SimplyRhino, London
February 12-14, 2020

Python Scripting for Rhino/Grasshopper

Long Nguyen

Rebrand.Ly/Python202002

Overview

- Essential Python programming concepts: variable, data, data types, conditional statements, loops, function, data structure
- The Python code editor in Rhino and Grasshopper
- Intro to RhinoCommon Library
- Diving deeper into Rhino geometries: Curve, Mesh, Surface, Brep
- Objected-oriented programming in Python
- Using Python with Grasshopper timer

- Randomness
- Generating and editing texts
- Reading/writing files

Three categories

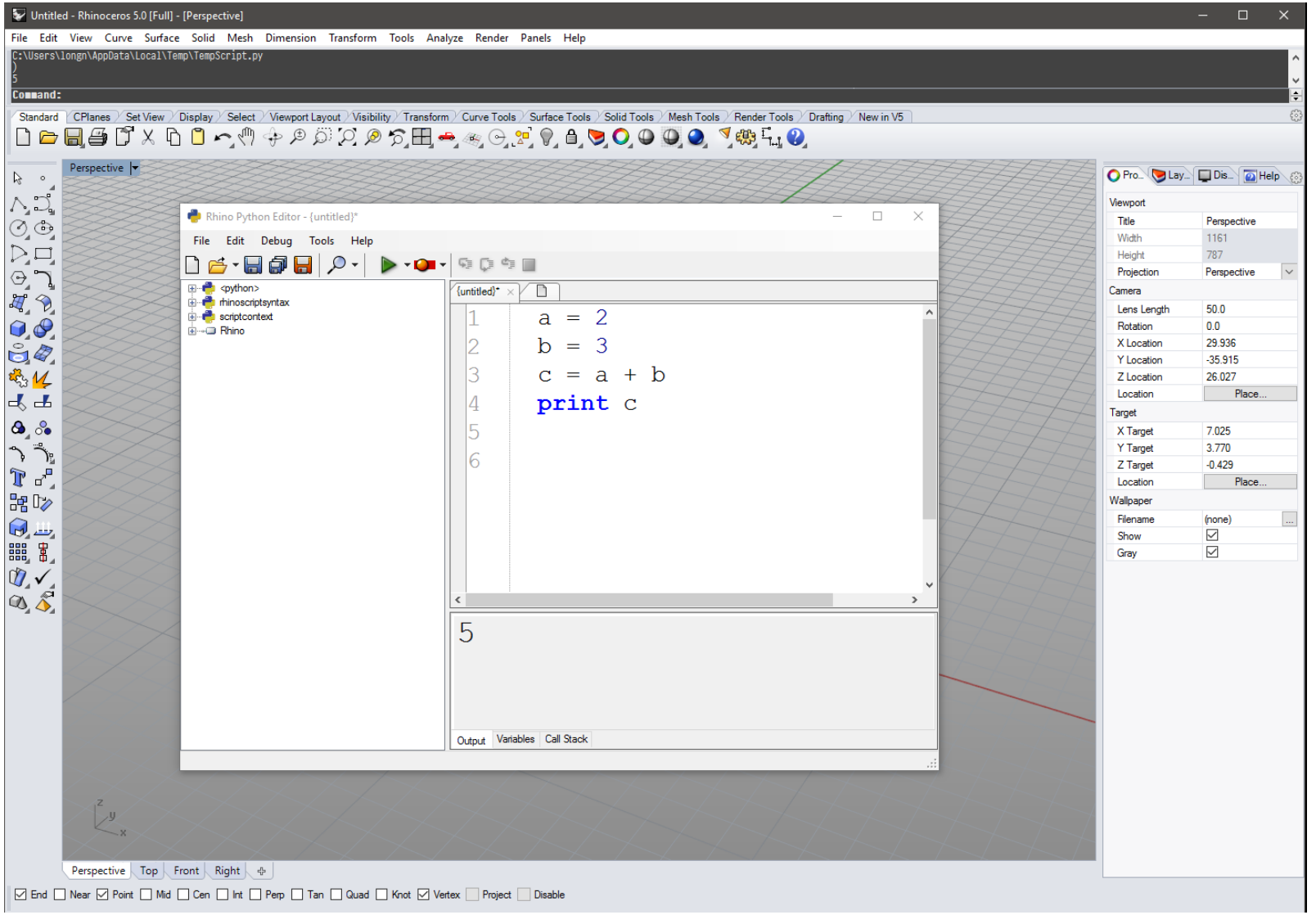
Python language

Rhino/Grasshopper API
programming library

Algorithms

Python in Rhino

The Python Code Editor in Rhino



Our very first script: Hello World!

Script

```
print "Hello World!"
```

Output Panel

```
Hello World!
```

Our very first script: Hello World!

Script

```
print "Hello World!"
print "Welcome to London"
```

Output Panel

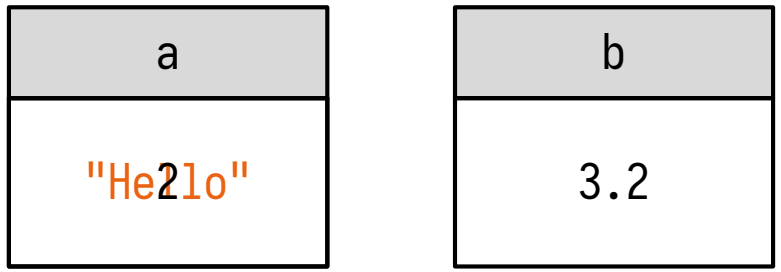
```
Hello World!
Welcome to London
```

Variables

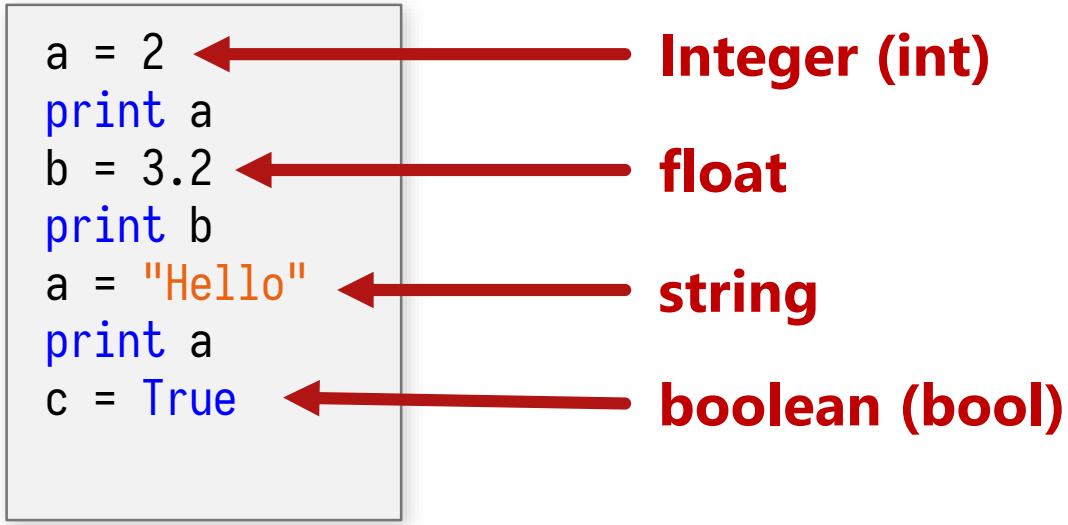
```
a = 2
print a
b = 3.2
print b
a = "Hello"
print a
```

```
2
3.2
Hello
```

Think of a variable is a container that store a certain value



Data Types



Later we will see data types defined by the Rhino programming library (RhinoCommon):

`Point3d, Vector3d, Mesh, Brep, RhinoDoc`

Variable's name vs. text

```
print "a"
a = 2
print a
```

```
a
2
```

What if ...

```
print a
a = 2
```

```
Message: name 'a' is not defined

Traceback:
  line 1, in <module>,
"C:\Users\longn\AppData\Local\Temp\TempScript.py"
```

Arithmetic operators

```
a = 2
b = 3
c = a + b
print c
c = c + 2
print c
d = c * 3
print d
```

```
5
7
21
```

Commenting the code

```
# This is a comment
a = 2
b = 3
c = a + b
print c
c = c + 2
# This is a another comment
print c
d = c * 3
print d
```

Using comments to “disable” a line of code

```
# This is a comment
a = 2
b = 3
c = a + b
print c
c = c + 2
# This is a another comment
print c
#d = c * 3
print d
```

Logical Operator

```
a = True
b = False
c = a or b
d = a and b
d = not d
```

Comparision operators

```
a = 3
b = 2
c = a > b #True
print a > 4 #False
print a == 3 #True
print a == 3 and c #True
print b and c #Error (why?)
```

Conditional Statement

```
a = 3
b = 2

if a > b:
    print "a is larger than b"
```

```
a = 3
b = 2

if a > b:
    print "a is larger than b"
else:
    print "a is not larger than b"
```

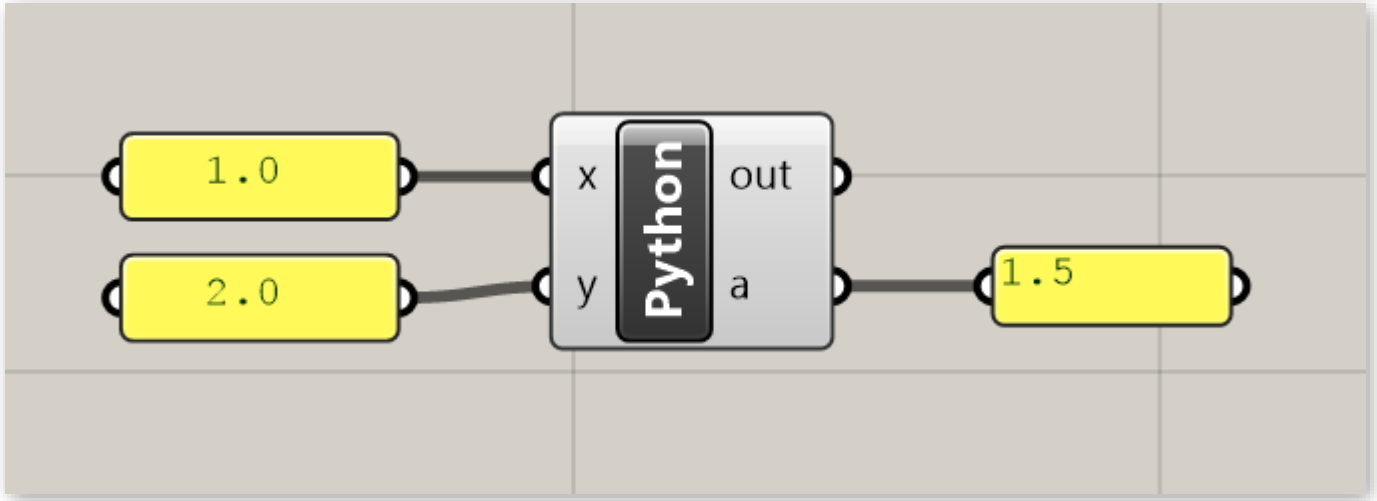
Nested Conditional Statements

```
a = 3
b = 2
c = 1

# Find the largest number among a, b and c

if a > b:
    if a > c:
        print "a is largest"
    else:
        print "c is largest"
else:  # b > a
    if b > c:
        print "b is largest"
    else:
        print "c is largest"
```

Python in Grasshopper

Simple Example:
Compute the average of two numbers

```
INPUTS
x: float
y: float

OUTPUTS:
a
```

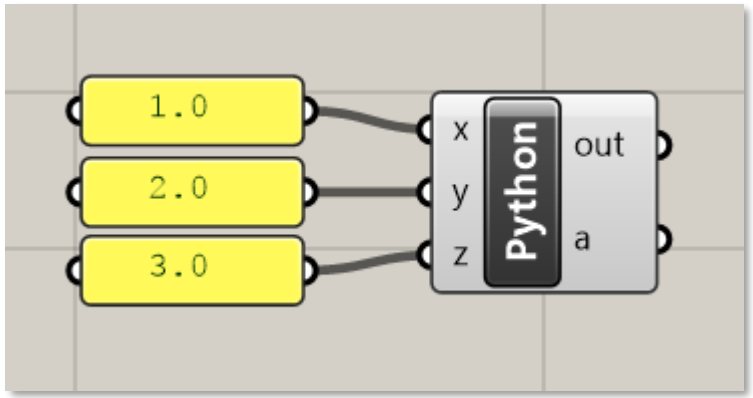



```
average = 0.5 * (x + y)
a = average
```

Using RhinoCommon from Python (in Grasshopper)

Create a point

```
INPUTS
x: float
y: float
z: float

OUTPUTS:
a
```




```python
import Rhino.Geometry as rg

myPoint = rg.Point3d(1.0, 2.0, 3.0)
a = myPoint
```

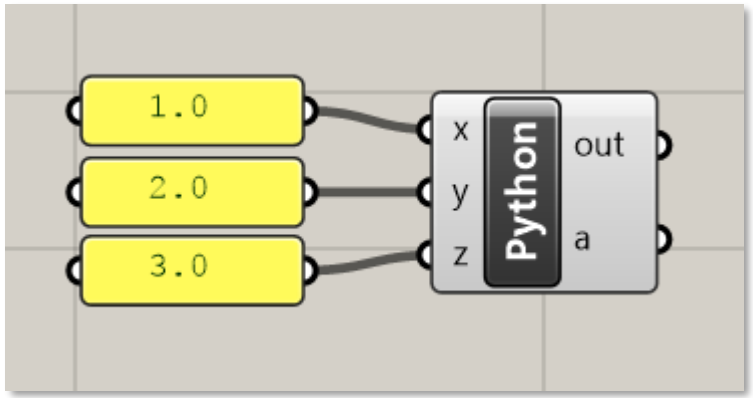
BEHIND THE SCENE:
"myPoint" is the a variable/container that store a value of type Point3d

myPoint
Point3d(1.0, 2.0, 3.0)

Create a point

INPUTS
x: float
y: float
z: float

OUTPUTS:
a



BEHIND THE SCENE:
"myPoint" is the a variable/container that store a value of type Point3d

```
import Rhino.Geometry as rg

myPoint = rg.Point3d(x, y, z) # Use input parameters
a = myPoint
```

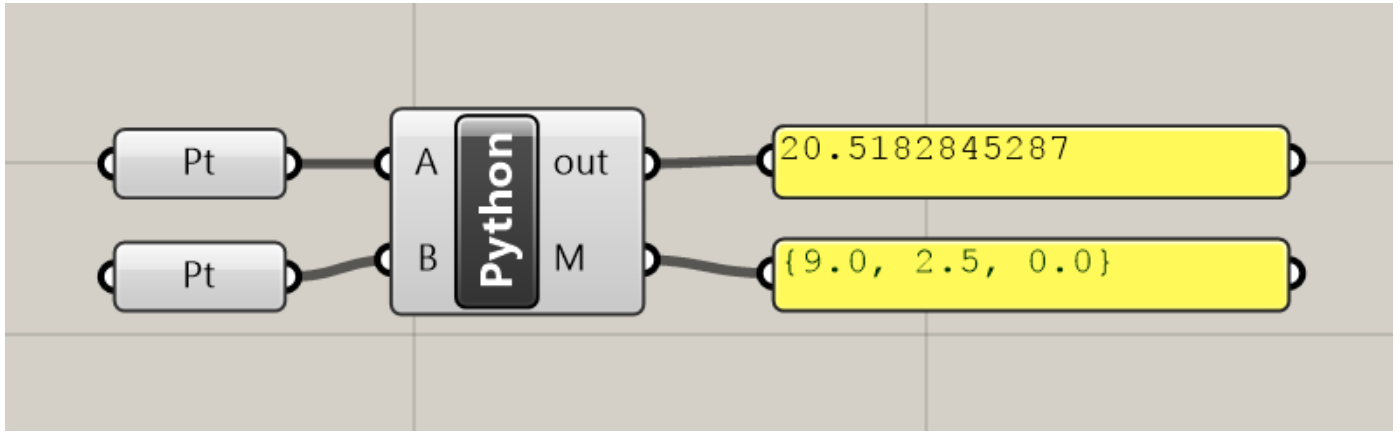
a
Point3d(1.0, 2.0, 3.0)

Point3d properties and methods

```
import Rhino.Geometry as rg

myPoint = rg.Point3d(x, y, z)
a = myPoint
print myPoint.X # accessing the X coordinate of myPoint
```

Compute mid-point between two points and the distance in between

```
INPUTS
A: Point3d
B: Point3d

OUTPUTS:
M
```




```python
import Rhino.Geometry as rg

midX = 0.5 * (A.X + B.X)
midY = 0.5 * (A.Y + B.Y)
midZ = 0.5 * (A.Z + B.Z)

M = rg.Point3d(midX, midY, midZ)

distance = A.DistanceTo(B)

print distance
```

Alternatively:

```python
M = 0.5 * (A + B)
```

Some other Rhino Geometry data types

INPUTS
none

OUTPUTS:
oPlane
oCircle

```
import Rhino.Geometry as rg

myVector = rg.Vector3d(0.0, 1.0, 0.0)
myPoint = rg.Point3d(2.0, 2.0, 2.0)
myPlane = rg.Plane(myPoint, myVector)

oPlane = myPlane

myCircle = rg.Circle(myPlane, 3)

oCircle = myCircle
```

Data types are important!

```
Plane(Point3d, Vector3d)
Plane(Point3d, Vector3d, Vector3d)
Plane(Point3d, Point3d, Point3d)

Circle(float)
Circle(Point3d, float)
Circle(Plane, float)
```

- A function/command can have more than one variant.
- Each variant takes in a different set of parameters.
- Python determines which is the right version to use SOLELY based on the data types of the input parameters

List

storing multiple items in “one” item

What is a list?

- A list allows us to store multiple items in a sequence
- The list itself can be treated like a single item

```
heroes = ["Batman", "Wolverine", "Superman"] # Create a list of three string values and
                                               store the list in the variable named heroes

print heroes # Textually display the content of the list

# Retrieve items from the list
firstHero = heroes[0]
lastHero = heroes[-1]

# Retrieve a sublist
sublist = heroes[0:2] # This will give a smaller list: ["Batman", "Wolverine“]
```

Lists functions

- Let's start with this list that contains the name of three superheroes

```
heroes = ["Batman", "Wolverine", "Superman"]
```

- Modifying an existing element in the list

```
heroes[1] = "Thor" # Change the 1st hero (i.e. Wolverine) to Thor
```

- Adding and removing elements to a list

```
heroes.append("Captain America") # Add a single new element to the end of the list
heroes.extend( ["Ant Man", "Robin"]  ) # append a new list to the current list
heroes.insert(1, "Hulk") # Insert "Hulk" between "Batman" and "Wolverine"
heroes.remove("Superman")
```

Other useful list functions

```
heroes = ["Batman", "Wolverine", "Superman"]

...

n = heroes.count("Batman") # Count how many times "Batman" appears in the list
heroes.sort() # Sort the items (in-place)
heroes.reverse() # Reverse the list (in-place)
elementCount = len(heroes) # Get the total length of the list
```

List and Rhino geometries

```
import Rhino.Geometry as rg

points = []

points.append(rg.Point3d(1.0, 1.0, 0.0))
points.append(rg.Point3d(2.0, 2.0, 0.0))
points.append(rg.Point3d(3.0, 3.0, 0.0))
points.append(rg.Point3d(4.0, 4.0, 0.0))

oPoints = points
```

Loops

“For” loop

```
for i in [0, 1, 2, 3, 4]:
    print i
```

=

```
print 0
print 1
print 2
print 3
print 4
```

Output Panel

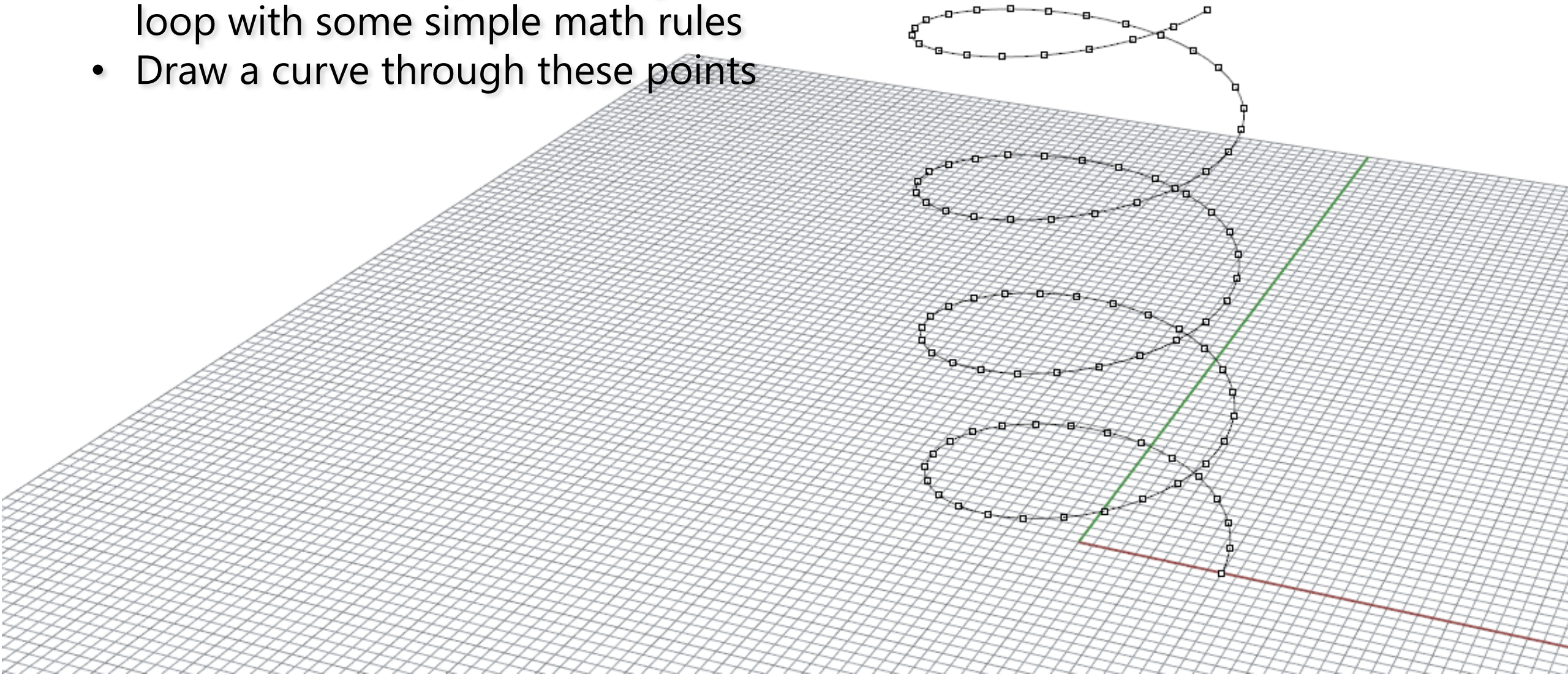
```
0
1
2
3
4
```

Creating geometries using for loop

```
import Rhino.Geometry as rg

points = []

for i in range(0, 50):
    myPoint = rg.Point3d(i, i, 0)
    points.append(myPoint)

oPoints = points
```

Live example:
List, loop and some maths

- Create and list of points using a “for” loop with some simple math rules
- Draw a curve through these points



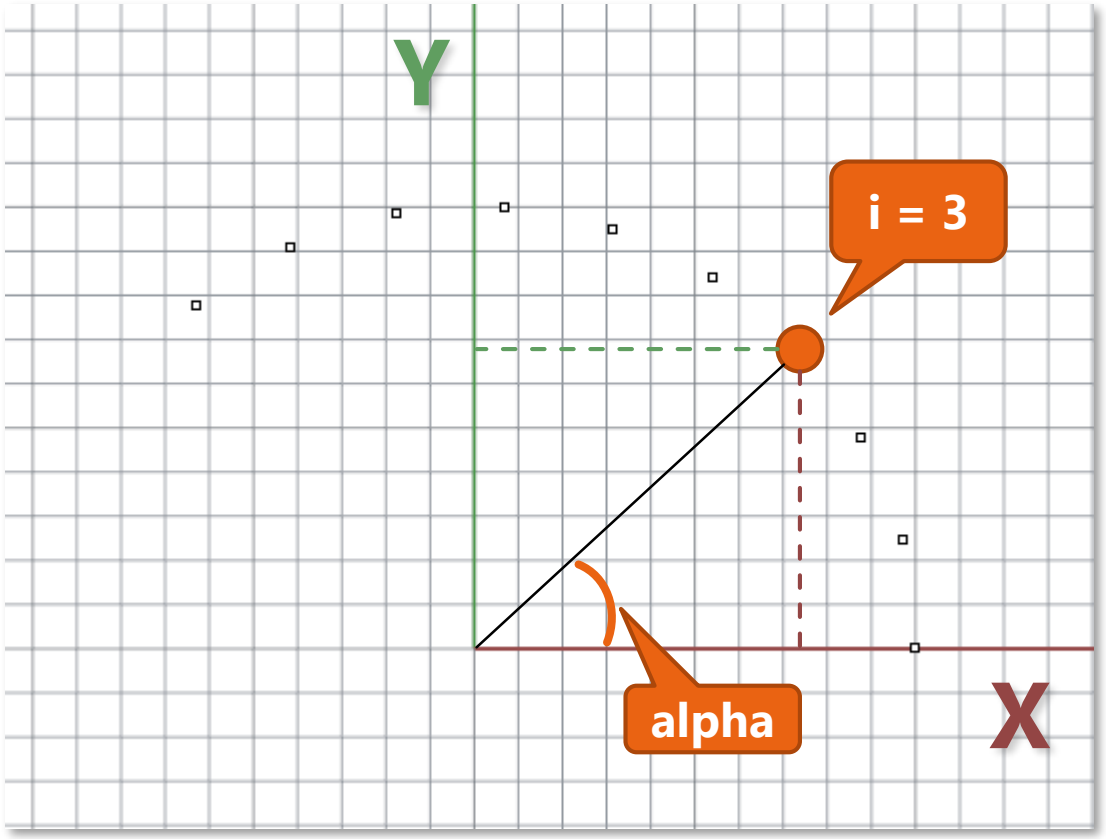
Live Example: Create a list using for loop

Step 1: Create points along a circle

```
import Rhino.Geometry as rg
import math

points = []

for i in range(0, 10):
    alpha = i * 0.25
    x = 10 * math.cos(alpha)
    y = 10 * math.sin(alpha)
    z = 0
    points.append(rg.Point3d(x, y, z))

oPoints = points
```

INPUTS
none

OUTPUTS:
oPoints
oCurves



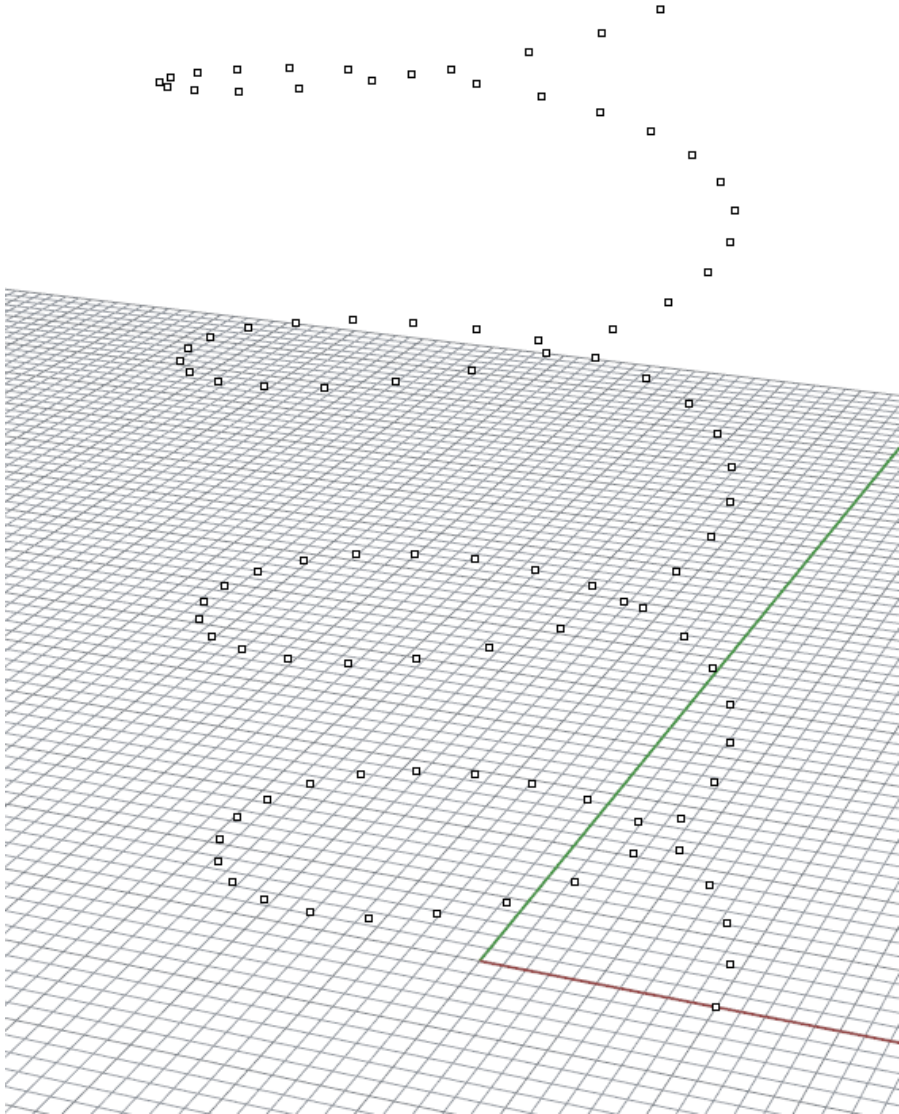
Live Example: Create a list using for loop

Step 2: raising the z coordinates to create the helix

```
import Rhino.Geometry as rg
import math

points = []

for i in range(0, 100):
    alpha = i * 0.25
    x = 10 * math.cos(alpha)
    y = 10 * math.sin(alpha)
    z = i * 0.4
    points.append(rg.Point3d(x, y, z))

oPoints = points
```

INPUTS
none

OUTPUTS:
oPoints
oCurve



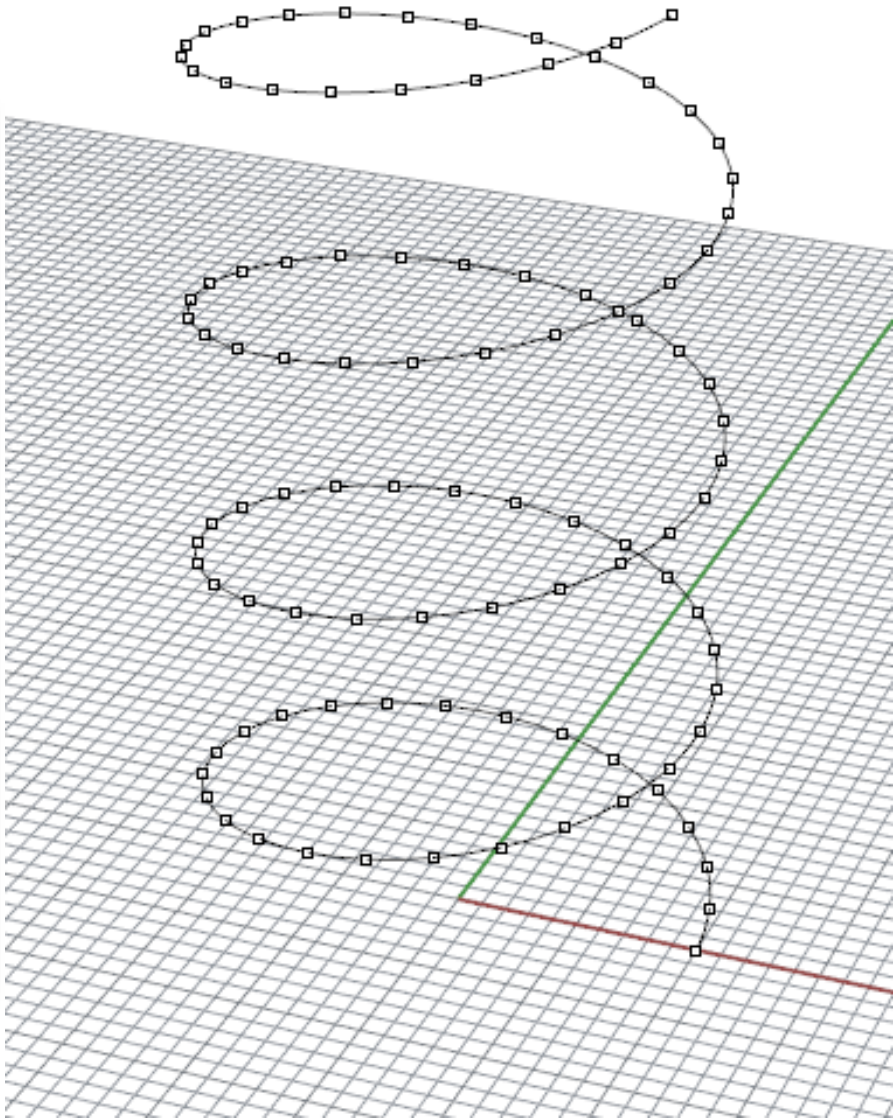
Live Example: Create a list using for loop

Step 3: draw a curve

```
import Rhino.Geometry as rg
import math

points = []

for i in range(0, 100):
    alpha = i * 0.25
    x = 10 * math.cos(alpha)
    y = 10 * math.sin(alpha)
    z = i * 0.4
    points.append(rg.Point3d(x, y, z))

oPoints = points
oCurve = rg.NurbsCurve.CreateInterpolatedCurve(points, 3)
```

INPUTS
none

OUTPUTS:
oPoints
oCurve



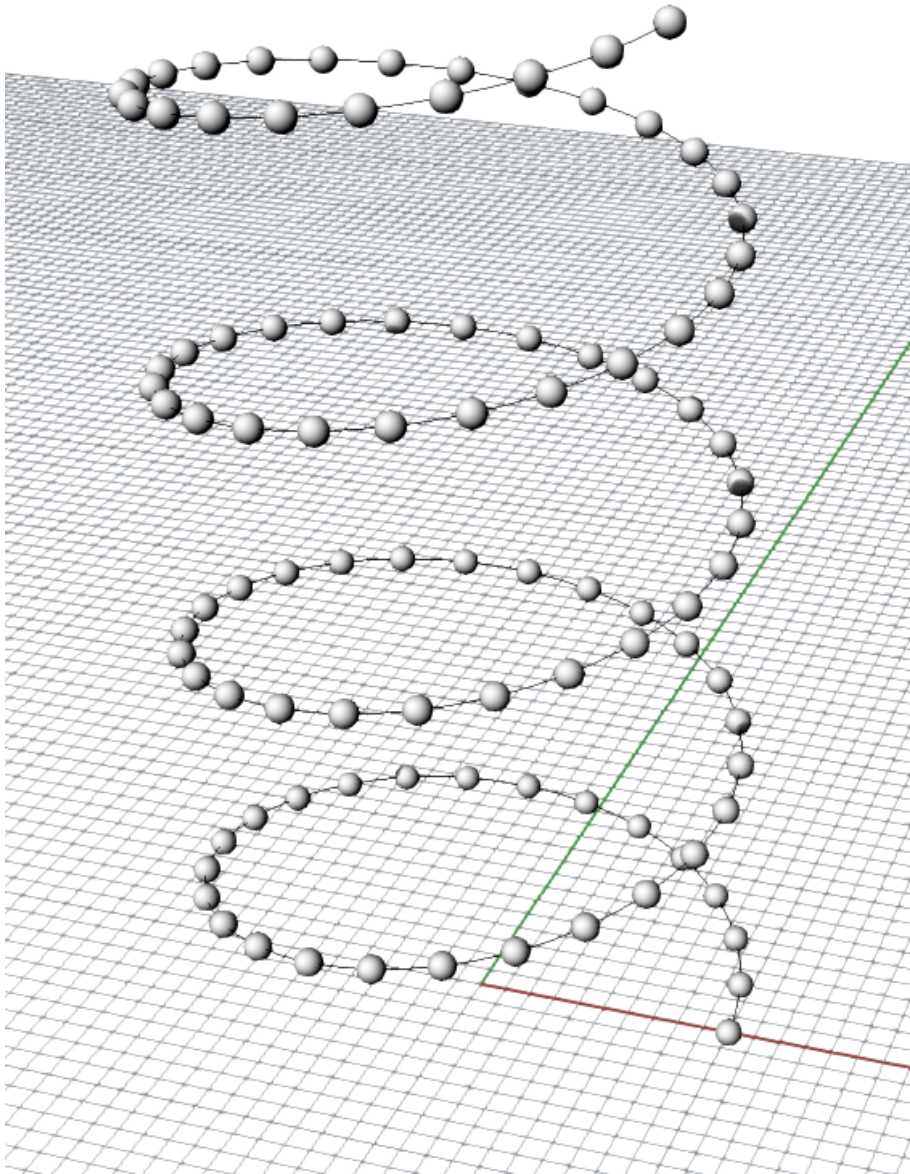
Live example:

Lists as input parameters

Live Example: Lists as input parameters

```
INPUTS
iPoints: List of Point3d

OUTPUTS:
oSpheres
```

```python
import Rhino.Geometry as rg

spheres = []

for i in range(0, len(iPoints)):
    center = iPoints[i]
    mySphere = rg.Sphere(center, 0.5)
    spheres.append(mySphere)

oSpheres = spheres
```

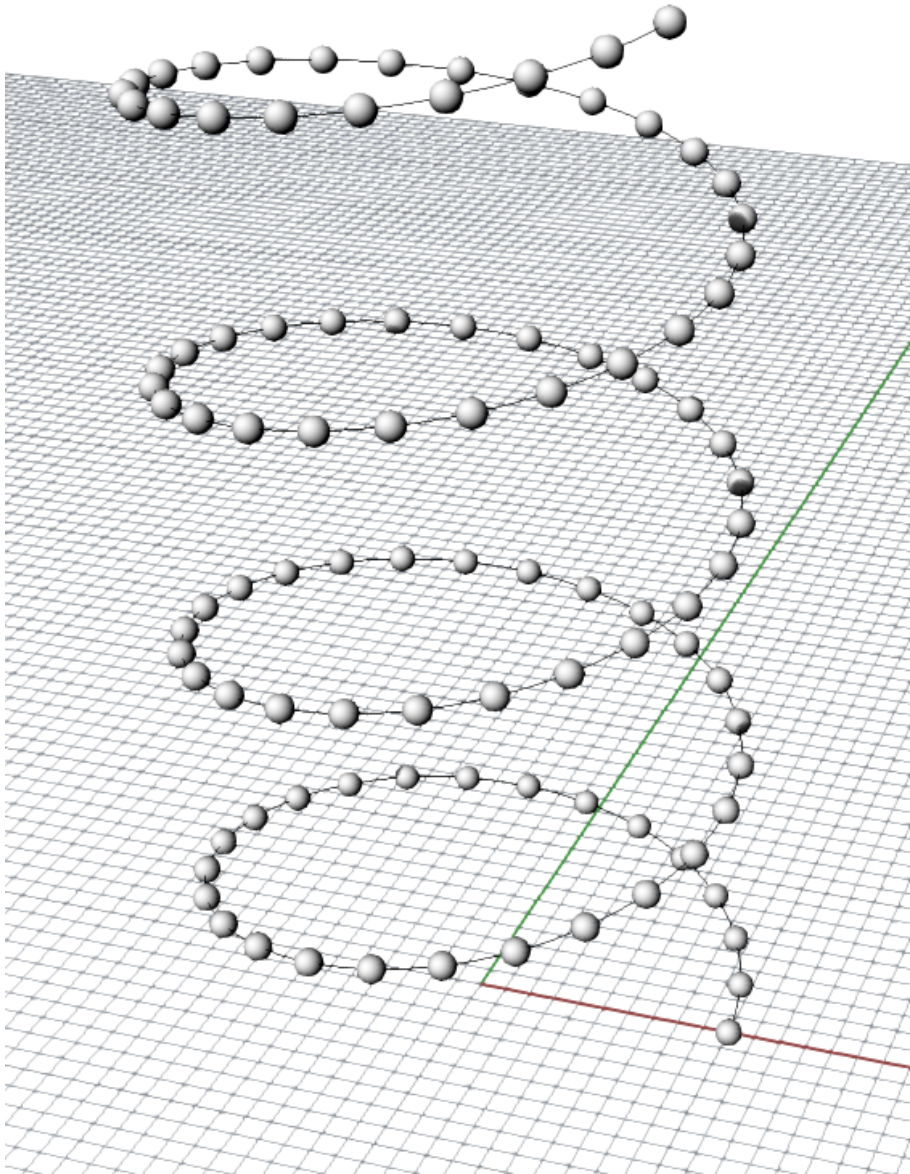


Live Example: Lists as input parameters

Accessing element of a list without knowing the current index i

```
INPUTS
iPoints: List of Point3d

OUTPUTS:
oSpheres
```

```python
import Rhino.Geometry as rg

spheres = []

for point in iPoints:
    center = point
    mySphere = rg.Sphere(center, 0.5)
    spheres.append(mySphere)

oSpheres = spheres
```

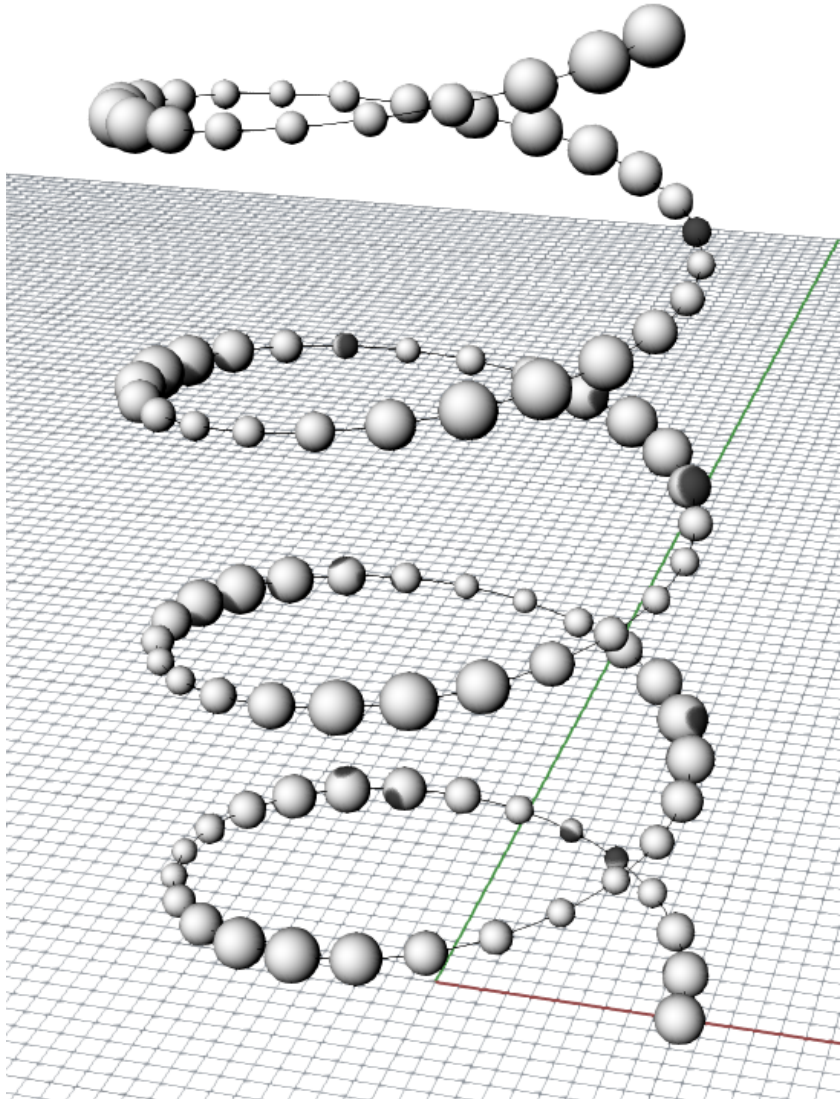


Live Example: Lists as input parameters

… but, with the index i we can do extra stuff

```
INPUTS
iPoints: List of Point3d

OUTPUTS:
oSpheres
```

```python
import Rhino.Geometry as rg
import math
spheres = []

for i in range(0, len(iPoints)):
    center = iPoints[i]
    r = 0.75 + 0.25 * math.cos(i)
    mySphere = rg.Sphere(center, r)
    spheres.append(mySphere)

oSpheres = spheres
```



Two common list programming patterns

Summation/Accumulation/Aggregation

Let's say we have a list of numbers

```
numbers = [2, 3, 2, 1, 6, 9, 8, 2]
```

We can use for loop to compute the sum of all elements

```
sum = 0

for i in range(0, len(numbers)):
    sum = sum + numbers[i]

print sum
```

Equivalently …

```
sum = 0

for number in numbers:
    sum += number

print sum
```

Summation/Accumulation

Compute the centroid of a set of input points

```
INPUTS
iPoints: List of Point3d

OUTPUTS:
oCentroid
```

```python
import Rhino.Geometry as rg

centroid = rg.Point3d(0, 0, 0)

for point in iPoints:
    centroid += point

centroid /= len(iPoints)

oCentroid = centroid
```

Finding the “extreme” element

Let’s say we have a list of numbers

```
numbers = [2, 3, 2, 1, 6, 9, 8, 2]
```

Find the smallest number

```
smallest = numbers[0]

for number in numbers:
    if number < smallest:
        smallest = number

print smallest
```

Finding the “extreme” element

Finding the nearest point relative to a reference point

```
INPUTS
iPoints: List of Point3d
iReference: Point3d

OUTPUTS:
oNearest
```

```
nearest = iPoints[0]

for point in iPoints:
    if point.DistanceTo(iReference) < nearest.DistanceTo(iReference):
        nearest = point

oNearest = nearest
```

Nested Loops

Nested loops: A for loop within a for loop

```
for i in range(0, 2):
    print "i = " + i
    for j in range(0, 3):
        print "  j = " + j
```

```
i = 0
  j = 0
  j = 1
  j = 2
i = 1
  j = 0
  j = 1
  j = 2
```

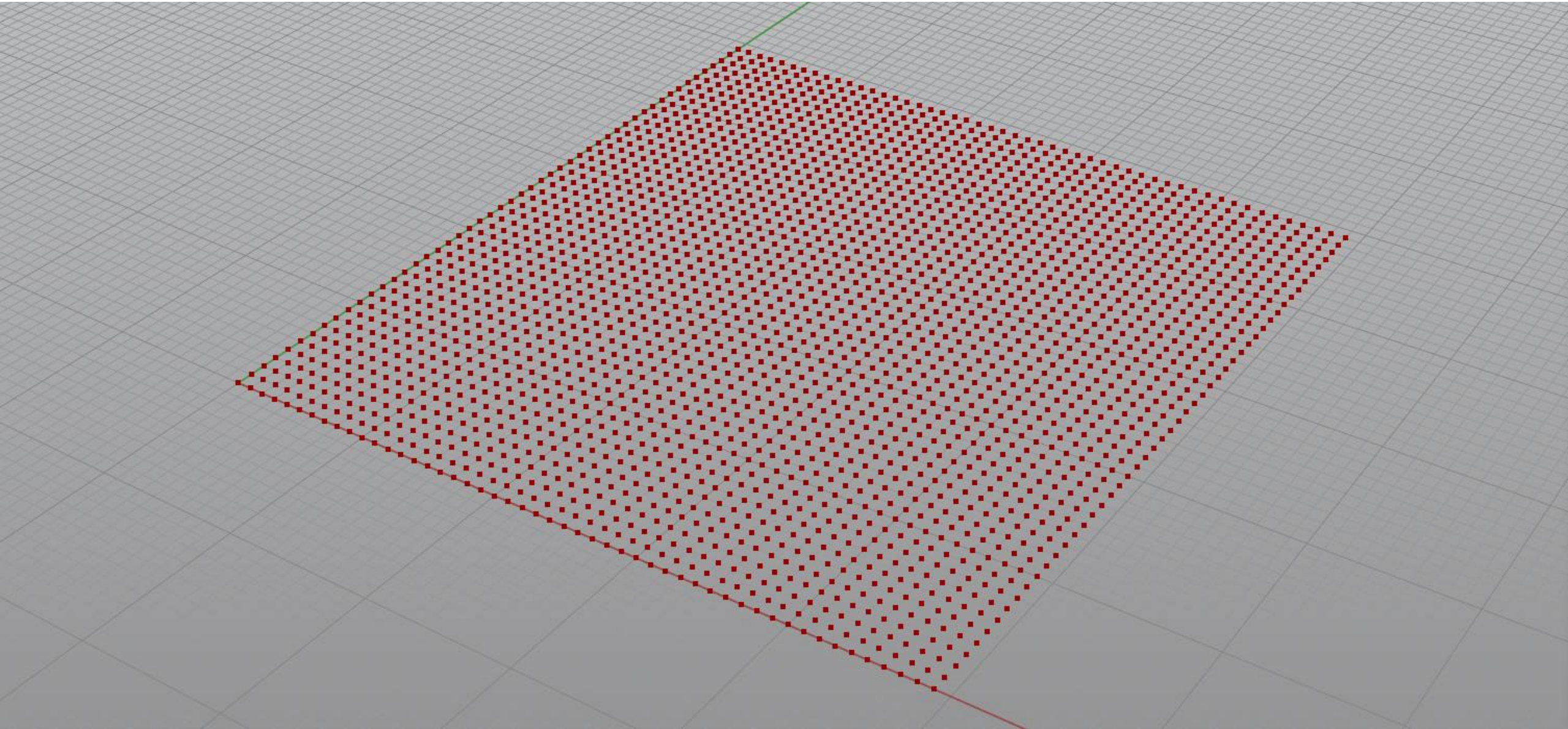
Creating a 2D grid of points

```
import Rhino.Geometry as rg
import math

points = []

for i in range(0, 50):
    for j in range(0, 50):
        points.append(rg.Point3d(i, j, 0.0))

oGeometry = points
```

INPUTS
none

OUTPUTS:
oGeometry

Creating a 2D grid of points

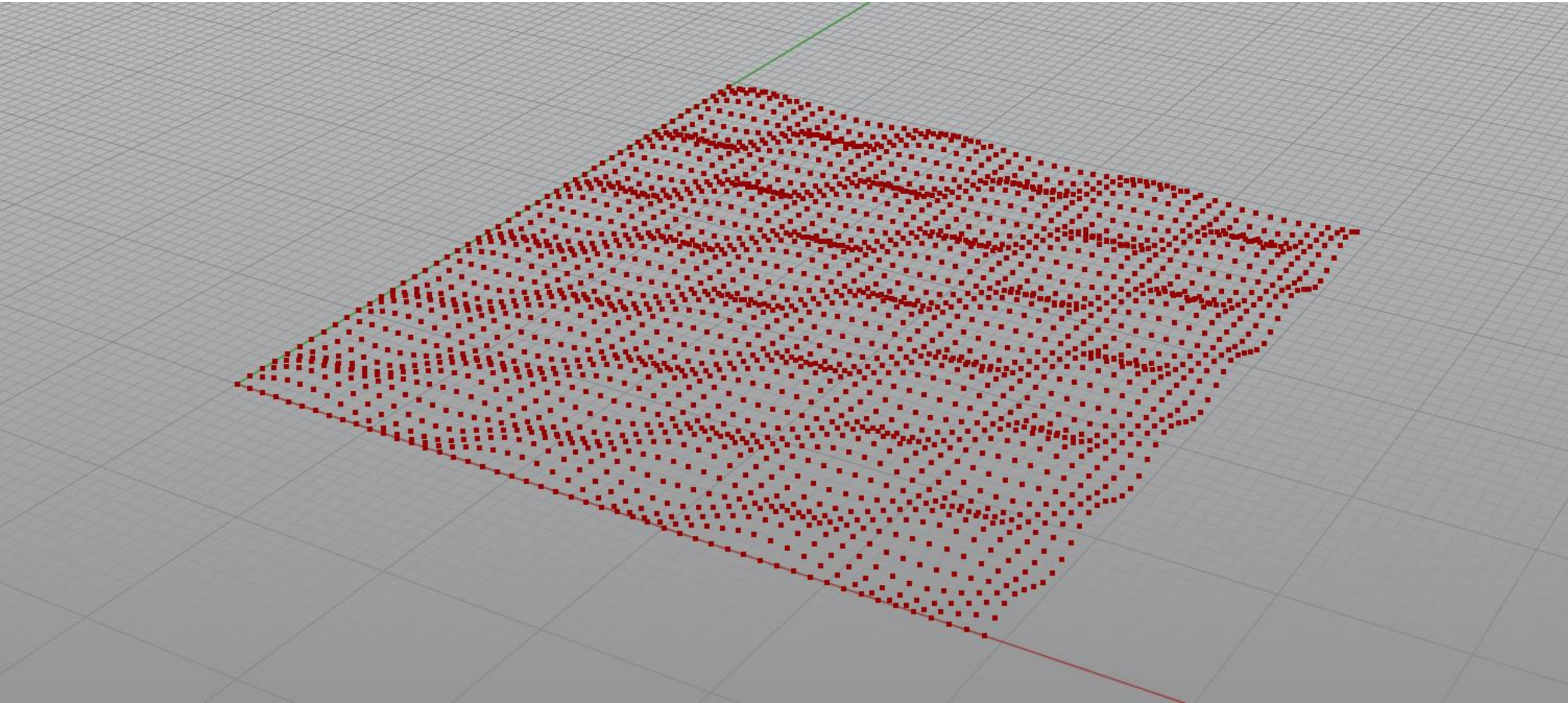


Adjusting the heights to create wavy effect

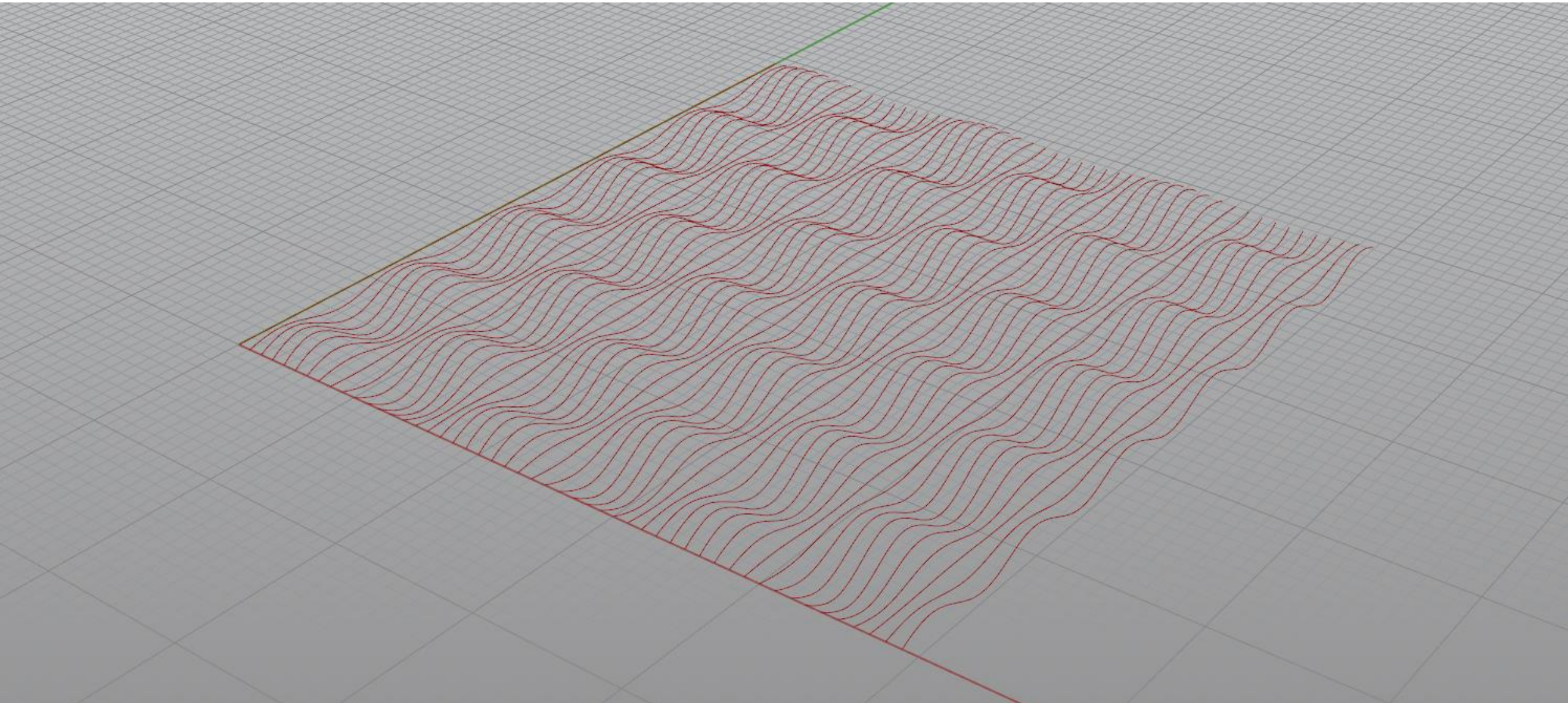
```
import Rhino.Geometry as rg
import math

points = []

for i in range(0, 50):
    for j in range(0, 50):
        z = math.sin(i * 0.4) * math.sin(j * 0.7)
        points.append(rg.Point3d(i, j, z))

oGeometry = points
```

```
INPUTS
none

OUTPUTS:
oGeometry
```

Adjusting the heights to create wavy effect



Creating a list of wavy curves



Creating a list of wavy curves

```
import Rhino.Geometry as rg
import math

curves = []

for i in range(0, 50):
    points = []
    for j in range(0, 50):
        z = math.sin(i * 0.4) * math.sin(j * 0.7)
        points.append(rg.Point3d(i, j, z))
    curve = rg.NurbsCurve.CreateInterpolatedCurve(points, 3)
    curves.append(curve)

oGeometry = curves
```

```
INPUTS
none

OUTPUTS:
oGeometry
```

Creating a loft surface from the wavy curves


```
import Rhino.Geometry as rg
import math

curves = []

for i in range(0, 50):
    points = []
    for j in range(0, 50):
        z = math.sin(i * 0.4) * math.sin(j * 0.7)
        points.append(rg.Point3d(i, j, z))
    curve = rg.NurbsCurve.CreateInterpolatedCurve(points, 3)
    curves.append(curve)

brep = rg.Brep.CreateFromLoft(curves, rg.Point3d.Unset, rg.Point3d.Unset, rg.LoftType.Normal, False)

oGeometry = brep
```

Nested Lists

Nested list: a list whose elements are also lists!

Define a nested list

```
myNestedList = [ [0, 1], [2, 3, 4], [2] ]
```

Or, equivalently…

```
myNestedList = []
myNestedList.append([0, 1])
myNestedList.append([2, 3, 4])
myNestedList.append([2])
```

Retrieving elements from a nested list

```
myNestedList = [ [0, 1], [2, 3, 4], [2] ]


firstSublist = myNestedList[1] # This will give the sublist [2, 3, 4]
n = myNestedList[1][0] # This will give the zeroth element in the first sublist
```

Nested List: an multidimensional structure to store stuff

A matrix can be described naturally by a nested list

$$\text{myMatrix} = \begin{pmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \end{pmatrix}$$

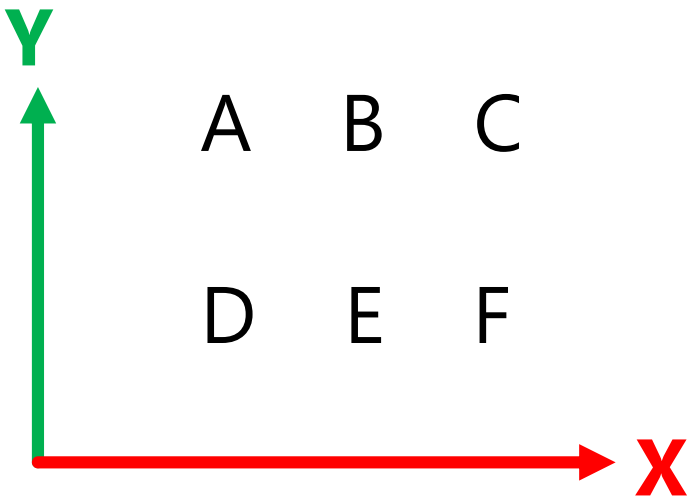
```
myMatrix = [ [0, 2, 4], [1, 3, 5] ]
```

Retrieving an item from the matrix is just a matter of using the index operator (the “square-bracket” operator)

```
# Retrieve the item at the zeroth row, first column
item = myMatrix[0][1]
print item # This will display “2”
```

Nested List: an multidimensional structure to store stuff

Guess what, we can store a 2D array of Point3d in a 2D nested list too

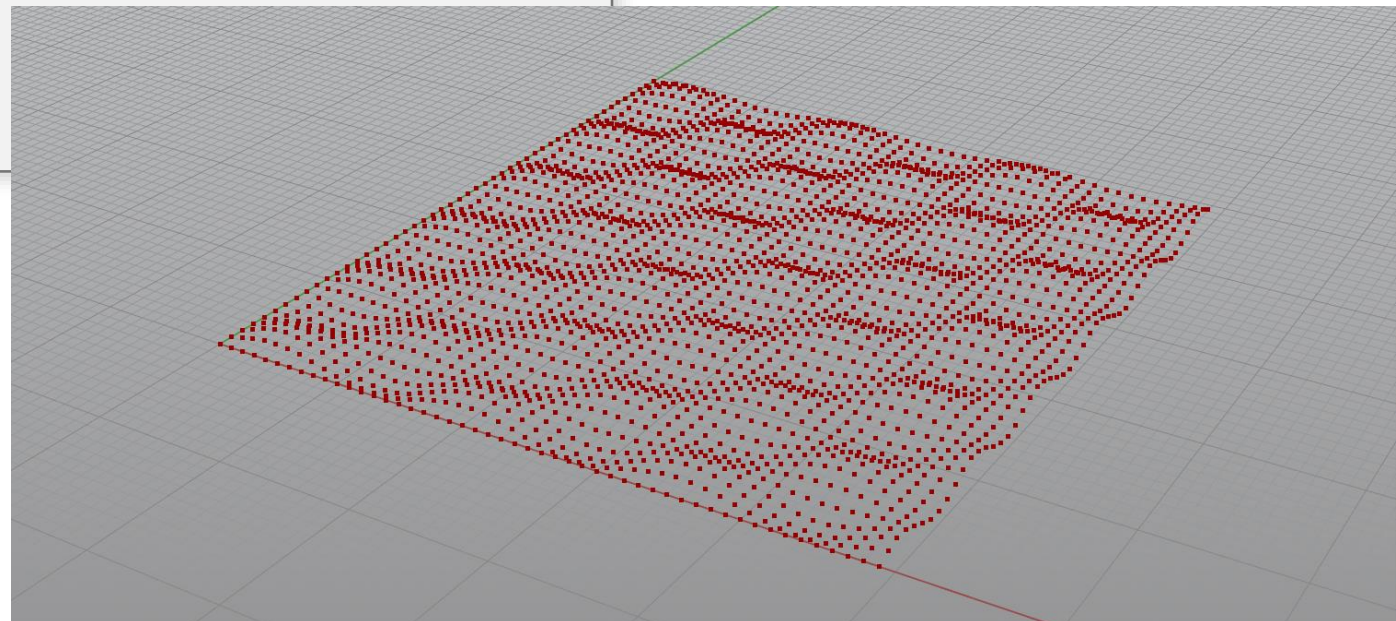



```
points = [ [A, B, C], [D, E, F] ]
```

```
# Retrieve the point at the first row, second column
p = points[1][2]
```

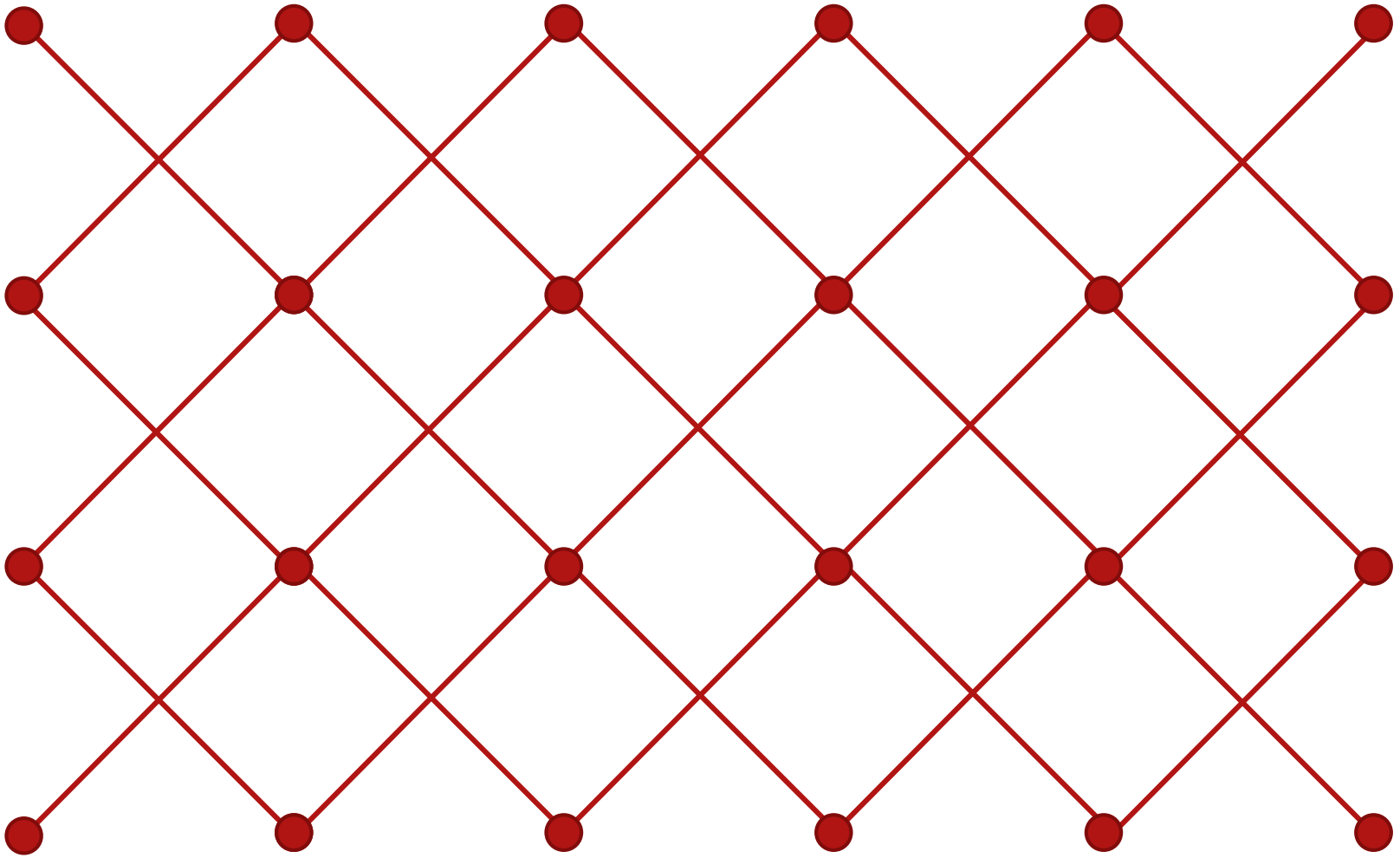
Previously, we stored a 2D array of points in a 1D list

```
import Rhino.Geometry as rg
import math

points = []

for i in range(0, 50):
    for j in range(0, 50):
        z = math.sin(i * 0.4) * math.sin(j * 0.7)
        points.append(rg.Point3d(i, j, z))

oGeometry = points
```



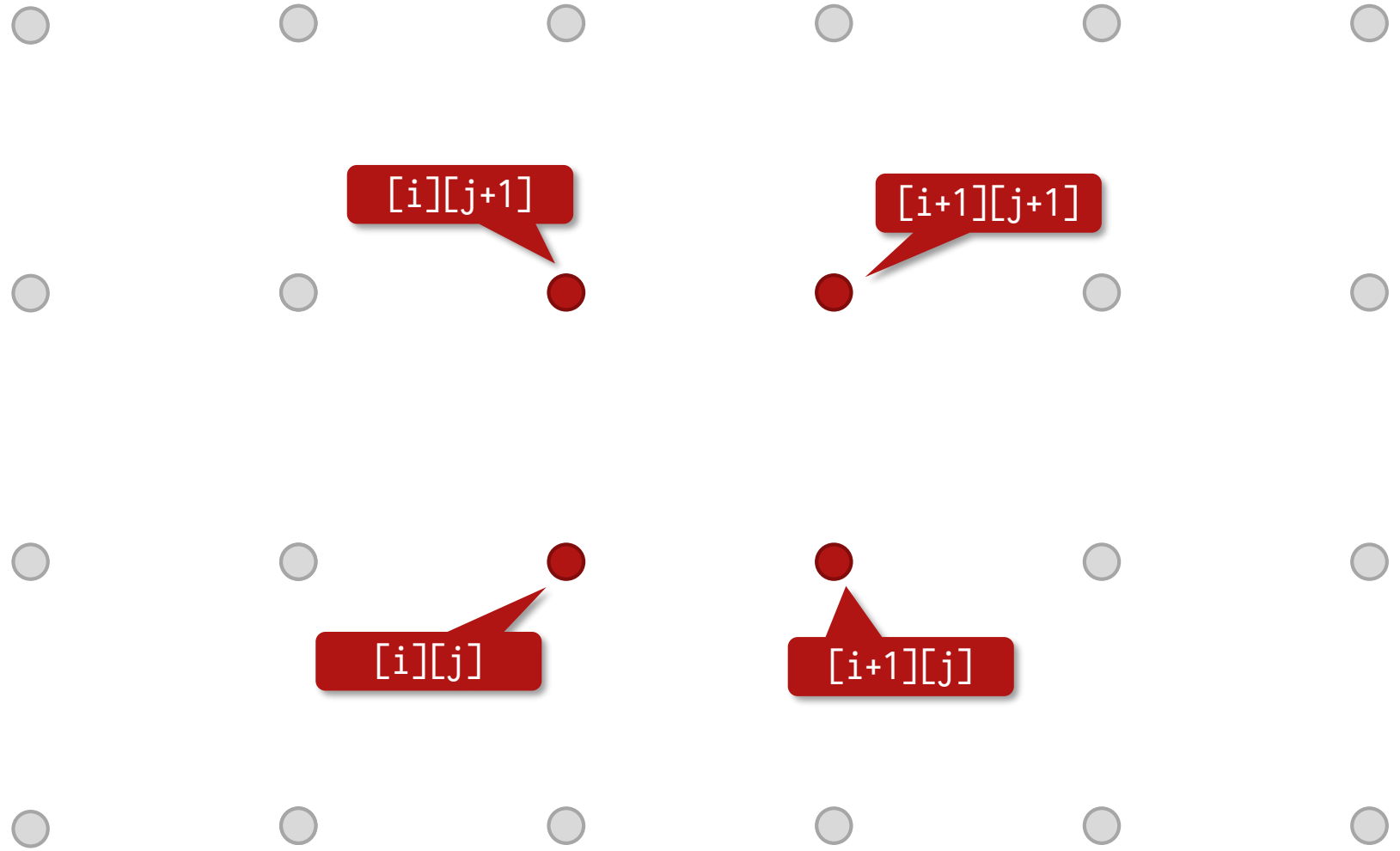
Doing math using 2D indices are kinda easier

- Lets say we want to create a diagrid pattern based on a 2D array of point



Doing math using 2D indices is kinda easier

- Lets say we want to create a diagrid pattern based on a 2D array of point

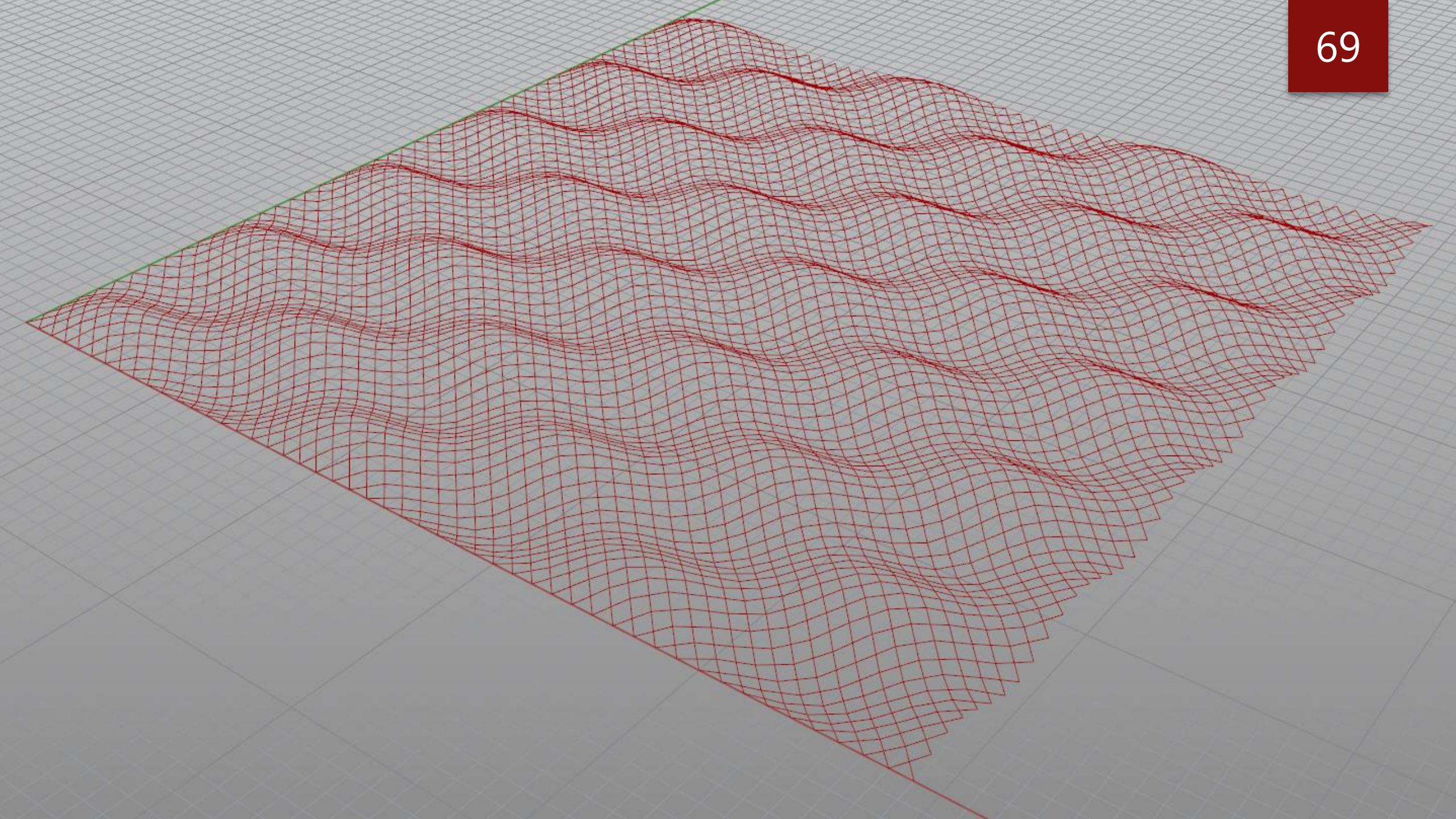


Doing math using 2D indices are kinda easier

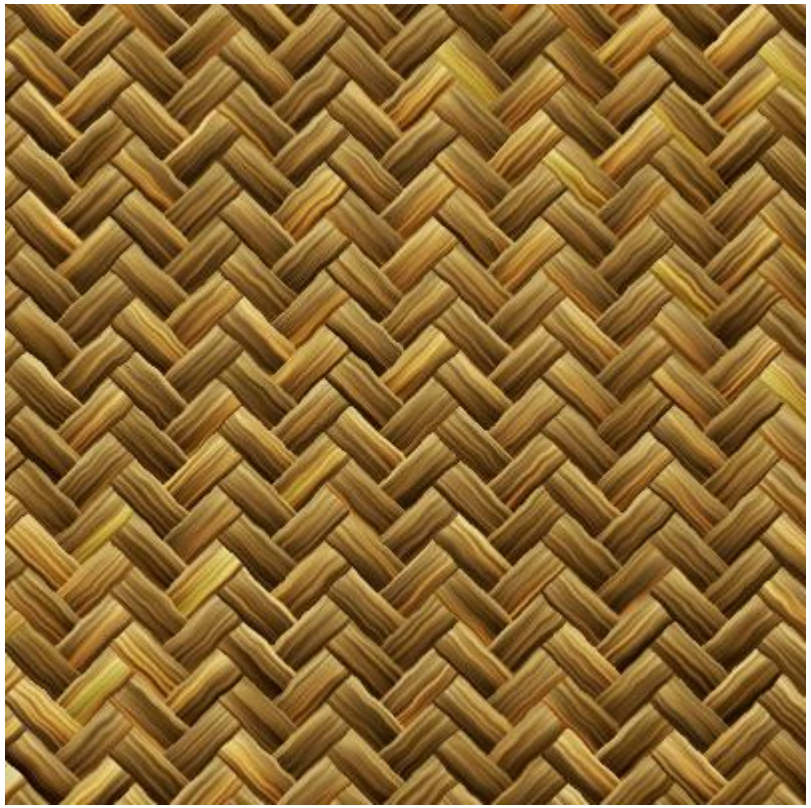
```
import Rhino.Geometry as rg
import math

points = []

for i in range(0, 50):
    row = []
    for j in range(0, 50):
        z = math.sin(i * 0.4) * math.sin(j * 0.7)
        row.append(rg.Point3d(i, j, z))
    points.append(row)

oGeometry = points
```

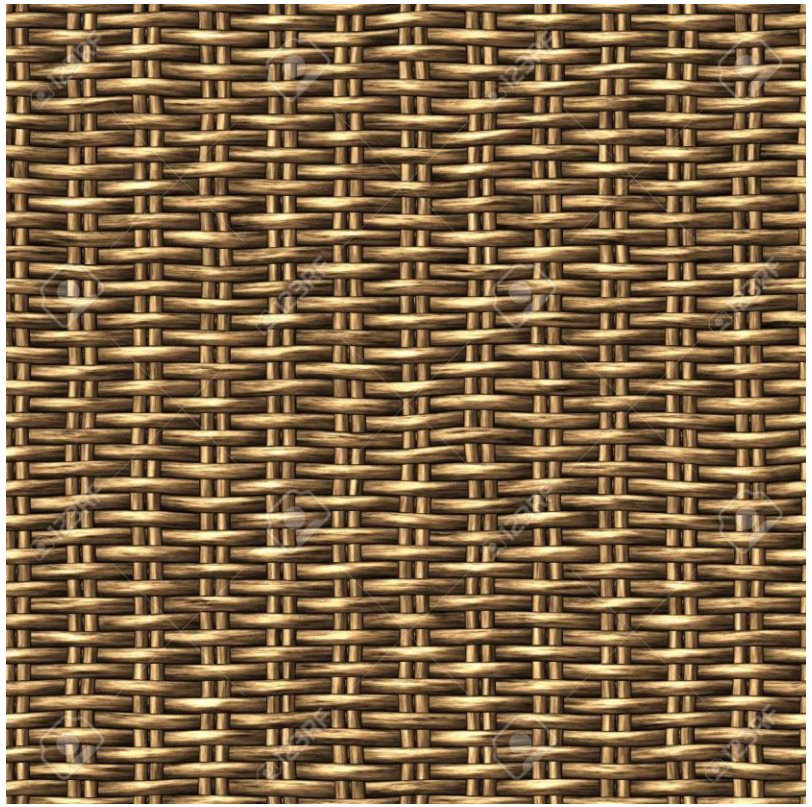
Doing math using 2D indices are kinda easier

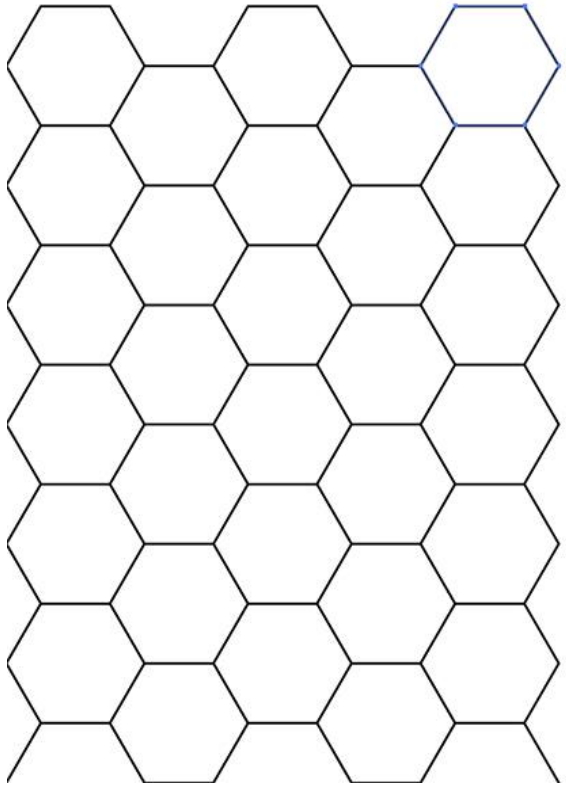
```
# ... continue from previous example

lines = []

for i in range(0, 49):
    for j in range(0, 49):
        line = rg.Line(points[i][j], points[i + 1][j + 1])
        lines.append(line)
        line = rg.Line(points[i + 1][j], points[i][j + 1])
        lines.append(line)

oGeometry = lines
```



Playing with 2D indexing logic
Endless patterned geometry possibilities







Functions

Sometimes codes need to be repeated multiple times

```
# Compute the average of a list of numbers

myNumbers = [2, 3, 4, 5]
sum = 0
for number in myNumbers
    sum += number
average = sum / len(myNumbers)
print average

# Compute the average of Another list of numbers

yourNumbers = [5, 0, 1, 2]
sum = 0
for number in yourNumbers
    sum += number
average = sum / len(yourNumbers)
print average
```

Output Panel

```
3.5
4
```

Functions: Package and reuse codes!

```
def ComputeAverage(numbers):
    sum = 0
    for number in numbers:
        sum = sum + number
    average = sum / len(numbers)
    print average

ComputeAverage([2, 3, 4, 5])
ComputeAverage([5, 0, 1, 2])

# Terminology: argument
```

Output Panel

```
3.5
4
```

Functions can have an output:

```
def ComputeAverage(numbers):
    sum = 0
    for number in numbers:
        sum += number
    average = sum / len(numbers)
    return average

result1 = ComputeAverage([2, 3, 4, 5])
result2 = ComputeAverage([5, 0, 1, 2])

# Do whatever you want with result1 and result2
print result1
print result2
```

Output Panel

```
3.5
4
```

Function can have multiple inputs, but only one output at most (But we can easily work around this)

```
def ComputeCylinderAreaAndVolume(r, h):
    area = 3.14 * r * r * 2 + 3.14 * r * 2 * h
    volume = 3.14 * r * r * h
    return (area, volume) # Package area and volume into a tuple

result = ComputeCylinderAreaAndVolume(2.1, 3.4)
area = result[0]
volume = result[1]
```

Function naming convention

```
def ComputeCylinderAreaAndVolume(r, h):
    area = 3.14 * r * r * 2 + 3.14 * r * 2 * h
    volume = 3.14 * r * r * h
    return (area, volume)
```

- Function name should use upper casing
- Function name typically should contains a verb
- Input parameter name should use camel casing (first letter is lower case and every new word start with upper case)

Function is like a black box

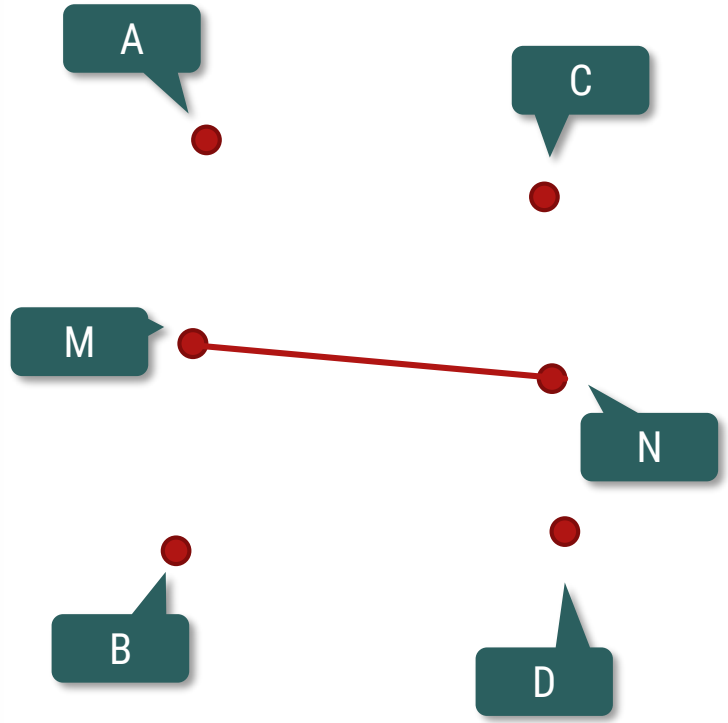
Hide the implementation details away

Function with RhinoCommon Geometry

```
def ComputeMidPoint(A, B):
    return (A + B) * 0.5
```

Invoke a function from within another function

```
import Rhino.Geometry as rg

def ComputeMidPoint(A, B):
    return (A + B) * 0.5

def CreateMidLine(A, B, C, D):
    M = ComputeMidPoint(A, B)
    N = ComputeMidPoint(C, D)
    return rg.Line(M, N)

oMidLine = CreateMidLine(rg.Point3d(0,0,0), \
rg.Point3d(0,3,0), rg.Point3d(3,0,0), rg.Point3d(3,3,0))
```



Three types of function that you will commonly see


```
# Type 1: normal Python functions that we have been dealing with today
ComputeCylinderAreaAndVolume(20, 2.0)

# Type 2: a function that is associated with specific data type (methods)
# For example: The method DistanceTo(), defined for datatype Point3d
myPoint3d.DistanceTo(yourPoint)

# Type 3: static function (static method) of a class
rg.Vector3d.Cross(myVector, yourVector)
rg.Curve.CreateInterpolatedCurve(...)
```

Vector3d.IsPerpendicularTo Method (Vector3d, Double)

Determines whether this vector is perpendicular to another vector, within a provided angle tolerance.

Namespace: Rhino.Geometry
Assembly: RhinoCommon (in RhinoCommon.dll) Version: Rhino 6.0

Syntax

C# VB

```
public bool IsPerpendicularTo(
        Vector3d other,
        double angleTolerance
)
```

Parameters

other
Type: Rhino.Geometry.Vector3d
Vector to use for comparison.

angleTolerance
Type: System.Double
Angle tolerance (in radians).

Return Value

Type: Boolean
true if vectors form Pi-radians (90-degree) angles with each other; otherwise false.