

This manual is designed for two types of readers. First, those who want to make an initial foray into text based programming and who may be currently using a visual programming environment such as Dynamo. Second, those users who have some scripting or programming experience and want harness a visual programming environment in order to access the related geometry and application libraries. DesignScript therefore acts as a two-way bridge between visual programming and text based programming.

Visual data flow programming uses incredibly simple rules for connecting nodes together. All nodes follow this simple graph-node convention allowing the user to easily discover more functionality even from unfamiliar nodes. As a result visual programming has an intuitive, easy access quality and allows the user with the minimum of training to be immediately productive and to explore new functionality. However this initial 'ease of use' advantage of visual programming can also become a disadvantage. As more nodes are added to the graph and as more of the arcs between the nodes cross the whole visual effect becomes very complex and eventually difficult to read. Essentially, the moment has come for the user to think about a more succinct way to represent his program logic. The next step into text based programming necessarily requires the acquisition of a number of new concepts, precisely defined terminology and unfamiliar syntax. There is more to learn but much more to be gained from text based programming than is possible with visual programming. Text based programming languages are more expressive. They use many more concepts and allow these to be combined in many more ways. This manual is based on the idea that the path to being a productive, expressive and accomplished programmer is through investing in this boarder understanding of the concepts, terminology and rules of programming. By learning DesignScript users will acquire familiarity with the well-established programming conventions that are transferrable to other programming languages. In addition DesignScript builds on these established conventions to offer additional and unique ways of programming not supported by other languages.

We can start by building on the experience of visual programming. While visual programming appears to be completely intuitive, only requiring the connecting together of nodes, it actually introduces some important programming concepts:

Dependency and automatic 'change propagation'

The concept of 'type'

Replication

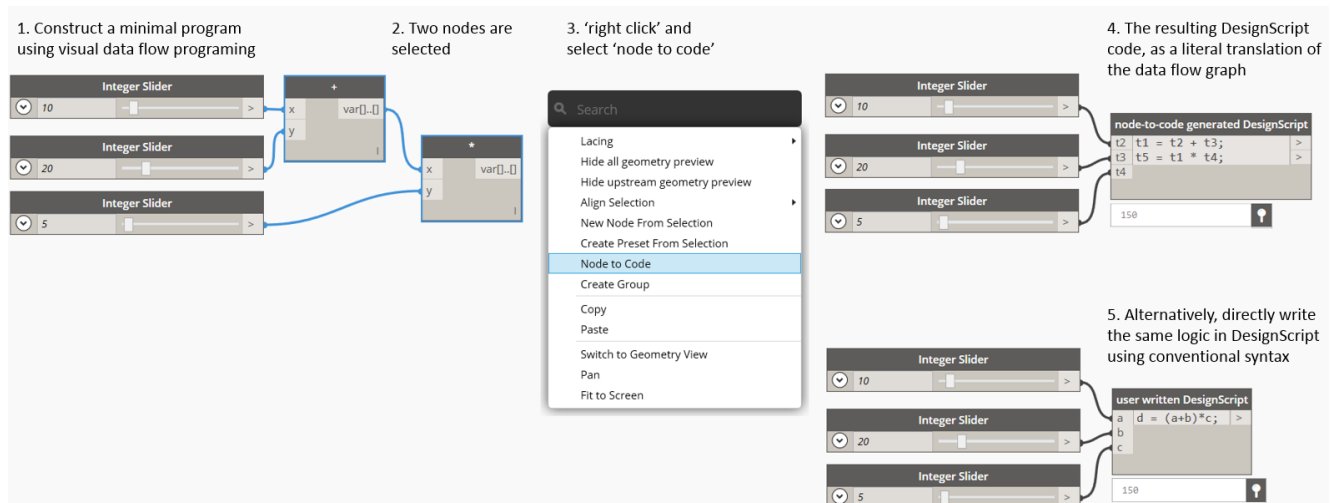
Zero based indexing

The concept of an executable programming statement, with dependent and independent variables

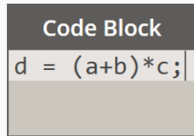
[the Code Block node essentially unifies visual data flow programming with text based programming using executable programming statements

[Section TBC]

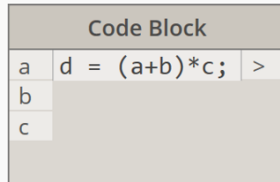
While this manual focusses on the functionality and syntax of the DesignScript language, there is still some minimal 'mechanics of use' which have to be presented in order to write and execute a DesignScript program. Most users will access DesignScript through the Dynamo user interface either by creating a visual program [1]. Dynamo uses the DesignScript language, compiler and execution processor as its computational engine. When using the Dynamo visual data flow programming environment, the user may not be aware that each node is in fact a DesignScript statement. The 'Node to Code' functionality [2] can be used to create the equivalent DesignScript program in a 'code block' node [3]. Effectively the 'node to code' simply exposes the underlying DesignScript code previously hidden within the Dynamo node and provides a convenient and accessible path for the novice programmer between visual and text based programming [4]. However, we can immediately observe that the resulting generated DesignScript code is no more than a one-to-one translation of the original visual program. Each line of DesignScript corresponds to a node in the graph. 'Node to Code' is very useful to create an initial program, but it is often advisable to refine this by editing the generated DesignScript code, for example by choosing more informative names for the variables or by combining statements so as to remove the need for intermediate variables [5]. This ability to write multiple statements to be written within one node, further reducing the visual complexity of the graph. Rather than creating a visual program and converting it to DesignScript via the node-to-code functionality, DesignScript code can be directly written into an empty code block node.



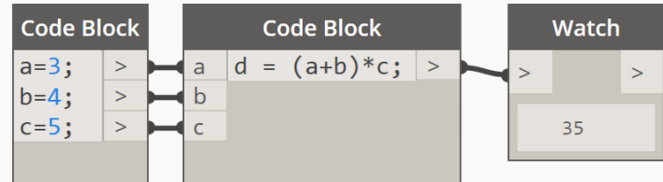
1. Double click to create an empty code block node and type the program statement



2. Clicking outside the code block node and input and output 'ports' are automatically generated

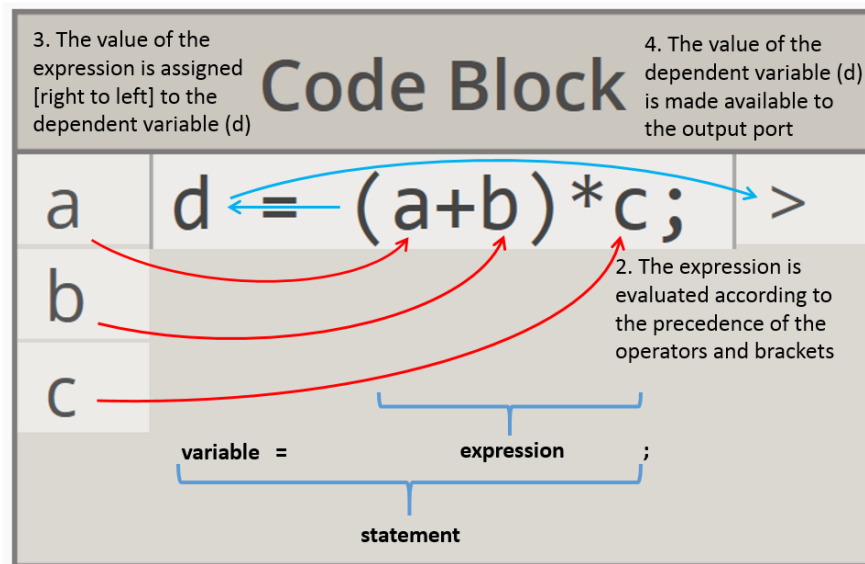


3. Connect up the input and output ports to provide sufficient data to run the program



There are a number of advantages of the Code Block node. As the user writes his DesignScript program any variable used in the expression on the right hand side of the '=' sign has an 'input port' automatically created for it on the left hand side of the code block node and any variable which exists on the left hand side of the '=' sign (whose value is being calculated) has an 'output port' automatically created for it on the right hand side of the Code Block node. This enables the user to connect a code block node to the necessary input and output blocks in the rest of his visual program.

1. The independent variables (a, b, c) in the expression are assigned their respective values from the 'input' ports



By convention visual data flow programs are based on a left-to-right flow, with inputs on the left and outputs on the right. By convention program statements are based on right to left flow, with the evaluation of an expression on the right, whose value is assigned right-to-left [represented by the '='] to a dependent variable on the left. Fitting a conventional [right to left] program statement into a data flow [left to right] visual convention is somewhat convoluted.

The user can edit the DesignScript code, and when Dynamo is running in 'automatic' mode, simply clicking outside the node (for example on the 'canvas') automatically executes DesignScript. All the examples in this manual can be copied into a Dynamo Code Block node and run.

It is important to note that the DesignScript language is independent of Dynamo. For example, Dynamo also implements a number of useful geometric and domain specific functions which can be programmed using DesignScript. We will use examples of these geometric classes and methods to illustrate the features of the DesignScript language. However, it should be noted that these classes and methods are not part of the DesignScript language. Other libraries, including geometry libraries can be built using C# development tools operating within the 'Zero Touch' conventions, and the resulting DLL's can be loaded into Dynamo using the 'file/import' command and accessed via the Dynamo UI and DesignScript.

There are some aspects of DesignScript which are currently not supported through the Dynamo User Interface, but these can be accessed by using a regular text editor to create a DesignScript source file [with the .ds extension]. These files can also be loaded using the Dynamo 'file/import' command.

DesignScript support two styles of programming: Imperative and Data Flow [sometimes referred as Associative programming] and has functionality which is common to both styles.

Data Flow programming is essentially a text based version of a visual programming using a graph or nodes and arcs (as in Dynamo). Data flow programming whether visual or text based uses the concept of graph dependencies to establish 'flow control' and is useful for modeling complex operations (such as geometric processes) applied to collection of objects, in a very succinct programming style with automatic change propagation. Data Flow programming is particularly relevant to users who are currently using visual programming and who are new the text based programming.

Imperative programming is characterized by explicit 'flow control' using for loops (for iteration) and if statements (for conditionals) as found in familiar scripting and programming languages such as Python, C# or Java. Imperative programming is useful to perform iteration, either stepping through a collection or to perform some iterative feedback or optimisation loop. Imperative programming can also be used for complex conditional logic. Imperative programming is particularly relevant to users who have experience with existing scripting and programming languages.

The two styles of programming address different computational tasks and essentially complement each other. The different styles of programming share a common notation which means that in some case the same code can be executed either as data flow or imperatively. In addition there are certain computational tasks that benefit from a combination of programming styles within the same program or indeed within the same function or method

- Aspects of DesignScript that are common to both Associative and Imperative programming:
 - **Types:** In DesignScript the 'explicit typing' of variables during declaration is optional. However there are certain operations where a particular type of variable is expected. Also different functions and methods may expect arguments of a defined type. So while variables do not have to be explicitly typed, the type they have at runtime may affect the behaviour of the program.

At the start of writing a program it may be a helpful strategy to use untyped variables, particularly when the programmer is in an exploratory mode and when flexibility is required to explore the option for variables to have different types of values. Predefining the type of a variable reduces this flexibility and may inhibit exploration. However as the program matures, the programmer may want to be more explicit about the type of different variables and to use explicit typing as part of the error checking process in his program.

- **Built-in types:** DesignScript supports the following built-in types: `int`, `double`, `bool`, `string`

```
a : int    = 10;    // whole numbers, typically used for counters, indices, etc
b : double = 5.1;   // with a decimal floating point
c : bool   = true;  // or false
d : string = "hello world";
e = null;          // undefined
```

- **Optional use of 'type':** DesignScript support both typed and un-typed variables

The type of a variable can be defined using the syntax

```
variableName : type;
variableName : type = expression; // initialize a variable with a value
```

In the absence of the type of a variable being explicitly defined, the type of a variable takes the type of the value assigned to it

```
a = 10; // variable 'a' has no predefined type,
        // but assumes the 'int' type by having an int (the value 10) assigned to it
```

- **Default type:** DesignScript has a default type called `var`.

```
a : var; // the variable 'a' is declared, but of an undefined type
```

- **Declaring variables:** DesignScript gives complete freedom to the programmer, as follows:

```

a;           // the variable is untyped and can have any value assigned to it
b = null;    // the variable is untyped and can have any value assigned to it
c : var;     // the variable is untyped and can have any value assigned to it
d : int;     // the variable is declared as a single int
e : int[];   // the variable is declared as a 1D collection of int's: all member must be int's
f : int[][]; // the variable is declared as a 2D collection of int's: all member must be int's
g : int[]..[]; // the variable can be a single int or a collection of int's of any dimension
h = { };     // the variable is an empty collection of any type and can also be a single value

```

[Note: the use of the 'int' type (above) is purely illustrative. Any type can be used, as appropriate.]

In summary:

- A variable can be untyped and therefore can be a single value of any type or a collection of any dimension of any type.
- Or a variable can be declared as a single value or as a collection of a specified dimension, or as a collection of any dimension; and the variable can be typed or untyped.

If it is important to declare the type of the variable and it is anticipated that a variable may be a single value or a collection of values, then the declaration `variable : type[]..[];` should be used.

If it is important to declare the type of the variable and it is anticipated that a variable could be of more than one type, then the type declared should be the most specialised common super type of the anticipated values for this variable. For example, if a variable could be a `Line` or an `Arc`, then it should be declared as `variable : Curve;` [as the common super type]

Similarly, if it is important to declare the type of a collection variable and, if it is anticipated that the variable will be a heterogeneous collection, then the type of the collection should be the most specialised common super type of the anticipated members. For example, if a variable collection could contain members that are `Line` or `Arc`, then the collection should be declared as `variable : Curve[];` [as the common super type] or `variable : Curve[]..[];` if it is anticipated that the dimension of the collection may change.

Therefore (more generally) the users should check the class hierarchy of the libraries he is using to ensure that he has selected the appropriate common super type.

- **Collections:** can be defined with the `{}` notation and evaluated by the `Count()` and `Rank()` functions

```

a = {1, 2, 3, 4};           // a 1D collection defined using the {} notation, with a list of values
b = Rank(a);                // b = 1
c = Count(a);               // c = 4
d = { { 1, 2 }, { 3, 4 } };
e = Rank(d);                // d = 4
f = Count(d);               // f = 4

```

- **Range Expression:** range expression can be used to generate numeric collections, in the form `start..end..inc`

```

a = 1..5; // a = {1,2,3,4,5} start..end [using 1 as the default increment] Dynamo 'Range' node
b = 1..9..2; // b = {1,3,5,7,9} start..end..increment Dynamo 'Range' node
c = 1..9..~3; // c = {1.0, 3.666667, 6.333333, 9.0} start..end..~approximateIncrement
d = 1..9..#3; // d = {1, 5, 9} start..end..#noOfCases
e = 1..#3..9; // e = {1, 10, 19} start..#noOfCases..increment Dynamo 'Sequence' node

```

- **Indexing:** the members of a collection can be accessed using indices and the `[]` notation

```

a = 1..9;           // a = {1,2,3,...7, 8, 9}.. using the default increment of '1'
b = a[0];           // b = 1 ... in DesignScript as with other languages, indexing starts from zero
c = a[-1];          // c = 9 ... negative indexing counts back from the end of the collection
d = a[1..5..2];     // d = {2, 4, 6} ... a collection of int's can be used to select a sub collection,
//                  // via a range expression
e = a[{ 2, 3, 0 }]; // d = {3,4,1} ... a collection of int's can be used to select a sub collection,
//                  // via some arbitrary collection
f = Count(a);       // f = 9 ... using the Count() function

```

Note: the following code will fail...

```
g = a[Count(a)]; // because indexing starts at zero, the last member of a collection is at [Count(a)-1]..
```

This provides the motivation for negative indexing where:

`a[Count(a)-1]` is equivalent to `a[-1]` (the count of `a` is applied automatically)

- **Rectangular Collection:** is a collection where all sub collections are the same dimension and length

```
a = { { 1, 2 }, { 3, 4 } }; // a rectangular 2D collection
b = Count(a[0]);          // b = 2... both the a[0] and a[1] subcollections and have the same length
c = Count(a[1]);          // c = 2
```

- **Ragged Collection:** is a collection where sub collections may have different dimensions and lengths

```
a = { { 1, 2 }, { 3, 4, 5 }, 6 }; // a ragged 2D collection
b = Count(a[0]);                // b = 2... a[0], a[1] and a[2] are subcollections of different lengths
c = Count(a[1]);                // c = 3
d = Count(a[2]);                // d = 1
```

- **Flexible collection building:** DesignScript supports flexible and direct ways to build collections

```
h = {};
h[2] = 5; // h = { null, null, 5 } here, a variable can be defied as a member of a collection,
          // provided that collection has been previous defined. The values of the
          // members of the collection prior to member which is explicitly defined will be null

h[1] = {4,5,6}; // h = {null {4, 5, 6}, 5} here, a member of collection that was originally a single value
               // (or a sub collection of a particular dimension) can be directly replaced by a different
               // single value or by a collection of a different dimension
```

In addition there are a number of functions available to build and manipulate collections, including to Insert, Remove and test for the presence of members in a collection [see the 'Language Function' tab in the DesignScript class library documentation]

TBC documentation of the following functions: IsHomogeneous, IsRectangular, InUniformDepth, NormalizeDepth, Flatten, Insert, Contains, IndexOf, Equals,

- **User defined types:** these are types defined as classes within DesignScript or via imported DLL's . DesignScript is an object-oriented language and uses the standard terminology of class, subclass, superclass, constructor, method, arguments and instance, as in:

```
instance = Class.Constructor(arg1, arg2,... argN);
```

A variable is an **instance** of a **class** using a particular named **constructor** or **method** with various input **arguments**.

- **Optional typing and the ability of a variable to change type:** We have seen that variables need not be explicitly typed and that they can take on the type of the value assigned to them. It is therefore possible for a variable to change its type, and this may affect the validity of how it is referenced in subsequent language statements, as illustrated by this example:

```
WCS = CoordinateSystem.Identity(); // define the world coordinate system;

a = Line.ByStartPointEndPoint(Point.ByCartesianCoordinates(WCS, 5, 5, 0),
                             Point.ByCartesianCoordinates(WCS, 10, 5, 0));
    // create variable 'a' as instance of the Line class

b = a.PointAtParameter(0.5);
    // referencing variable 'lineA' to make the variable 'point1'
    // [point at parameter on the line];

a = Arc.ByThreePoints(Point.ByCartesianCoordinates(WCS, 5, 5, 0),
                    Point.ByCartesianCoordinates(WCS, 5, 7, 0),
                    Point.ByCartesianCoordinates(WCS, 10, 10, 0));
    // variable 'a' is an instance of the Arc class...
    // the method PointAtParameter is still valid because this is
    // defined on the Curve class of which Line and Arc are subclasses
```

In this example, variable 'b' is still valid because the `.PointAtParameter()` method is defined in the *Curve* class and *Line* and *Arc* are both subclasses of *Curve*. These issues, particularly with reference to geometry are discussed in more detail in the subsequent section in this manual on Geometry [pages 20-22].

More generally, it is possible for a variable to be defined without a type or to change its type. However, when it is referenced (either as an instance used with a method or as an argument) then it may be required to be of specific type. Therefore the users should check the class hierarchy of the libraries he is using to ensure that such type changes are valid.

- **Properties:** user defined types (or classes) can have properties which are accessed via the `.` operator [the dot operator] Properties may have a defined type.

```
a = Point.ByCoordinates(5,5,0);
b = a.X; // b = 5 .. access the X coordinate property of the Point variable 'a'
```

Note: most properties of the geometry objects in ProtoGeometry are 'read only' and cannot be assigned to.

A constructor [such as `Point.ByCoordinates(5,5,0);`] is effectively setting a set of related (and sufficient) properties 'in one go'. DesignScript intentionally restricts the change of a single property [such as the X coordinate of a point] in isolation, outside the known context provided by such method.

```
a.X = 10; // not allowed, instead a method should be used to define a.X in the context of other properties
```

- Imperative Programming: accessed via the `[Imperative]` directive applied to a block:

Imperative programming is useful to perform iteration, either stepping through a collection or to perform some iterative feedback or optimisation loop.

Imperative programming uses conventional **for** and **while** loops and **if .. else** statements to explicitly define 'flow control'.

- **Program execution:** In the absence of such flow control the next statement in the program is executed, for example:

```
1 a; b; // define the variables to be output at the top or outer scope
2 [Imperative]
3 {
4   a = 10; // original value of 'a'
5   b = a * 2; // calculating 'b' based on the current value of 'a'.. 'b' = 20
6   a = 15; // changing the value of 'a' will NOT cause the value of 'b' to change
7 }
```

If this code fragment is executed in single step debug, the following sequence of statements will be executed: 4, 5, 6.

- **Iteration:** defined by a **for** loop, as: **for (variable in collection) {...}**
defined by a **while** loop, as: **while(condition_is_true) {...}**

An example of a **for** loop:

```
a = {}; // declare an empty collection at the global scope

[Imperative]
{
  for(i in 0..5) // for loop
  {
    a[i] = i;
  }
}
b = a; // b = {0, 1, 2, 3, 4, 5}
```

An example of a **while** loop:

```
a = {}; // declare an empty collection at the global scope
i = 0; // define the initial value for 'i'

[Imperative]
{
  while(i <= 5) // test if I satisfies the defined condition
  {
    a[i] = i;
    i = i+1; // increment i
  }
}
b = a; // b = {0, 1, 2, 3, 4, 5}
```

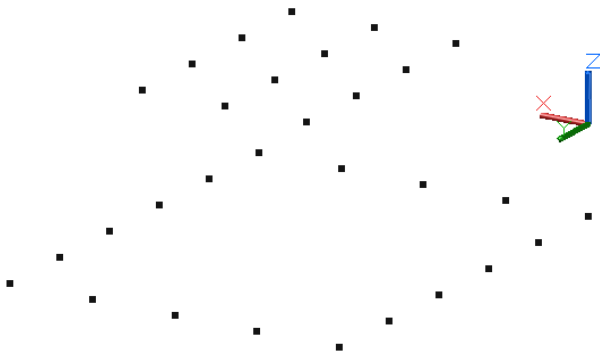
- Conditional: defined by an **if..else** statement as: *if (condition_is_true) single_statement_if_true*
if (condition_is_true) { statements_if_true}
if (condition_is_true) { statements_if_true}
else {statements_if_false}

An example of a **if..else** conditional embedded in a double **for** loop

```
iCount = 4;
jCount = 5;

resultPoints = { }; // define an empty collection

[Imperative]
{
  for(i in 0..iCount)
  {
    for(j in 0..jCount)
    {
      if ((i == 0) || (i == iCount) || (j == 0) || (j == jCount))
      {
        // points are the periphery of the array to move down
        resultPoints[i][j] = Point.ByCoordinates(i, j, -1);
      }
      else
      {
        // points are the periphery of the array to move up
        resultPoints[i][j] = Point.ByCoordinates(i, j, 1);
      }
    }
  }
}
```



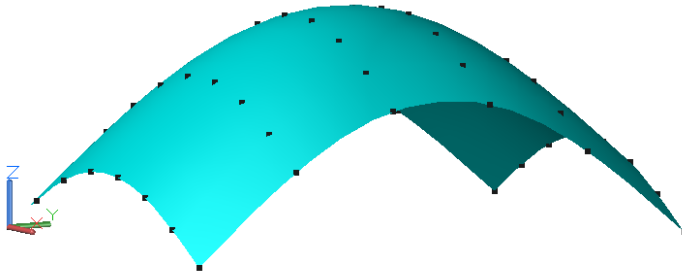
- o A more complex example of iteration:

```
surfacePoints_2D_array : Point[][]; // define a 2D array of Points
surface : NurbsSurface; // define a surface

[Imperative]
{
    xSize    = 10;
    ySize    = 15;
    xHeight  = 2;
    yHeight  = 4;
    numColsX = 8;
    numColsY = 6;

    for(i in 0..numColsX)
    {
        for(j in 0..numColsY)
        {
            surfacePoints_2D_array[i][j] = Point.ByCoordinates( i * (xSize / numColsX), // x coordinates
                                                                j * (ySize / numColsY), // y coordinates
                                                                (Math.Sin(i * (180 / numColsX)) * xHeight)
                                                                + (Math.Sin(j * (180 / numColsY)) * yHeight) ); // z coordinates
        }
    }

    surface = NurbsSurface.ByPoints(surfacePoints_2D_array).SetColor(Color.Cyan); // create a surface
}
```



Compare this script with the same geometry created Associatively on Page 10.

- **Associative Programming:** accessed via the `[Associative]` directive:

Associative programming uses the concept of graph dependencies to establish 'flow control.' Changes to 'upstream' variables are automatically propagated to 'downstream' variables. Associative programming in DesignScript also implements two additional concepts: 'replication' and 'modifiers'.

With replication, anywhere a single value is expected a collection may be used instead and the execution is automatically *replicated* over each element. The combined result of dependencies and replication is that is easy to program complex data flows (including geometric operations) involving collections. An upstream variable may change from being a single value to a collection or from a collection to another collection of different dimensions or size, so the downstream dependent variables will automatically follow suit and also become collections of the appropriate dimension and size. This makes Associative programming incredibly powerful, particularly in the context of generating and controlling design geometry.

With modifiers, each variable can have multiple states, which might reflect the geometric modeling sequence. For example a geometric variable might be created (say as a curve) and then it can be 'modified' by being trimmed, projected, extended, transformed or translated. Without the concept of modifiers each state or modeling operation would require to be a separate variable and this would force the user to have to make up the names of all these intermediate variables. Modifiers avoid imposing this naming process on the user.

Dependencies, replication and modifiers can all be combined to represent the typical modeling operations found in architecture and constructions. Buildings are composed of collections of components. Typically these collections are often the product of a series of standard operations across all members. On the other hand, within such collections there may be special conditions where different or additional modeling operations are required to be applied to a sub collection of members. Modifiers enable these special conditions to be identified and for additional modeling operation applied.

Associative programming involves the following concepts:

- **Assignments and Dependencies**

In associative mode, except where otherwise noted, all DesignScript statement are of the form:
`variable = expression;`... for example:

```
a = 10;    // a is defines as int with the value 10;
b = a * 2; // b is defined by the expression 'a * 2'
```

These statements define relationships between the variable (on the left hand side of the statement) and references to other variables within the expression (on the right hand side of the statement). These relationships define a graph.

[Note: In the following code fragment, we have added line numbers so as to be able to describe the execution order. These line numbers should be removed before executing these fragments in DesignScript]

```
1 a; b;           // define the variables to be output at the top or outer scope
2
3
4 a = 10;         // original value of 'a'
5 b = a * 2;      // define 'b' as dependent on 'a'.. 'b' initially = 10, then = 30
6 a = a + 5;      // changing the value of 'a' will change the value of 'b' (now = 30)
7
```

If this code fragment is executed in single step debug, the following sequence of statements will be executed: 4, 5, 6, 5.

In associative programming, statement 5 is not just executed once (in sequence, after statement 3).

In associative programming, statement 5 establishes a persistent relationship between the variable **b** and variable **a**.

When the value of variable **a** is re-defined in statement 6, the Associative update mechanism in DesignScript will execute all statements that depend on variable **a**, which (in this example) includes statement 5. Hence the execution sequence: 4, 5, 6, 5.

This can be compared to the exact same code fragment executed imperatively, as follows:

```
1 a; b;           // define the variables to be output at the top or outer scope
2 [Imperative]
3 {
4   a = 10;        // original value of 'a'
5   b = a * 2;     // calculating 'b' based on the current value of 'a'.. 'b' = 10
6   a = a + 5;     // changing the value of 'a' will NOT cause the value of 'b' to change
7 }
```

If this code fragment is executed in single step debug, the following sequence of statements will be executed: 4, 5, 6.

In imperative programming, statement 5 is just executed once (in sequence, after statement 3).

In imperative programming, statement 5 does not establish a persistent relationship between the variable 'b' and variable 'a'.

- **Collections and Replication:** a collection can be used where a single value is expected

```

1 a; b; // define the variables to be output at the top or outer scope
2
3
4 a = 10; // original value of 'a'
5 b = a * 2; // define 'b' as dependent on 'a'
6 a = { 5, 10, 15 }; // redefine a as a collection.. the value of b is now = {10, 20, 30}
7

```

In this example, when **a** becomes a collection of values, the expression **a * 2** is executed for every member of **a** and the resulting collection of values are assigned to **b**. In associative programming, the existence of variable as a collection is propagated to all dependent variables, in this example 'b'. This propagation of collections is called 'replication'.

Note: The replication in this example is equivalent to the following Imperative code:

```

a; b; // define the variables to be output at the top or outer scope
[Imperative]
{
    i = 0;
    a = { 5, 10, 15 }; // define a as a collection..

    for(aa in a) // explicit 'for' loop
    {
        b[i] = aa * 2;
        i = i + 1;
    }
} // the value of b is now = {10, 20, 30}

```

We can observe that replication saves considerable program space and programmer attention.

If this code fragment is executed in single step debug, the following sequence of statements will be executed: 4, 5, 6, 5.

Note that variables can be untyped, but if they are declared as typed then DesignScript assumes that the user wants to restrict the values that can be assigned to that variable and will report errors if an erroneous assignment is attempted. For example, if **a** and **b** are defined as single **int**'s, below:

```

a : int; // explicitly defined as a single int
b : int; // explicitly defined as a single int

a = 10;
b = a * 2;
a = { 5, 10, 15 }; // changed into an array of int's ... this will fail

```

- **Zipped replication:**

When there are multiple collections within the same expression, we need to control how these are combined. With 'zipped' replication, when there are multiple collections, the corresponding member of each collection is used for each evaluation of the expression. This works well when all collections are the same dimension and length. If collections are of different lengths, then the shortest collections determines the number of times the expression is evaluated, and hence the size of the resulting collection.

```

a; b; c; // define the variables to be output at the top or outer scope

a = {1, 5, 9};
b = {2, 4, 6};
c = a + b; // zipped replication operation .. c = {3, 9, 15}

```

- **Cartesian replication** controlled by **Replication Guides**

When there are multiple collections, we need to control how these are combined. With 'cartesian' replication, each member of one collection is evaluated with every member of the other collections, so that resulting collection is the 'cartesian product' of the input collections.

The order in which the cartesian product is created is controlled by 'replication guides' in the form **<n>**, which define the sequence of the replication operations. This must be a continuously increasing sequence of **int**'s starting at 1. This sequence is equivalent to the order of the nested **for** loops that would have had to be written in an Imperative script

```

a; b; c; d; // define the variables to be output at the top or outer scope

a = {1, 5, 9};
b = {2, 4 };
c = a<1> + b<2>; // cartesian replication c = { { 3, 5 }, { 7, 9 }, { 11, 13 } }
d = a<2> + b<1>; // changing the sequence of replication guides changes the resulting collection
// d = { { 3, 7, 11 }, { 5, 9, 13 } }

```

Replication guides can be used in any expression with any term which is a collection no matter where that collection appears in the expression, for example as arguments in a function call or as a literal collection or as a range expression.

- Using replication guides with arguments:

```

a; b; c; d; // define the variables to be output at the top or outer scope

def intAdd : int (first : int, second : int) = first + second; // define a function

a = {1, 5, 9};
b = {2, 4 };
c = intAdd(a<1>, b<2>); // cartesian replication c = { { 3, 5 }, { 7, 9 }, { 11, 13 } }
d = intAdd(a<2>, b<1>); // changing the sequence of replication guides changes the
// resulting collection d = { { 3, 7, 11 }, { 5, 9, 13 } }

```

- Using of replication guides with range expressions:

```

c; d; // define the variables to be output at the top or outer scope

c = (1..9..#3)<1> + (2..4..2)<2>; // cartesian replication c = { { 3, 5 }, { 7, 9 }, { 11, 13 } }
d = (1..9..#3)<2> + (2..4..2)<1>; // changing the sequence of replication guides changes the
// resulting collection d = { { 3, 7, 11 }, { 5, 9, 13 } }

```

Note: in the term “((1..9..#3))<1>”, the range expression 1..9..#3 has to be wrapped in brackets so that the replication guide <2> applies to the resulting evaluation of the range expression. If the term was “1..9..#3<1>” then the replication guide <1> would only apply to the value 3.

- Using replication guides with Instance methods:

Replication and replication guides can be used in any expression which uses collections. This includes those expressions where an instance method is being called and the instance is a collection and the other arguments to that method are also collections.

If no replication guides are used, then the result will be a zipped collection (as would be expected)

```

a; b;
a = Point.ByCoordinates(1..5, 1, 0);
b = a.Translate(0, 1..5, 0);

```

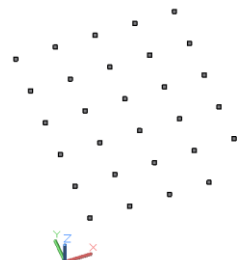


If replication guides are used, then the result will be a cartesian collection

```

a; b;
a = Point.ByCoordinates(1..5, 1, 0);
b = a<1>.Translate(0, (1..5)<2>, 0);

```



Note: the syntax to apply replication guides to an instance variable that is a collection. The replication guide comes immediately after the term to which it is to be applied.

- Combining zipped and cartesian replication

In the following example, we are taking the cartesian product of one 1D array and another 1D array [to create an intermediate 2D array] and then 'zipping' this intermediate 2D array with another 2D array

```
surfacePoints_2D_array : Point[][]; // define a 2D array of Points
surface : NurbsSurface; // define surface

xSize = 10;
ySize = 15;
xHeight = 2;
yHeight = 4;
numColsX = 8;
numColsY = 6;

xCoords_1D_array = 0..xSize..numColsX; // 1D array
yCoords_1D_array = 0..ySize..numColsY; // 1D array

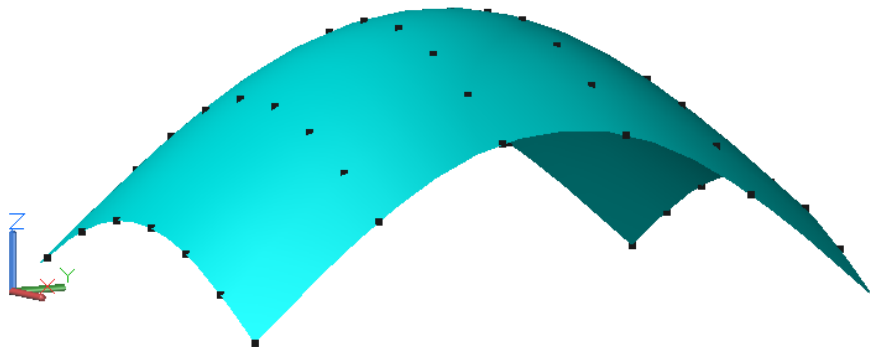
xSineWave_1D_array = (Math.Sin(0..180..numColsX) * xHeight); // 1D array
ySineWave_1D_array = (Math.Sin(0..180..numColsY) * yHeight); // 1D array

zHeight_2D_array = xSineWave_1D_array<1> + ySineWave_1D_array<2>; // using cartesian replication
// adding a 1D array to another 1D array creates a 2D array

surfacePoints_2D_array = Point.ByCoordinates(xCoords_1D_array<1>,
                                             yCoords_1D_array<2>,
                                             zHeight_2D_array<1><2>);

// this operation is taking the cartesian product of xCoords_1D_array and yCoords_1D_array
// [to create a 2D array] and then 'zipping' this 2D array with zHeight_2D_array

surface = NurbsSurface.ByPoints(surfacePoints_2D_array); // create a surface
```



Compare this script with the same geometry created Imperatively on Page 7.

Note: replication works well with rectangular collection, but the results may be undefined when used with ragged collections.

- Modifiers:** a variable can have a sequence of states

As we have seen, a variable may be defined in one statement (where it is on the left hand side of the '=' sign and has a value 'assigned' to it) and a variable may be referenced in a subsequent expression (on the right hand side of the '=' sign).

```
a = 10; // a is defined as int with the value 10;
b = a * 2; // b is defined by the expression 'a * 2'
```

We now introduce the concept in Associative programming of a 'modifier' in which a previously defined variable appears on both left and right hand side of statement, so as to have its original value 'modified'.

```
a; b; // define the variables to be output at the top or outer scope

a = 10; // original value of 'a'
b = a * 2; // define 'b' as dependent on 'a'
a = a + 1; // modify 'a' to be 1 more than its original value... now a = 11
// the value of 'b' is now = 22
```

Essentially, the variable 'a' has multiple states

```
state 1: a = 10; // original value of 'a'
state 2: a = a+1; // modify 'a' to be 1 more than its original value... now a = 11
```

when a variable is referenced, the value of its final state is used. In this case, when the statement

```
b = a * 2;
```

is evaluated, the value of the second (and final) state of 'a' is used (i.e. 11)

Modifiers are extremely useful to represent compound modeling operations, for example:

```
start = Point.ByCoordinates(10, 0, 0);
end   = Point.ByCoordinates(10, 5, 0);
a     = Line.ByStartPointEndPoint(start, end);
a     = a.ParameterTrim(0.2, 0.8); // trim the line;
a     = a.Translate(1, 1, 0);      // move the trimmed line;
```

Modifiers avoid having to give a separated variable name to each operation. In this case the intermediate states of are not visible or accessible.

Alternatively, the methods can be 'chained' together

```
start = Point.ByCoordinates(10, 0, 0);
end   = Point.ByCoordinates(10, 5, 0);
a     = Line.ByStartPointEndPoint(start, end).ParameterTrim(0.2, 0.8).Translate(1, 1, 0);
```

If intermediate states are required to be accessible, then each state must be assigned to a different variable:

```
start = Point.ByCoordinates(10, 0, 0);
end   = Point.ByCoordinates(10, 5, 0);
a     = Line.ByStartPointEndPoint(start, end);
b     = a.ParameterTrim(0.2, 0.8); // trim the line;
c     = b.Translate(1, 1, 0);      // move the trimmed line;
```

○ Modifying a member of collection:

A member of a collection can be modified, without breaking the integrity of the collection.

```
a; b; // define the variables to be output at the top or outer scope

a = 1..9..4; // initially a = {1,5,9}
b = a * 2;   // define 'b' as dependent on 'a', 'b' initially = {2, 10, 18}
           // subsequently 'b' = {2, 10, 2}
a[-1] = a[-1] + 1; // modify a member of the collection
           // a = {1,5,10} note: use of negative indexing from the end of the collection
```

Because a member of the collection **a** has been modified, this will trigger a re-computation of dependent statement

```
b = a * 2;
```

However, this re-computation will use the collection **a** including the modification to **a[-1]**

The use of modifier saves having to use explicit names for the intermediate states in a sequence of modelling operations.

Summary: A statement where the variable on the left hand side is also referenced within the expression on the right hand side is effectively a 'modifier' of that variable, e.g. **a = a + 1;**

○ In-line conditional: an in-line conditional is defined as:

```
variable = boolean_expression ? expression_if_true : expression_if_false;
```

for example:

```
a; b; // define the variables to be output at the top or outer scope

a = 4; // variable 'a' as a single value
b = a<2?10:20; // make the value of 'b' depend conditionally on the value 'a': 'b' = 20; //
```

- **In-line conditional with a collection**

A new collection can be built from an existing collection, using replication, where the individual members of the existing collections are evaluated, and the '*expression_if_true*' and the '*expression_if_false*' are used to build the new collection, for example:

```
a; b;           // define the variables to be output at the top or outer scope

a = 0..5;       // 'a' = { 0, 1, 2, 3, 4, 5 }
b = a<2?10:20; // 'b' = {10, 10, 20, 20, 20, 20}; // build collection 'b' by evaluating
// each member of the collection 'a'
```

- **Understanding the differences between Associative and Imperative programming:**

- Associative programming supports graph based dependencies and uses:
 - replication and replication guides,
 - modifiers
 - in associative programming a program statement not only defines that the value of a variable will be calculated based on references to other variables, but also defines a persistent dependency relationship between the variable whose value is being computed and the references to the other variables.
 - once a dependencies has been established, a subsequent change to these other variables in successive statements will cause the variable to be recomputed. In single-step debug mode the execution cursor may apparently move backwards through the source code as statements are executed and the value of variables are recomputed based on these dependencies.
 - in the absence of graph based dependencies, statements are executed in lexical order
- Imperative programming supports explicit 'flow control':
 - iteration with **for** and **while** loops
 - conditionals with **if..else** statements
 - in the absence of such flow explicit control statements are executed in lexical order
 - in imperative programming a program statement defines the value of a variable to be calculated based on references to other variables, but this is a 'one-time' operation. A subsequent change to these other variables in successive statements does not cause the variable to be recomputed.
 - forward references are not allowed: a variable cannot be computed from variables which have yet to been defined.

- **Additional functionality common to both Associative and Imperative language interpretation:**

- **Functions** are first class elements of the language,
Function must be defined in the global scope (the outermost block)

Functions are defined using the **def** key word as:

```
def function_name (argument_list) { program statements }
def function_name : return_type (argument_list) { program statements }
```

The *argument_list* is a comma separated list, with optional types

By convention function names start with an uppercase letter and argument names start with a lower case letter.

For example, with untyped arguments

```
def foo(x) = x * 5; // x is untyped, but the function will fail if it is not an int or a double
```

The function can be called with different arguments.. some which work and others which fail:

```
a = foo(10);           // a = 50.. with an int
b = foo(10.1);         // b = 50.5.. with a double
c = foo(myPoint);      // c = null.. representing failure
```

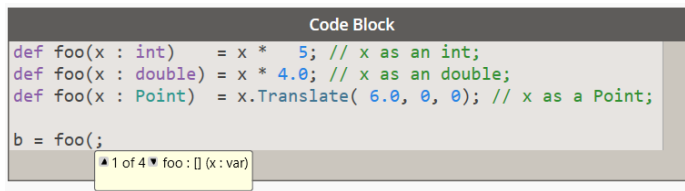
- **Function Overloading** it is possible to have multiple definitions of the same function with different types of arguments, for example:

```
def foo(x : int)      = x * 5;           // x as an int
def foo(x : double)  = x * 4.0;        // x as an double
def foo(x : Point)   = x.Translate( 6.0, 0, 0); // x as a Point
```

In addition the type of the return argument can be explicitly defined, for example:

```
def foo : int (x : int)      = x * 5;           // x as an int
def foo : double (x : double) = x * 4.0;        // x as an double
def foo : Point (x : Point)  = x.Translate( 6.0, 0, 0); // x as a Point
```

In the Dynamo UI use defined functions are immediately recognized and in this case because there are overloaded functions with the same name, the 'autocomplete' offers the user the alternative versions of the functions.



Depending on the type of the argument the appropriate version of the function will automatically be called, but if an argument is provided for which there is no overloaded method, then this will fail.

```
a = foo(10);           // a = 50.. with an int: DesignScript will call: def foo:int (x : int)
b = foo(10.1);         // b = 50.5.. with a double: DesignScript will call: def foo:double(x : double)
c = foo(myPoint);      // c = myPoint translated : DesignScript will call: def foo:Point(x : Point)
d = foo(yourLine);     // d = null.. DesignScript will call: def foo(x) which will fail
```

A function may have a single statement, of the following form: `def foo(x) = x * 5;`

Or alternatively a function may have multiple statements, in which the last must be a return statement defined using the `return` keyword, in the form:

`return = expression;` A function can return any type.

The `return` statement should be the last statement of the function and not nested within a block, for example:

```
def foo : int (x)
{
  y : int;           // define a local variable
  y = x * 5;         // use the local variable together with arguments
  return = y + 1;    // define a return value
}
```

- **Scoping issues with functions.** If we consider the function definition `foo`, above and within the same script have the statements

Generally, all variables within a function must either be defined locally or must be passed in as arguments. The following examples show various conditions where variable which are in scope are used within the function, but the function would fail if the scope changed or the variable referenced was no longer available. This could be the origin of major programming errors.

```
y = 1;
z = foo(y+10);
```

then because the variable `y` is defined both in the outer scope of the function and within the function, the execution of `foo` will change the value of `y` in the outer scope. This kind of side effect is potentially a source of errors.

In the case above, a variable was being assigned to within a function and had the same name as a variable in the main script, but whether or not the variable was present in the main script the function would still execute.

In the next example, a variable in the outer scope is referenced in a function

```
def foo (x)
{
    return = y * x;    // define a return value
}

y = 2;

z = foo(10); // z = 20
```

If `y` is commented out then it will not be available within `foo` and the function will fail, below:

```
def foo (x)
{
    return = y * x;    // this will fail
}

// y = 2; comment out 'y' so that it is not available within foo

z = foo(10); // z = null
```

This means that function `foo` can only operate in the context where the variable `y` is defined in the outer scope. One of the motivations for creating a function is that it can eventually be moved to a function library and used more generally. For the function to fulfill this role, it cannot reference variables in its outer scope.

As such when the code is maturing it is often helpful to ensure that all variables within a function are defined locally or are arguments.

○ Programming with Functions:

Functions are first class features of the DesignScript language and as such they can be assigned to variables and passed arguments, for example:

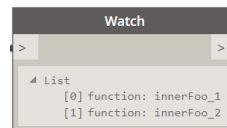
```
def innerFoo_1 (x : int)
{
    return = x * 5;    // define a return value
}

def innerFoo_2 (x : int)
{
    return = x + 5;    // define a return value
}

def outerFoo ( func : var, x : int)
{
    return = func(x);    // call the function
}

y = outerFoo(innerFoo_1, 10); // y = 50 .. call outerFoo with innerFoo1
z = outerFoo(innerFoo_2, 10); // z = 15 .. call outerFoo with innerFoo2
```

In addition a collection of functions can be defined, and the appropriate function can be selected by indexing into the collection, for example.



```
fooCollection = { innerFoo_1, innerFoo_2};
```

```
z = outerFoo(fooCollection[-1], 10); // use the last function in the fooCollection... y = 15
```

However, neither a DesignScript class nor a DesignScript method can be an argument.