ANN MCNAMARA

AN INTRODUCTION TO PYTHON SCRIPTING IN AUTODESK MAYA

SIGGRAPH '21 Educator's Forum, August 09-13, 2021, Virtual Event, USA
ACM 978-1-4503-8363-9/21/08.
10.1145/3450549.3464409

Contents

Listings

1
Introduction

1.1 Motivation

Coding is a highly desirable skill. Learning to code can boost problem-solving and logic skills. Perhaps more importantly, coding empowers automation. Scripts can handle mundane and repetitive tasks in an efficient and precise manner. This course will offer a gentle introduction to the Python programming language using a hands-on interactive format. It is intended to be a foundational course. The goal is to cover the basics but provide enough information for attendees to build on later. We will take a "creative computing" approach by applying Python scripts in the Autodesk Maya environment. Python scripting can automate many tasks in Maya, from running simple commands to developing plug-ins. Examples will be presented first as concepts in pure Python, then subsequently applied in Maya. The hope is that connecting Python language foundations to visual tasks will motivate reluctant programmers to experiment and explore the possibilities. Attendees will learn how to automate simple tasks using the magic of scripting. The course will cover data types and variables, lists, conditional statements, iteration, functions, and Object-Oriented Programming. By the end of the course, attendees should walk away with a fundamental understanding of the Python language, Maya commands the ability to write basic scripts for Maya. Hopefully, they will have the tools, confidence, and initiative to explore more advanced scripts independently. Attendees should have Autodesk Maya, Python, and Visual Studio Code pre-loaded on their devices if they intend to follow along.

1.2 Focus Areas

This course will focus on the fundamentals of the Python programming language, but through the lens of scripting in Autodesk Maya. Examples will be presented first as a concept in pure Python, then subsequently applied in Maya. The hope is that connecting Python language foundations to visual tasks will motivated reluctant programmers to experiment and explore the possibilities. We will spend about 20 to 25 minutes on each topic to allow for breaks and questions.

1. Variables & Data Types
2. Lists
3. Conditional Statements
4. Loops

5. Functions
6. Object Oriented Programming

1.3 *A note on code listings*

Each code listing, as shown in Listing 1.1, represents a python file that is also included with these Course Notes. All the files can be downloaded from the authors web page, linked here. Comments appear in green and are included to provide documentation for the human reader. Python ignores any text after a #.

```
# A first example in python
# print statements will display the
# result of evaluating the expressions
# everything in green
# after a # is a comment and is for the
# human reader and ignored in the program

print 10            # output 10
print 10 + 10       # output 20
print 10 * 10       # output 100
print 5  * 10       # output 50
print 40/10         # output 4
print 42/10         # output 4 (only whole number portion)

print 'Hello'       # output Hello
```

Listing 1.1: python code example (1.1_firstExample.py)

1.4 *Setting Up Python & Autodesk Maya*

There are some steps that are necessary to set up communication between Python and Autodesk Maya.

1. Download and install Autodesk Maya 2020. If you are a student (or faculty member) can select the educational version by providing your student id. (or you can download a 30-day free trial if you are not a student or while you wait for your student id to be approved.)
   - Student Version
   - 30 Day Trial Version
2. Download and install Visual Studio Code. Visual Studio Code is a text editor that allows extensive and specialized extensions/plugins that help speed up your workflow. Youcan think of VSC as Notepad/-TextEdit but with extendable functionality.
   - Visual Studio Code
3. Install Maya Code (click the green install button and open in Visual Studio Code). This extension allows you to see the Maya hierarchy within Visual Studio code, colors the words of your script for readability (syntax highlighting), and provides intellisense (smart autocomplete) for MEL scripting.
   - Maya Code

4. Put the following two lines of code in a file called **userSetup.mel**. Maya will execute this file every time it opens - the code basically tells VSCode how to "talk" to Autodesk Maya. You can create the file in notepad or any text editor you like (even VSCode)

```
commandPort -name "localhost:5678" -sourceType "python" -echoOutput;
commandPort -name "localhost:7001" -sourceType "mel" -echoOutput;
```

5. Put the following line of code in a file called **userSetup.py**. Maya will automatically execute this file on startup when Maya opens. We are just going to save ourselves from typing this line of code over and over.

```
import maya.cmds as cmds
```

6. Be careful with the file names. They need to be *exactly* as listed in bold

7. Place both **userSetup.mel** and **userSetup.py** in the following folder:

   - MAC:
     `/Users/username/Library/Preferences/Autodesk/maya/2020/scripts`
     For example on my mac my folder is called
     `/Users/annmcnamara/Library/Preferences/Autodesk/maya/2020/scripts`
   - PC:
     `..\MyDocuments\maya\<Version>\scripts`
     The version is 2020 as that is what you just downloaded so the path should be
     `..\MyDocuments\maya\2020\scripts`

8. Once you have the files correctly in the directory open Maya. Open Visual Studio Code if you don't have it open and create a new python script (File->New). Type in the following two lines of code and save the file as **test.py** (File->Save As) - make sure you save it as test.py, the .py extension at the end tells VSCode that it is a python file.

```
cmds.polyCube(name='myCube')
cmds.polySphere(name="mySphere")
```

9. Now the moment of truth, right-click anywhere in your script test.py window in Visual Studio Code and you should see an option called Maya: Send Python Code to Maya, or you can use use the hotkey combo: shift plus alt plus M, since you will be doing this all throughout it might be faster. The very first time you do this you may need to scroll down to the bottom and open the command palette. if you start typing Maya at the > then select Maya: Send Python Code to Maya, it should work thereafter.

10. If you have followed all the instructions carefully you should see a sphere and a cube in your Maya window with the names mySphere and myCube, as shown in Figure 1.1. When first drawn the sphere will be on top of the cube so you wont be able to see the cube unless you move the sphere.

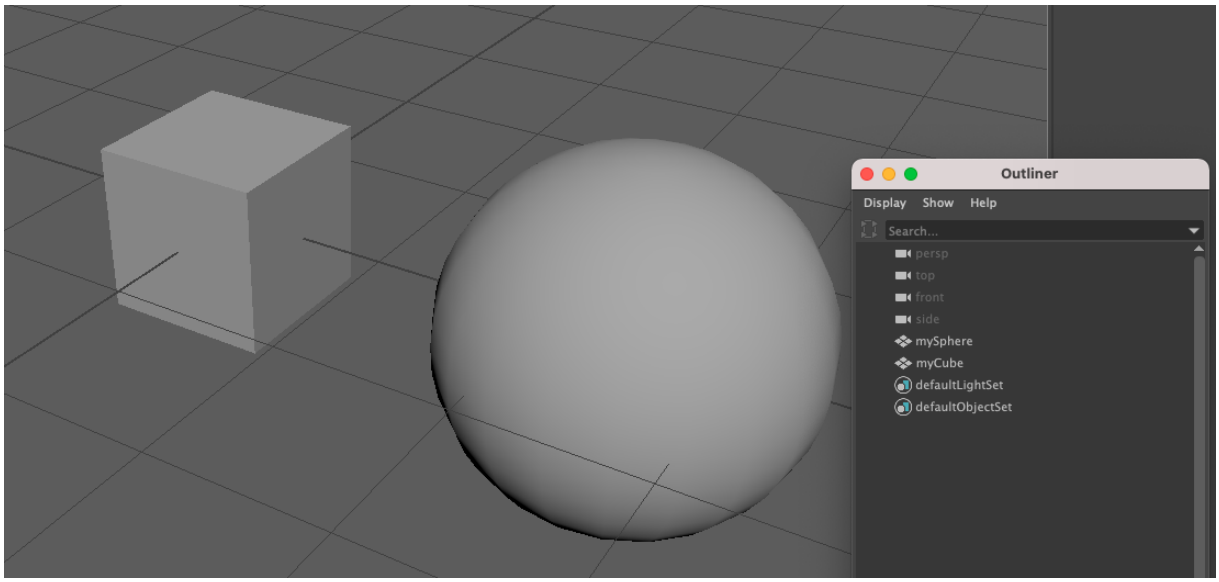

Figure 1.1: A sphere and a cube using python for Maya.

2
Data Types & Variables

There are many different types of data we wish to create, use and manipulate. Python provides may types and can guess what type we intend. This means we don't have to tell Python which type we want - it knows from the value we provide.

The main data types in Python are:

- int (a whole number)
- float (a number with a decimal point)
- str (string of characters)
- tuple (a collection of objects)
- list (an ordered collection of objects)
- dict (key value pairs)

In this course we will only cover *int, float, str, and lists*. Lists are so prolific in Maya we will devote a entire section to covering lists.

*2.1 The **int** data type (integers, whole numbers)*

The data type to hold integers is called *int* for short. Integers are whole numbers, like 2, 4, 6, 8, 10. Let's work through the code in listing 2.1 to see what the int data type can do. In listing 2.1, the number 10 is typed several times. We don't want to type 10 every time. Furthermore, what if we decided we wanted to use a different value for our calculations. Luckily we can store values in *variables*.

Variables are just names that store values so if tell python that

```
theNumber = 10    # this creates a variable called theNumber to store the value 10
```

then I can just use the *variable name* theNumber instead of typing out the number 10. This is kind of handy since if for some reason I decide later that theNumber should be some value other than 10 I can just change it one time! In Python we create variables as follows

```
variableName = value          #This is an assignment
```

We don't need to tell Python it is an integer - Python knows because we set the value to a whole number. Also notice how I capitalized the first letter of the second word. This is called *camel case* and is a convention used when writing Python code. Examples of integer operations are shown in Listing 2.1.

```
# int - int is short for integer
# an int is a whole number.
print 10        # 10 is a whole number
# We can do arithmetic with integers
print 10 + 10   # + for addition
print 10 - 10   # - for subtraction
print 10 * 10   # * for multiply
print 5  * 10   # doesn't have to just be the number 10
print 40 / 10   # / for divide

print 43 % 10 # See what \% did
# it is called modulo and gives us the remainder!
# can be very useful for cycling - we will see this later in the course

# We can check what type python thinks we are storing using the type function

print (type (10))  # tells us its an int

theNumber = 10
print("THE INTEGER is ", theNumber)
print theNumber
print theNumber + theNumber
print theNumber * theNumber
print 5  * theNumber
print 40 / theNumber


# now lets change the value of theNumber
theNumber = theNumber * 2
print("THE INTEGER is ", theNumber)
print theNumber
print theNumber + theNumber
print theNumber * theNumber
print 5  * theNumber
print 40/theNumber
```

Listing 2.1: The int data type (2.1_intDatatype.py)

2.2 *The float data type (floating point numbers)*

The data type to hold numbers with decimal points, floating point numbers, is called *float* for short. floats are whole numbers, like 2.45, 4.782, 6.1, 8.9862, 10.4. Let's work through the code in listing 2.2 to see what the float data type can do. In listing 2.2, we create a floating point number and apply some arithmetic operators.

Note we can add,subtract, multiply, divide and modulo a combination of floats and ints. However, the results may not be as expected (see listing 2.2.

```
# float - float is short for floating point number
# A float is a number with a decimal point.
print 3.14        # 3.14 is a decimal number
# We can do arithmetic with floating point numbers (floats)
print 2.50 + 2.50  # + for addition
print 3.14 - 1.0   # - for subtraction
```

```
print 2.2  * 3.3   # * for multiply

# Declare a variable called theFloat
# to hold the value 3.2
theFloat = 3.2


# We can check what type python thinks we are storing using the type function
print type(theFloat)  # <type 'float'>

print("THE FLOAT is ", theFloat)
print theFloat
print theFloat + theFloat
print theFloat * theFloat
print 5  * theFloat
print 40 / theFloat

# now lets change the value of theFloat
theFloat = theFloat * 2
print("THE FLOAT is ", theFloat)
print theFloat
print theFloat + theFloat
print theFloat * theFloat
print 5  * theFloat
print 40/theFloat   # result is of type float
print .25/5.0

print .25/4
print 4/.25

print 2.5 % .3 # See what % did
#  modulo and gives us the remainder
```

Listing 2.2: The float data type (2.2_floatDatatype.py)

2.3 *The str data type (strings)*

The data type to hold words, *strings of characters* is called a string, *str* for short. Strings are just words, like 'hello','siggraph', 'Bl1nk1'. Let's work through the code in Listing 2.3 to see what the string data type can do. In listing 2.3, we create a string object and apply some string operators.

```
## Strings - strings of characters
greeting = 'Hello'
print greeting
myGreeting = "Mirning"  #typo is deberate here -we will fix it in a minute
print myGreeting
print myGreeting + ' ' + greeting


print dir(myGreeting) # these are all the operations I can do on strings
print(help(myGreeting.replace)) # replace sounds promising

print myGreeting  # orignial with typo

myGreeting.replace('ir', 'or') # I will use replace to fix my typo
print myGreeting  #let's check it worked
# !!
# nothing changed!
# this is because strings are IMMUTABLE
```

```
# we  cant MUTATE them once we create them
# we can however create a new string with the values we want

newMyGreeting = myGreeting.replace('ir', 'or')
print myGreeting
print newMyGreeting

# What about arithmetic
print newMyGreeting + 3.5  # this will give you an error
# we also say it will throw an error
# the error is cannot concatenate 'str' and 'float' objects
# Python doesnt know how to add an integer to a string
# which kind of makes sense

# but it does know how to multiply them!
print newMyGreeting * 3
```

Listing 2.3: The string data type (2.3_stringDatatype.py)

2.4 *Summary*

Data types are the building blocks of any programming language. Data types are stored in variables. Python variables can store different data types. The type of a Python variable depends on the value that is stored in it. As we have seen in the examples so far, we can do different things with different types of data.

3
Variables & Data Types in Maya

3.1 maya.cmds

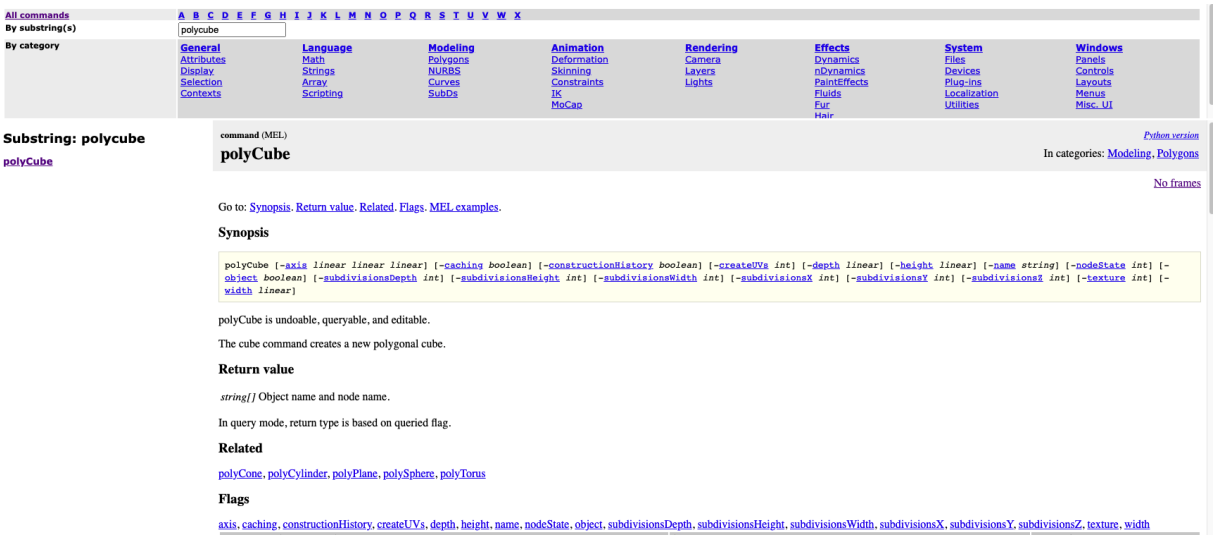

Figure 3.1: Maya cmds

Maya provides the **cmds** API to allow us to program Maya functions. At the beginning of each Python program that uses Maya cmds, we must first **import the maya.cmds API**.

```
import maya.cmds as cmds
```

Here we could import maya.cmds as any name we would like, but a popular convention is to use cmds. I have seen it written as

```
import maya.cmds as mc
```

where mc is short for maya.cmds, again you can call it anything you like. For this course we will use the first way and then precede our calls to the API with cmds. An example to to create a cube would be:

```
cmds.polyCube()
```

This line of code will create a polygon cube in Maya with all the default values. If we want to set specific values at creation time we can look to the Maya command reference, Figure 3.1 If we search for **polyCube**

the documentation will list all the *flags*, or attributes, available to set. Each flag has a long name and an abbreviation. The examples in this course use the long names for completeness but once you get used to the flags it's very handy to be able to use the abbreviations.

The polyCube() command has the following flags, axis, caching, constructionHistory, createUVs, depth, height, name, nodeState, object, subdivisionsDepth, subdivisionsHeight, subdivisionsWidth, subdivisionsX, subdivisionsY, subdivisionsZ, texture, width. It is not necessary to remember all of these, or to use them every time you create a cube. It is good to know what is available. An example to create a cube called "box" with a width of three and height of two would be as follows. We are going to store this in the variable called aBox. This is separated from the name we set to "box" using the **-name** flag. that name, "box" will be used inside Maya to name the object.

```
cmds.polyCube(name='box', width=3, height=2))
```

Once you are familiar with the long names you can use the short names. The following line of code is exactly the same as the line above, but the abbreviated versions of the flag names are used.

```
cmds.polyCube(n='box', w=3, h=2))
```

The documentation typically includes an example of how to use the Maya cmd. This example is taken directly from the documentation and illustrates how to create a cube and query the width attribute. The documentation does use the abbreviated version of the flags.

```
import maya.cmds as cmds

cmds.polyCube( sx=10, sy=15, sz=5, h=20)

#result is a 20 units height rectangular box

#with 10 subdivisions along X, 15 along Y and 20 along Z.

cmds.polyCube( sx=5, sy=5, sz=5 )

#result has 5 subdivisions along all directions, default size

# query the width of a cube

w = cmds.polyCube( 'polyCube1', q=True, w=True )
```

For completeness, the above code is repeated but with the full names for each flag.

```
import maya.cmds as cmds

cmds.polyCube( subdivisionsX=10, subdivisionsY=15, subdivisionsZ=5, height=20)

#result is a 20 units height rectangular box

#with 10 subdivisions along X, 15 along Y and 20 along Z.

cmds.polyCube(subdivisionsX=5,subdivisionsY=5, subdivisionsZ=5 )
```

```
#result has 5 subdivisions along all directions, default size

# query the width of a cube

cubeWidth = cmds.polyCube( 'polyCube1', query=True, width=True )
```

Both code segments produce the exact same results. The second one where the flags are written in full is a little easier to read and understand when you are starting out. Once the flags become familiar it is easier to use the abbreviated versions. Note while this example stored the width in the variable called **cubeWidth** (or w) we do no actually do anything with this value, if we wanted to examine it we could print it out.

```
print cubeWidth
```

Using maya.cmds in Python allows us to create many objects including spheres, torii, cylinders, planes and disc. To create a sphere, we can use polySphere()

```
cmds.polySphere() # Create a default sphere

cmds.polySphere(name='globe', radius=2) # Create a sphere with the name globe and a radius of 2

cmds.polySphere(n='globe', r=2) # Same as preceding line with flag abbreviations
```

3.1.1 *Getting Quick Help on flags*

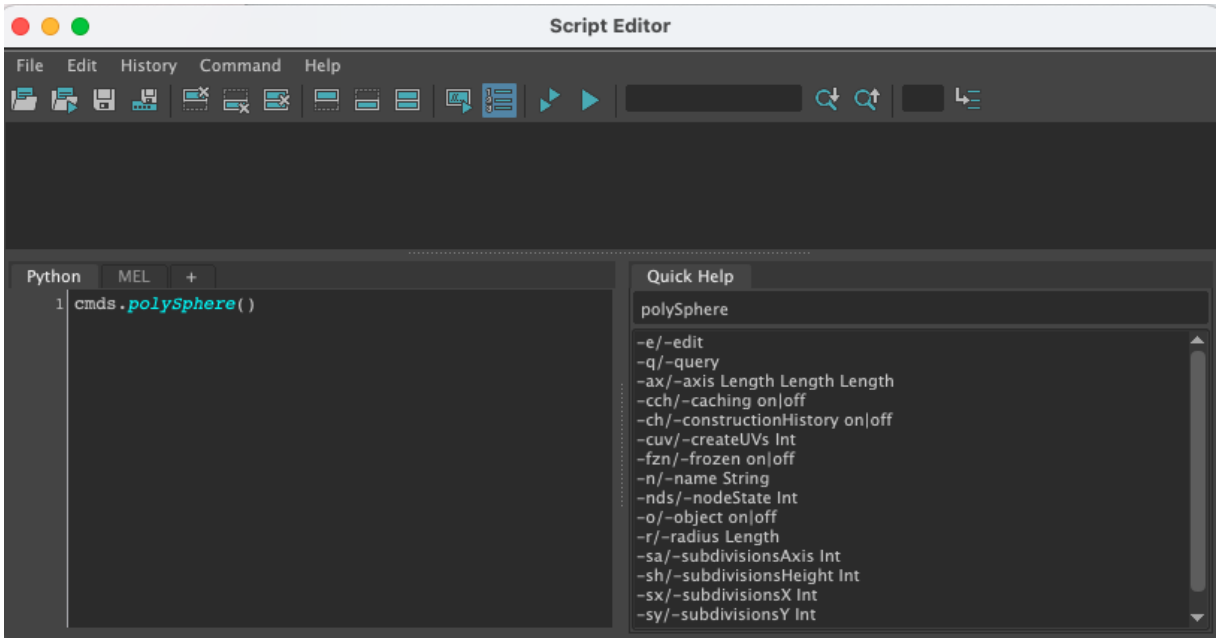

Figure 3.2: Getting Quick Help

The script editor in Maya has a nice feature to enable a listing of all the flags called *Quick Help*. To activate this, simply type the maya cmd into the Python tab of the script editor, right click and choose *Quick help*. This reveals a list of all possible flags along with the data type of the expected argument. Figure 3.2 shows the Quick Help for cmds.polySphere(). We can see, for example, that **-r** is the abbreviation for **radius**, and that we need to provide a length for the radius. We can also see that -sx is short for subdivisionsX and we are expected to provide a **int** data type.

3.2 *Variables in Maya*

```
##### Variables in MAYA ####
# In lab one you created some objects in Maya
import maya.cmds as cmds

cmds.polySphere()
cmds.polySphere()

#now I have 2 spheres in Maya but I can't quickly access them
# Let's delete them and use variables

cmds.select(all=True)
cmds.delete()

bigSphere  = cmds.polySphere()
lilSphere  = cmds.polySphere()

cmds.scale(5, 5, 5, bigSphere)  # if you scale to zero you wont see your object!
cmds.move(5, 0, 0, bigSphere)

cmds.scale(.5, .5, .5, lilSphere)  # if you scale to zero you wont see your object!
cmds.move(-5, 0, 0, lilSphere)
```

Listing 3.1: Maya Variables (3.1_mayaVariables.py)

```
# Create a Blinn Shader and assign it as the default shading group
import maya.cmds
# Create a new yellow shader
print 'A short script to create a yellow blinn shader'
print 'assign it as default, and create a sphere'

# create a new shader
yellowBlinn = cmds.shadingNode( 'blinn', asShader=True )

# set the color
cmds.setAttr( yellowBlinn+".color", 0.99, 0.91, 0.0, type='double3' )

# create a shading group
blinnSG = cmds.sets(renderable=True,noSurfaceShader=True, empty=True,name='blinnSG')

# Connect the shader
cmds.connectAttr( yellowBlinn+".outColor", blinnSG+".surfaceShader", force=True)

# Make it the default
cmds.setDefaultShadingGroup( blinnSG )

# Create a Sphere
# called yellowSphere
# with a radius of 3
cmds.polySphere(name = "yellowSphere", radius = 3)

# clear the active list
cmds.select( clear=True )
```

Listing 3.2: Creating and assigning a shader (3.2_yellowShader.py)

4
Lists

Python Programming Language does not have arrays like some other programming languages To hold a sequence of values, then, it provides the *list* class. A Python list can be seen as a collection of values. List are important for interacting with Maya So we want to make sure we can create, use and manipulate lists.

In this section we will cover

- Creating lists
- Accessing individual elements in lists
- Slicing lists

A frequent operation on lists is to iterate over them This just means we visit each item (or a subset of items) in the list. For example, we might want to visit every geometric object in Maya and scale it up or down by some factor. We will cover iteration later when we discuss loops. First we will just look at creating and accessing list elements.

4.1 Creating Lists

To create a list we can just **list the items, separated by commas, in square brackets []**. Just like with variables we don't have to tell Python what data type we need stored, Python will do it for us automatically. Technically this is called dynamically-typed. We can also create an empty list using empty []. To quickly populate a list we can use the **range** function which will provide a list of values in a range of two numbers, from a start value and up to but not including the end number. If no start value is provided then range will set the start value to zero. Listing 4.1 creates some lists in Python.

```
numberList = [0, 1, 2, 3, 4, 5]
print numberList
print type(numberList)  # okay its a list

days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']  # I forgot Sunday lets add it later
print days
print type(days)  # okay its a list

# We can also create an empty list
emptyList = []
```

```
print emptyList

# or set the list to a range
longList = list(range(0, 1931))  # all the numbers forom 0 to 1930
print longList
```

Listing 4.1: Creating Lists (4.1_createList.py)

Data types can be mixed in Python lists. Lists do not have to contain just numbers, or just strings, for example. Technically lists can even contain other lists. A list with mixed data types is shown in Listing 4.2.

```
# We are not forced to stay with the same type so we can mix them

mixedList = ["Hello", 5, 7.0, "Twenty"]
print mixedList
print type(mixedList)
```

Listing 4.2: Creating Mixed Lists (4.2_mixedList.py)

The length of a list (or number of elements) can be found using the function **len**. We can apply mathematical operators to lists. For example, adding two lists concatenates the lists into a single list. Elements will appear in the order the lists are added.

```
# To see how long a list is I can use len (short for length)
longList = list(range(0, 100))  # all the numbers forom 0 to 1930
print len(longList)

# It is possible to add or concatenate lists
numberList = [0, 1, 2, 3, 4, 5]
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']

newList = numberList + days
print (newList)
print len(newList)

aNewList = days + numberList
print(aNewList)

# We can aslo use multiplication like we did on strings
# to replicate all items in the list.
print(numberList * 3)
```

Listing 4.3: Lists can be concatenated using + (4.3_listConcat.py)

4.2 *Accessing objects in a list*

We often want to just access one element in a list We do this using square brackets and the **index** of the element we want. Python like many other languages begins indexing at 0 so the first element is listName[0] and the last element is listName[n-1] where n is the number of items in the list. Indexes are always integers (they cannot be a float for example).

We can also locate an element in a list using **index** with the value of the object I want to locate. This is shown in lines 19 and 23 of 4.4. On line 19 an error will be thrown as the “ " are missing around the string Monday.

The elements of lists can be directly changed by assigning a new value to them using listname[index] = newValue. Examples can be seen in Listing 4.4.

```
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']  # I forgot Sunday lets add it later

print (days[0])  #  Monday
print (days[2])  #  Wednesday
print (days[6])  # ERROR - oh look the list index is out of range

print (days[5])  # I have 6 days so far - the indices run from 0 to 5
## Python will tell you if you forget and the index is out of range

# There is also a neat trick to get the last element
print (days[-1])

# This also works for the second to last element, and so on
print days[-2]


# IF I want to know where an item is in my list i.e. whats it index I can do that using index

print days.index(Monday)  # ERROR - WHOA it says Monday is not defined
# Oh wait Python thinks Monday is variable name but I really meant it to be a string
# I forgot my quotes!

print 'Index of Monday is', days.index('Monday')

# Lists are MUTABLE - we can change them in place
print(days[1])
days[1] = "Moonday"
print (days)
```

Listing 4.4: Accesssing individual list elements (4.4_listElements.py)

4.3 *Checking for list membership*

Python uses the key word **in** to test if a value is an member of a list. If we want to see if an element is in a list we can use **in**. This will return either *true* or *false*, and is known as a **boolean** value.

```
# Boolean - a True or false
# we can use the keyword in to check if an element is in a list

numberList = [0, 4, 45, 78, 9, 3]
print (1 in numberList)
print(78 in numberList)
```

Listing 4.5: To check if an element is in a list use **in** (4.5_listMember.py)

4.4 *Slicing a list*

We can create a new list as a subset of existing list using *slicing*. We provide a start value and an an end value to represent the indexes for the sequence of elements we want to extract. To slice a list we provide the start index and end index separated by a colon. Listing 4.6 shows an example. It is important to remember that slicing goes up to but *does not include* the stop value.

```
##### Slicing List #####
### Sometimes we only want part of a list so we can slice them up using
### the start index : stop index
### This returns items from start index up to stop index MINUS 1
### Lets see
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']

print days[2:5]

#notice we get the elements up to but not including 5

# if we get items from the beginning of the list up to an index we can leave our the start index
print days[:5]  # from begining of list up to index 4

# and we can do the same to go right to the end of the list
print days[3:] #goes from index 3 to the end of the list

# we can leave out both to get the whole list
print days[:]

# thats just the same as print days
print days
```

Listing 4.6: Slicing Lists (4.6_slicing.py)

4.5 *Lists in Maya*

Listing 4.7 shows an example of using lists in Maya. The program first imports the Maya **cmds** API to allow us to use all the Maya commands in Python. Lines 6 selects all objects in the scene and line 7 deletes them so we can start fresh. The program uses **cmds.polySphere()** to create three spheres. Suppose I want to maintain a list of all the spheres. I can use **cmds.ls** with a wildcard on object name, in this case ***Sphere**, (line 25) and now I have a list that contains all the spheres in the scene and I can manipulate them individually. Note we could have assigned the spheres to a variable upon creation, but the point of the example is to show lists in Maya.

```
##### Lists in MAYA ####
# Lets take a look at some lists in Maya
import maya.cmds as cmds

# Lets clear the screen
cmds.select(all=True)
cmds.delete()

# Create some spheres using cmds.polySphere
# We will give each sphere a name but use the defaults
# for all other attributes.
cmds.polySphere(name = 'bigSphere')
cmds.polySphere(name = 'mediumSphere')
cmds.polySphere(name = 'lilSphere')

# bigSphere    = cmds.polySphere(name = 'bigSphere')
# mediumSphere = cmds.polySphere(name = 'mediumSphere')
# lilSphere    = cmds.polySphere(name = 'lilSphere')


# clear any selections
cmds.select(clear=True)
```

```

# Now I have 3 named spheres in Maya but I can't quickly access them
# they all have Sphere in their name
# I can use * (wildcard) to select them
# so any object with Sphere in the name gets selected

# List all selected objects named "*Sphere"
allTheSpheres = cmds.ls( '*Sphere', selection=False )
print allTheSpheres
print type(allTheSpheres)  # a list!

print allTheSpheres[0]
print allTheSpheres[1]
print allTheSpheres[2]

cmds.scale(5, 5, 5, allTheSpheres[0])  # if you scale to zero you wont see your object!
cmds.move(5, 0, 0,  allTheSpheres[0])

cmds.scale(.5, .5, .5, allTheSpheres[1])  # if you scale to zero you wont see your object!
cmds.move(-5, 0, 0,    allTheSpheres[1])

cmds.scale(2, 2, 2, allTheSpheres[2])  # if you scale to zero you wont see your object!
cmds.move(0, 5, 0,  allTheSpheres[2])

# Well imagine if we had 1000 spheres - clearly I dont want to repeat my code
# 1000 times and type in all the indexes from 0 to 999 - how tedious!
# Luckily iterating over lists (and other things) is pretty common in programming languages
# so much so we have constructs called LOOPS to visit each item in a list
# There are a few different kinds of loop constructs in python and we will cover soon!
```

Listing 4.7: Maya Lists (4.7_mayaLists.py)

4.6 *Extra Listings*

```
##### Lists in MAYA ####
# Lets take a look at some lists in Maya
import maya.cmds as cmds

# The ls command returns the names
# (and optionally the type names) of
# objects in the scene
print(cmds.ls())

# TRY THE FOLLOWING FILTERS ALL THE NODES in  your Maya Scene
# We can filter them  - let look at quick help
# type cmds.ls in the script editor - highlight ls
# and right click then quick help

print (cmds.ls(transforms= True))

print (cmds.ls(shapes= True))

print (cmds.ls(cameras= True))  #note we got the shape nodes

print (cmds.ls(lights= True))

print (cmds.ls(geometry= True))

# one of the most common flags is selection - or sl for short
```

```
# you will likely come across this one if you look at existing scripts
# TRY THE FOLLOWING COMMAND WITH NO SELECTION THEN SOME OBJECTS SELECTED IN MAYA
print (cmds.ls(sl=True))  #only the selected objects are returned
# if NO objects are selected then you will get [] - an empty list

#THIS WILL TELL YOU WHAT TYPE OF NODE WE HAVE
selection = cmds.ls(sl=True, showType=True)
print selection
```

Listing 4.8: Maya Lists (4.8_mayaLists.py)

```
# List all selected objects named "*Sphere*"
sphereList = cmds.ls( '*Sphere*', sl=False )  # This will give me a list

# Check its a list
print type(sphereList)

# lets print it out
print sphereList
# but look
# I have the Shape node too -
# I really just want the first one for each
# This is called the transform node -
# I can specify this in my cmds.ls

sphereList = cmds.ls( '*Sphere*', sl=False, type='transform' )
# This will give me a list
print sphereList
# thats better now I just have the
# transform nodes of any spheres with Sphere in the name
```

Listing 4.9: Select objects by (partial) name (4.9_mayaSpheres.py)

5
Conditional Statements

So far all the statements we have typed are executed in sequence. The first statement, followed by the second and so on. Often we want to execute a line of code, or a group of lines of code depending on some scenario. For example, if the object is a sphere make it blue. If the object is not already in a list add it to the list. For this type of branching, we can use control structures Control structures direct the order of execution of statements in the program. In this chapter, we will look at one control structure the if and if/else statement. Each line of code is called a statement

5.1 Boolean Values

In most programming languages decisions are made using boolean logic. A boolean can only have one of two values True or False.

```
itsRaining = True

print (itsRaining)
```

5.2 The if-Statement

In Python, if statements are what is known as a conditional. A conditional statement allows us to execute different blocks of code based on the value of a conditional statement. The statement will evaluate to True or False.

```
if itsRaining:

  print("Better take an Umbrella")
```

Note the format of the if statement. Python uses indentation to group statements into blocks of code.

```
if condition:

  statement(s). # statement(s) will only execute if the condition is True
```

We can compare values using comparison operators. These operators allow us to compare two values. The value returned as a result of these comparisons is either **True** or **False**.

Listing 5.1 shows the if statement in action.

Operator	Name	Example	Outcome
==	equal	a == b	True if a and b are same value, False otherwise
!=	not equal	a != b	True if a and b are different values, False otherwise
>	greater than	a > b	True if a is larger than b, False otherwise
>	less than	a < b	True if a is smaller than b, False otherwise
>=	greater than or equal	a >= b	True if a is equal to or larger than b, False otherwise
<=	greater than or equal	a <= b	True if a is equal to or smaller than b, False otherwise

```
# The if statement

# we can compare values using the following
# == for equal to (we cant use a single = because thats for assignment)
# < > less than and greater than
# <= >= less than or equal and great than or equal

a = 10
b = 10

if a == b:
    print 'a is equal to b'  # this will print

# lets change the value of b
b = 5

if a == b:
    print 'a is equal to b'  # this will not print

if a != b:
    print 'a is not equal to b'  # this will print

if a > b:
    print 'a is greater than b'  # this will print

if a < b:
    print 'a is less than b'  # this will not print

if a >= b:
    print 'a is greater than or equal to b'  # this will print

if a <= b:
    print 'a is less than or equal to b'  # this will notprint

# setting b back to 10
b = 10

if a > b:
    print 'a is greater than b'  # this will NOT print

if a < b:
    print 'a is less than b'  # this will NOT print

# the two if statements above will evaluate to FALSE
# both a and b have a value of 10
# 10 is not larger  than 10 so the first one will not print
# 10 is not smaller than 10 so the second will not print

# in this case we need to use >= and <=
# for the conditions to evaluate to true
```

```
51
52 if a >= b:
53     print '** a is greater than or equal to b'  # this will NOT print
54
55 if a <= b:
56     print '** a is greater than or equal to b'  # this will NOT print
```

Listing 5.1: Python if statements (5.1_if.py)

5.3 *The if-else Statement*

The if-else statements allows us to include an alternate block of statements to execute if the condition evaluates to false. In Listing 5.2 the *modulo* operator, which returns the remainder, is used to determine if a number is odd or even. The code begins using two separate if statements to check for this. In the first if statement checks if the remainder is equal (==) to zero, in which case the number is even. The second if statement checks if the remainder is NOT equal to zero, in which case the number is odd. In this case we know if the number is not even then it is odd - if the first if statement evaluates to *false*, then we know the number is odd so we can just use **if-else** rather than writing two if statements. The format of the if-else statement is:

```
if condition:

  statement(s) # Do these if condition is true

else:

  statement(s) # Do these if condition is not true
```

```
1  # Remember our modulo operator %
2  # Let's use it to see if a number is odd or even
3
4  theNumber = 14
5
6  if theNumber % 2 == 0:
7      print theNumber + " is even"
8
9  # Logical equivalence
10 # ==
11
12 # what if we want to say not equal
13 # then we use !=
14
15 if theNumber %2 != 0:
16     print theNumber + "is odd"
17
18 # note the format
19 # if testIsTrue:
20 #     Do this statement
21
22 # But now I just wrote two separate statements
23 # to check if a number is odd or even
24 # I can use an else statement to execute when the
25 # if test is incorrect
26
27 # the format is
```

```
# if testIsTrue:
#     Do this statement
# else:
#     Do this alternate statement

# Let's combine the 2 separate statements

if theNumber %2 != 0:
    print theNumber + "is even"
else:
    print theNumber + "is odd"

# remember the indentation.
# #
# The amount of indentation matters: A missing or extra space in a
# Python block could cause an error or unexpected behavior.
# Statements within
# the same block of code need to be indented at the same level.
```

Listing 5.2: Python if-else statements. Only one of the blocks of code will be executed. (5.2_ifelse.py)

5.4 *The if-elif-else Statement*

The if else statement above only has 2 conditions. Sometimes our decisions are more complex. We can use elif (short for else if) for those times. It is important to note that only one set of statements will be executed. The format of the if-else statement is:

```
if condition:

  statement(s) # Do these if condition is true

elif condition2:

  statement(s) # Do these if condition is not true and condition2 is true

elif condition3:

  statement(s) # Do these if condition and condition2 are not true but condition3 is true

elif condition4:

  statement(s) # Do these if condition, condition2 \& condition3

               # are not true but condition4 is true

...

else

  statement(s) # Do these if none of the conditions above this statement are true
```

An example of if-elif-else is shown in Listing 5.3. This examples prints a greeting based on the time of day. We have not seen the **input** statement before. The input statements allows values to be entered (by the user) via the keyboard. The string inside the input statement is the prompt that is shown to to the user.

```
# if-elif-else allows us to choose statments from
# a bunch of decisions

# Lets say we want a greeting depending on time of day

theTime = input("Please enter the hour of time (24 hour format)?")
# we have not seen this before!
# input allows us to take values from the keyboard/user

print theTime # just to make sure we entered it correctly

# This will pring out one greeting based on the number entered
# If the number entered does not fall into any of the categories
# then the else statment "Good night" will be printed

if theTime< 10:
 print "Good morning"   # Good Morning is printed if time is less than 10
elif theTime<12:              # now we know theTime is > 10
 print "Soon time for lunch"
elif theTime<18:              # now we know theTime is > 12
  print "Good day"
elif theTime<22:              # now we know theTime is > 18
 print "Good evening"
else:                         # now we know theTime is > 22
  print "Good night"
```

Listing 5.3: Python if-elif statements. Again only one of the blocks of code will be executed. (5.3_ifelif.py)

5.5 *if-statements in Maya*

We can use if-statements in Maya to control the flow of our code. For example, if we have spheres with a radius smaller than a certain threshold we can use an if statement to increase the radius. Listing 5.4 creates three objects then moves them if their x translation value is at 0.0. Once we move the objects, if we run the if statements again then they would not affect the placement of the objects.

```

# Let's begin by creating some Shapes
aCube     = cmds.polyCube(name = 'cube', width = 5, height =7)
aCylinder = cmds.polyCylinder(name = 'cylinder', height = 5, radius = 2)
aTorus    = cmds.polyTorus(name = 'torus', radius = 10, sectionRadius=5)

# By default all the object will be placed at the origin
# and so they will all be on top of each other
# LETS MOVE ANYTHING that is at 0 - we will just use xtranslate

# to see what we are dealing with let's pring the aCube variable
print aCube

# notice this prints a list that has a name and a shape
# we just want the name so we can access it using the index
# As we saw with lists

xTranslation = cmds.getAttr(aCube[0]+'.translateX')  # you can do this for translateY and translateZ also

print xTranslation  # to begin this is 0.0

# if it is at 0.0 we want to move it
# let's just move it 10
```

```
24 if xTranslation == 0.0:
25     cmds.setAttr(aCube[0]+'.translateX', 5)
26 
27 # Cylinder
28 xTranslation = cmds.getAttr(aCylinder[0]+'.translateX')
29 if xTranslation == 0.0:
30     cmds.setAttr(aCylinder[0]+'.translateX', 10)
31 
32 # Torus
33 xTranslation = cmds.getAttr(aTorus[0]+'.translateX')
34 if xTranslation == 0.0:
35     cmds.setAttr(aTorus[0]+'.translateX', 15)
```

Listing 5.4: Using if statements to control object location(5.4_mayaif.py)

Code listing 5.5 shows an example of how we can use the if-statement to query attributes, in this case the light intensity, and perform some action based on the attribute value. On line 53, for example, we are using an if statement to check if the light intensity attribute is less than (<) 1, and if it is then we set the intensity to 1.

```
1  # we import our maya.cmds library as usual
2  import maya.cmds as cmds
3  
4  # Lets clear the screen
5  cmds.select(all=True)
6  cmds.delete()
7  
8  # Lets say we wanted to select all the lights in our scene
9  # If you remember from last week we can do that using
10 # cmds.ls
11 print (cmds.ls(lights= True))
12 
13 # Instead of printing the lights let save them to a list and then
14 # we can use an if statement to see if we have lights
15 # in the scene
16 
17 sceneLights = cmds.ls(lights=True)
18 print sceneLights
19 if len(sceneLights) == 0:
20     print 'There are no lights in the scene'
21 else:
22     print 'Let there be light!'
23 
24 # Well there are no lights in the scene
25 # Let's create a light
26 # A directional light just has a direction
27 #
28 brightRedLight = cmds.shadingNode('directionalLight',asLight = True, n='brightLight')
29 print brightRedLight
30 cmds.setAttr(brightRedLight+'.intensity', 1)
31 cmds.setAttr(brightRedLight+'.color', 1, 0, 0, type='double3')
32 
33 cmds.polySphere()
34 
35 #lets delete all the lights and start over
36 sceneLights = cmds.ls(lights=True)
37 print sceneLights
38 cmds.delete(sceneLights)
39 
40 # There are various types of Lighting that you can choose
41 # for your 3D works; Ambient Light,
```

```
# Directional Light,
# Point Light, Spot Light,
# Area Light, and Volume Light.

ambientLight = cmds.ambientLight()
dirLight     = cmds.directionalLight()
pointLight   = cmds.pointLight()
spotLight    = cmds.spotLight(intensity=100)
areaLight    = cmds.areaLight()

cmds.setAttr(dirLight+'.intensity', .1)
if cmds.getAttr(dirLight+'.intensity') < 1:
    cmds.setAttr(dirLight+'.intensity', 1)
    cmds.setAttr(dirLight+'.color', 1, 0, 0, type='double3')
```

Listing 5.5: Using if statements to query attributes (5.5_mayaifLights.py)

6
Iteration (loops)

Iteration means executing the same block of code over and over, potentially many times. A programming structure that implements iteration is called a loop. Often we want to repeat a statement, or a group of statements, *called a block* of code. Previously we saw if statements had a test and expression. code is executed depending on if the expression returned **True** or **False**.

6.1 while Loops

Now we will look at while loops. There are a couple of different forms of while loops. A while loop **repeatedly** evaluates the test and, if its true it executes the statement, or block of code.

Let's take a look, there are several parts to a loop:

1. initialize
2. evaluate expression (test)
3. execute (if expression is true)
4. update (change the test variable)

Listing 6.1 contains the code for a simple countdown loop. The program prints the numbers from ten down to zero, followed by a "lift off". The initialization is done before we enter the while loop, **number=10**. The while loop will test the expression provided, **number>0**. If the expression evaluates to **True** then the block of code within the while loop is executed. If the expression evaluates to **False** then the program *falls out of the loop* and continues execution with the next line of code **print 'lift off'**. Each time around the loop Python evaluates the expression and repeats this process until the expression evaluates to **False**. It is important that within the while loop that the value being tested in the expression is **updated**. Failure to update the test value will result in an **infinite loop** (when using Maya with Python this will mean forcing Maya to quit). The other thing to note with while loops is that they execute zero or more times. For example, if we had set the value of number to zero in the initialization then the code within the while loop would never execute because the test expression would evaluate to **False**.

```
# A while loop REPEATEDLY TESTS the expression and,
# if its true it excutes the line (or block) of code

# Let's take a look
# if you can remember initialize, test expression, execute, update
# you should be good

number = 10         # Here the variable number is initialized to 10

while number > 0:   # this is my test, before proceeding the while loop
                    # checks if the value in number is greater than 0
                    # if this evluates to True the code block is executed
                    # otherwise the program proceeds to the next line of code
                    # after the while loop
    # execute
    print number    # In this loop we just print the number
    # update
    number = number-1 # the update must change the value in your test

print 'lift off'    # once complete execution resumes wih the next line code
# QUESTION: what happens if we miss the update.
#
# QUESTION: what happens if the test evaluates to False to begin with
#
# Now you try - can you do one
# that counts UP to 10 instead of down from 10
```

Listing 6.1: Using a while loop to print a count down (6.1_while.py)

In Listing 6.1 the while loop will execute ten times, until number reaches zero. However, *there are many situations where we will not know in advance how many times a loop will execute*. For example, when we allow the user to influence how many times we need to execute the body of the loop. Imagine a game, when the game ends the user is presented with the option to play again or end gameplay. Listing 6.2 shows an example. It is a little more complex because we use a dialog box to capture the input from the user, and this is a little more involved in Maya than say at the command line. In Listing 6.2, if the user enters any value except 'n' then the loop continues execution. Once the user enters 'n' the Boolean value playAgain is set to **False** and the loop will be terminated.

```
# What about if its not a number
# Maybe we want to keep playing a game until
# the user enter No

# initialize
playAgain = True    # This will control my loop
user_input = 'n'    # This will store user input
                    # if the user doesnt enter something other than n
                    # then the playAgain will be set to False and the
                    # loop will be complete

while playAgain: #test  # playAgain is a boolean.  I could say while playAgain == True
                        # but it will either be True or False anyway

    # Pretend we have some game in here
    # At the end we ask do you want to play again
    # We can use a promptDialog to get input from the user
    # If the user does not enter a value other than n and hit OK
    # the loop will end

    result = cmds.promptDialog(
```

```
                title='Play Again',
                message='Do you want to Play Again:',
                button=['OK', 'Cancel'],
                defaultButton='OK',
                cancelButton='Cancel',
                dismissString='Cancel')
    # if they press cancel or x to dismiss we will
    # not change the value of user_input so the loop will end

    if result == 'OK':
        user_input = cmds.promptDialog(query=True, text=True)
    print user_input
    # now we can put our knowledge of if statements to work
    if user_input == 'n':
        print 'Bye, Bye, Bye'
        playAgain = False  # note here that we dont have to do
                           # anything if user input is anything other than n
                           # because we wont change the value
    else:
        print 'Another Round!'

# While Loop
# INIT: playAgain = True
# TEST: while playAgain (note playAgain is boolean so its already True/False)
# UPDATE:  playAgain = False
```

Listing 6.2: Using a while loop govern game play (6.2_whileLoop.py)

Another example of a while loop is shown in Listing 6.3. Here we are capturing numeric input from the user so we can use the simpler command **input()**. Again, in this case, we do not know how many times the loop will execute. We know it will certainly execute once because we initialize the test variable to False directly before we enter the loop. An alternate way to test would be to use the **not** operator, bf ! and use the following test in place of line 13.

```
while !guessedTheNumber:
```

```
# Lets play a game
# I want the user to guess a number between 1 and 10
# I can set this up as follows

secretNumber     = 7 # the user cant see this
                     # we could use a random function to set this
                     # but for now this is fine
                     # the example just illustrates while loops

guessedTheNumber = False    # this is a Boolean value
                            # it can only have the values True or False

while guessedTheNumber == False:    # we just initialized guessedTheNumber to False
                                    # so we know this loop will execute at *least* once
    # allow the  user to enter a guess
    guess = input("Guess a number between 1-10")    # taking number input is simpler

    # Use an if statment to check if the number the user entered
    # matches the secret number
    if guess == secretNumber:
        print "\n YAY you guessed correct.  The secret number is: ", guess
        # NOTE This is the update - we will only set guessedTheNumber to True
        # Which will end the loop WHEN the user enters the correct number
```

```
        # If the correct number is never entered then the loop will continue forever
        # (since we said between 1-10 we hope this will never happen)
        guessedTheNumber = True
    else:
        print "\n Oh Oh, you didn't guess correctly try again"

# Things to consider
# What is the initialization of the above loop
# What is the test
# what is the update
# why didnt I need to do anything in the else to update?
```

Listing 6.3: Using a while loop to play a guessing game (6.3_guess.py)

While loops are used extensively in programming, particularly in cases where we do not know in advance the exact number of times the block of code will execute. If we do know beforehand exactly how many times we want the loop to execute we can use *for loops*, sometimes called counted loops. We will look at for loops in the next section.

6.2 *for Loops*

Another way to accomplish iteration is to use a *for loop*. These work similarly to while loops, but are a little bit neater. Let's revisit Listing 6.1, but instead of a while loop we will use a for loop. The difference here is that the code in a for loop runs a fixed number of times. The for loop will iterate over a sequence, executing the block of code each time. Before we examine the for loop, we will introduce the range function. The range function is used in for loops so often that you could be forgiven for thinking it was part of the syntax. But it is a built-in function in Python which returns numbers in a range.

6.2.1 *The range function*

The range function returns numbers within a given range and takes the form of

```
range(startValue, stopValue, step)
```

If the **step** is negative, then the start value must be greater than the stop value or range will return an empty sequence [].

```
# range
# starts counting at 0 and then provides
# the numbers up to but not including 10
print range(10)          # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# same as above
print range (0, 10)      # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# can provide a start that is not zero
# range(start, stop)
print range (2, 10)      # [2, 3, 4, 5, 6, 7, 8, 9]

# dont have to increment by 1
# range(start, stop, increment)
print range (2, 10, 2)   # [2, 4, 6, 8]

# remember up to but not including stop
print range (0, 101, 10) # [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

```
print range (0, 100, 10) # [0, 10, 20, 30, 40, 50, 60, 70, 80, 90]

# to count down provide a negative decrement
print range (10, 0, -1)  # [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Listing 6.4: The range function is often used in conjunction with the for-loop (6.4_range.py)

6.3 *The for-loop structure*

The for loop is structured as follows:

```
for element in sequence:

     block of code
```

Again Python uses indentation to indicate the block of code that belongs in the for-loop. Listing 6.5 shows our some examples of for loops. In each loop the element iterates over the sequence.

```
# for loops
# sometimes called counted loops
# because the iterate over the code block a fixed number of times

# the strucuture is
# for element in sequence:
#   block of code

# note the for loop iterates over a sequence
# there is no need for init, test, update like in
# our while loop.   Here we simply visit each element
# of the sequence

# The sequence is often a list

for element in [1, 2, 3]:
    print element

for element in ['Howdy', 'How', 'Are', 'You']:
    print element

# can do more than just print
totalSum = 0
for element in [10, 23, 5]:
    totalSum += element

print totalSum
```

Listing 6.5: Examples of for loops in Python (6.5_for.py)

Listing 6.6 shows our count down program using a for loop instead of a while loop. The syntax is a more concise than for the while loop. If you know in advance the number of times you want to iterate you can use a for loop.

```
# This does exacly the same as our while loop
# that printed the numbers from 10 to zero
# in steps of -1

# the for loop in this case is a little more concise
# use for loops when you know in advance that the
```

```
# loop must execute a fixed number of times

for element in range(10, 0, -1): # prints each element from 5 to 10 (up to stop)
    print element                # in steps of -1

print 'lift off'
```

Listing 6.6: Examples of for loops in Python (6.6_countDown.py)

The range function is commonly used to set up the sequences that the for loop will iterate over. Listing 6.7 shows the use of range in for loops. Note the variable name **element** can be any name you choose. For example, if you had a list called **days** that contained a list of days of the week your code might read at follows:

```
[for day in days:]
```

```
# for loops iterate over a sequcence
n = 10

for element in range(n):
    print element               # prints each element from 0 to 9


for element in range(5, 11):    # prints each element from 5 to 10 (up to stop)
     print element

for element in range(5, 11, 2):  # prints each element from 5 to 10 (up to stop)
                                 # in steps of 2
     print element

for element in range(10, 0, -1): # prints each element from 5 to 10 (up to stop)
    print element                # in steps of -1

for element in range(10):        # prints each element from 0 to 9
    print element

for i in range(10, 0, 1):       # this will print nothing because range returns []
    print element

# nested loops can iterate over 2 values
for x in range(1, 11):          # prints times tables from 1 to 10
    for y in range(1, 11):
        print x*y
    print '------'
```

Listing 6.7: Examples of for loops in Python (6.7_for.py)

6.4 *Loops in Maya*

We can use loops in Maya to automate repetitive tasks. If I wanted to create twenty spheres, doing this by hand may become cumbersome. With a while loop I can automate it using a few lines of code. Listing 6.8 illustrates the use of a while loop to draw five spheres but it would be easy to change the number of spheres to draw just by changing the value initially assigned to **numSpheres** on line 9. The result of running this code is shown in Figure 6.1. If we wanted to create more (or less) spheres changing the value of numSpheres is simply a case of changing the number 5 on line 9 the number of spheres required.

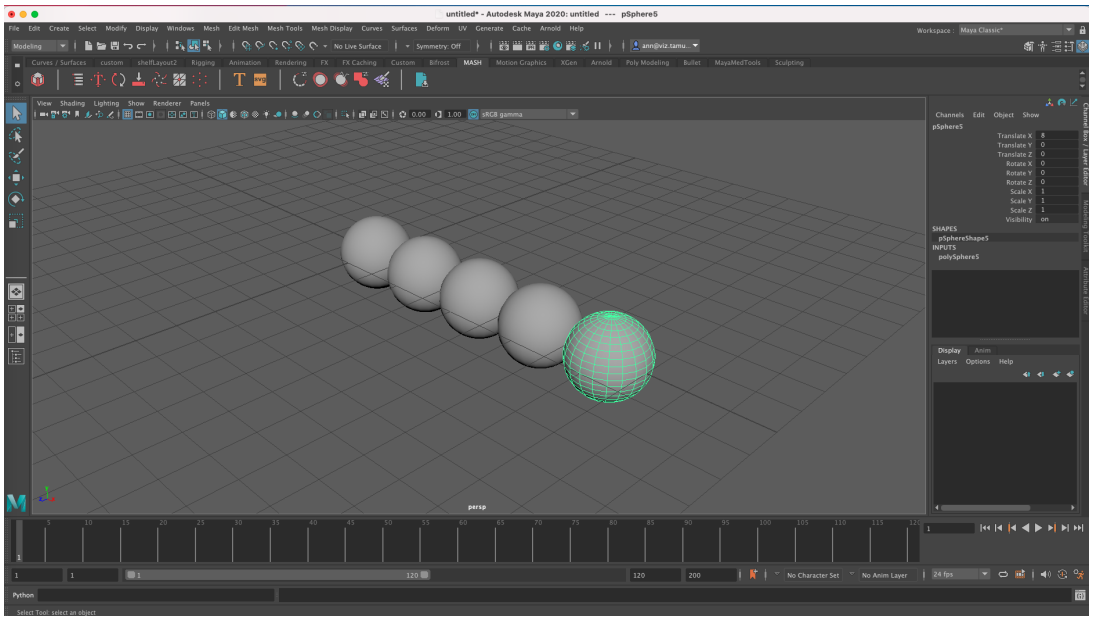

Figure 6.1: The result of a while loop to draw five spheres in Maya

```python
# using a while loop to draw multiple spheres
import maya.cmds as cmds

# Completely reset the Maya scene (similar to using "cmds.select(all=True) cmds.delete()")
cmds.file( force=True, new=True )


sphereCount = 0      # initialize my counter, sphereCount, to 0
numSpheres  = 5      # here I will draw 5 spheres, but its easy to change
xTranslate  = 2      # Give myself an offset to I can space out the spheres

while sphereCount < numSpheres:
    print sphereCount               # will print the counter, from 0 to 4

    sphere = cmds.polySphere()      # draw a sphere, we will overwrite the sphere variable each time
    cmds.move(xTranslate * sphereCount, 0, 0, sphere)  # move the sphere over a little
    sphereCount = sphereCount + 1     # UPDATE increment the sphereCount

# Lets run that loop again
# What if I want 10/20/30 spheres!
```

Listing 6.8: Using a while loop create and place spheres (6.8_mayaWhile.py)

We can also use for loops in Maya to automate repetitive tasks such as creating multiple objects, or a grid of objects.

```python
# for loops in maya
# this program works the same as listing 6.8
import maya.cmds as cmds

cmds.file( force = True, new = True)    # create a new file

sphereCount = 10         # this can be changed easily
xTranslate  = 2          # as can this

for eachSphere in range(sphereCount):  # runs from 0 to 9 so 10 spheres
```

```
    sphere = cmds.polySphere()
    cmds.move(xTranslate*eachSphere, 0, 0, sphere)
```

Listing 6.9: Using a while loop create and place spheres (6.9_mayaFor.py)

```
# for loops in maya
# import maya commands
import maya.cmds as cmds

cmds.file( force = True, new = True)    # create a new file

# using nested loops to make a grid of objects
sphereCountInX = 10
xTranslate  = 2

sphereCountInY = 10
yTranslate  = 2

for xSphere in range(sphereCountInX):  # runs from 0 to 9 so 10 spheres
    x = xTranslate * xSphere
    for ySphere in range(sphereCountInY):  # runs from 0 to 9 so 10 spheres
        y = yTranslate * ySphere
        sphere = cmds.polySphere()
        cmds.move(x, y, 0, sphere)
```

Listing 6.10: Using a nested for loop create and place spheres (6.10_mayaNestedFor.py)

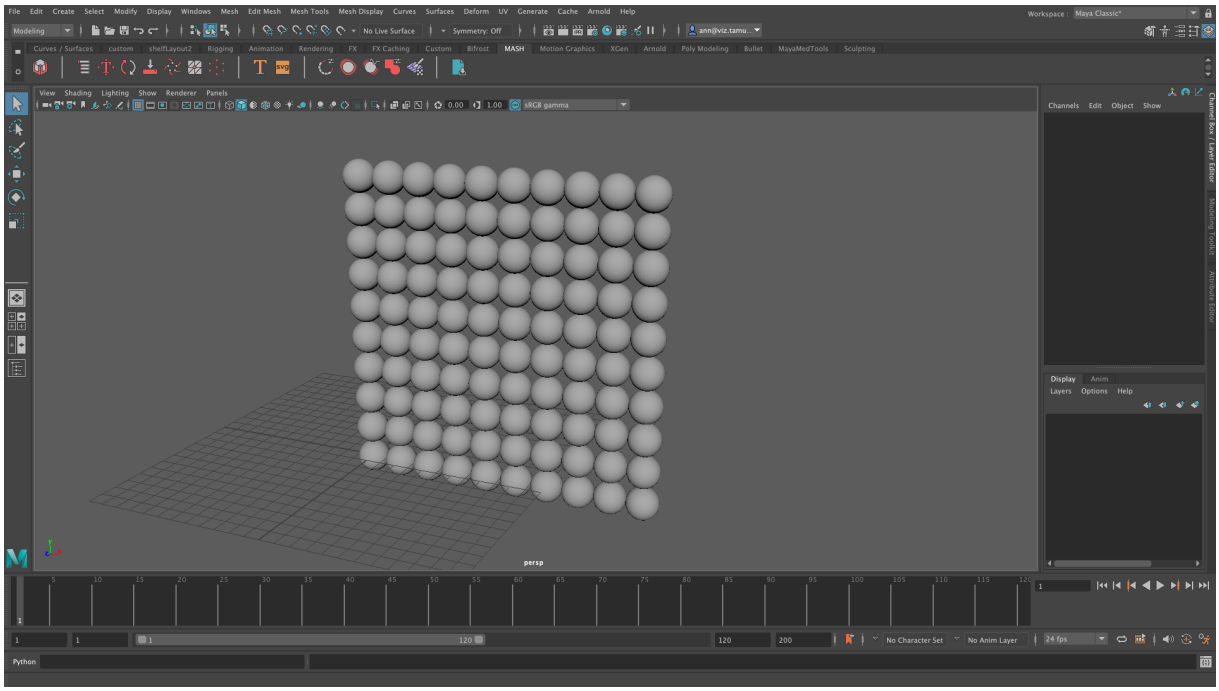

Figure 6.2: The result of a nested for loop to draw a grid of spheres in Maya

```
import maya.cmds as cmds

cmds.file( force = True, new = True)    # create a new file

# list all objects
geoList = cmds.ls()  # this selects everything in the scene
```

```
print geoList

# Hmm we dont want ALL THAT lets just get our geometry

geoList = cmds.ls( geometry=True )
print geoList

# this will scale everything by 5 in the y direction
# the for loop will iterate over each geo object
# and scale it
for geo in geoList:
    cmds.scale(1, 5, 1, geo)
```

Listing 6.11: Using a for loop scale up all geometry in a scene (6.11_geo.py)

Combining the two previous examples it is easy to see how useful this could be to quickly generate scenes. Listing 6.12 shows the code that result in a rudimentary “city" scene in just a few lines of code. This could also be applied to other repeating scenes, like a neighborhood of houses, a forest of trees or a garden of flowers. This code uses the**random** library. The function **random.randint(start, stop)** which returns a random number between the start value given and the stop value given. In Listing 6.12 we can place the cubes on a grid and then scale them a random amount in y.

```
import maya.cmds as cmds
import random
cmds.file( force = True, new = True)    # create a new file

# using nested loops to make a grid of objects
gridX = 10
xTranslate  = 2

gridY = 10
yTranslate  = 2

for xElem in range(gridX):  # runs from 0 to 9 so 10 spheres
    x = xTranslate * xElem
    for yElem in range(gridY):  # runs from 0 to 9 so 10 spheres
        z = yTranslate * yElem
        cube = cmds.polyCube()
        cmds.move(x, 0, z, cube)

geoList = cmds.ls( geometry=True )
print geoList

# this will scale everything by a randome amount in the y direction
# the for loop will iterate over each geo object
# and scale it
for geo in geoList:
    cmds.scale(1, random.randint(2,10), 1, geo)

cmds.select(clear = True)
```

Listing 6.12: Using a for loop scale up all geometry in a scene (6.12_city.py)

6.5 Summary

Iteration is one of the main ways to control the flow of a program. It is widely used in all programming languages, but is especially useful in Maya where we can use it to help quickly populate environments with

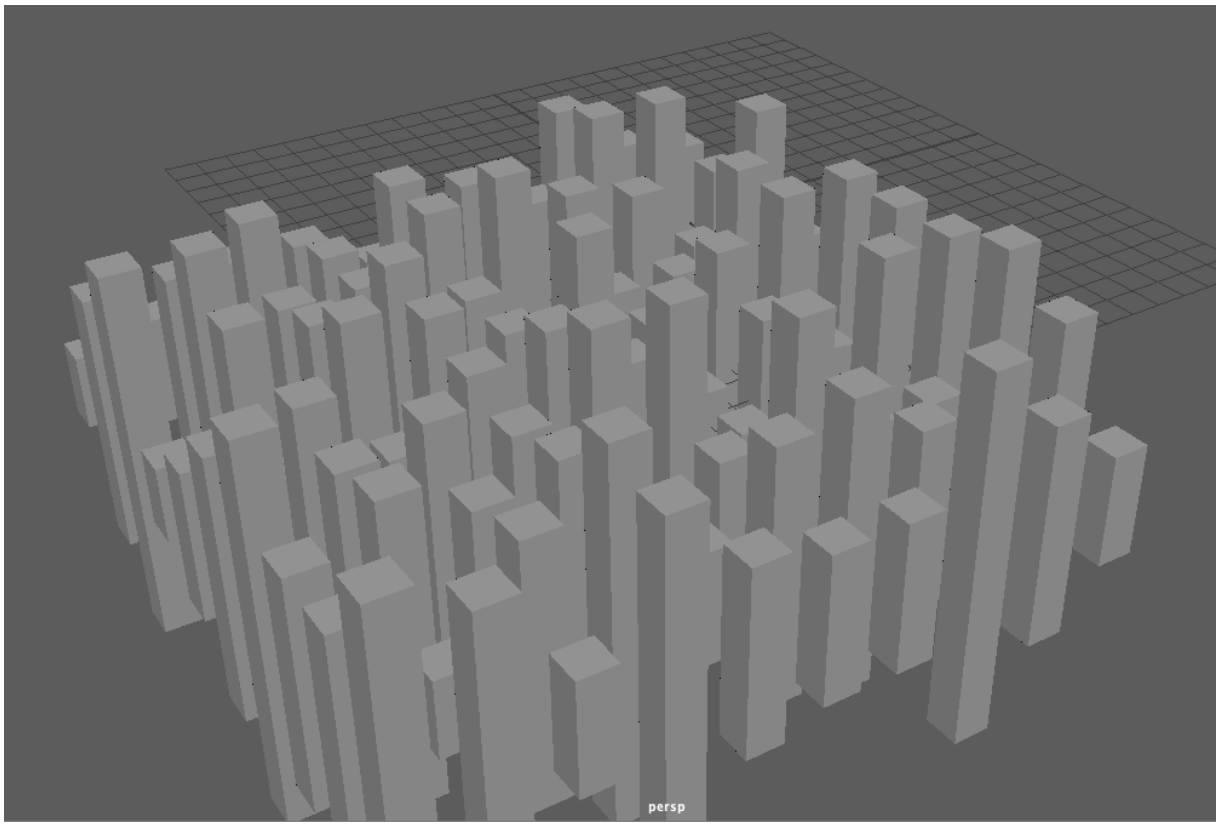

Figure 6.3: The result of a nested for loop to draw a grid of rectangles of random sizes to emulate a city in Maya

objects, or scale all the spheres in the scene, or intensify all the lights in a scene. Loops really are are a very useful tool for scripting in Maya.

7
Functions

7.1 What are functions

Functions are a convenient way to package up a group of statements that accomplish a specific task. Once we package our code in a function it can be reused over and over again. We *call* a function by using its name, and we provide any input the function might need, and store any results the function might return back to us. We have already encountered many functions but we just used them (ex. scale(), move(), select(), print()). These are **built in functions**, or functions that come with the programming language. As programmers we want to write our own functions to solve our own particular problems.

You use functions in programming to bundle a set of instructions that you want to use repeatedly or that, because of their complexity, are better self-contained in a sub-program and called when needed. *That means that a function is a piece of code written to carry out a specified task.*

To carry out that specific task, the function may or may not need multiple pieces of information provided to it, these are called **inputs**. (Functions can have optional inputs.). When the task is carried out, the function might have a value to return, so a function may or may not return values. (Functions can have optional outputs/return-values.)

Generally, a good rule of thumb is if you've written the same code twice, then you should think about putting it into a function. One of the objectives of good programming is to keep things DRY (Don't Repeat Yourself), that way if you need to make a change it will only need to happen in one place and it also makes debugging far quicker and simplified.
There are a few steps to define a function

1. Use the keyword **def** to declare a function and give it a name NOTE the name should reflect what the function does - believe me you want to name your functions well

2. If you have parameters they go inside () after the function name and then end that line with a :

3. Add your code indented

4. End your function with the keyword "return" and a value if you want the function to return anything (if not then it will simply continue to the next line of code)

```
def functionName(parameter1, parameter2...):
    code to execute
    return value # optional

# the function name should describe the action
# to INVOKE a function we have to CALL it

functionName(argument1, argument2). # this is calling the function
# You can pass any number of arguments, each argument is mapped to each parameter in the function
# If you have more than a few parameters you might # want to rethink if the function could be
# separated into multiple function, maybe you are # not trying to just do one thing.
```

The following function prints a greeting, this function takes a value “name" and a value “greeting" then prints the values. These input values, name and greeting, are called parameters.

```
def greet(name, greeting):
    print greeting + ' ' + name

# So that by itself does NOTHING
# to INVOKE a function we have to CALL it

greet("Jack", "Hello")
greet("Jane", "Good Morning")
greet("Sophie", "Good Afternoon")
```

7.2 *Passing Arguments & Returning values*

Let’s write a simple function to sum two numbers. We can call the function sum, it will accept two arguments, and return the sum and print the result of adding those two numbers. This code is shown in Listing 7.1.

```
# Lets add 2 numbers

def sum(firstValue, secondValue):
    print firstValue+secondValue


sum()       # NOTE THE ERROR
sum(1)      # Again ERROR
sum(2, 3)   # Always a good idea to test values you know the answer to
sum(5, 5)
sum(17, 3)
sum(2251416516, 315645613415684)  # then move on to more complex

sum(2, 3, 3)  # NO, invalid because we never define a third input
```

Listing 7.1: A function to sum two numbers (7.1_sum.py)

Listing 7.1 simply outputs the results, using the print statement. Often we want to use the result of a function in another calculation, or even as an argument to another function. Listing 7.2 illustrates some of

these concepts. First in Line 9, while the function will execute and return a value we have not printed it out so we won't see any result. The next two lines print the result. Line 15 shows how we can store the result of a function in a variable, theSum, and then use that result in later calculations, or even as an argument to functions. Finally, we can pass variables as arguments.

```
# What if I dont want to just print something and just return the value?

# Let's update our sum function
def sum(firstValue, secondValue):
    return firstValue + secondValue # Here I am returning the answer
                                    # I am not printing it I am sending it back to
                                    # where the function was called

sum(3, 2) # This will run the function, but we wont be able to see the output!
print sum(3, 2)  # here the sum(3, 2) is evaluated by calling the function and returning the value
print sum(5, 6)

# But maybe I just want to use that result in another calculation...

theSum       = sum(50, 30)
doubleTheSum = 2 * theSum
print doubleTheSum

# Of course we can pass variables as arguments
x = 10
y = 40

print sum(x, y)
```

Listing 7.2: A function to sum two numbers (7.2_return.py)

7.3 *Functions in Maya*

```
# MAYA
import maya.cmds as cmds
import random
#This Creates a new file
cmds.file( force=True, new=True )

# create a sphere
def createSphere(theRadius):
    print 'Creating A Sphere'
    sphere = cmds.polySphere(radius = theRadius)
    return sphere

# scale an object - this will work for any object
def scaleObject(theObject, scaleX, scaleY, scaleZ):
    cmds.select(clear = True)
    cmds.select(theObject)
    cmds.scale(scaleX, scaleY, scaleZ, theObject)

# uniformScale
#   2 parameters
#   theObject
#   scaleFactor - this will work on any object
def uniformScale(theObject, scaleFactor):
        cmds.scale(scaleFactor, scaleFactor, scaleFactor, theObject)

```

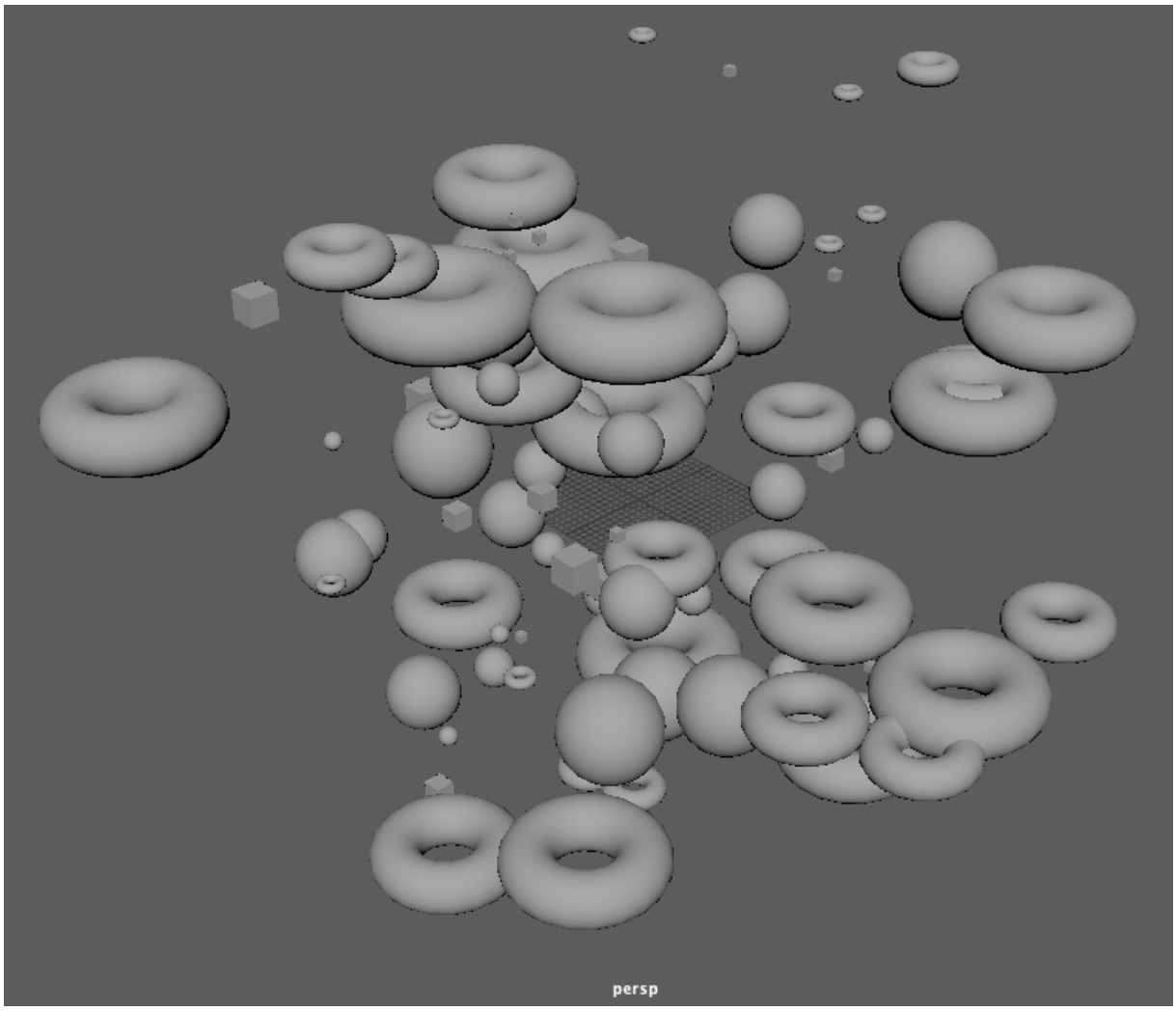

Figure 7.1: The result of using functions to create, size and randomly place objects in a scene

```
26 def randomPlace(theObject, gridSize):
27     xTranslate = random.randint(0, gridSize)
28     yTranslate = random.randint(0, gridSize)
29     zTranslate = random.randint(0, gridSize)
30 
31     cmds.move(xTranslate, yTranslate, zTranslate, theObject)
32 
33 
34 mySphere   = createSphere(5)
35 scaleObject(mySphere[0], 1, 3, 4)
36 
37 myCube = cmds.polyCube()
38 scaleObject(myCube[0], 1, 8, 1)
39 
40 myTorus = cmds.polyTorus()
41 
42 uniformScale(mySphere[0], 10)
43 uniformScale(myCube[0], 10)
44 
45 #This Creates a new file
```

```
cmds.file( force=True, new=True )

numberOfSpheres = 30
for i in range(numberOfSpheres):
    sphereRadius = random.randint(1, 5)
    sphere = createSphere(sphereRadius)
    randomPlace(sphere[0], 55)

numberOfCubes   = 20
for i in range(numberOfCubes):
    cubeSize = random.randint(1, 3)
    cube     = cmds.polyCube()
    uniformScale(cube[0], cubeSize)
    randomPlace(cube[0], 60)

numberOfTorii   = 40
for i in range(numberOfTorii):
    torusSize = random.randint(1, 6)
    torus     = cmds.polyTorus()
    uniformScale(torus[0], torusSize)
    randomPlace(torus[0], 70)


cmds.select(clear=True)
```

Listing 7.3: Using Maya functions to create size and randomly place objects in a scene (7.3_mayaFunctions.py)

In Listing 7.4 the code first uses a loop to call the functions, then packages that loop into its own function so it can be called with a single line of code. Here we introduce *default parameter values*. If no argument values are provided when the function is called then the function will use the default values. Figure 7.2 shows the result of calling this function with fifty spheres, and an a thirty by thirty by thirty grid. Of course we can repeatedly call the function, we could even put the function call inside a loop, and this would result in a denser grid as shown in Figure 7.3.

```
# Remember a function should really just do one thing
# We have 2 functions one creating the sphere and shading it
import maya.cmds as cmds
import random

#This Creates a new file
cmds.file( force=True, new=True )

# Creates a sphere, prints out its name, and returns the sphere.
# we can assign default values to the parameters
# here if no values are passed in the the default values are used
def createSphere(theRadius = 3, xScale = 1, yScale = 1, zScale = 1):
    sphere = cmds.polySphere(radius = theRadius)
    print 'Creating: ' + sphere[0]
    return sphere

# Shades the object based off rgb values.
def shadeObject(theObject, red = 1.0, green = 1.0, blue = 1.0):
    shadingNode = cmds.shadingNode( 'blinn', asShader=True )
    cmds.setAttr( shadingNode+".color", red, green, blue, type='double3' )
    shadingGroup = cmds.sets(name=theObject+'SG', empty=True, renderable=True, noSurfaceShader = True)
    print shadingGroup
    cmds.connectAttr(shadingNode+'.outColor', shadingGroup+'.surfaceShader')
```

```
    cmds.select(theObject)
    cmds.sets(e=True, forceElement=shadingGroup)

# Scales the object, but uses default values
def scaleObject(theObject, xScale = 1, yScale = 1, zScale = 1):
    cmds.scale(xScale, yScale, zScale, theObject)

# Scales the object uniformly
def uniformScale(theObject, scaleFactor = 1):
    cmds.scale(scaleFactor, scaleFactor, scaleFactor, theObject)

# places the object randomly in a 3D grid
def randomPlace(theObject, gridSize):
    xTranslate = random.randint(0, gridSize)
    yTranslate = random.randint(0, gridSize)
    zTranslate = random.randint(0, gridSize)

    cmds.move(xTranslate, yTranslate, zTranslate, theObject)


sphere = createSphere(2)
shadeObject(sphere[0], 1, 0, 0)

cube = cmds.polyCube()
cmds.move(-2, 0, 0, cube)
shadeObject(cube[0], 1, 0, 1)  # Change this to see different colors
scaleObject(cube[0],1, 8, 5)

# Lets make something colorful
numSpheres  = 30
gridSize    = 40
for i in range(numSpheres):
    xTranslate = random.randint(0, numSpheres)
    yTranslate = random.randint(0, numSpheres/10)
    zTranslate = random.randint(0, numSpheres)

    sphereRadius = random.randint(1, 3)
    sphere  = createSphere(sphereRadius)
    shadeObject(sphere[0], random.random(), random.random(), random.random())
    randomPlace(sphere, gridSize)

# Completely reset the Maya scene (similar to using "cmds.select(all=True) cmds.delete()")
cmds.file( force=True, new=True )

##################################################
# Now lets actually put that loop into a function so
# we can call it easily.

# Returns a random integer based off the number of objects
def myRandom(start = 0, numObjects = 100, divideBy = 1):
    return random.randint(start, numObjects/divideBy )

# Creates a specified number of spheres, randomly moves them, and adds a random color
def lotsOfSpheres(numObjects = 100, gridSize=100):
    for i in range(numObjects):
        sphereRadius = random.randint(1, 3)
        sphere  = createSphere(sphereRadius)
        shadeObject(sphere[0], random.random(), random.random(), random.random())
        randomPlace(sphere, gridSize)
        cmds.select(all=True, clear=True)

```

```
# lotsOfSpheres()  # Calling like this will use the default values
lotsOfSpheres(50, 30)
```

Listing 7.4: Using Maya functions to create shade and randomly place objects in a scene (7.4_mayaFunctions.py)

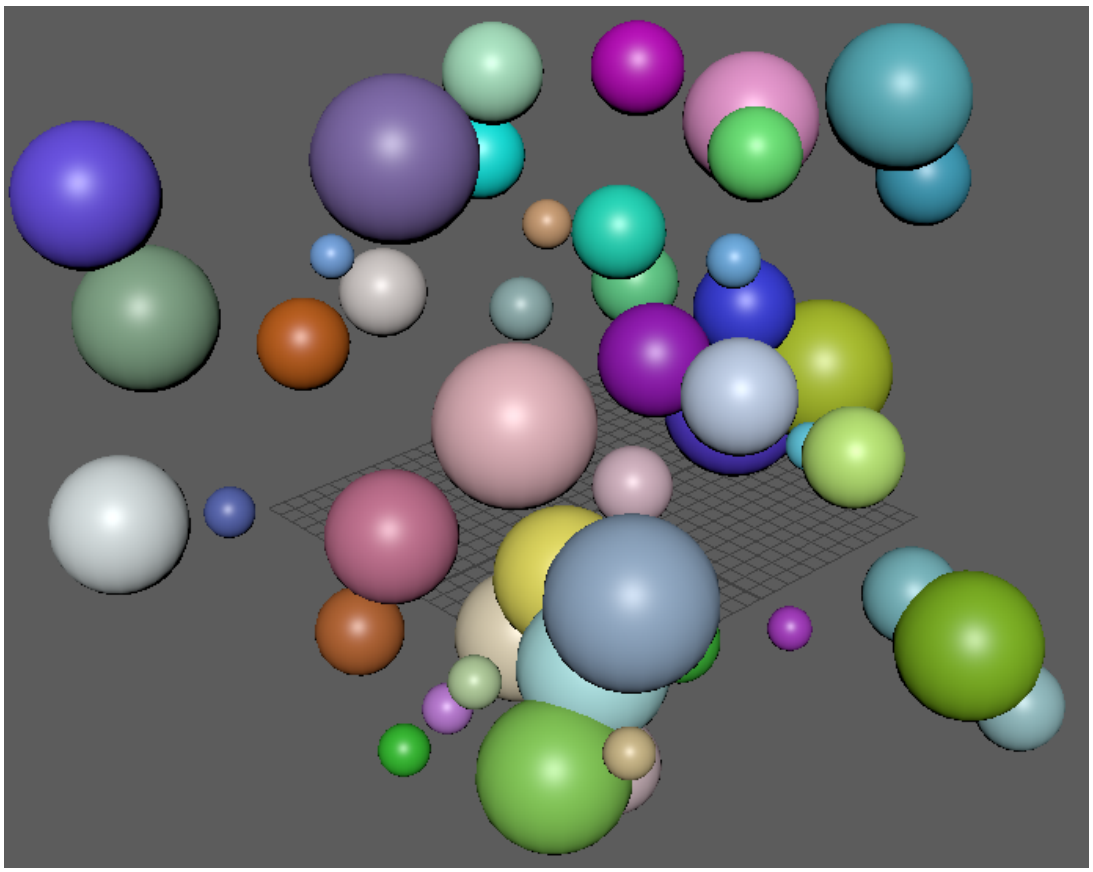

Figure 7.2: The result of using functions to draw multiple spheres in random colors on a 3D grid in Maya

7.4 *Summary*

Functions are a very useful way not only to organize our code but also to package up blocks of code that carry out a specific tasks. One of the biggest benefits of functions is the fact that they can be *reused*. Functions can be called with different parameters, leading to different results. This allows us to write our code once and use it thereafter. It also facilitates collaboration across teams as once a function is written (and tested) then it can be used by other people. In fact, we have been using functions from the very beginning. Even the humble **print** statement is a (built-in) function in Python.

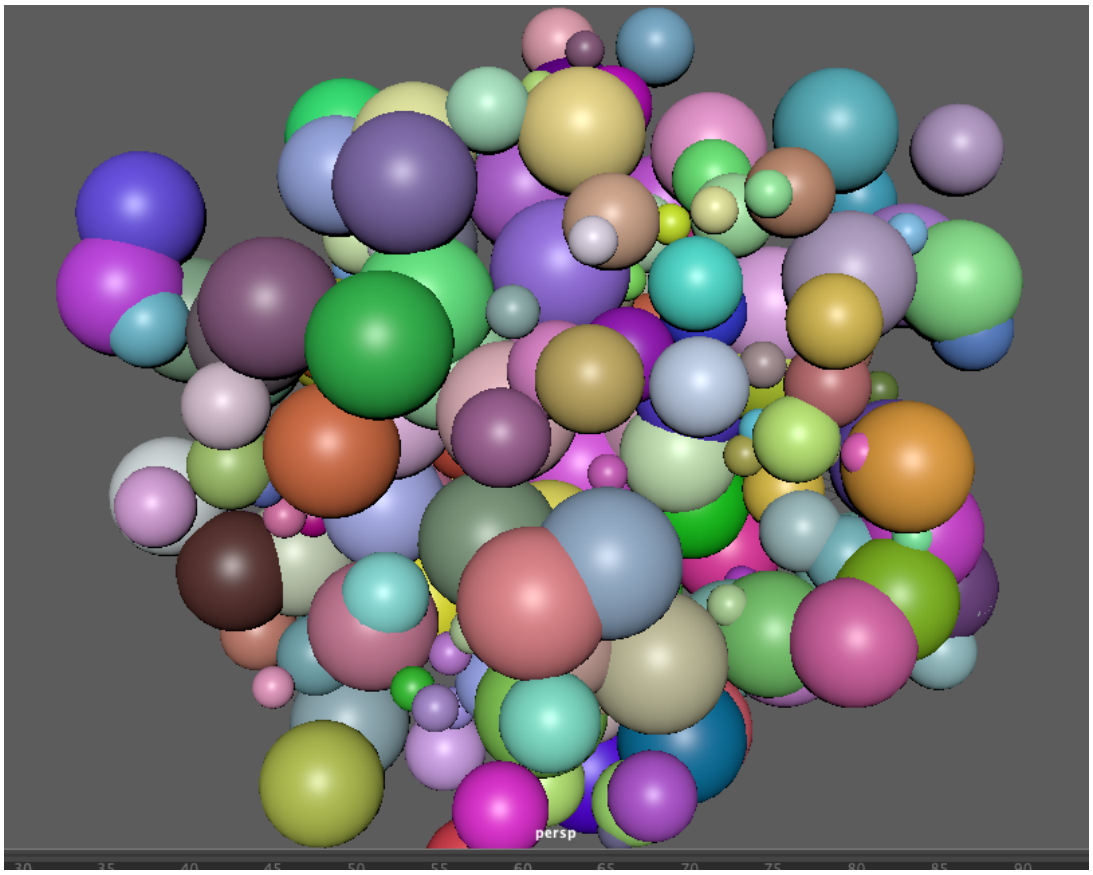

Figure 7.3: The result of repeatedly calling a function to draw multiple spheres in random colors on a 3D grid

8
Object Oriented Programming

8.1 Classes and Objects

We have seen many built-in types in both Python (strings, lists) and Maya (spheres, cubes, lights). All these are in fact **objects**. We have been using objects all along. The Object Oriented Paradigm (OOP) facilitates the creation of user defined types, called objects. The blueprint for an object is called a class. A class encapsulates data and methods. The data stores the attributes of an object and the methods define the functionality.

8.2 Object Oriented Programming in Maya

```
import maya.cmds as cmds
import random

#This Creates a new file
cmds.file( force=True, new=True )


class Worm:
    def __init__(self, size):
        self.size = size    # this is the number of body parts
        self.body = []      # we will keep the body parts as a list

    def create(self):
        for ball in range(self.size):
            sphere = cmds.polySphere(radius=ball+.5)
            cmds.move(0, ball*(ball+.15), 0, sphere)
            self.body.append(sphere[0])
        print self.body

    def move(self, xTranslate):
        for ball in self.body:
            cmds.move(xTranslate, 0, 0, ball, relative=True)

    def setColor(self):
        red = random.random()        # if we wanted a certain hue
        green = random.random()      # we could set the other two to 0
        blue = random.random()
        shadingNode = cmds.shadingNode( 'lambert', asShader=True )
        cmds.setAttr( shadingNode+".color", red, green, blue, type='double3' )
        shadingGroup = cmds.sets(name=self.body[0]+'SG', empty=True, renderable=True, noSurfaceShader = True)
        cmds.connectAttr(shadingNode+'.outColor', shadingGroup+'.surfaceShader')
```

```
        for ball in self.body:  # loop over all our body parts and assign the shader
            cmds.select(ball)
            cmds.sets(e=True, forceElement=shadingGroup)

# This creates three Worm objects from the same class (Worm)
mediumWorm = Worm(5)
mediumWorm.create()
mediumWorm.move(1)
mediumWorm.setColor()

littleWorm = Worm(3)
littleWorm.create()
littleWorm.move(-7)
littleWorm.setColor()

bigWorm = Worm(7)
bigWorm.create()
bigWorm.move(11)
bigWorm.setColor()
```

Listing 8.1: Using Object Oriented Programming create shade and place objects in a scene (8.1_simpleClass.py)

We can add any functionality to our worm class that we like. The nice thing about Object Oriented Programming is that you can control behavior. For example, if we only want our creature to move in a certain way then we can control that. Let's add a *walk* method to our Worm class. This method will allow the Worm object to move from the current location to the current location plus the **howFar** parameter provided. For simplicity, we set keyframes at 0 and 100 but you could set them at any times you like by changing the t= values on line 64 and 65. You could even decide to pass the keyframe times as parameters.

```
import maya.cmds as cmds
import random

#This Creates a new file
cmds.file( force=True, new=True )


class Worm:
    def __init__(self, size):
        self.size = size    # this is the number of body parts
        self.body = []      # we will keep the body parts as a list

    def create(self):
        for ball in range(self.size):
            sphere = cmds.polySphere(radius=ball+.5)
            cmds.move(0, ball*(ball+.15), 0, sphere)
            self.body.append(sphere[0])
        print self.body

    def move(self, xTranslate):
        for ball in self.body:
            cmds.move(xTranslate, 0, 0, ball, relative=True)

    def setColor(self):
        red = random.random()        # if we wanted a certain hue
        green = random.random()      # we could set the other two to 0
        blue = random.random()
        shadingNode = cmds.shadingNode( 'lambert', asShader=True )
```

```
        cmds.setAttr( shadingNode+".color", red, green, blue, type='double3' )
        shadingGroup = cmds.sets(name=self.body[0]+'SG', empty=True, renderable=True, noSurfaceShader = True)
        cmds.connectAttr(shadingNode+'.outColor', shadingGroup+'.surfaceShader')
        for ball in self.body:  # loop over all our body parts and assign the shader
            cmds.select(ball)
            cmds.sets(e=True, forceElement=shadingGroup)

    def walk(self, howFar):
        for ball in self.body:
            start = cmds.getAttr(ball+'.translateX')
            cmds.setKeyframe(ball, value = start, attribute = 'translateX', t=0)
            cmds.setKeyframe(ball, value = start+howFar, attribute = 'translateX', t=100)


# This creates three Worm objects from the same class (Worm)
littleWorm = Worm(3)
littleWorm.create()
littleWorm.move(-7)
littleWorm.setColor()

mediumWorm = Worm(5)
mediumWorm.create()
mediumWorm.move(1)
mediumWorm.setColor()

bigWorm = Worm(7)
bigWorm.create()
bigWorm.move(11)
bigWorm.setColor()

littleWorm.walk(5)
mediumWorm.walk(20)
bigWorm.walk(10)

# sphere = cmds.polySphere()
# cmds.setKeyframe(sphere[0], value = 0, attribute = 'translateX', t=0)
# cmds.setKeyframe(sphere[0],value = 10, attribute = 'translateX', t=100)
```

Listing 8.2: Adding a “walk" to the Worm class (8.2_wormClass.py)

8.3 Summary

This was a very brief overview just to get you familiar with Object Oriented Programming and the structure for defining classes and creating objects. OOP is very powerful and a elegant way to encapsulate attributes and behavior.

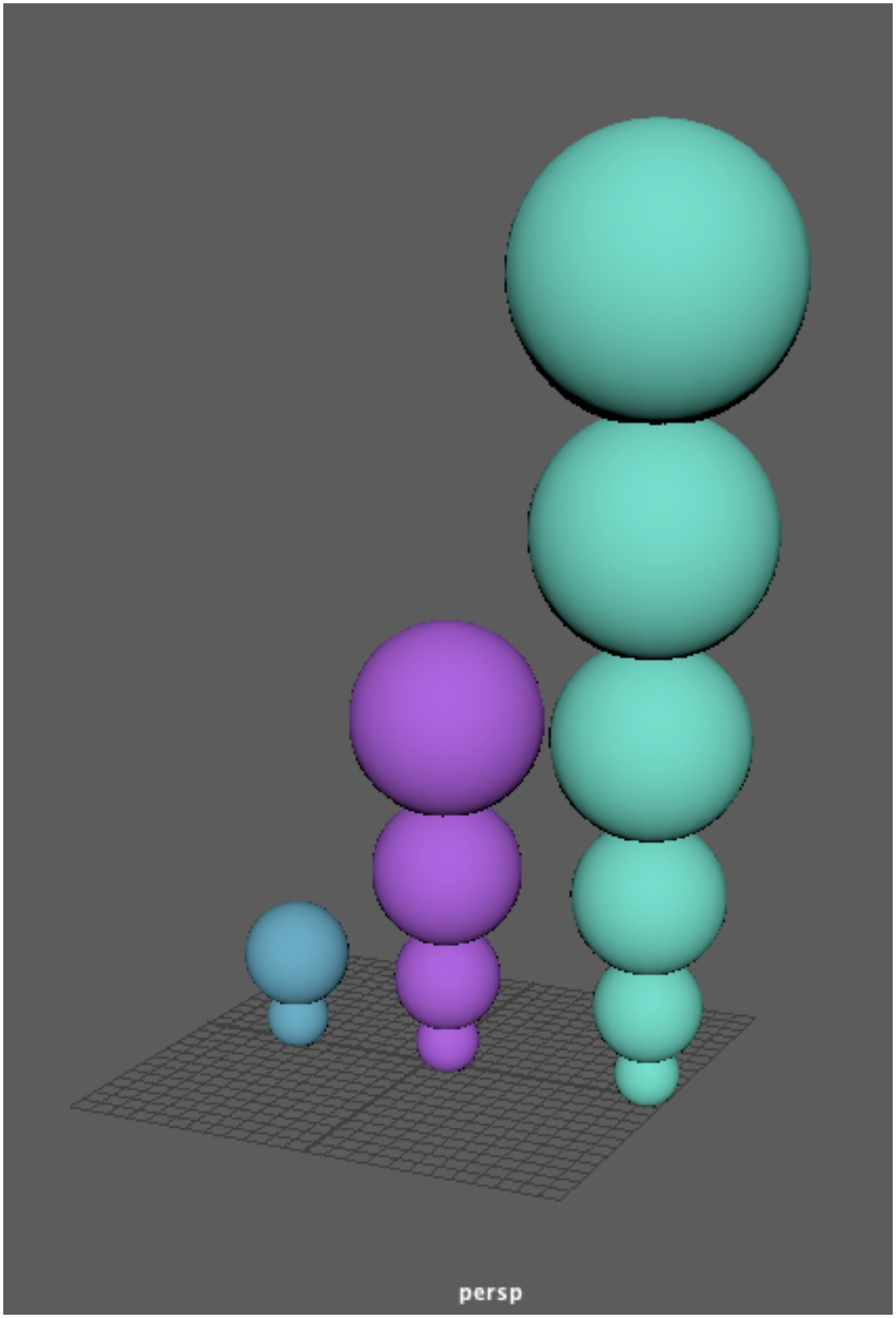

Figure 8.1: The result of creating three worm objects, each time this code is run the colors will be different as the shader is assigned randomly.

9
Resources

9.1 Resource Links

1. Python in Maya, Autodesk
2. Using Python, Autodesk
3. Maya Python Commands Documentation, Autodesk
4. Maya Programming with Python Cookbook, Adrian Herbez
5. Beginning Python for Maya, Chris Zurbrigg
6. Python Scripting for Maya Artists, Chad Vernon
7. Maya Python for Newbies