

# SCRIPTING INTRODUCTION

## Python for Rhino5

Section 1: Introduction

Section 2: Primitives & Variables

Section 3: Methods

Section 4: Control Structures

Section 5: Arrays

## Section 1: Introduction

## Introduction: Note

THIS IS A WORK IN PROGRESS

This scripting presentation covers a basic introduction to scripting in the Python language for Rhino5. Content for this presentation has been compiled from many sources and is the result of years of hacking, writing and trying to understand scripting and programming myself. I owe a great deal of thanks to Raymond Kettner, Chris Lasch, David Rutten, Skylar Tibbits, Axel Kilian, Kyle Steinfeld and Pat Palmer. Much of the text included in this document has been compiled from notes taken during a summer course on programming that I took within the Computer and Information Science Dept at UPenn. All content has been compiled and translated for use with Python for Rhino5.

## Introduction: references

Rhino

<http://www.rhino3d.com/>

<http://blog.rhino3d.com/>

Grasshopper

<http://grasshopper.rhino3d.com/>

Microsoft DotNET Framework 2.0

<http://www.microsoft.com/downloads/details.aspx?familyid=0856eacb-4362-4b0d-8edd-aab15c5e04f5&displaylang=en>

RhinoScript.org

<http://www.rhinoscript.org/>

RhinoScript Wiki

<http://en.wiki.mcneel.com/default.aspx/McNeel/RhinoScript.html>

RhinoPython

<http://python.rhino3d.com/>

<http://download.rhino3d.com/IronPython/5.0/RhinoPython101/>

Python

<http://learnpythonthehardway.org/>

<http://proquest.safaribooksonline.com/sso/incommon>

1. Beazley, D. Python essential reference (4th ed.) New York: Addison-Wesley Professional, 2009.
2. Summerfield, M. Programming in Python 3: a complete introduction to the Python language (2nd ed.) New York: Addison-Wesley, 2010.

Stylianios Dritsas

<http://www.dritsas.net/>

Axel Kilian

<http://designexplorer.net/>

David Rutten

<http://www.grasshopper3d.com/profile/DavidRutten>

Skylar Tibbits

<http://www.sjet.us/>

Cornell Scripting Google Group

[cornellarchscripting@googlegroups.com](mailto:cornellarchscripting@googlegroups.com)

Rhino API

Help/Plug-ins/Rhinoscript

## Introduction: Scripts are like Recipes

Scripts can perform mathematical actions, respond to variable conditions, allow for user interaction and they have the ability to control their 'flow'. Scripts can behave dynamically.

Every script deals with:

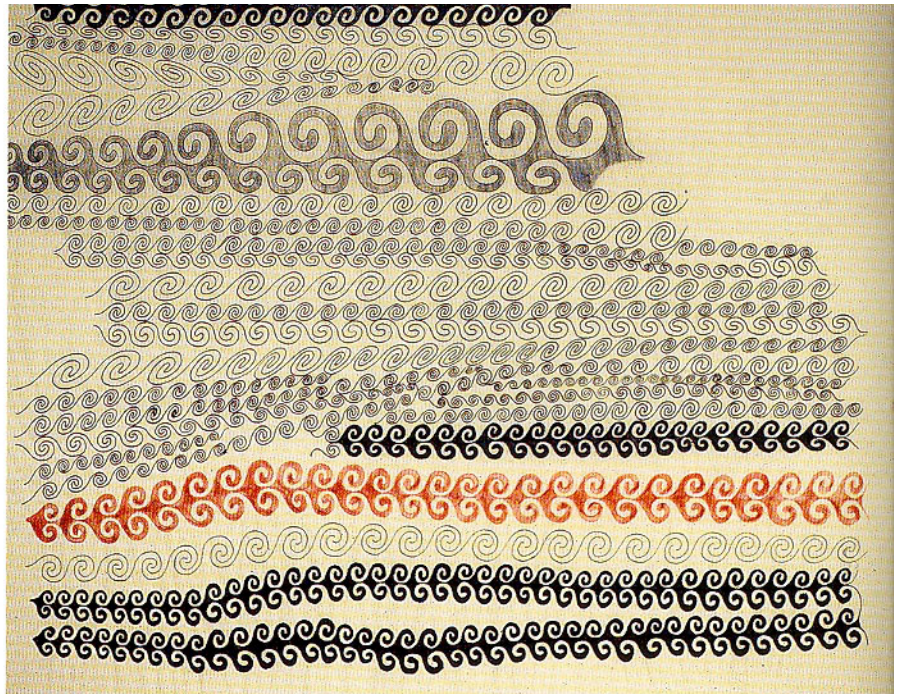
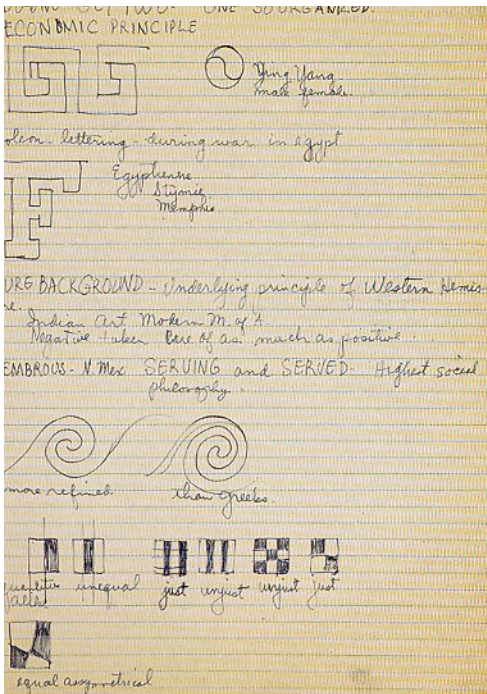
1. Flow control (skipping and repeating lines)
2. Variable control (logical and mathematical operations – the storage of data)
3. Input and output (how I interact with the script)

Python is a programming language. It was released in 1991.

It is a language that you can learn to write, and the computer can be made to understand.

It is a powerful language but it is not simple!

Python is object-orientated, meaning that objects are *mutable*.



Ruth Asawa



## Introduction: Syntax and Semantics

The way in which programming languages deal with flow control, variable control and input and output is called the syntax. The syntax is a set of rules that define what is and isn't valid. To begin to deal with the logic of scripting, one must learn the syntax of the programming language.

Syntax is the "grammar" of the language. The syntax of Python is large, but finite. Syntax must be absolutely correct including case (capitalization).

The Python editor will point out every syntax error. These error messages may be helpful.

Semantics is the "meaning" of your program. Semantic errors cause your answers to be wrong. You may or may not get error messages. If your program is not doing what you want it to do, though it runs, the error is semantic.

Python has syntax and semantics. We will start learning them both immediately.

Python also has "packages". Packages are sort of like vocabulary bundles they are also called "libraries", "base classes" or "base libraries". Some of you may get into this area.

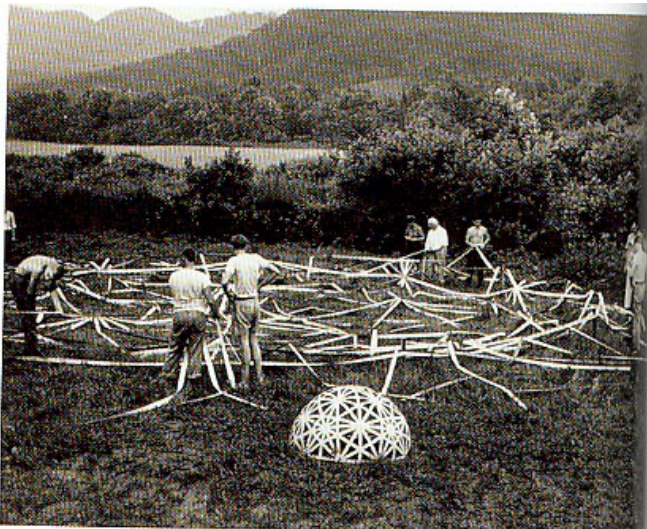
Things to watch out for:

Python is Case Sensitive.

Python is Indent Sensitive. Indentation delineates the scope of loops, conditionals, functions and classes.

You DON'T need to declare variables and types. This is great news!

The '#' symbol is used to make comments in Python.



Bucky Fuller, Black Mountain College

## Introduction: Integrated Development Environment (IDE)

Context is an IDE (Integrated Development Environment) an editor, for writing programs, a debugger, to help find mistakes, a viewer, to see the parts of your program. You may also simply use the PythonScript editor within Rhino. For our purposes, we will use the Python Editor in Rhino5.

It is free and downloadable  
<http://www.context.cx/>

### Rhino Monkey

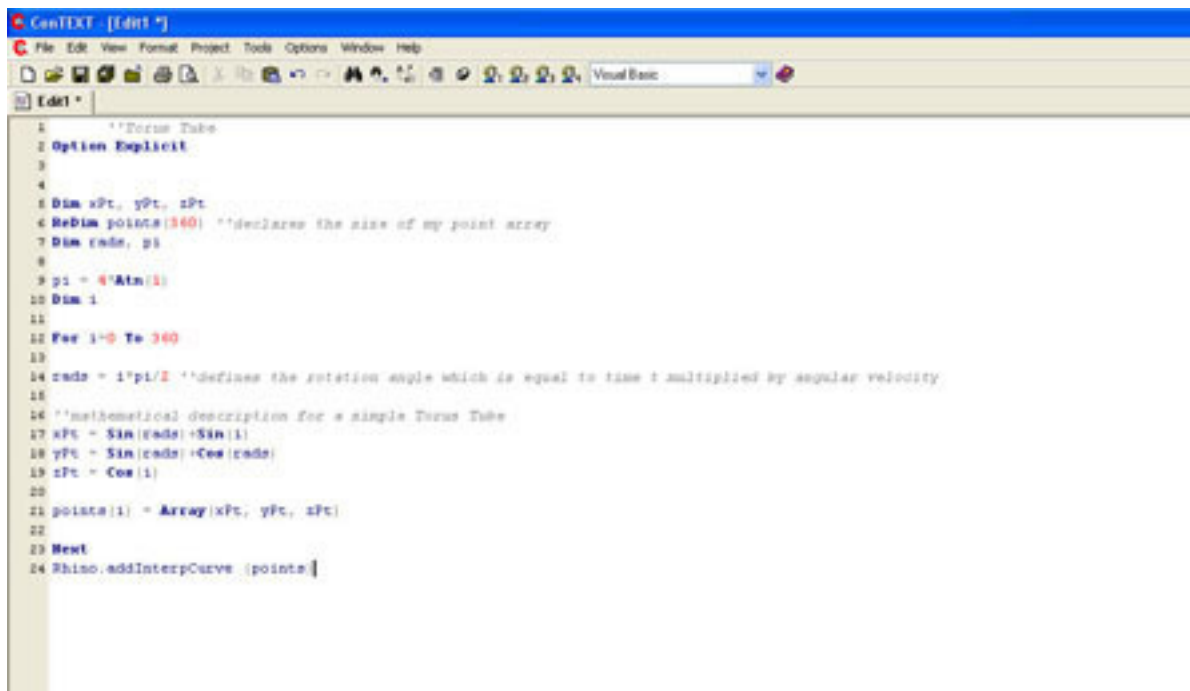
Monkey is a new script editor in Rhino4 which can be used to edit, run debug and compile scripts. It runs directly with Rhino4. It is also a DotNET plugin. You need Microsoft DotNET Framework 2.0 to run it.

### DotNET Framework 2.0

<http://www.microsoft.com/downloads/details.aspx?familyid=0856eacb-4362-4b0d-8edd-aab15c5e04f5&displaylang=en>

### Monkey

<http://en.wiki.mcneel.com/default.aspx/McNeel/MonkeyForRhino4.html>

A screenshot of the Context IDE window. The title bar reads 'Context [Edit \*]'. The menu bar includes 'File', 'Edit', 'View', 'Format', 'Project', 'Tools', 'Options', 'Window', and 'Help'. The toolbar contains various icons for file operations and execution. The main text area displays a Python script for creating a Torus Tube. The script includes comments in English and uses mathematical formulas for point calculation. The code is as follows:

```
1 '''Torus Tube
2 Option Explicit
3
4
5 Dim xPt, yPt, zPt
6 ReDim points(360) 'declares the size of my point array
7 Dim rads, pi
8
9 pi = 4*Atn(1)
10 Dim i
11
12 For i=0 To 360
13
14 rads = i*pi/180 'defines the rotation angle which is equal to time t multiplied by angular velocity
15
16 ''mathematical description for a single Torus Tube
17 xPt = Sin(rads)*Sin(i)
18 yPt = Sin(rads)*Cos(rads)
19 zPt = Cos(i)
20
21 points(i) = Array(xPt, yPt, zPt)
22
23 Next
24 Rhino.AddInterpCurve (points)
```



# Introduction: Rhino API (Application Programming Interface)

Help/Plug-ins/Rhinoscript

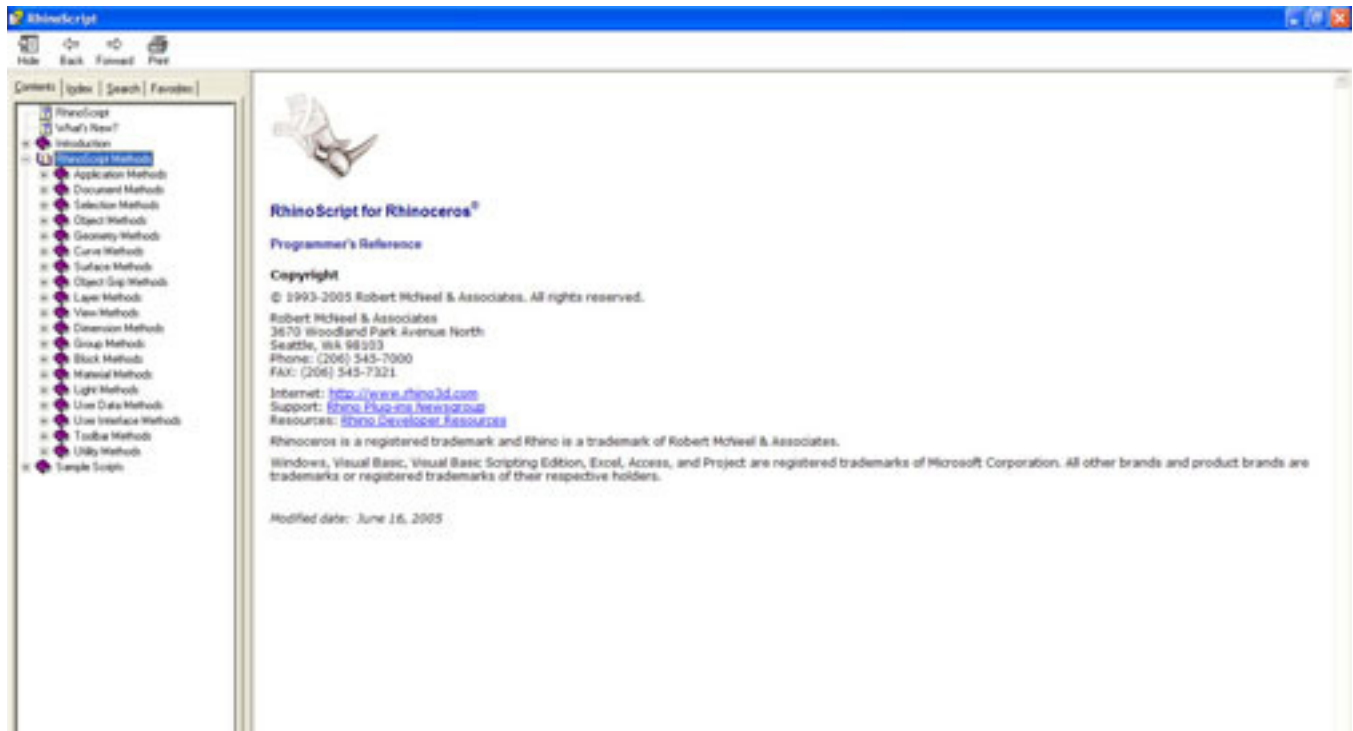
The Rhino API is the library of custom functions written for Rhino on top of Visual Basic Script.

Example:

```
Rhino.Print  
Rhino.GetPoint  
Rhino.AddPoint
```

To access these custom methods and functions in Python, you must import them. To do this, type the below at the top of your new script within the EditPythonScript window.

```
import rhinoscriptsyntax as rs
```



## Section 2: Primitives & Variables

## Primitives and Variables: Introduction to Primitives

Primitives are the “basic” data values. There are eight types of primitives:

**boolean** -- used for true and false values

**char** -- used for single characters (letters, etc.)

**byte, short, int, long** -- four different kinds of integer (whole number) values

**float, double** -- two different kinds of decimal numbers (numbers with a decimal point)

For the most part, we will work with boolean, char, int and double.

Note: Python is considered to be a high level programming language, meaning that it is fairly close to human language. It falls far above macros or machine code (low level code) and slightly higher than C or C#. One benefit of this is that we don't have to worry about allocating memory through the declaration of primitive types, which means that we don't need to declare variables!! So, this page and the next several pages are for your information only. We won't actually use these types in our code, but it's important to know the differences as your variables will still take on the data values.

## Primitives and Variables: int

The most important integer type is int

An int is a “whole” number (no decimal point)

Numbers occupy memory in the computer

Larger numeric types require more memory

byte: 1 byte   short: 2 bytes   int: 4 bytes   long: 8 bytes

An int can be between about two billion and negative two billion

If you just write a number, such as 25, Python assumes it is an int

Hence it is easier to work with int values than with the other integer types (byte, short, and long)

Use int in preference to other integer types, but remember with Python, we don't have to declare our variables!

Example:

```
num = 5
```

Example:

```
num = rs.GetReal("line length")  
line = rs.AddLine([9,7,2], [num,0,0])  
print "curve inserted with id", line
```

## Primitives and Variables: byte and short

A byte can be between -128 and 127

A short can be -32768 to 32767

Why these numbers?

These are “round numbers” in binary; for example,

0111 1111 1111 1111 is binary for 32767

1000 0000 0000 0000 is binary for -32768

The first bit is the sign bit: a 1 means it's a negative number

Use byte or short only when you know the numbers are all small

There are millions of numbers to remember

If you can avoid them, don't use bytes or shorts!

## **Primitives and Variables: long**

Long integers are for when two billion isn't large enough for your needs

A long can be as long as about 19 digits

A long occupies twice as much space as an int

Arithmetic on long values is slower

Use long only when you need really big numbers



## **Primitives and Variables: double**

A double represents a “real” number

Also sometimes called “floating point”

These are numbers with a decimal point

A double has about 15 digits of accuracy

If you just write a real number, such as 1.37, Python assumes it is a double

Hence it is easier to work with double values than with float values

Use double in preference to float

## **Primitives and Variables: float**

float is the other kind of “real,” or “floating point” number

float has about 8 digits of accuracy

Arithmetic with float is not faster

Use float only to save space when there are millions of numbers involved

Basically, you should only work with int and double!

## Primitives and Variables: string

Another important type is the String

A String is an Object, not a primitive type. Strings are used to store text by adding quotes around text or numbers.

A String is composed of zero or more chars

```
"this is a string"
```

Quotes will turn numbers into text.

```
x = "7"
```

```
n = "8"
```

```
print x+n = 78
```

We can convert a point to a string in Rhino like this

Example:

```
myPoint = rs.GetPoint ("Pick first Point")
```

## Primitives and Variables: comments

A comment is a note to any human reading the program; comments are ignored by GC and Rhino

A comment starts with `#` and goes to the end of the line in Python

A comment may be put after a statement (on the same line) to say something about that particular statement

A comment may be put on a line by itself, to say something about the following statements

Example:

```
#swap the values of x and y  
rad = x  
#save old value of x in radx  
radx = y  
#replace old value of x with y  
x = radx
```

## **Primitives and Variables: approximations**

Integers are precise, but real numbers are always approximate (inaccurate)

Computers always use the binary system internally

Many numbers that can be expressed precisely in decimal format, but cannot be represented precisely in binary

For example, the numbers 1.1, 1.2, 1.3, and 1.4 can only be approximated in binary

Two numbers that look the same may actually be subtly different. Python may round up or down depending on the numbers that we are working with, so, keep this in mind.

## Primitives and Variables: variables

Sometimes you know what a number is

You have 10 toes

There are 60 min. in an hour

PI is 3.141592653589793238

Numbers written like this are called *literals*

You can use literals any place in Python where you can use a number

Sometimes you need to use names instead:

`typeofSquare, myCircle, numRows`

Names like this are called *variables*

The value of a variable may change



## Primitives and Variables: variables

Before you use a variable, decide what type it is: int, double, char, ... and what value it has. Remember, you don't need to declare it (Python will automatically do this for you based on the value of the variable).

There are two reasons for this:

1. Different types require different amounts of space
2. So Python can prevent you from doing something meaningless

You might assign an initial value to your variable, or compute a value, or read a value in

## Primitives and Variables: using variables

You declare variables like this:

```
myRad = 3.6  
x = 3
```

When you define a variable, Python automatically declares its type and finds space for it

The amount of space Python needs to find depends on the type of the variable

Think of a variable as a special “box,” with a shape designed to hold a value of a particular type

An int variable is four bytes long and there’s a special place for the sign bit

A float variable is also four bytes long, but the bits are used differently--some are used to tell where the decimal point goes

## Primitives and Variables: giving value to variables

A variable is just a name for some value

You have to supply the actual value somehow

You can assign values like this:

```
numPoints = 100
```

Every variable has a type of value that it can hold

For example,

`name` might be a variable that holds a String (sequence of characters)

`myCircle` might be a variable that holds an integer value

`isFloppy` might be a variable that holds a boolean (true or false) value

## Primitives and Variables: initializing variables

You can give a variable an initial value when you declare it:

```
numPoints = 100
```

```
myRad = 3.5
```

```
numPts = 50
```

You can change the value of a variable many times:

```
numPoints = 200
```

```
myRad = myRad + 20.0
```

```
numPts = 23
```

## Primitives and Variables: arithmetic

Primitives have operations defined for them

int and double have many defined operations, including

- + for addition
- for subtraction
- \* for multiplication
- / for division

## Primitives and Variables: order of operations

Operations with higher precedence are done before operations with lower precedence

Multiplication and division have higher precedence than addition and subtraction:

$$2 + 3 * 4 \text{ is } 14, \text{ not } 20$$

Operations of equal precedence are done left to right:

$$10 - 5 - 1 \text{ is } 4, \text{ not } 6$$

Operations inside parentheses are done first

$$(2 + 3) * 4 \text{ is } 20$$

Parentheses are done from the inside out

$$24 / (3 * (10 - 6)) \text{ is } 2$$

Parentheses can be used where not needed

$$2 + (3 * 4) \text{ is the same as } 2 + 3 * 4$$

[ ] and { } cannot be used as parentheses! You will learn why later.



## Primitives and Variables: assignment statements

An assignment statement has the form:

`variable = expression`

Examples:

`myRad = 3.5`

(The expression can be as simple as a single literal or variable)

`area = pi * radius * radius`

`numPoints = 50`

`numPoints = numPoints + 10`

This means “add 10 to the value in numPoints”

## Primitives and Variables: parentheses, braces and brackets

( ) are parentheses

Signifies a Function; ex. Series (start, finish, increment)

(0,1, 0.2) returns {0,0.2,0.4,0.6,0.8,1.0}

{ } are braces

Signifies a Dictionary

[ ] are brackets

Signifies an index to a member within an array. It could be a tuple or a list.

ex. [3] for collection {0,0.2,0.4,0.6,0.8,1.0} returns 0.6

note: arrays are zero-based

We will come back to arrays shortly.

## **Primitives and Variables: new vocabulary**

**primitive:** one of the 8 basic kinds of values

**literal:** an actual specified value, such as 42

**variable:** the name of a “box” that can hold a value

**type:** a kind of value that a literal has or that a variable can hold

**declare:** to specify the type of a variable

**operation:** a way of computing a new value from other values

**precedence:** which operations to perform first (which operations precede which other operations)

**assignment statement:** a statement that associates a value with a name

**initialize:** to assign a “starting” value

## Section 3: Methods

## Methods: Assignment Statements-review

Values can be assigned to variables by assignment statements

The syntax is: `variable = expression;`

The expression must be of the same type as the variable

The expression may be a simple value or it may involve computation

Examples:

```
name = "Jenny"  
count = count + 50  
area = (4.0 / 3.0) * 3.1416 * radius  
isFloppy = false
```

When a variable is assigned a value, the old value is discarded and totally forgotten

## Methods: Functions

A function is a named group of declarations and statements. Think of it as a mini script or algorithm that you can call at any point within your larger script. It's code where you need it!

```
def day():  
    day = "Thursday"  
    print ("It is " + day + "!")  
day()
```

We "call," or "invoke" a function by naming it in a statement:

```
day( )
```

This should print out:

```
It is Thursday!
```

**If you don't "call" the function, it will sit there and you won't get an error message, but it won't run!**



## Methods: class organizations

A class may contain data declarations and methods (and constructors, which are like methods), but not statements

A method may contain (temporary) data declarations and statements

class Example

```
    someMethod( ):
        yetAnotherVariable
        yetAnotherVariable = 5 #statement inside method is OK
```

## Section 4: Control Structures

## **Control Structures: simple control structures**

You can't do very much if your program consists of just a list of commands to be done in order

The program cannot choose whether or not to perform a command

The program cannot perform the same command more than once

Such programs are extremely limited!

Control structures allow a program to base its behavior on the values of variables

## Control Structures: boolean

boolean is one of the eight primitive types

booleans are used to make yes/no decisions

All control structures use booleans

There are exactly two boolean values, true ("yes") and false ("no")

boolean, true, and false are all lowercase

booleans are named after George Boole, the founder of Boolean logic

boolean variables are declared like any other kind of variable, but its value is true or false

```
taskCompleted = false
```

boolean values can be assigned to boolean variables:

```
taskCompleted = true
```

## Control Structures: boolean

Arithmetic comparisons result in a boolean value of true or false

There are six comparison operators:

<	less than
<=	less than or equals
>	greater than
>=	greater than or equals
==	equals
!=	not equals

There are three boolean operators:

&&	"and"--true only if both operands are true
	"or"--true if either operand is true
!	"not"--reverses the truth value of its one operand

Example:

```
(x > 0) && !(x > 99)
```

"x is greater than zero and is not greater than 99"

## Control Structures: string concatenation

You can concatenate (join together) Strings with the + operator

In fact, you can concatenate any value with a String and that value will automatically be turned into a String

Be careful, because + also still means addition

```
x = 3  
print ("I have "+x+" dogs.")
```

The above prints *I have 3 dogs.*

## Control Structures: if statements

An if statement lets you choose whether or not to execute one statement, based on a boolean condition

Syntax: `if (boolean_condition) statement;`

Example:

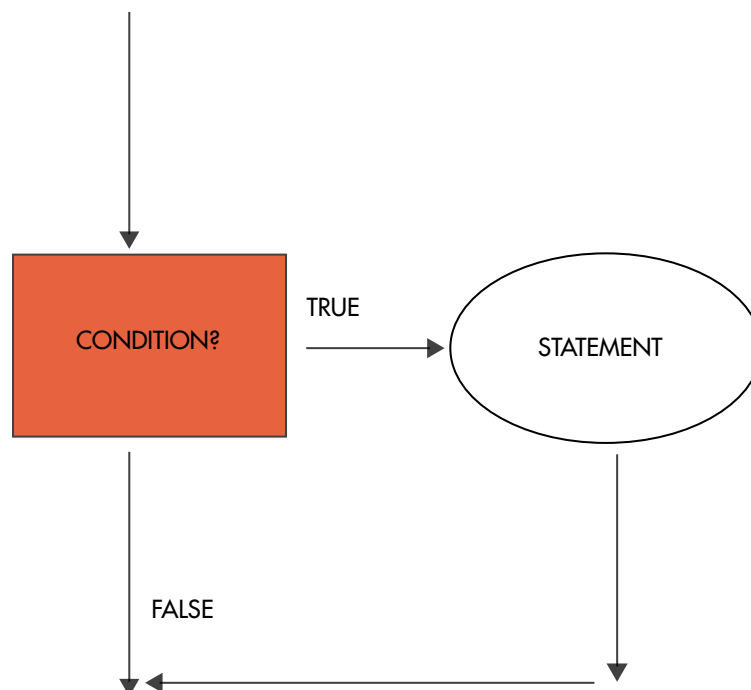
```
if (x < 100):  
  x = x + 1 #adds 1 to x, but only if x is less than 100
```

An if statement may have an optional else part, to be executed if the boolean condition is false

Syntax: `if (boolean_condition) statement; else statement;`

Example:

```
if (area > 10):  
  
    point1 = {4,4.5,8}  
  
elif (area = 3):
```



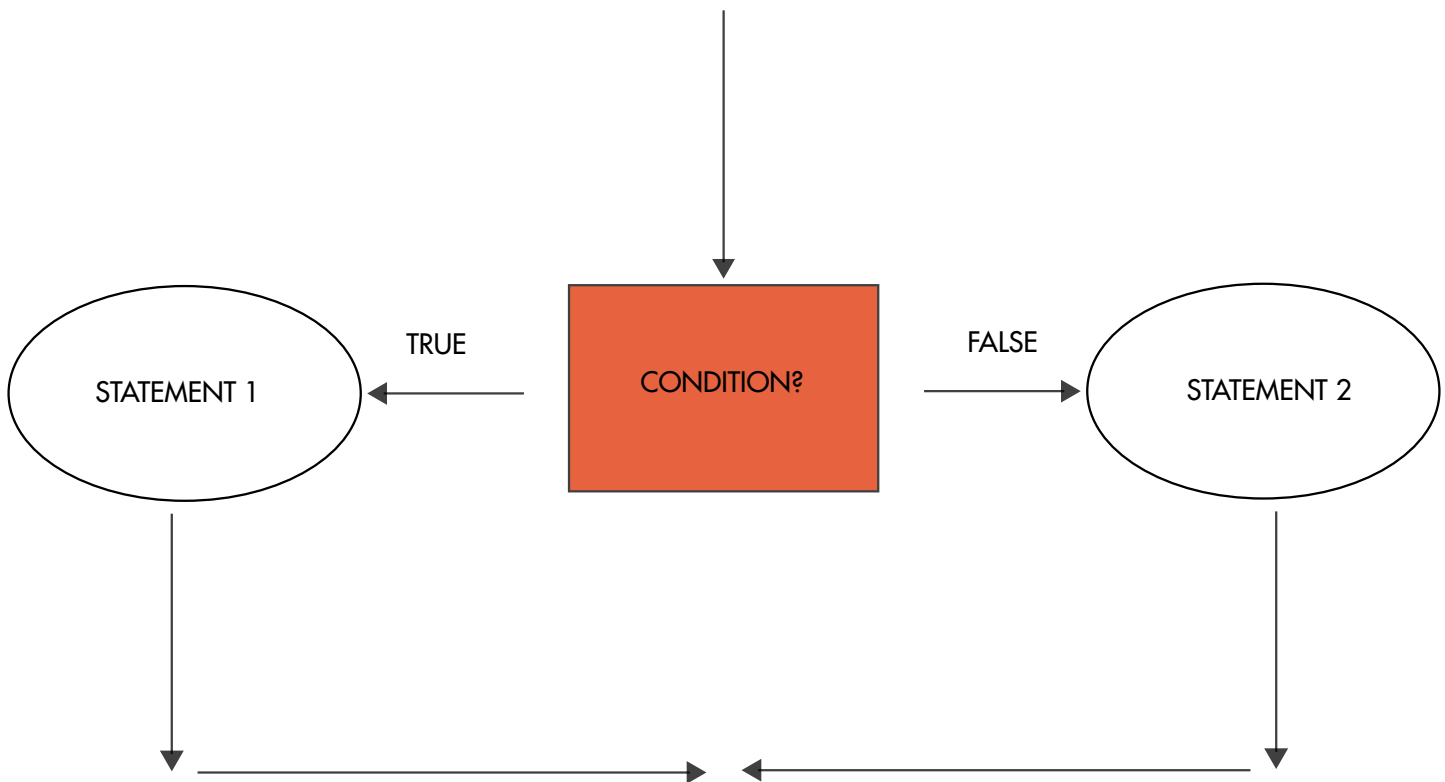
## Control Structures: if else statements

The if-else statement chooses which of two statements to execute

The if-else statement has the form:

`if (condition) statement-to-execute-if-true; else statement-to-execute-if-false`

Either statement (or both) may be a compound statement





## Control Structures: while loop

This is the form of the while loop:

`while (condition) statement`

If the condition is true, the statement is executed, then the whole thing is done again

The statement is executed repeatedly until the condition becomes false

If the condition starts out false, the statement is never executed at all

A while loop will execute the enclosed statement as long as a boolean condition remains true

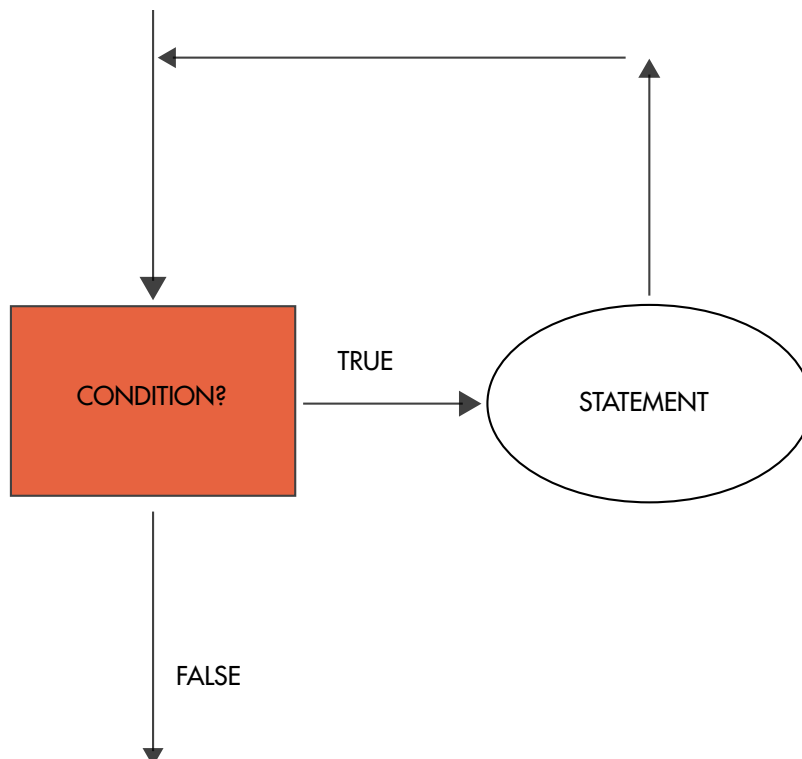
Syntax: `while (boolean_condition) statement`

Example:

`while (list.Count < 20):`

`Add(list.Count = 50)`

Danger: If the condition never becomes false, the loop never exits, and the program never stops



## Control Structures: method calls

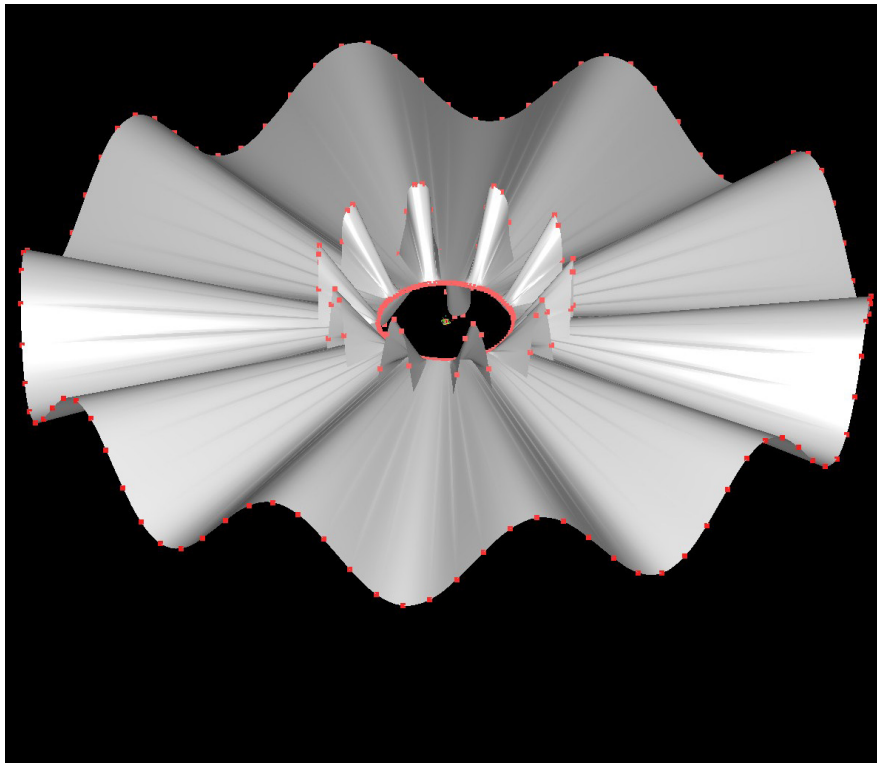
A method call is a request to an object to do something, or to compute a value

A method call may be used as a statement

Some method calls return a value, and those may be used as part of an expression

Example:

```
x = r * math.sin(i*360/(nP-1))
```



## Control Structures: a Complete Python Script

```
import rhinoscriptsyntax as rs
import math
```

```
def (typeofFlange):
```

```
    #Flange variables
    points = [] #collection
    nP = 100
    nSin = 10
```

```
    #conditional assignment
    if (typeofFlange == "floppy"):
```

```
        t = 3
        r=15
        amp=10 #conditional variables
```

```
    if (typeofFlange == "extrafloppy"):
```

```
        t = 3
        r= 30
        amp=20
```

```
    #iteration to calculate points
    for i in rs.frange(0, 100, 1):
```

```
        x = r * Sin(i*360/(nP-1))
        y = r * Cos(i*360/(nP-1))
        z = amp*Sin(i*nSin*360/(nP-1))
```

```
    #Feature
    point = rs.AddPoint([x,y,z])
    points.append(point)
```

```
rs.AddInterpCurve(points)
```

```
flange("floppy")
```

## Control Structures: a complete Python Script

### #3,8 Torus knot

```
import rhinoscriptsyntax as rs
import math
```

```
points = [] #declares the size of my point array
pi = math.pi
```

```
rs.EnableRedraw(False)
```

```
for t in rs.frange(0,360,1):
```

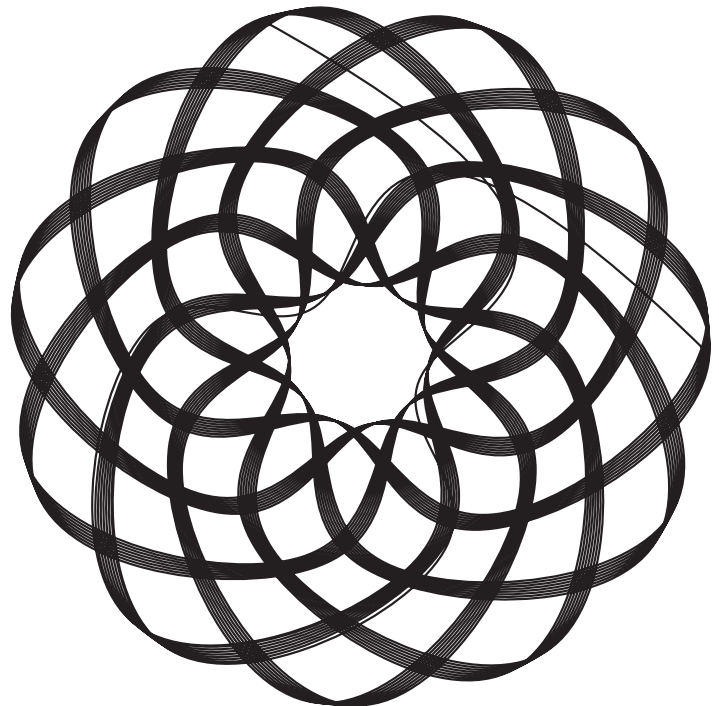
```
    rads = t*pi/2 #defines the rotation angle which is equal to time t multiplied by angular
                  velocity
```

```
    #mathematical description of a 3,8 Torus knot
```

```
    x = (3+ 2*math.cos(rads))*math.cos(t)
    y = (3+2*math.cos(rads))*math.sin(t)
    z = 2*math.sin(rads)
```

```
    n = rs.AddPoint([x, y, z])
    points.append(n)
```

```
rs.AddInterpCurve(points)
rs.EnableRedraw(True)
```



## Section 5: Arrays: Tuples, Lists, Dictionaries

## Arrays: a problem with simple variables

One variable holds one value

The value may change over time, but at any given time, a variable holds a single value

If you want to keep track of many values, you need many variables

All of these variables need to have names

What if you need to keep track of hundreds or thousands of values?

An **array** lets you associate one name with a fixed (but possibly large) number of values

All values must have the same type

The values are distinguished by a **numerical index between 0 and array size minus 1**

A Point is an Array: x, y, z

myArray[0]=x

myArray[1]=y

myArray[2]=z

	0	1	2	3	4	5	6	7	8	9
myArray	12	43	6	83	14	-57	109	12	0	6

## Arrays: indexing into arrays

To reference a single array element, use

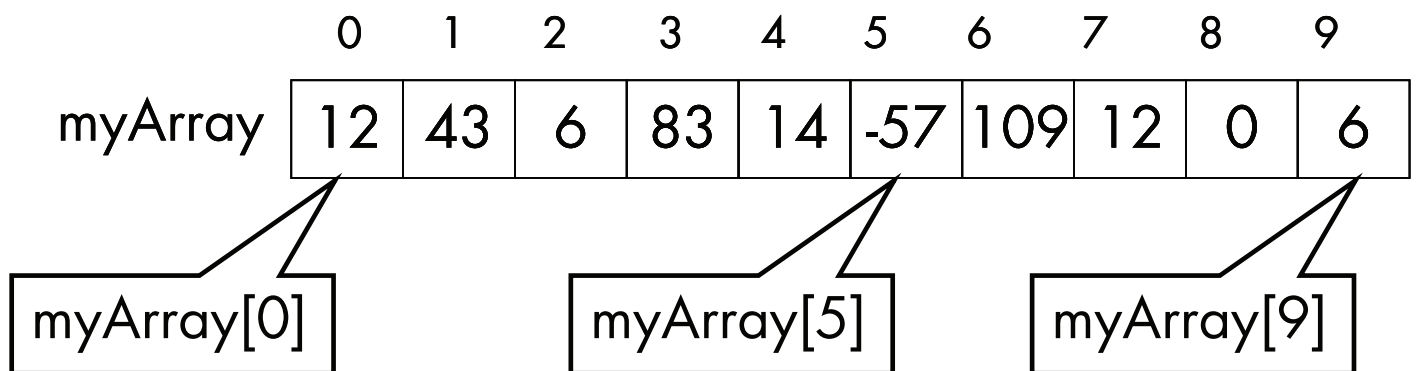
`array-name [ index ]`

Indexed elements can be used just like simple variables

You can access their values

You can modify their values

An array index is sometimes called a subscript



## Arrays: using array elements

Examples:

```
x = myArray[1];      #sets x to 43  
myArray[4] = 99;     #replaces 14 with 99  
m = 5; y = myArray[m]; #sets y to -57  
z = myArray[myArray[9]]; #sets z to 109
```

	0	1	2	3	4	5	6	7	8	9
myArray	12	43	6	83	14	-57	109	12	0	6



## Arrays: array values

An array may hold any type of value

All values in an array must be the same type

For example, you can have:

an array of integers (ints)

an array of Strings

an array of Dogs

In this case, all the elements are Dogs; but they may belong to different subclasses of Dogs

For example, if you have a class Bulldog extends Dogs, then you can put Bulldog in your array of Dogs

This is because a Bulldog is a Dog

You can even have arrays of arrays, for example, an array of arrays of int

# Arrays & Mutability: Tuples, Lists, Dictionaries

(adapted from RhinoPythonPrimer, p.24, section 4.5)

1. **Tuples** are not mutable. The values in the array are static or fixed.

```
x = (5,1)
y = x
x = (3,4)
print(y)
```

The answer is (5,1)

2. **Lists** are mutable, but be careful.

```
x = [9,1]
y = x
x.append(3)
print y
```

The result = (9,1,3)

```
x = [9,1]
y = x[:]
x.append(3)
print y
```

The result = (9,1)

3. **Dictionaries** are also mutable.

```
x = {1:'t',2:'r'}
y = x
x[3] = 'n'
print y
```

The result = {1:'t',2:'r',3:''}     -OR-

```
x = {1:'t',2:'r'}
y = x.copy()
x[3] = 'n'
```

## Arrays: Tuples, Lists, Dictionaries cont-

### Tuple

```
t = 12345, 5421, 'hello!' # Creating a Tuple with a variable name t
print(t[0]) # print the first value of the Tuple t
# This returns 12345 - the first value inside the Tuple
print(t)
# This returns (12345, 54321, 'hello!') - all of the values within the Tuple
```

### List

```
myList = [] #This creates an empty list with the variable name myList
myList.append(5)
myList.append(6)
print myList[0]
# This returns 5 - the first element (0th item) in the list
```

## Arrays: array assignment

You can declare more than one variable in the same declaration:

```
int a[ ], b, c[ ], d; #notice position of brackets  
a and c are int arrays  
b and d are just ints
```

Another syntax:

```
int [ ] a, b, c, d; #notice position of brackets  
a, b, c and d are int arrays
```

When the brackets come before the first variable, they apply to all variables in the list

Array assignment is an object assignment

Object assignment does not copy values

```
Flange p1; Flange p2;  
p1 = new Flange("Blue");  
p2 = p1; #p1 and p2 refer to the same Flange
```

Array assignment does not copy values

```
int[ ] a1; int[ ] a2;  
a1 = new int[10];  
a2 = a1; #a1 and a2 refer to the same array
```

## Arrays: an array's size is not a part of its type

When you declare an array, you declare its type; you must not specify its size

Example:

```
String names = [ ]
```

When you define the array, you allocate space; you must specify its size

Example:

```
names = new String[50]
```

This is true even when the two are combined

Example:

```
String names[ ] = new String[50]
```

When you assign an array value to an array variable, the types must be compatible

The following is not legal:

```
double[ ] dub = new int[10] #illegal
```

The following is legal:

```
int[ ] myArray = new int[10]
```

...and later in the program,

```
myArray = new int[500] #legal
```

Legal because array size is not part of its type

## Arrays: Redim vs Redim Preserve in RhinoScript (not Python)

ReDim : is for declaring the size of the array. It allows us to create dynamic arrays.

ReDim Preserve : is for redeclaring the size of an array without losing the data already stored in the array.

Example1:

```
Dim myArray
ReDim myArray(2) "myArray length is equal to the last index number + 1, so, 3
myArray(0) = "a"
myArray(1) = "b"
myArray(2) = "c"
```

```
Redim myArray(3)
"this will clear the strings previously stored in myArray
"therefor myArray(0) is now empty
```

Example2:

```
Dim myArray
ReDim myArray(2)
myArray(0) = "a"
myArray(1) = "b"
myArray(2) = "c"
```

```
Redim Preserve myArray(3)
"this does not clear the strings previously stored in myArray
myArray(3) = "d"
```

These are both needed when your going to fill individual positions within an array.

You don't need redim in this example. This is a static array.

```
Dim myArray2
myArray2 = Array("a", "b", "c")
```

## Arrays: Array Use Examples

Suppose you want to find the largest value in an array of scores of 20 integers:

```
largestScore = 0  
  
for i in rs.frange(0, 20, 1):  
    if (scores[i] > largestScore):  
        largestScore = scores[i]
```

Suppose you want to find the largest value in an array scores and the location in which you found it:

```
largestScore = scores[0]  
index = 0  
for i in rs.frange(0,20,1):  
    if (scores[i] > largestScore):  
        largestScore = scores[i]  
        index = i
```

## Arrays: Initializing Arrays

There is a special syntax for giving initial values to the elements of arrays

This syntax can be used in place of `new type[size]`

It can only be used in an array declaration

The syntax is: `{ value, value, ..., value }`

Examples:

```
primes = { 2, 3, 5, 7, 11, 13, 19 }  
String languages = { "C", "C++" }
```

The elements of an array can be arrays

Once again, there is a special syntax

Declaration: `pt[i][j] = new Point()` or `Point[i][j]`

Definition: `Point = [10][15]`

Combined: `int[i][j] point = new int[10][15]`

The first index (10) is usually called the **row** index; the second index (15) is the **column** index

An array like this is called a two-dimensional array



## Arrays: Arrays of Arrays or 2D Arrays

```
int[ ][ ] table = new int[3][2] or,  
int[ ][ ] table = { {1, 2}, {3, 6}, {7, 8} }
```

For example,

table[1][1] contains 6  
table[2][1] contains 8, and  
table[1][2] is "array out of bounds"

	0	1
0	1	2
1	3	6
2	7	8

Example:

A curve is a collection of points, it is a 2D array.

```
points = []
```

```
points(0) = array (4,5,7)
```

```
points(1) = array (9,2,4)
```

```
points(2) = array (1,2,9)
```

```
points(3) = array (5,2,8)
```

```
rs.AddPoint(points)
```

**That's It for Now!**