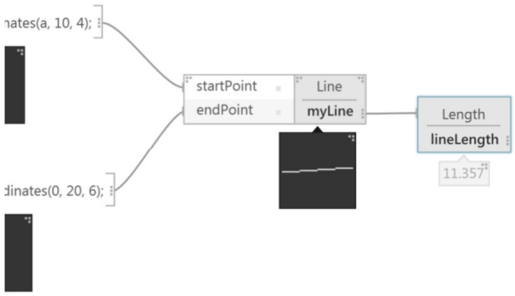
Selecting the command will convert the selected nodes to DesignScript code. Here we can see that DSS has assigned variable names based on the variable names in the Line and Length nodes, `Var5` and `Var8` respectively. Note that your names may be slightly different.



DSS allows you to label your nodes before converting them to code to assist the transition. Press Control-Z or select Edit → Undo from the DSS menu to undo your Node to Code operation, restoring your graph back to the old point.



Node variable names are changed by double clicking on the label under the node name, in this case `Var5` (or similar). The variable label will change allowing you to type in a new name. We can change the name of `Var5` to a more descriptive `myLine`. Similarly, we can change to `Var8` a more descriptive `lineLength`.



Now, when we select the nodes ad convert them to code, the generated code has more descriptive names.

DSS-4: DesignScript Studio Limitations

DesignScript Studio currently doesn't support the following DesignScirpt language features. As always, check designscript.org for the latest information on the state of the DesignScript language.

Not Supported:

- Classes
- Functions
- Language blocks: [Imperative] and [Associative]
- Modifier blocks
- Inline conditionals