Please do not redistribute slides without prior permission.

Getting Started with Scripting in Python

16:00 - 16:50 Fri, October 27, 2023

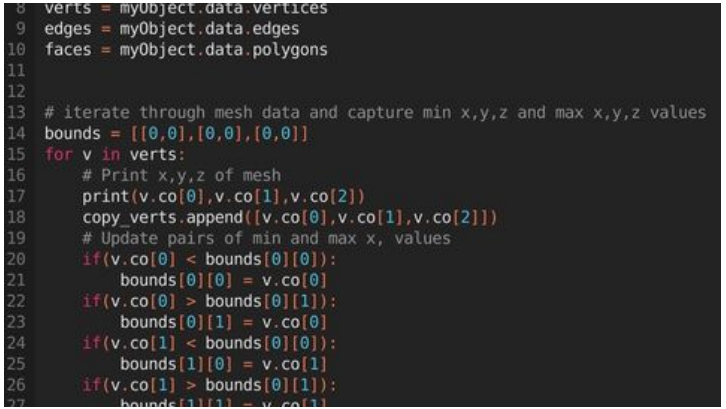
50 minutes | Introductory Audience

Social: @MichaelShah
Web: mshah.io
Courses: courses.mshah.io
YouTube: www.youtube.com/c/MikeShah

Abstract

The abstract that you read and enticed you to join me is here!

Blender 3D is a powerful tool for 3D modeling, animation, rigging, texturing, drawing, vfx, and more -- but what happens when a feature is not available in your respective domain? Good news -- you can create it yourself! In this talk, I will be showing beginners how they can get started creating their first add-on to the Blender 3D ecosystem using Python. This talk will show you how to get started with the scripting interface for artists with minimal programming experience, or programmers who want to write tools that integrate into the Blender 3D ecosystem. Folks will leave this presentation understanding how to write, package, and find more information to develop awesome scripts where they need!



```
verts = myObject.data.vertices
edges = myObject.data.edges
faces = myObject.data.polygons


# iterate through mesh data and capture min x,y,z and max x,y,z values
bounds = [[0,0],[0,0],[0,0]]
for v in verts:
    # Print x,y,z of mesh
    print(v.co[0],v.co[1],v.co[2])
    copy_verts.append([v.co[0],v.co[1],v.co[2]])
    # Update pairs of min and max x, values
    if(v.co[0] < bounds[0][0]):
        bounds[0][0] = v.co[0]
    if(v.co[0] > bounds[0][1]):
        bounds[0][1] = v.co[0]
    if(v.co[1] < bounds[0][0]):
        bounds[1][0] = v.co[1]
    if(v.co[1] > bounds[0][1]):
        bounds[1][1] = v.co[1]
```

Warning -- this talk may take you on a journey of spending even more time using Blender 3D to create awesome creations.

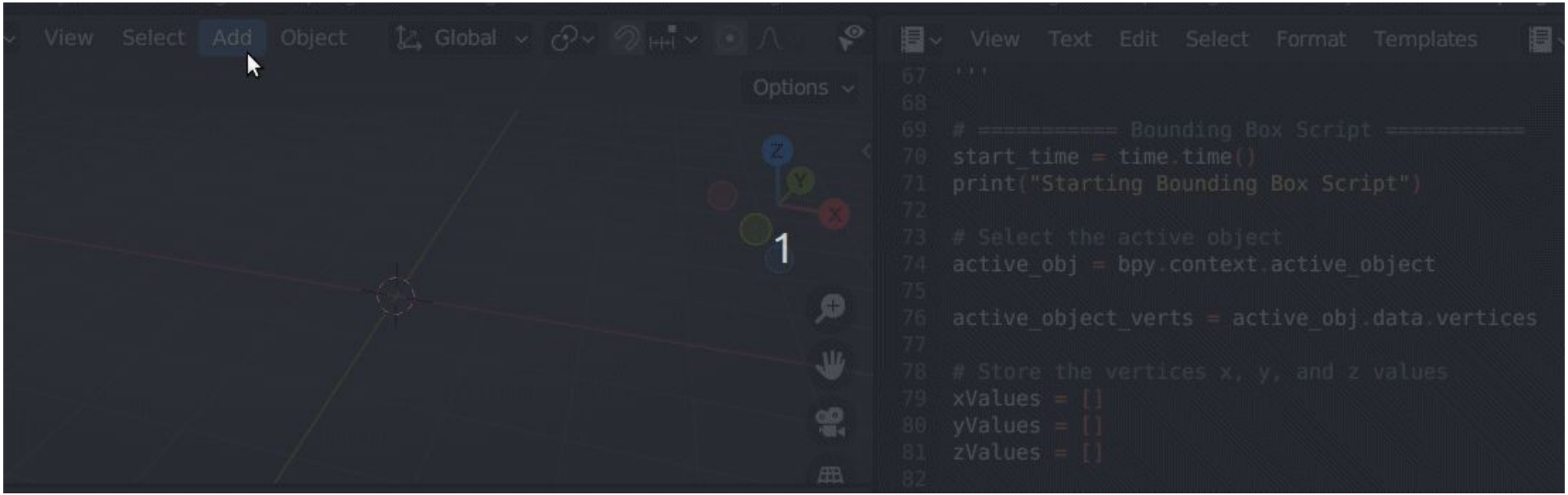
E	Rated ‘E’ For Everyone!
	(Yup, let’s continue to make Blender3D fun for everyone involved)

Here is what we are creating!

(So you know if you should stick around or hop into another session)

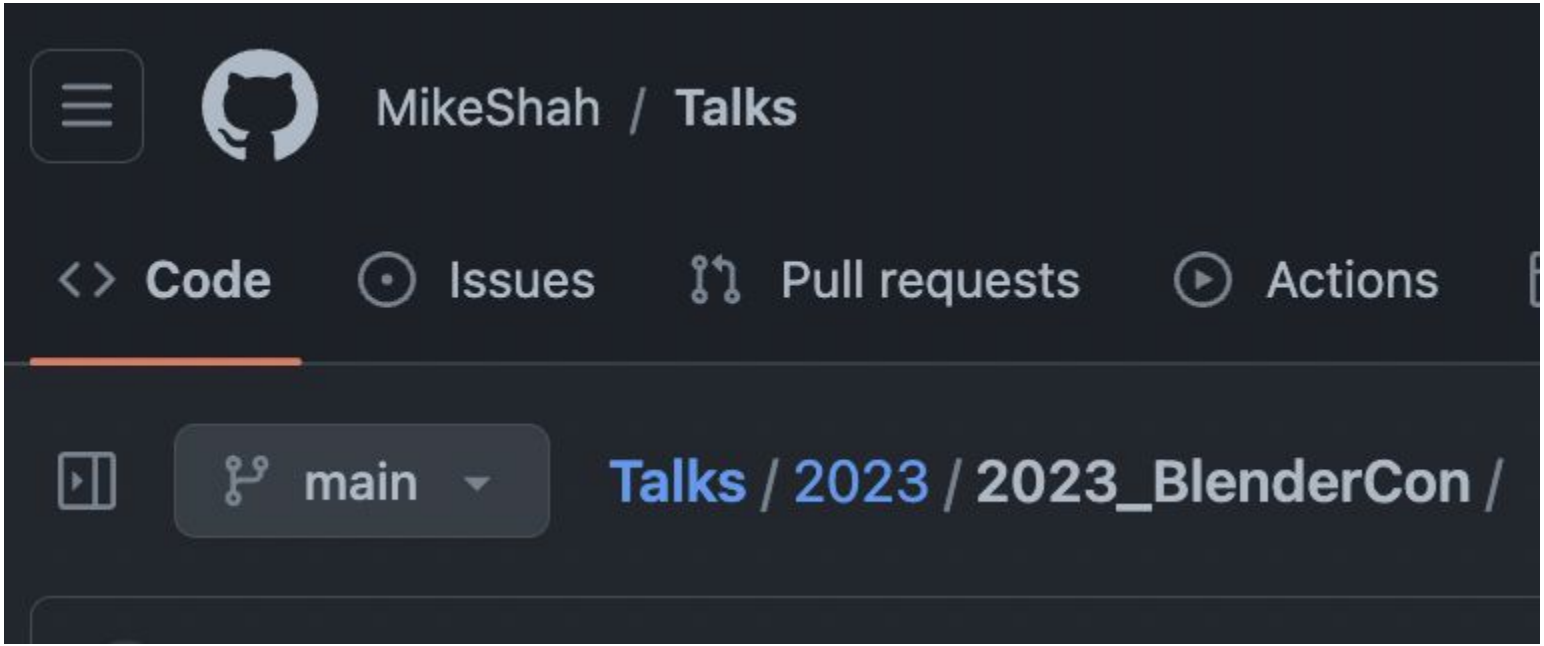
Result of Today's Presentation

- Creating a Bounding Box programmatically in Python



Code for the talk (or Google my name and find talk listed on website)

- Located here: https://github.com/MikeShah/Talks/tree/main/2023/2023_BlenderCon



Your Tour Guide for Today

by Mike Shah

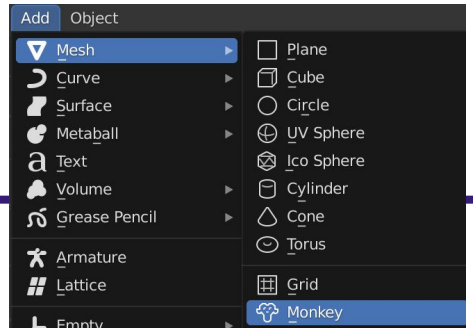
- Associate Teaching Professor at Northeastern University in Boston, Massachusetts.
  - I **love** teaching: courses in computer systems, computer graphics, geometry, and game engine development.
  - My research is divided into computer graphics (geometry) and software engineering (software analysis and visualization tools).
- I do consulting and technical training on modern C++, DLang, Concurrency, OpenGL, and Vulkan projects
  - Usually graphics or games related -- e.g. Building 3D application plugins
- Outside of work: guitar, running/weights, traveling and cooking are fun to talk about

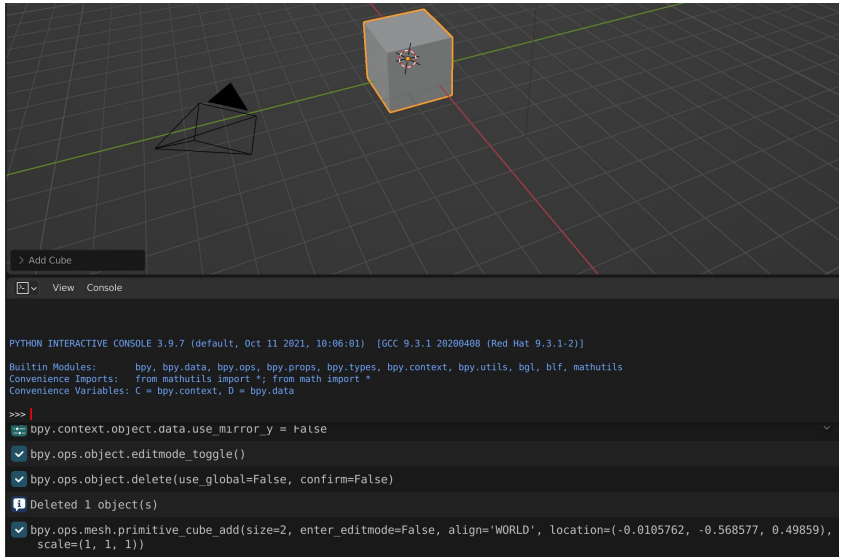
Web
www.mshah.io
YouTube
https://www.youtube.com/c/MikeShah
Non-Academic Courses
courses.mshah.io

Programming Blender 3D

Origin of this Talk

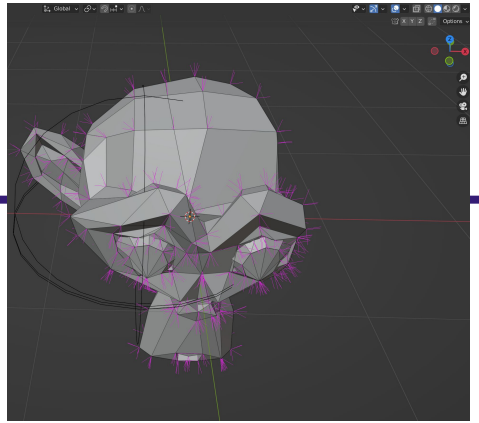
- The idea of this talk was born out of a computational geometry course that I teach
  - Within that course we implement several geometry algorithms in C++ and the SDL2 library in two-dimensions
- In order to start implementing in 3D however, I could not assume students knew OpenGL/Vulkan/Metal/D3D
  - So what better tool than Blender 3D which had many mesh operations and an easy scripting interface to access them.
  - Teaching students a concrete skill (i.e. Blender 3D) is also a win for me!

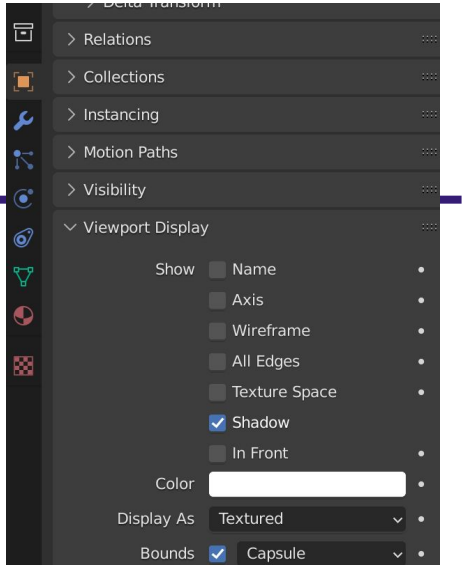


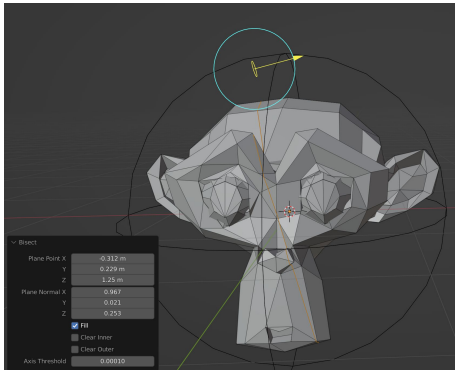


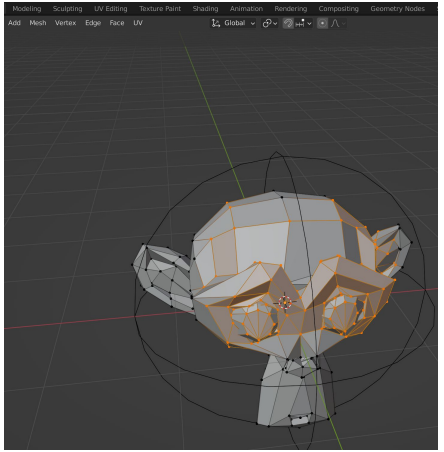
Brainstorming (1/2)

- So I thought of several ideas of how to get students started:
  - Computing Normals
  - Bisection
  - Convex Hull
  - Bounding Boxes
- I settled on bounding boxes, as it touches on enough interesting ideas for programming in Blender 3D
  - The rest remained candidates for incorporating into a final project!
  - (And homework for you now!)





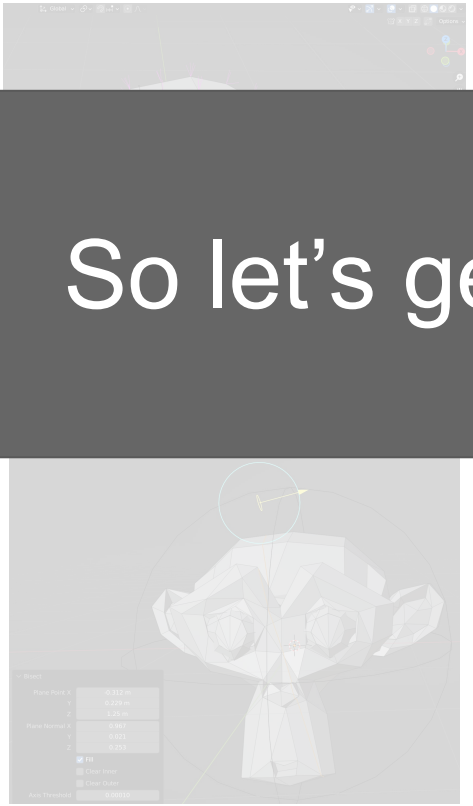




Brainstorming (2/2)

- So I thought of several ideas of how to get students started:
  - Computing Normals
  - Bisection
  - Convex Hull
  - **Bounding Boxes**
- I settled on bounding boxes, as it touches on enough interesting ideas for programming in Blender3D
  - The rest remained candidates for incorporating into a final project!
  - (And homework for you now!)

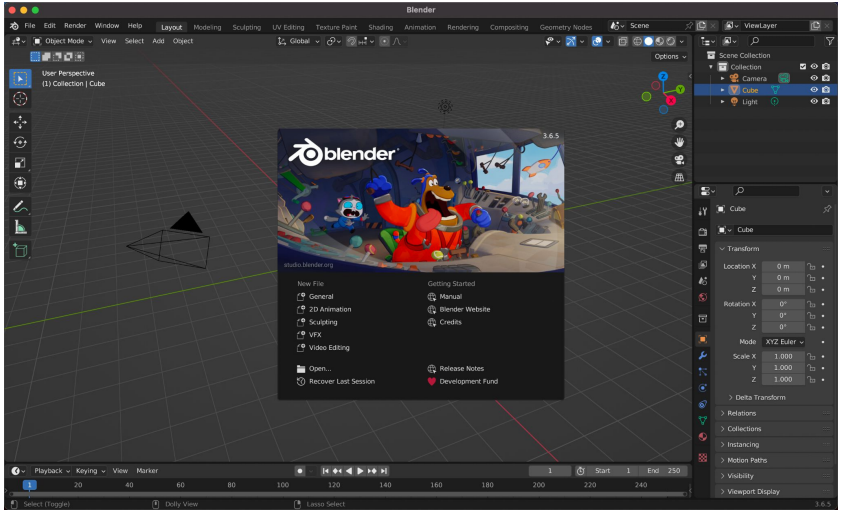
So let’s get started!



Writing Python Scripts in Blender 3D

Install Blender 3D

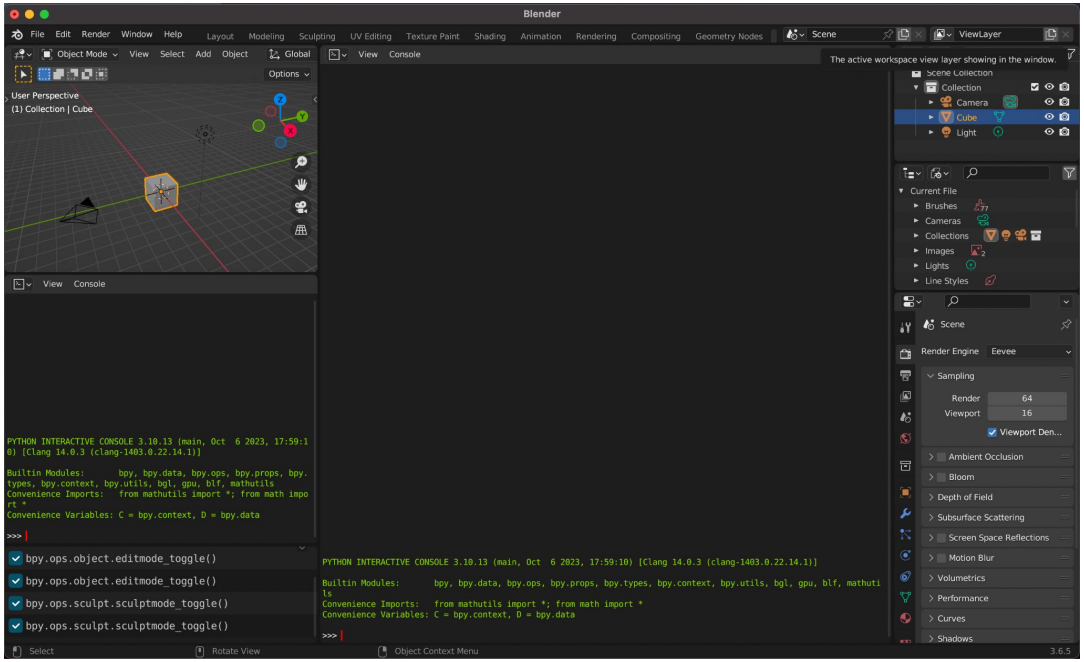
- I'll assume you have installed Blender 3D
  - I'll write scripts using Blender 3.6.5, but these scripts should largely be compatible with most every version of Blender 3.x.x
    - Nothing too fancy going on today
- The last assumption I'll make is that you have used Blender 3D at least a little bit
  - Minimum Requirements: You can navigate with the mouse, extrude some faces on a cube, and have spent a few hours in the program
  - But that's about it! That's great news if you're a programmer building plugins to support a project, and great news if you're already an expert artist!



https://www.blender.org/download/

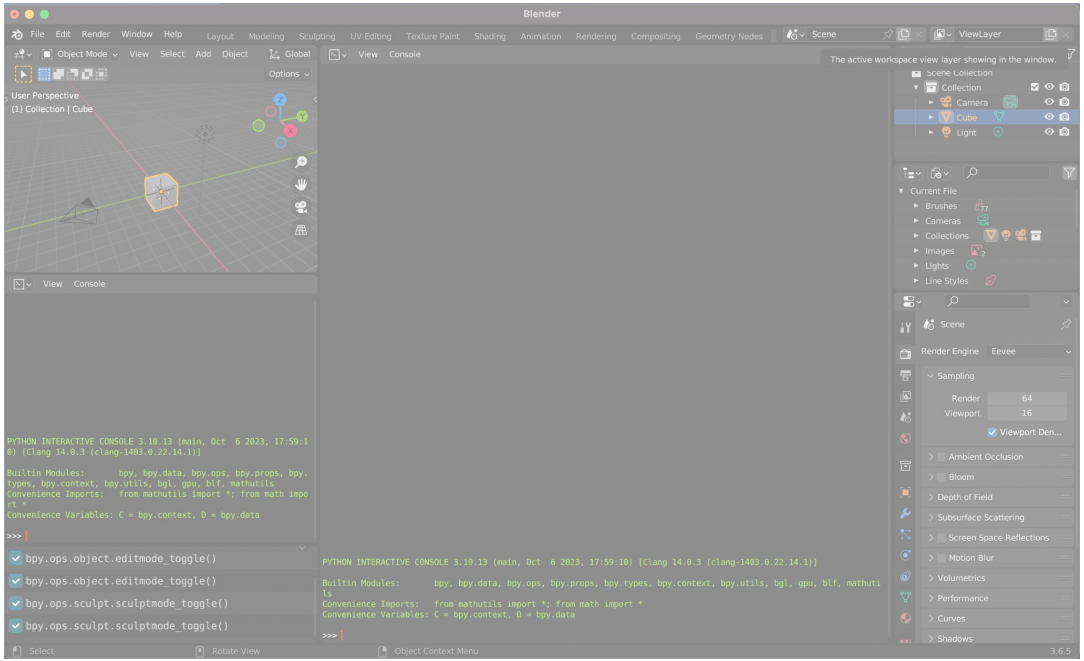
Scripting Layout (1/5)

- We are primarily going to be working from the **Scripting Workspace**
  - Here's what the scripting layout looks like



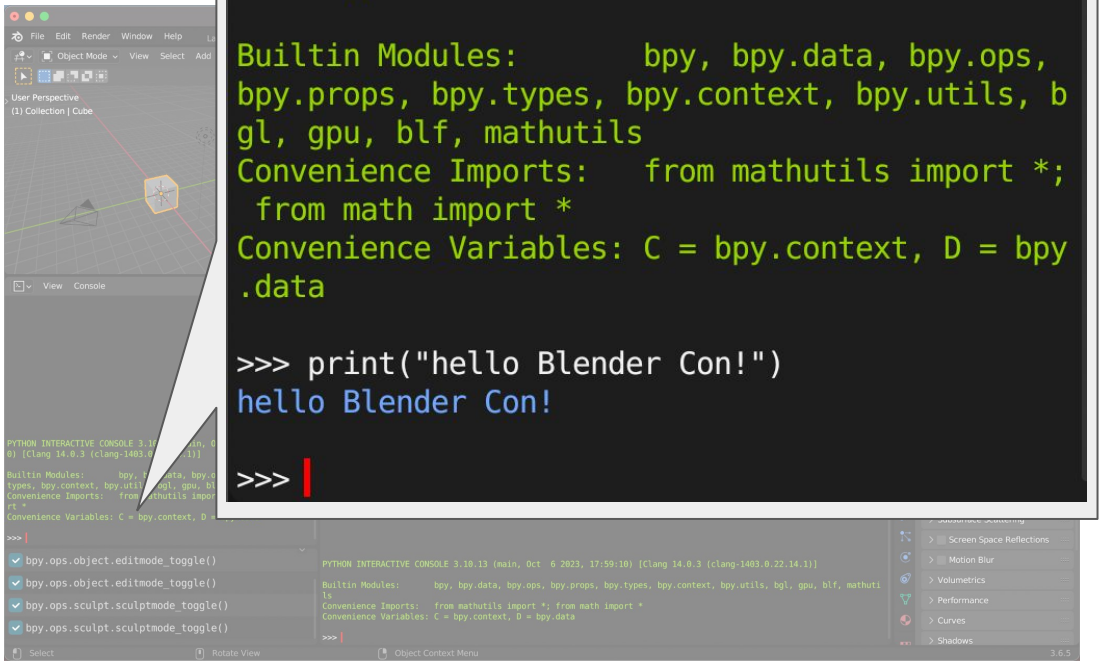
Scripting Layout (2/5)

- **Script Workspace**
  - For scripting!
- Python Console
  - Useful for typing in commands, querying information, and getting fast feedback
- Info Log
  - Tells you the results of operations occurring in the viewport (e.g. moving around)
- Text Editor
  - Used for executing larger scripts



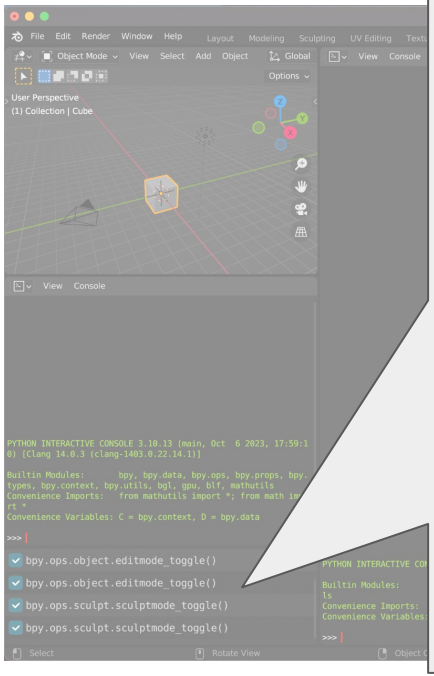
Scripting Layout (3/5)

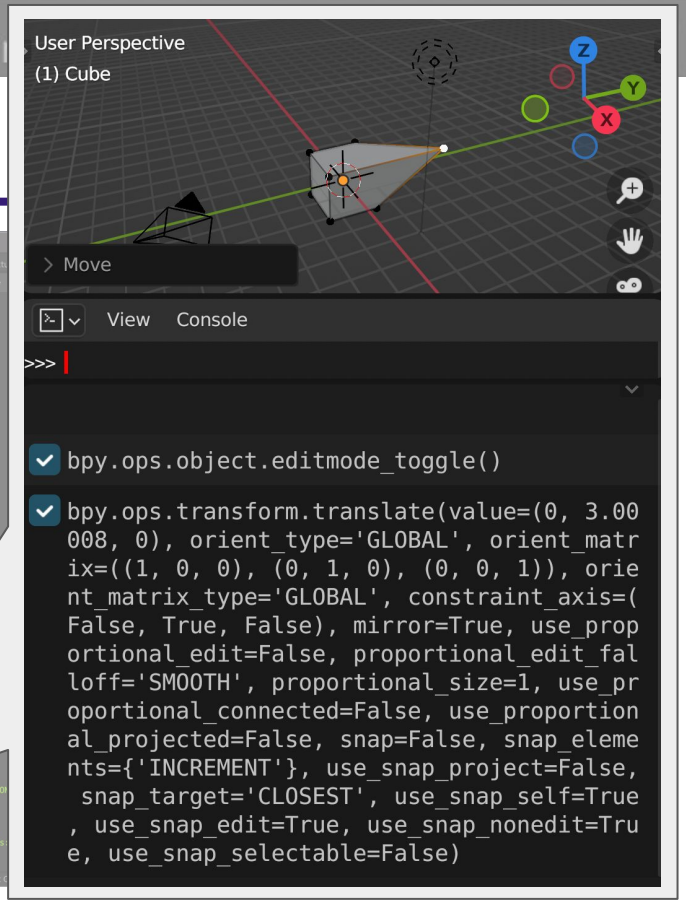
- Script Workspace
  - For scripting!
- **Python Console**
  - Useful for typing in commands, querying information, and getting fast feedback
- Info Log
  - Tells you the results of operations occurring in the viewport (e.g. moving around)
- Text Editor
  - Used for executing larger scripts



Scripting Layout (4/5)

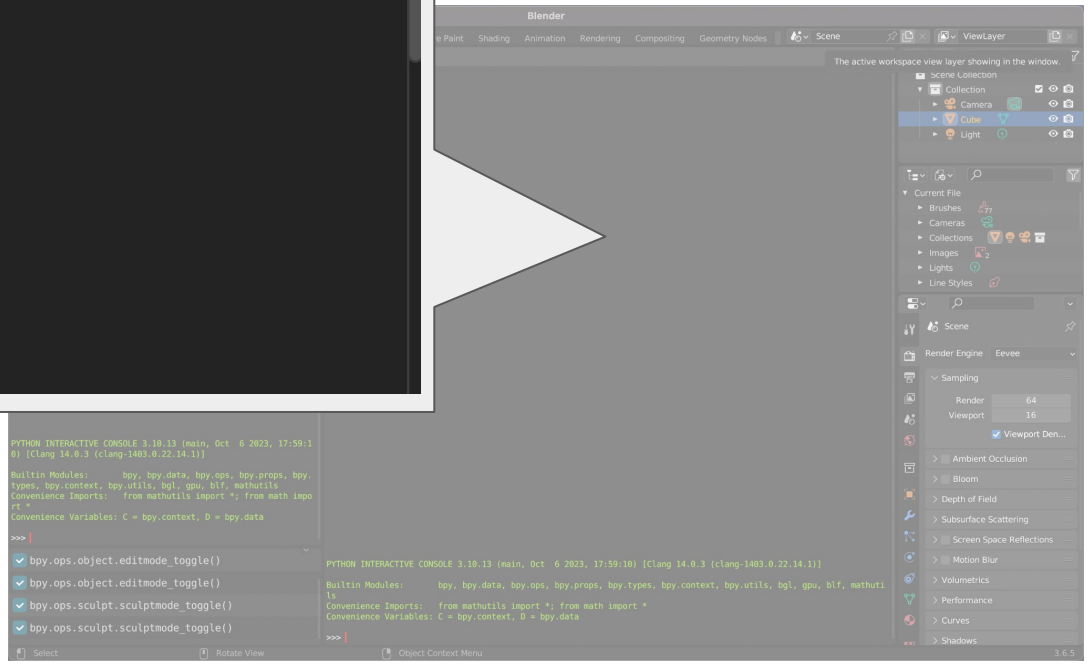
- Script Workspace
  - For scripting!
- Python Console
  - Useful for typing in commands, querying information, and getting fast feedback
- **Info Log**
  - Tells you the results of operations occurring in the viewport (e.g. moving around)
- Text Editor
  - Used for executing larger scripts





View Text Edit Select Format Templates script.py

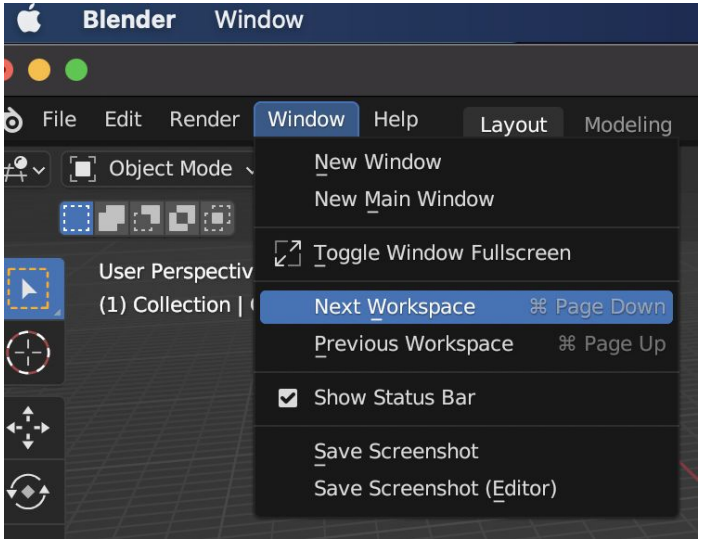
```python
# Note:
# We do have a way to compute the bounding box
# bpy.context.selected_objects[0].bound_box[0][0]
import bpy

import time

start_time = time.time()

print("Starting Script")

# Select the active object
myObject = bpy.context.active_object

# Let's print out the name of the object
print(myObject.name)
```

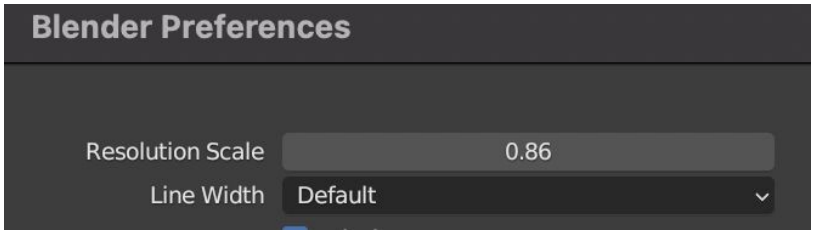


- Info Log
  - Tells you the results of operations occurring in the viewport (e.g. moving around)
- **Text Editor**
  - Used for executing larger scripts

(Aside) If you cannot find the Scripting Workspace

- Scripting Workspace is usually the last tab.
  - Just scroll over the menu bar and scroll your mouse wheel
  - Or otherwise use ‘page down’
- If you’re on a Mac laptop with a small screen, you can navigate to the next workspace
  - Mac users without a page down button:
    - `Cmd + fn + down`
- Another option is to scale down your display a bit
  - Edit -> Preferences -> Adjust Resolution Scale

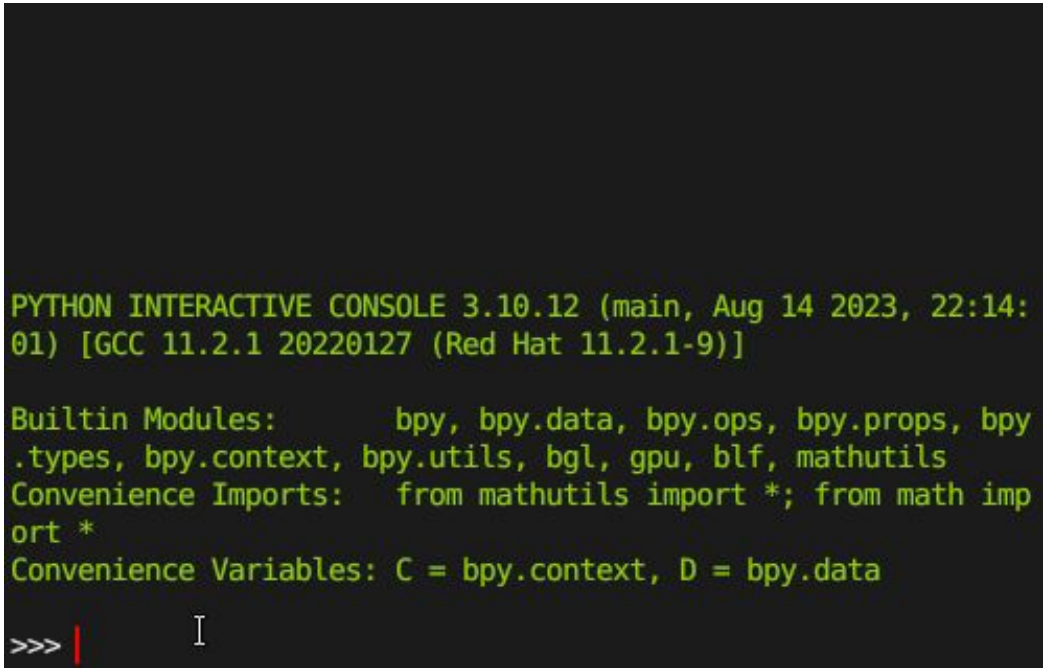




Your First (But not last!) Blender 3D Python Script

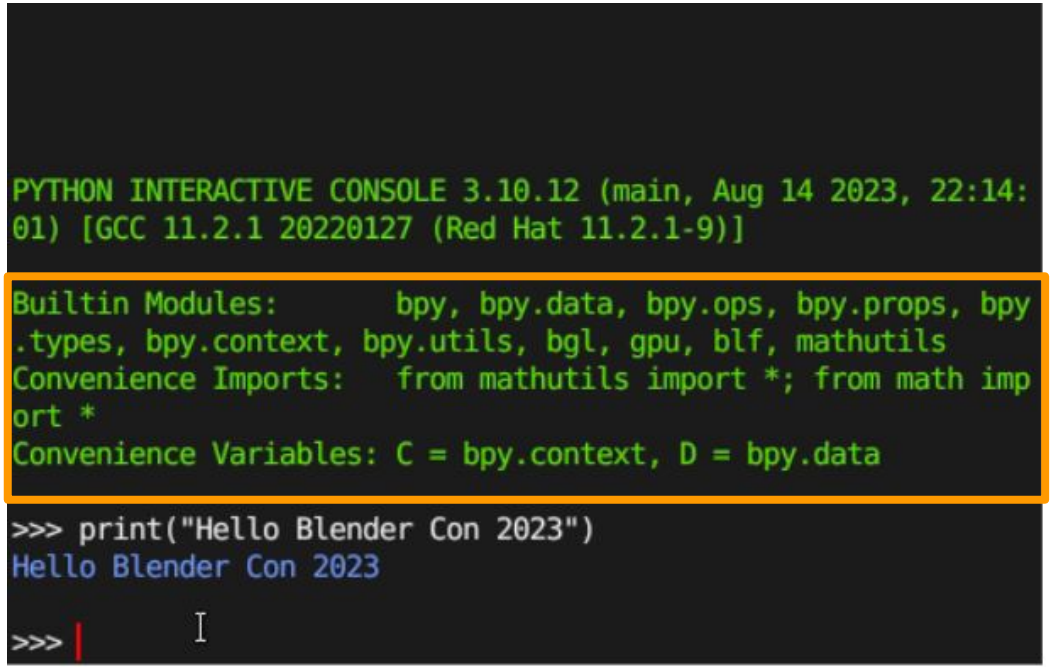
Your First Script (1/2)

- From the **Python Console** you can type in your first command:
  - `print("some_string")`
    - This should output text to the console
- Wonderful -- congratulations on your first script!



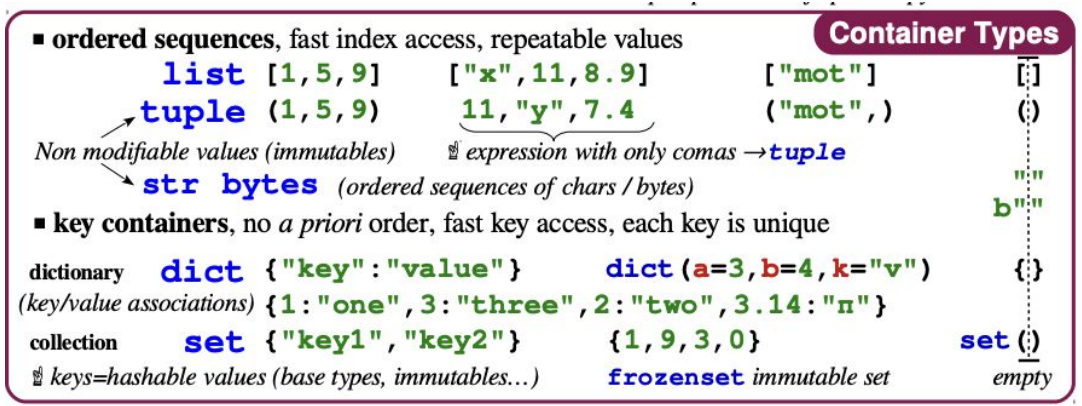
Your First Script (2/2)

- We'll talk a little bit about some of these important **Builtin Modules** throughout this talk.
  - Inside Blender, the python console, automatically loads these for us
  - Later on, in our scripts, we will manually import these modules.
- The other thing to note is that we are using Python `3.10.13`
  - Your version may differ, but you'll want a relatively recent version of Python
    - i.e. Python 3.10.XX or greater is ideal moving forward



(Aside) Python Cheat Sheet

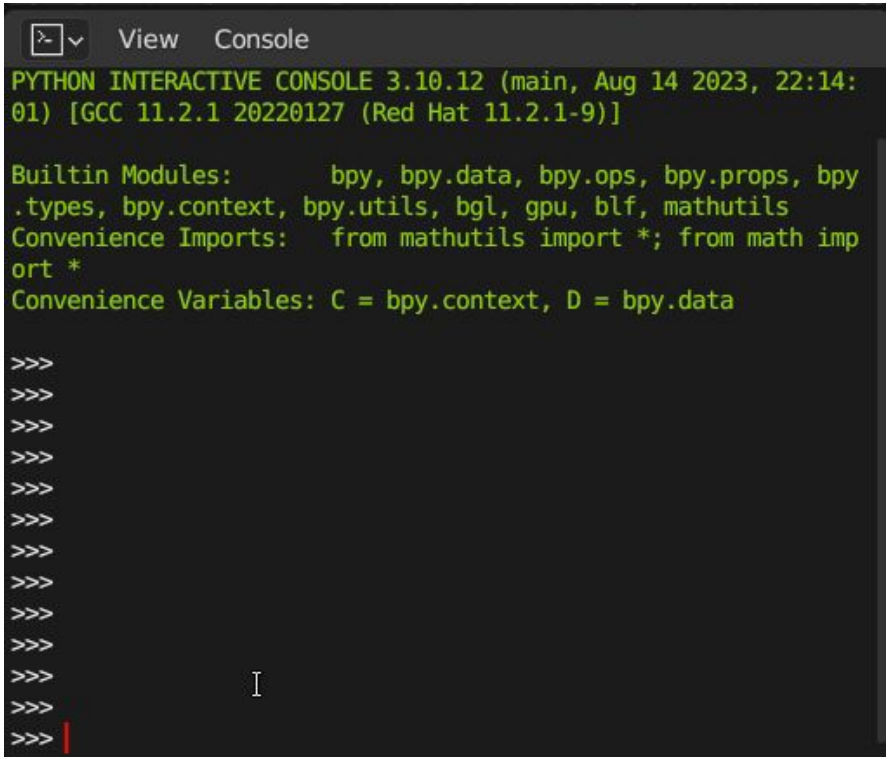
- I'll assume you have some amount of Python
  - Here's a brief cheat sheet on the right
- If you're pretty comfortable with:
  - lists, dictionaries, iteration, and classes you're all ready!



https://perso.limsi.fr/pointal/_media/python:cours:mementopython3-english.pdf

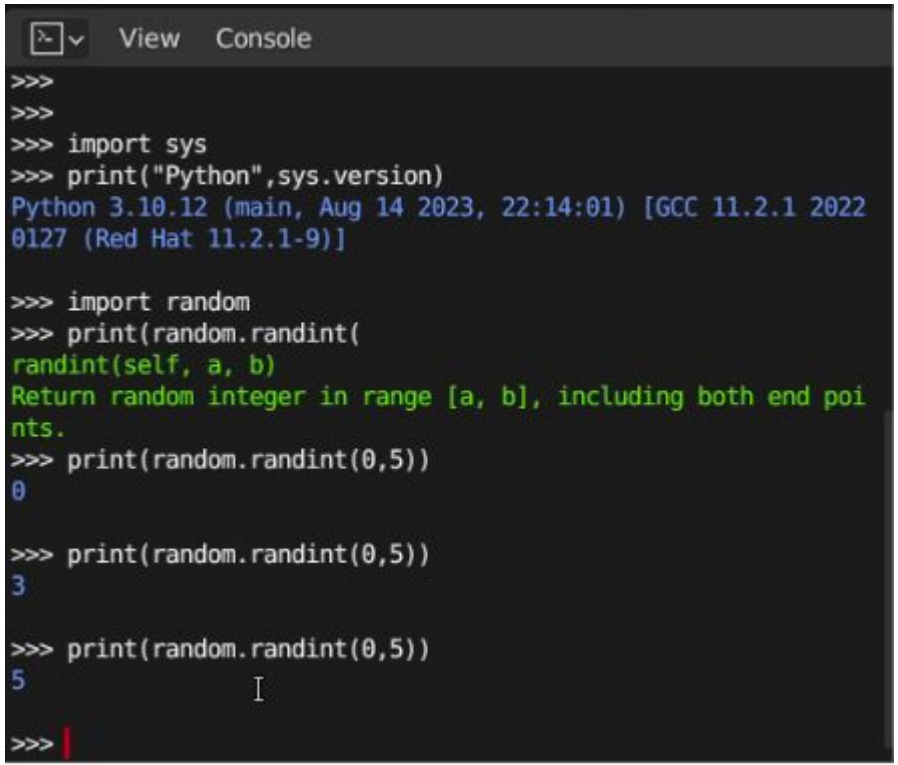
The Power of Python at Your Fingertips! (1/2)

- At this point -- you have all of Python available!
  - The example to the right shows importing ‘sys’ and ‘random’ libraries.
  - The ‘`sys.version`’ tells us what version of Python we have (incase you missed it)
    - Generally speaking, we want Python version 3+ for this tutorial.
  - For printing the random numbers, try repeating the command a few times



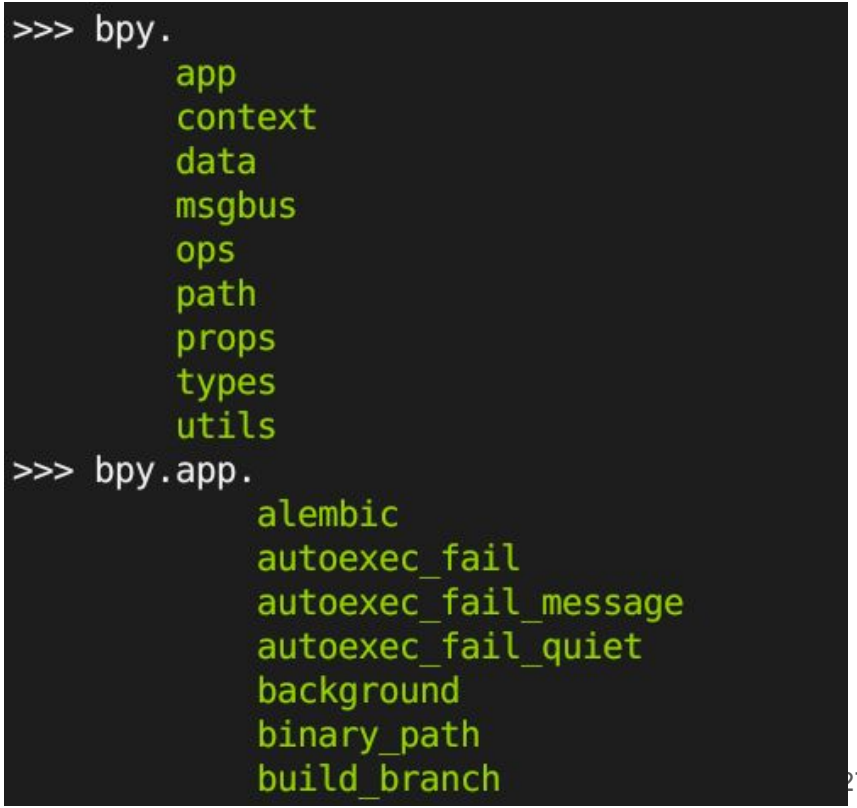
The Power of Python at Your Fingertips! (2/2)

- (Still capture of the code)



(Aside) Console Productivity Tip(s)

- To save yourself time, and re-execute a command, press the ‘up’ and ‘down’ arrow keys to cycle between your command history
  - Pressing ‘enter’ again will execute the command again (try with the random numbers)
- Use ‘tab’ to autocomplete text that you start typing.
  - This is a big time saver for typing out functions, variable names, etc.
    - As a learner, this is also useful for exploring which commands are available.



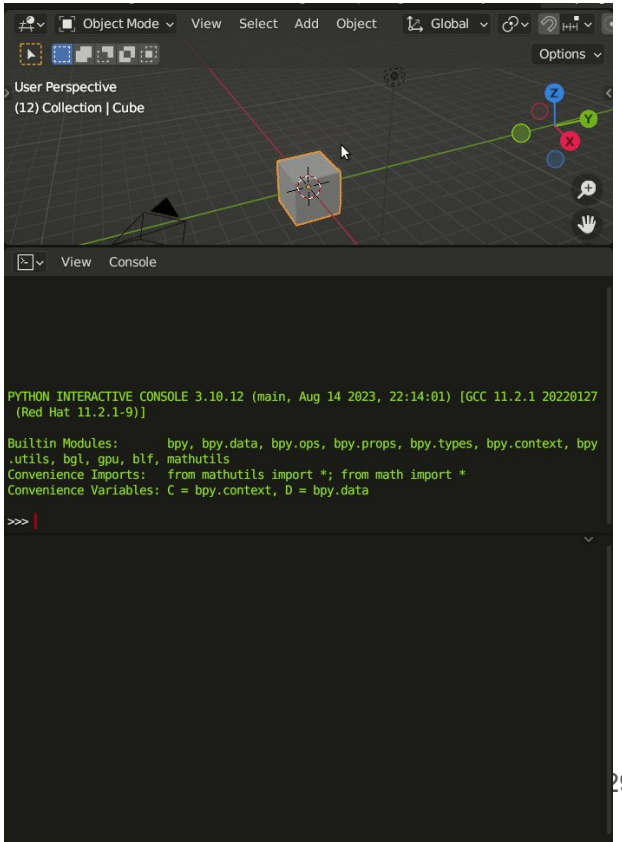
Another Script -- Script for Timing

- Again, demonstrating usage of Python
  - It's nice feedback to the user
  - I've found if you have a timer for when the operation starts ,and some sort of reporting when the operation is finished
    - From a development standpoint -- it's useful for performance, and otherwise knowing when your operation is done as well
  - But -- it's not just Python that we have access to


```
# Text for copy & pasting
import time
start_timer = time.time()
print("elapsed",time.time() - start_timer)
```

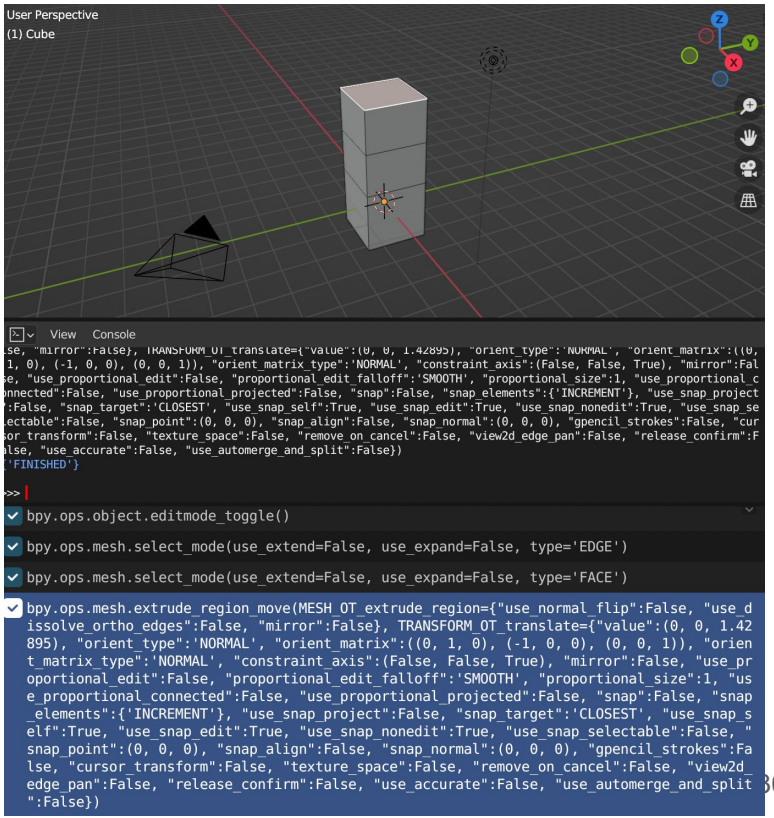
The Power of Blender at Your Fingertips! (1/2)

- What's also very neat about Blender3D is learning some of the commands 'as you normally use blender'
  - Take a moment to modify the geometry of the cube
  - As you modify the geometry, you'll observe the **info log** is updated!
- Wow -- observe that we can see the script actions as they take place!



The Power of Blender at Your Fingertips! (2/2)

- Exercise: Try copying and pasting the previous ‘extrude’ command from the **info log**, into the **python console** which repeats the extrude of the selected face.



Getting Help on Your Journey (1/2)

- **help(...)**
  - From the **Python Console** you can type ‘help’ on any module, function, class, or even a variable.
  - Exercise: Try help(bpy), help(bpy.data)
    - (Remember, these modules have been imported for us already)
    - You can use ‘help’ on any module to start exploring some of the ‘classes’ and ‘functions’ available.
- **type(...)**
  - This is useful for querying the type of something that has already been created.

```
>>> help(bpy)
Help on package bpy:

NAME
    bpy - Give access to blender data and utility functions.

PACKAGE CONTENTS
    ops
    path
    utils (package)

SUBMODULES
    props
    types
```

```
>>> myObject = bpy.context.active_object
>>> type(myObject)
<class 'bpy_types.Object'>

>>>
```

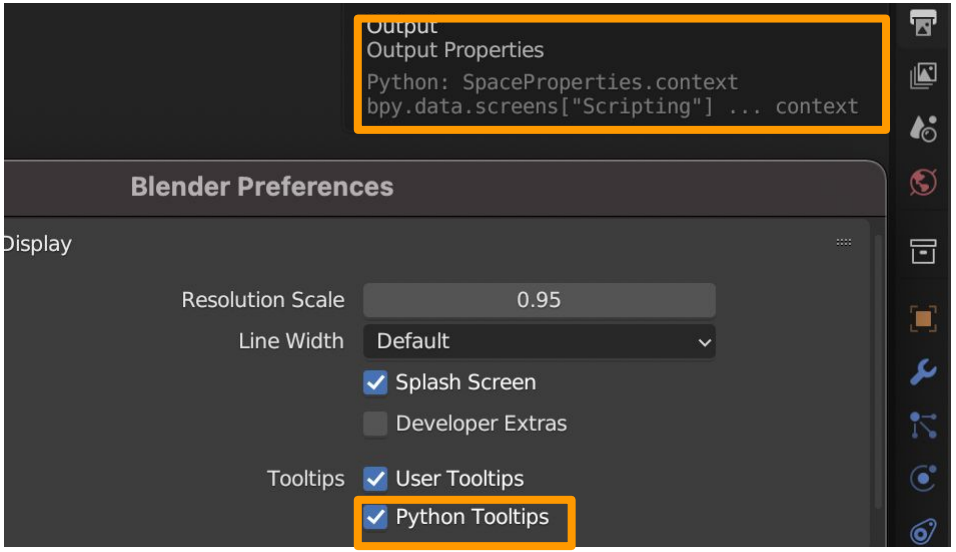
Getting Help on Your Journey (2/2)

- The Python API Documentation online is a great resource
  - Note: I recommend downloading a copy to be used offline for faster browsing
    - (Also useful for long airplane rides :))

https://docs.blender.org/api/current/index.html

Enabling ‘Python Tooltips’ for Developers

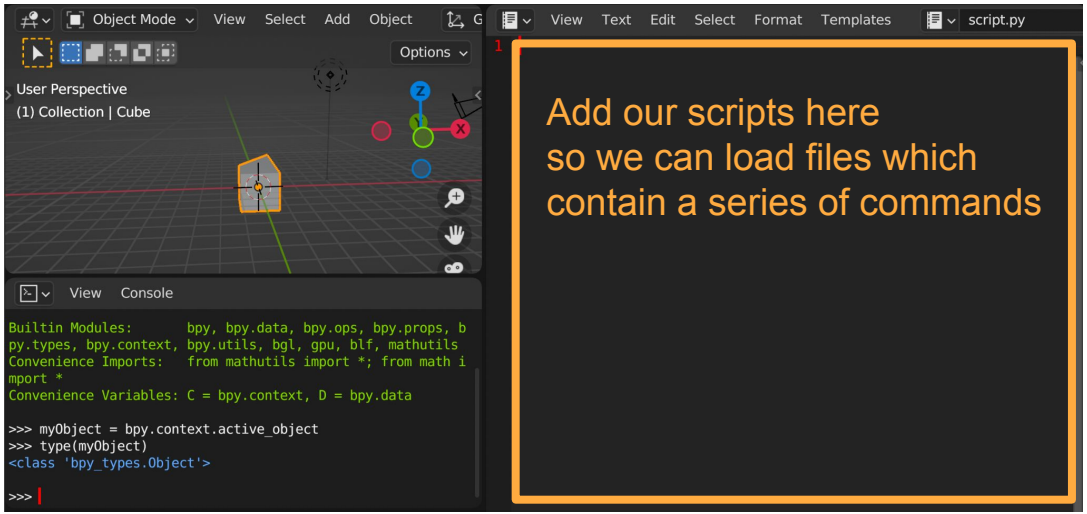
- Another very useful way to explore the Python API is to **enable ‘Python Tooltips’**
  - This is done in the ‘preferences’ modal.
  - Enabling ‘Python Tooltips’ will show you additional information about various tools you are use to clicking on -- and guide you to the python API.
    - (See example in the top-right)



Using the Internal Text Editor

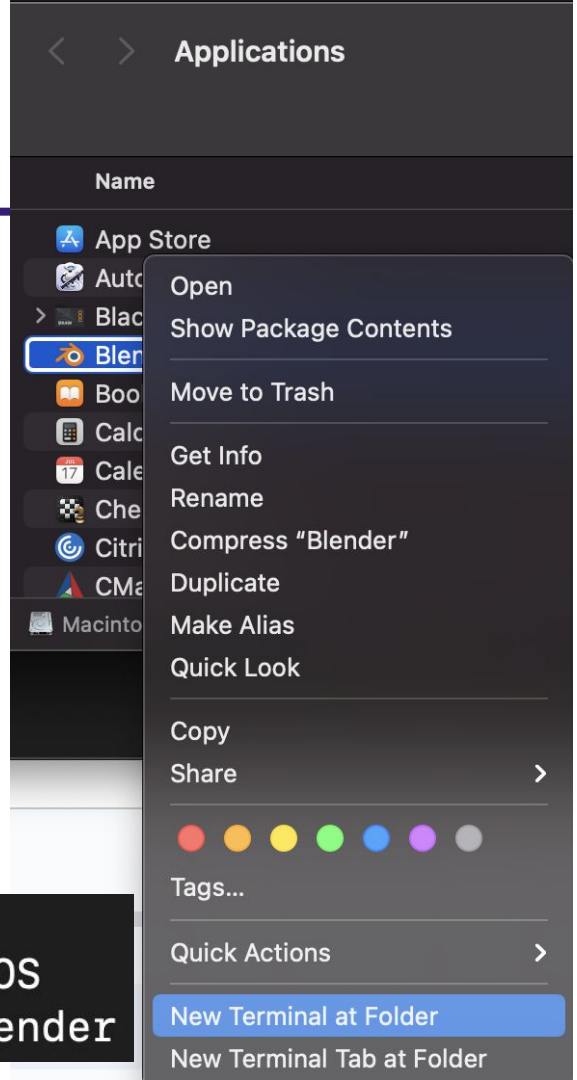
Blender 3D Internal Text Editor

- At some point we likely will want to be able to create larger scripts that execute a series of commands to perform some work.
  - For this presentation we'll write our scripts in the Text Editor
  - Note: You can use your favorite Text Editor (VIM, VSCode, etc.) to also write your scripts.



Tip: Launch Blender from Terminal

- In order to help us debug and ‘print’ out text, it is most useful to launch Blender from the terminal.
  - Then when we execute our scripts we will get text output on the terminal where we launched.
  - On Mac
    - You will then use ‘Option + P’ to run your script
  - On Linux
    - You will use ‘alt+p’



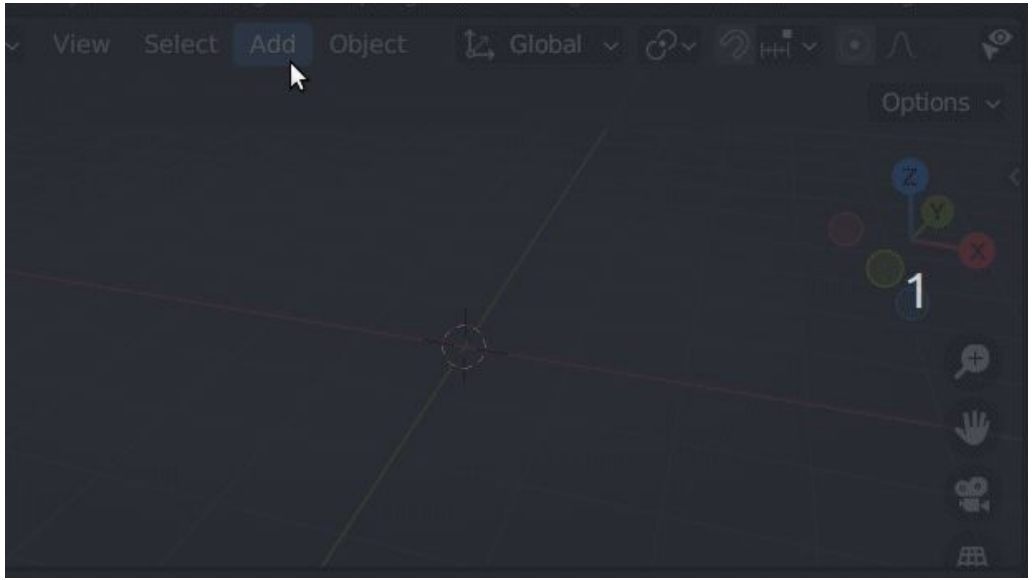

```
mike@Michaels-MacBook-Air MacOS % pwd
/Applications/Blender.app/Contents/MacOS
mike@Michaels-MacBook-Air MacOS % ./Blender
```

Solving the Bounding Box Problem with Python Scripting

Gathering our Tools from the Python API

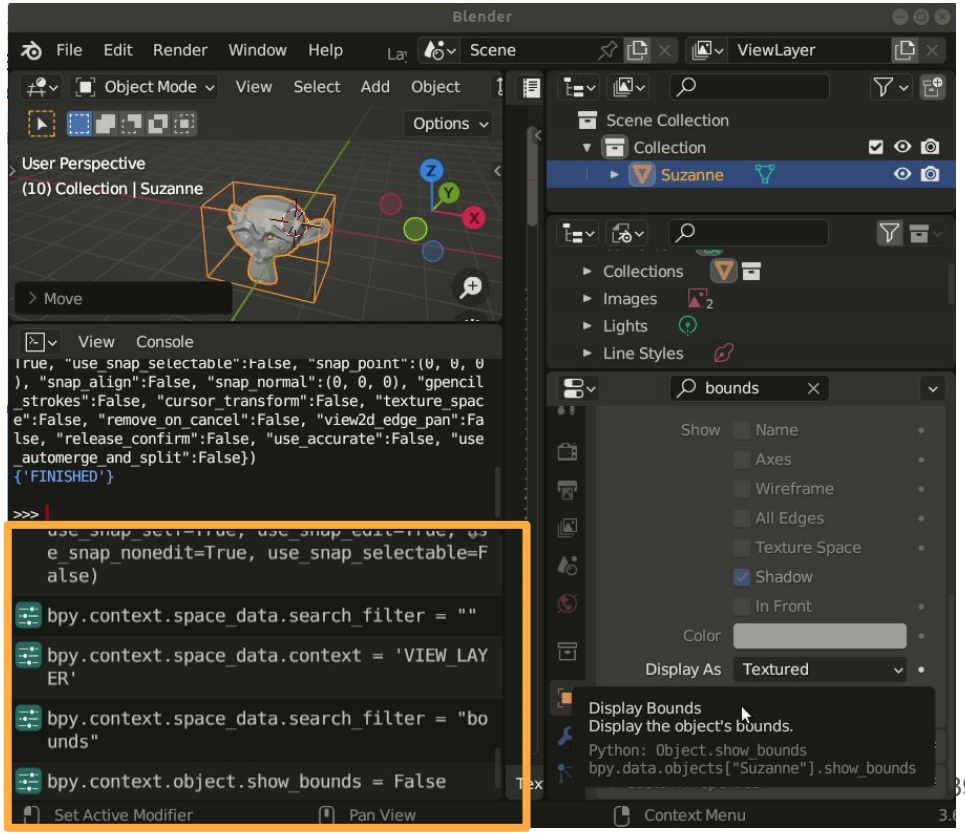
Creating a Bounding Box Programmatically

- So as was shown at the start of the talk, let's begin our journey creating a bounding box
  - Now this is something that Blender3D already has the capability to do
  - However, learning how to do so from scratch will expose us to Blender's API through Python.



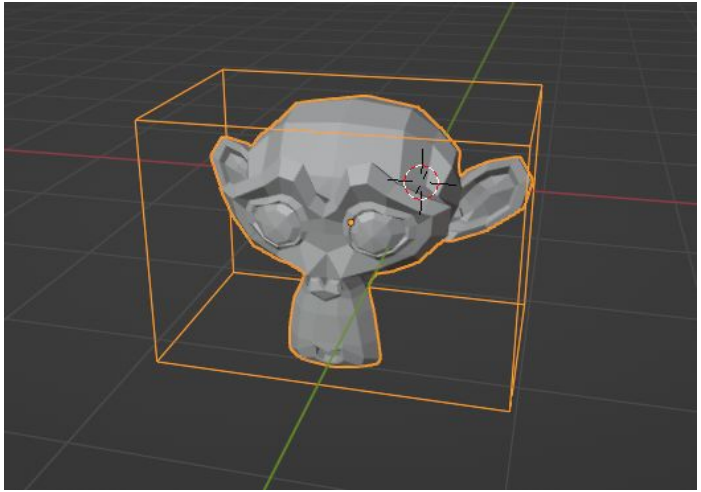
Creating a Bounding Box Programmatically -- built-in

- Now of course *you could* call:
  - `.show.bounds = True`
  - That's not really in the spirit of this assignment...
- However, this does introduce the 'bpy.context' (see the bottom-left of info log) module which is of use



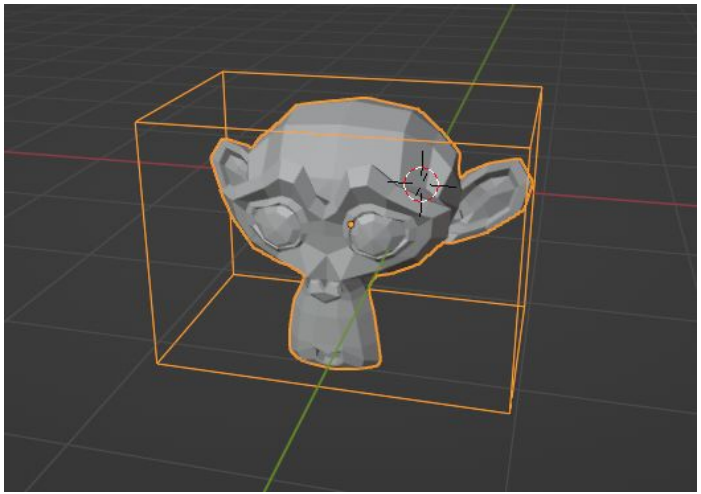
Exercise: How do you compute the bounding box? (1/2)

- Now if you had to compute the bounding box from scratch -- how would you do it?
  - (If you're watching this in the future you can pause the video and write out a solution)
  - For my current audience, I'm going to forward us to one solution -- there's a couple ways to approach this



Exercise: How do you compute the bounding box? (2/2)

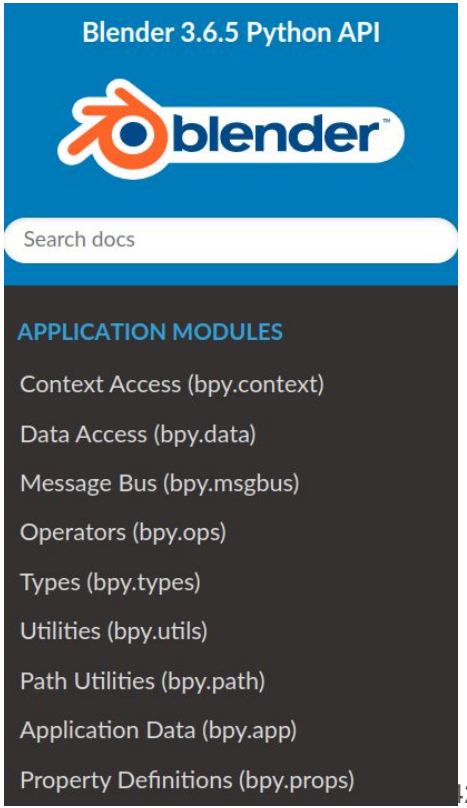
- Simplest solution
  - Iterate through all of the vertices
    - Keep track of both the minimum and maximum x,y,z values
- Another solution for obtaining the bounds is to otherwise use:
  - `myObject.bound_box`
  - This returns the ‘8’ vertices of the bounding box
- (Aside: This is an axis-aligned bounding box, but we can apply a transform to get an oriented-bounding box)



```
>>> myObject.bound_box
bpy.data.objects['Suzanne'].bound_box

>>> print(myObject.bound_box)
<bpy_float[8], Object.bound_box>

>>> print(myObject.bound_box[0])
<bpy_float[3], Object.bound_box>

>>> print("x of first corner:",myObject.bound_box[0][0])
x of first corner: -1.3671875
```

A few Blender Python (bpy) modules of Importance (1/2)

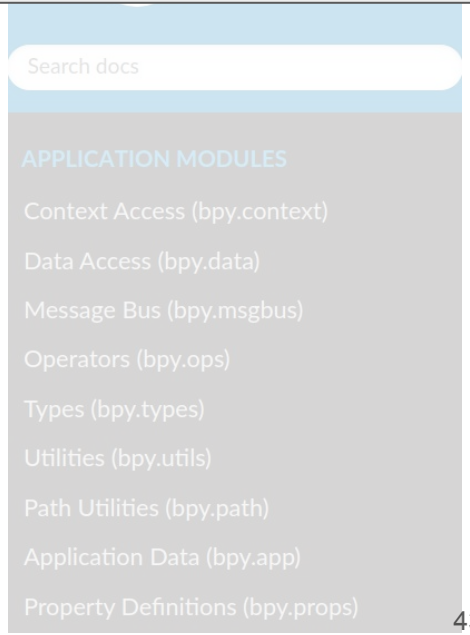
- bpy
  - This is the main module of the programming interface in Blender.
- bpy.context
  - This module captures the current state of the user interaction
    - (e.g. selection or current mode)
  - Note: This is often aliased as ‘C’ for ‘bpy.context’
- bpy.data
  - This is the storage of blender objects
    - Anything found within bpy.data.objects is something that can be displayed in the Blender 3D viewport
      - (e.g. camera, lights, curves, meshes, etc.)
  - Note: This is often aliased to ‘D’ for ‘bpy.data’
- bpy.ops
  - Functions that can be invoked in the interface



A few Blender Python (bpy

- bpy
  - This is the main m… the progr…
- **bpy.context**
  - This module captures the current state of the user interaction
    - (e.g. selection or current mode)
  - Note: This is often aliased as ‘C’ for ‘bpy.context’
- **bpy.data**
  - This is the storage of blender objects
    - Anything found within bpy.data.objects is something that can be displayed in the Blender 3D viewport
      - (e.g. camera, lights, curves, meshes, etc.)
  - Note: This is often aliased to ‘D’ for ‘bpy.data’
- bpy.ops
  - Functions that can be invoked in the interface

- Both of these modules are going to be important for us to work in
  - One for selecting our object of interest
  - The second for getting data

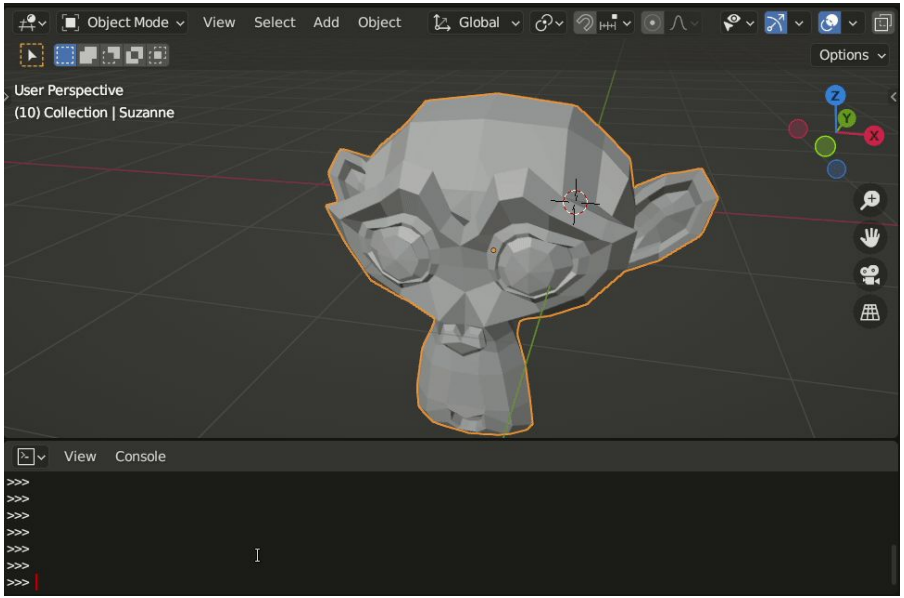


Bounding Box

Implementation

`bpy.context` and selecting the current object

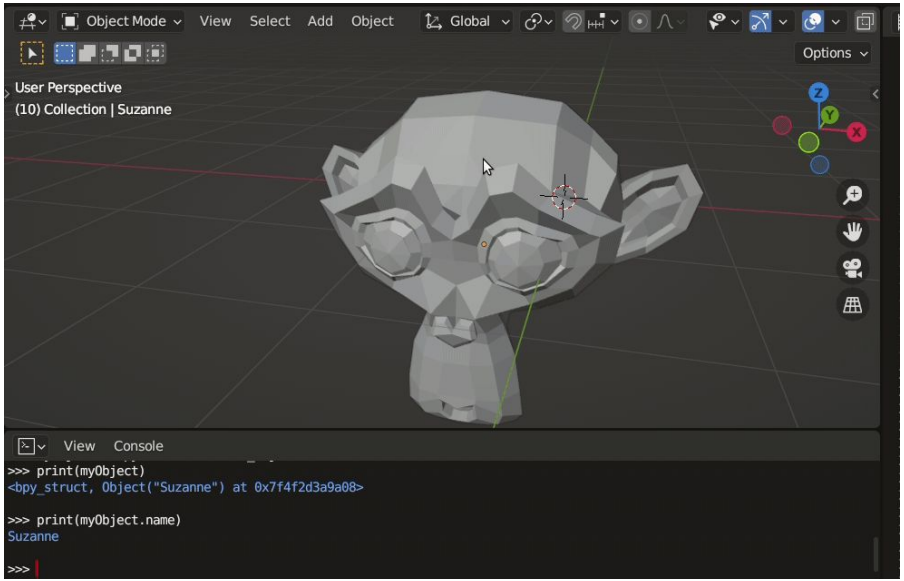
- So again the **`bpy.context`** is useful for telling us what is going on in an ‘area’ of our screen.
- Usually these are ‘read-only’ types of things we can get
  - But it’s very useful for instance if we want to store a variable to our currently selected object
- e.g.
  - `myObject = bpy.context.active_object`



Demonstrates getting a handle to the active object

Acquiring the Geometry of our current object

- As we know, 3D objects are often defined by:
  - vertices, edges, and polygons (3 or more edges)
- # Now let's acquire some data
  - `verts = myObject.data.vertices`
  - `edges = myObject.data.edges`
  - `faces = myObject.data.polygons`
- Note: When we access an objects ‘data’, sometimes you’ll hear this referred to as a **data-block**



Demonstrates getting vertex, edge, and polygon information

Computing the Bounds (1/3)

- I've opted for as simple of an algorithm as possible

```
76  active_object_verts = active_obj.data.vertices
77
78  # Store the vertices x, y, and z values
79  xValues = []
80  yValues = []
81  zValues = []
82
83  # Only compute bounding box based on
84  # the vertices if they are selected
85  for v in active_object_verts:
86      # if v.select == True:
87          xValues.append(v.co[0])
88          yValues.append(v.co[1])
89          zValues.append(v.co[2])
90
91  # Iterate through the values we have stored
92  # and grab the bounds
93  minx =  min(xValues)
94  maxx =  max(xValues)
95  miny =  min(yValues)
96  maxy =  max(yValues)
97  minz =  min(zValues)
98  maxz =  max(zValues)
```

Computing the Bounds (2/3)

- First grab the vertices
  - We're going to want our own 'List' of vertices to work with (and later generate some geometry)
  - Note: I have commented out to only compute bounding box on selected vertices (line 87) -- try to play around with that on your own time ;)
    - Hint: May or may not need to be in edit mode.

```
76  active_object_verts = active_obj.data.vertices
77
78  # Store the vertices x, y, and z values
79  xValues = []
80  yValues = []
81  zValues = []
82
83  # Only compute bounding box based on
84  # the vertices if they are selected
85  for v in active_object_verts:
86      # if v.select == True:
87          xValues.append(v.co[0])
88          yValues.append(v.co[1])
89          zValues.append(v.co[2])
90
91  # Iterate through the values we have stored
92  # and grab the bounds
93  minx =  min(xValues)
94  maxx =  max(xValues)
95  miny =  min(yValues)
96  maxy =  max(yValues)
97  minz =  min(zValues)
98  maxz =  max(zValues)
```

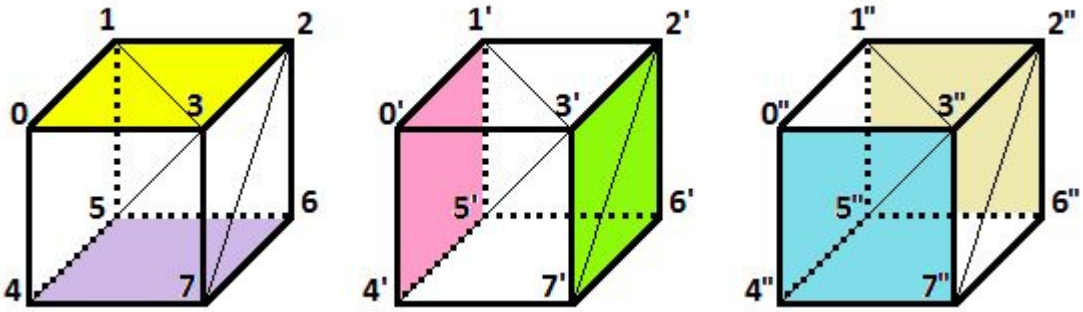
Computing the Bounds (3/3)

- Finally, compute the bounds
  - The min and max functions are useful here for searching through a range

```
76  active_object_verts = active_obj.data.vertices
77
78  # Store the vertices x, y, and z values
79  xValues = []
80  yValues = []
81  zValues = []
82
83  # Only compute bounding box based on
84  # the vertices if they are selected
85  for v in active_object_verts:
86      # if v.select == True:
87          xValues.append(v.co[0])
88          yValues.append(v.co[1])
89          zValues.append(v.co[2])
90
91  # Iterate through the values we have stored
92  # and grab the bounds
93  minx =  min(xValues)
94  maxx =  max(xValues)
95  miny =  min(yValues)
96  maxy =  max(yValues)
97  minz =  min(zValues)
98  maxz =  max(zValues)
```

Creating a ‘Bounding Box’ (1/2)

- Now that we have the boundaries, we need to create a ‘box’ object
- In order to generate a ‘mesh’ we have a few choices
  - Some folks with graphics programming, may go ahead and want to create the ‘indexed cube’ and calculate the vertices, edges, and polygons (with the correct winding order)



https://alg.manifoldapp.org/api/proxy/ingestion_sources/85b8a903-6d37-4a26-b376-552ce0ef528b

Creating a ‘Bounding Box’ (2/2)

- Now that we have the boundaries, we need to create a ‘box’ object
- In order to generate a ‘mesh’ we have a few choices
  - Some folks with graphics programming, may go ahead and want to create the ‘indexed cube’ and calculate the vertices, edges, and polygons (with the correct winding order)
  - A second choice, is to simply generate a cube from blender, and reposition the corner vertices
    - This does the hard work of preserving the connectivity for us.

```
bpy.ops.mesh.primitive_cube_add(enter_editmode=False, align='WORLD', location=(0, 0, 0), scale=(1, 1, 1))
cube_temp = bpy.context.active_object

# Now let's acquire some data
cube_verts = cube_temp.data.vertices
cube_edges = cube_temp.data.edges
cube_faces = cube_temp.data.polygons
```

bmesh

Blender Mesh Format

BMesh (bmesh)

- The BMesh API allows us to work with the internal mesh editing tools in blender.
  - i.e. Basically any operations that you'd like
- Probably most important for us is to just be able to grab data and put it into a mesh.
- There might come a time where you want to perform more interesting operations
  - (Note: When you run a script, you lock the mesh by operating on it, modify the mesh, and returns control to a user)

BMesh Module (bmesh)

This module provides access to blenders bmesh data structures.

Introduction

This API gives access the Blender's internal mesh editing API, featuring geometry connectivity data and access to editing operations such as split, separate, collapse and dissolve. The features exposed closely follow the C API, giving Python access to the functions used by Blender's own mesh editing tools.

For an overview of BMesh data types and how they reference each other see: BMesh Design Document.

```
# This example assumes we have a mesh object selected

import bpy
import bmesh

# Get the active mesh
me = bpy.context.object.data
```

```
>>> type(myObject.data)
<class 'bpy_types.Mesh'>
```

```
# Get a BMesh representation
bm = bmesh.new()   # create an empty BMesh
bm.from_mesh(me)   # fill it in from a Mesh


# Modify the BMesh, can do anything here...
for v in bm.verts:
    v.co.x += 1.0


# Finish up, write the bmesh back to the mesh
bm.to_mesh(me)
bm.free()  # free and prevent further access
```

https://docs.blender.org/api/current/bmesh.html#module-bmesh

Iterating through data

- It’s useful for us to store the vertices, edges, and faces in our own data structure to generate ‘a new mesh’ for our bounding box
  - The code below demonstrates how to ‘iterate’ through each of vertices, edges, and ‘polygons’ (i.e. faces)
    - Please be careful as to **not modify** the original data -- observe we are copying into our own list
    - Modifying a data structure while iterating could be unsafe
      - (‘search iterator invalidation’)
- Note: These blocks of code could be condensed further -- optimize at your level of Python!
  - List comprehension, unzip list, etc.

```
# Now let's acquire some data
cube_verts = cube_temp.data.vertices
cube_edges = cube_temp.data.edges
cube_faces = cube_temp.data.polygons
```

```
cube_temp_verts = []
for v in cube_verts:
    entry = [v.co[0],v.co[1],v.co[2]]
    cube_temp_verts.append(entry)


cube_temp_edges = []
for segment in cube_edges:
    entry = []
    for pair in segment.vertices:
        entry.append(pair)
    cube_temp_edges.append(entry)

# Loop through all of our faces
# and figure out the indices
cube_temp_faces = []
for idx,polygon in cube_faces.items():
    entry = []
    for vertInPolygon in polygon.vertices:
        entry.append(vertInPolygon)
    cube_temp_faces.append(entry)
```

Building Our Bounding Box

- Here is the little hack where I just need to reassign the vertices of our ‘cube’
  - There’s a pattern here you can follow
    - (Hint: It happens look like a truth table if you have taken a logic or discrete math subject)

```
140  # Create the bounding box with some vertex positions setup correctly
141  bounding_verts = cube_temp_verts.copy()
142
143  # Can print off the cube verts
144  # in order to see the pattern
145  # print(cube_temp_verts)
146
147  # Can otherwise think like a truth table
148  bounding_verts[0] = [minx,miny,minz]
149  bounding_verts[1] = [minx,miny,maxz]
150  bounding_verts[2] = [minx,maxy,minz]
151  bounding_verts[3] = [minx,maxy,maxz]
152  bounding_verts[4] = [maxx,miny,minz]
153  bounding_verts[5] = [maxx,miny,maxz]
154  bounding_verts[6] = [maxx,maxy,minz]
155  bounding_verts[7] = [maxx,maxy,maxz]
```

Building Our Mesh

- Finally it's time to create our mesh
  - We'll give it a unique name
  - Populate the mesh from our collection of vertices
    - Importantly using the bounding_verts
    - The edge and face relationship remains the same as a standard cube
- At line 164 and 166, observe that we need to do two steps
  - One to create the object
  - A second step to add it to our scene ('Collection' being the default scene)
- And finally, as an added touch at line 169 -- set the display_type to 'WIRE'
  - (Which I learned by clicking around the user interface)

```
157 # New mesh name
158 bounding_name = "bounding_"+active_obj.name
159 # Create a new 'empty mesh'
160 bounding_mesh= bpy.data.meshes.new(bounding_name)
161 # Populate the mesh with geometry data
162 bounding_mesh.from_pydata(bounding_verts,cube_temp_edges,cube_temp_faces)
163 # Create the new object with a name and associated mesh
164 bounding_object = bpy.data.objects.new(bounding_name, bounding_mesh)
165 # Link in the mesh to a scene so we can actually view it.
166 bpy.data.collections['Collection'].objects.link(bounding_object)
167
168 # Set the bounding box to wireframe by default
169 bpy.data.objects[bounding_name].display_type = 'WIRE'
```

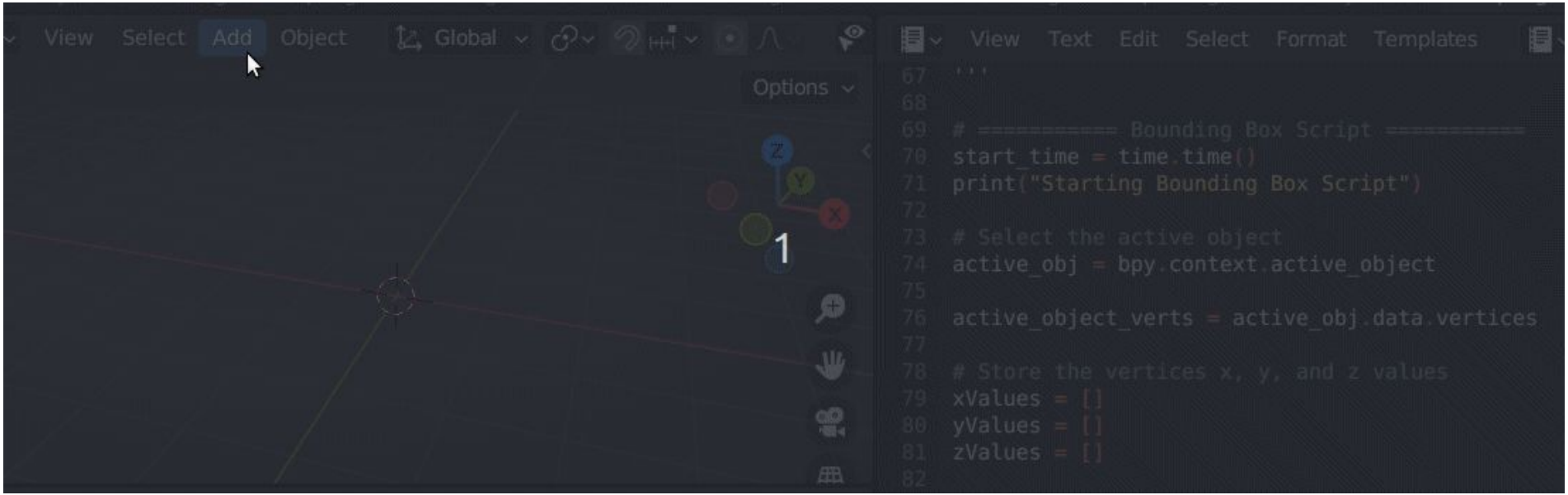
One Final Step

- In order to orient our bounding box to the object, we again have two strategies:
  - Set the transform (scale, rotation, and location) equivalent to the object
  - or
  - Make the bounding object a child of the selected object
    - Thus inheriting the transformations
- Either is fine -- the point is to play around and be creative
  - (Though making the child may be easier to maintain and organize in your scene!)

```
171  # Last step is to apply scale, rotation, and transform to the object
172  #bounding_object.scale = myObject.scale
173  #bounding_object.rotation_euler = myObject.rotation_euler
174  #bounding_object.location = myObject.location
175
176  # Another option is to make our target object the parent
177  # so that the bounding box transforms with it.
178  bounding_object.parent = active_obj
```

The Final Result!

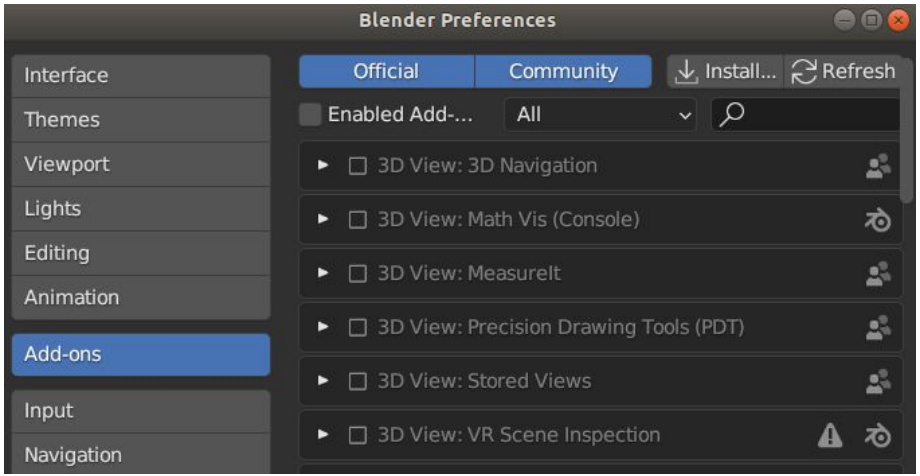
- Creating a Bounding Box programmatically in Python



Your Script as an Add-On

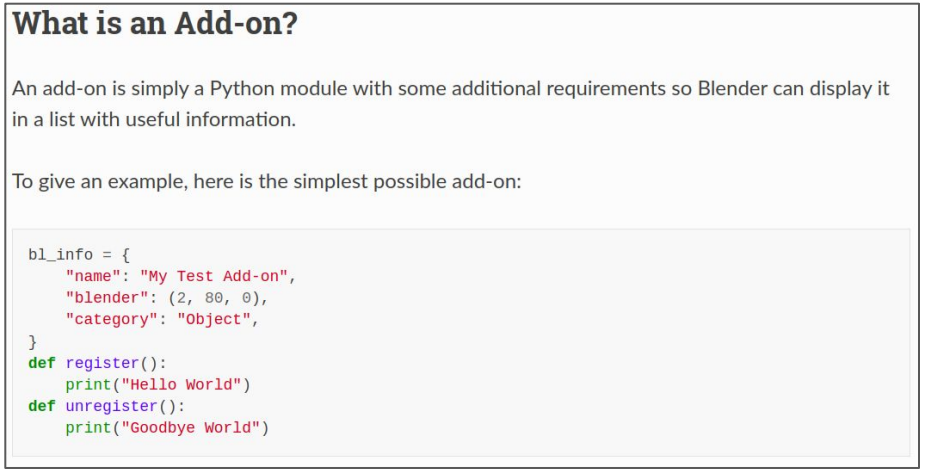
Making our Script available as a Plugin to the World

- Running our script through the Text Editor is perfectly fine
  - However -- it becomes much easier to share and use if we create an ‘add-on’
  - Some add-ons are official, and others are from the community (like you) that we can choose from.



Step 1: Prep

- The first thing we need is to prep our script as an add-on
  - The `bl_info` dictionary populates our plugin with meta-data importantly with:
    - A name
    - Category
  - `register()` and `unregister()` are function calls that take place when we first add our plugin



What is an Add-on?

An add-on is simply a Python module with some additional requirements so Blender can display it in a list with useful information.

To give an example, here is the simplest possible add-on:

```
bl_info = {
    "name": "My Test Add-on",
    "blender": (2, 80, 0),
    "category": "Object",
}
def register():
    print("Hello World")
def unregister():
    print("Goodbye World")
```

https://docs.blender.org/manual/en/latest/advanced/scripting/addon_tutorial.html

Step 2: Make our command useable

- We can make things slightly more interesting by adding our command to the search (F3) command menu.

```
def menu_func(self, context):
    self.layout.operator(ObjectMoveX.bl_idname)

def register():
    bpy.utils.register_class(ObjectMoveX)
    bpy.types.VIEW3D_MT_object.append(menu_func)  # Adds the new operator to an existing menu.

def unregister():
    bpy.utils.unregister_class(ObjectMoveX)
```


https://docs.blender.org/manual/en/latest/advanced/scripting/addon_tutorial.html

Step 3: Prepare ‘execute’ function

- Wrap the work that we previously did into a class
  - This inherits from the ‘Operator’ type, in that we then use our function as an ‘operator’
  - Then... (next slide)

```
1
2  bl_info = {
3      "name": "Compute Bounding Box BCon",
4      "blender": (3, 00, 0),
5      "category": "Object",
6  }
7
8  # Note: ^ This dictionary (bl_info) must be at the top of our file
9  #       (Usually with one space above)
10
11 import bpy  # Blender Python API
12 import time # Used for time keeping
13
14
15 class ObjectComputeBoundingBox(bpy.types.Operator):
16     """Simple example showing you how to compute bounding box""" # Use this as
17     bl_idname = "object.computebunding_box"            # Unique identifier fo
18     bl_label = "Compute Bounding Box BCon"                     # Display nam
19     bl_options = {'REGISTER', 'UNDO'}  # Enable undo for the operator.
20
21     def execute(self, context):        # execute() is called when running the
22
23         # ========== Bounding Box Script ==========
24         start_time = time.time()
25         print("Starting Bounding Box Script")
26
27         # Select the active object
28         active_obj = bpy.context.active_object
29
```


https://docs.blender.org/manual/en/latest/advanced/scripting/addon_tutorial.html

Step 4: Try it out!

- Test it out -- and we're done!
  - Of course -- for another tutorial we can create a menubar icon and further continue our adventure...maybe next year?

https://docs.blender.org/manual/en/latest/advanced/scripting/addon_tutorial.html

Wrapping Up

Summary

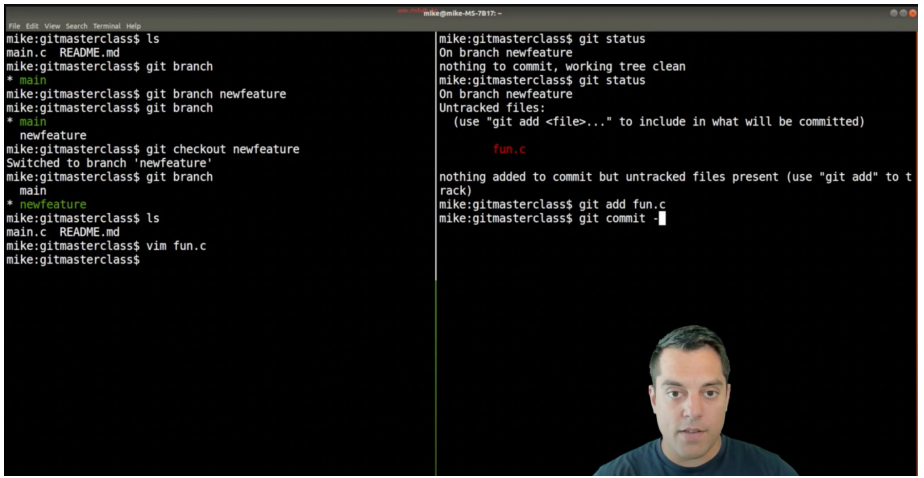
- Today we took an introductory look at Blender 3D's Python API
  - We briefly looked at some of the main modules
  - We solved (or resolved) a non-trivial problem in creating a bounding box
    - We showed how to create an add-on from this script.

Homework: New Feature Ideas of our Script

- What happens if we add new geometry to our mesh?
  - We need a way to poll and recompute the bounding box
  - Investigate handler callbacks here: https://docs.blender.org/api/current/bpy.app.handlers.html
- How about adding an option to creating a bounding sphere?
  - Just need to compute the maximum of the bounds on each axis to use as a diameter.
- Abstraction
  - As an exercise -- think about which chunks of code could go into their own functions
  - Perhaps we could encapsulate this into a classes or files
    - As our scripts get larger, it's important to get a little bit organized.
- Resiliency
  - Add some try/except blocks where necessary to make the code a bit more resilient

Other Essential Skills - Version Control for Text-based Files

- If you're diving into more programming, version control of your scripts becomes important
  - I'd recommend using 'git' and 'github' (to host the git repository) as a general skill
    - Git Beginner Masterclass (Free)
  - If folks are already using a tool like 'perforce' to manage art assets, that will also work fine too.



Further resources and training materials

- Best Practices
  - https://docs.blender.org/api/current/info_best_practice.html
  - Goto resource for questions on code structure, performance recommendations, etc.

Random Useful Ideas (If Time) (1/2)

- Check the blender version release
  - # Might be useful for checking compatibility with some feature
  - `import bpy`
  - `bpy.app.version`
    - or
  - `major,minor,micro = bpy.app.version`
  - `print(major)`

Random Useful Ideas (If Time) (2/2)

```
# Import our main module
import bpy


# A custom handler that runs only for the ‘Cube’
def my_handler(scene):
    if bpy.context.active_object.name == "Cube":
        print("Cube changed", scene.frame_current)


# ‘Install’ the handler (i.e. function) that will
# run when we do something interesting.
bpy.app.handlers.depsgraph_update_post.append(my_handler)
```

Thank you Blender Con 2023!

Getting Started with Scripting in Python

Social: @MichaelShah
Web: mshah.io
Courses: courses.mshah.io
YouTube: www.youtube.com/c/MikeShah

Thank you!

Extra