

Blender - Python API

#2



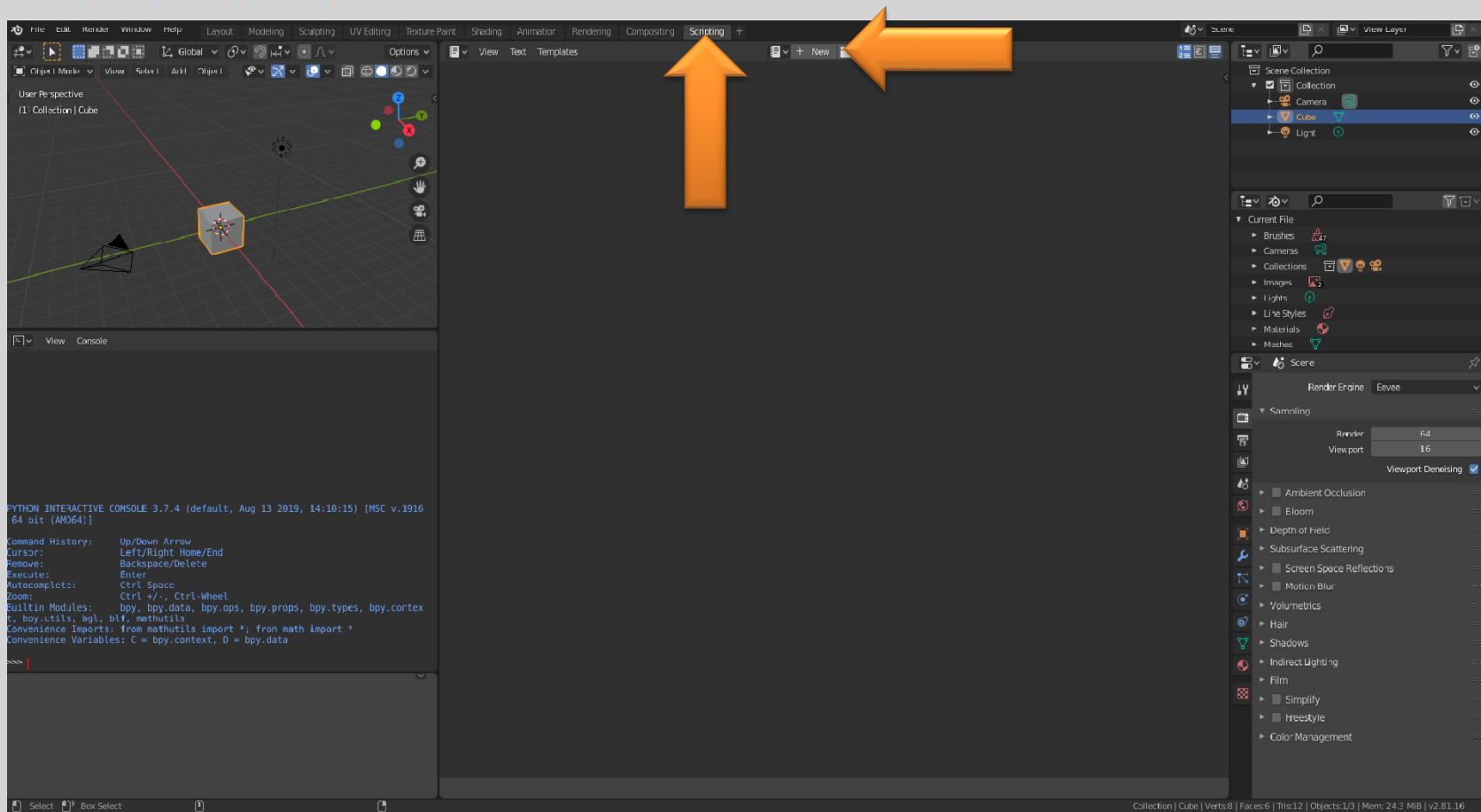
Serdar ARITAN

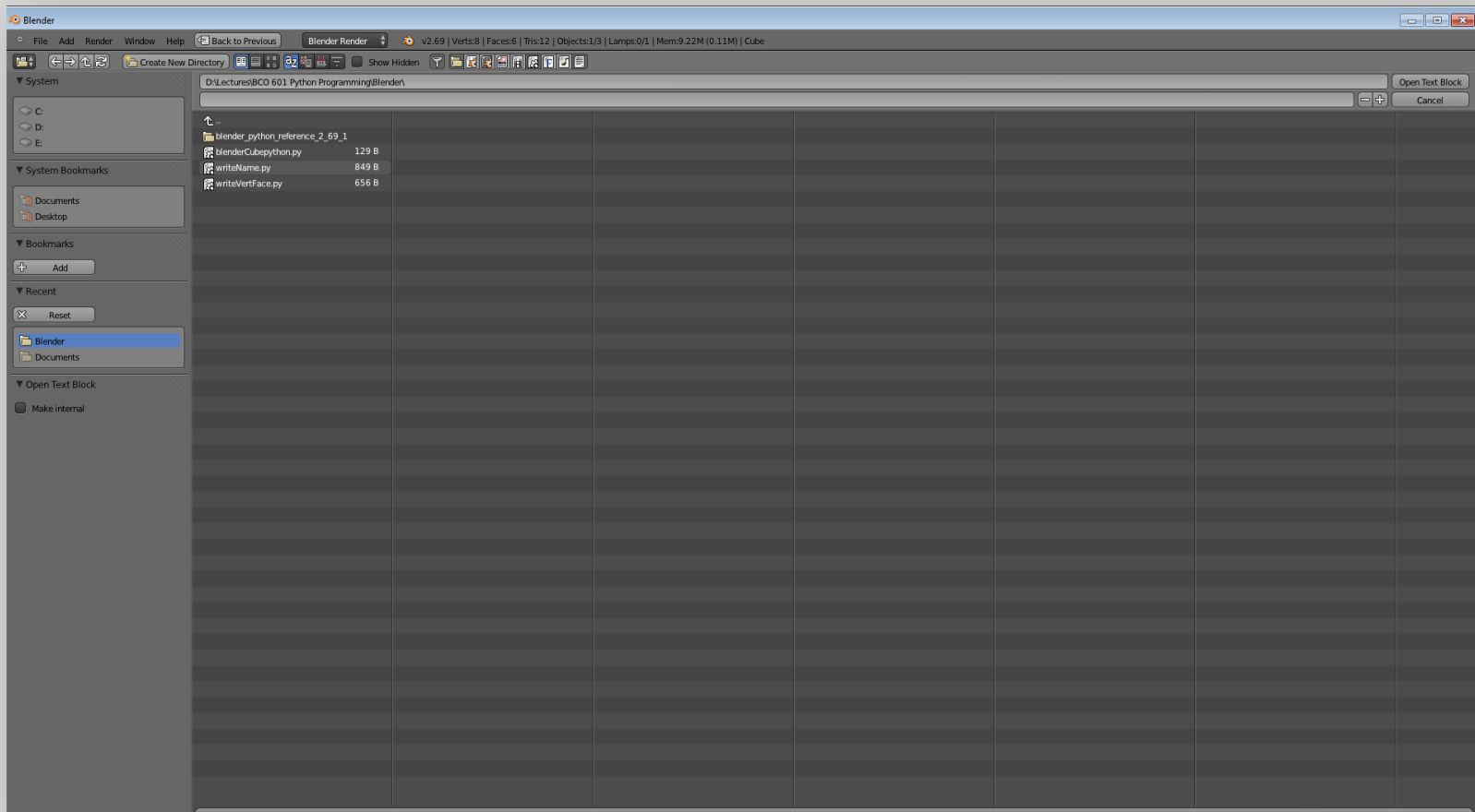
Department of Computer Graphics
Hacettepe University, Ankara, Turkey

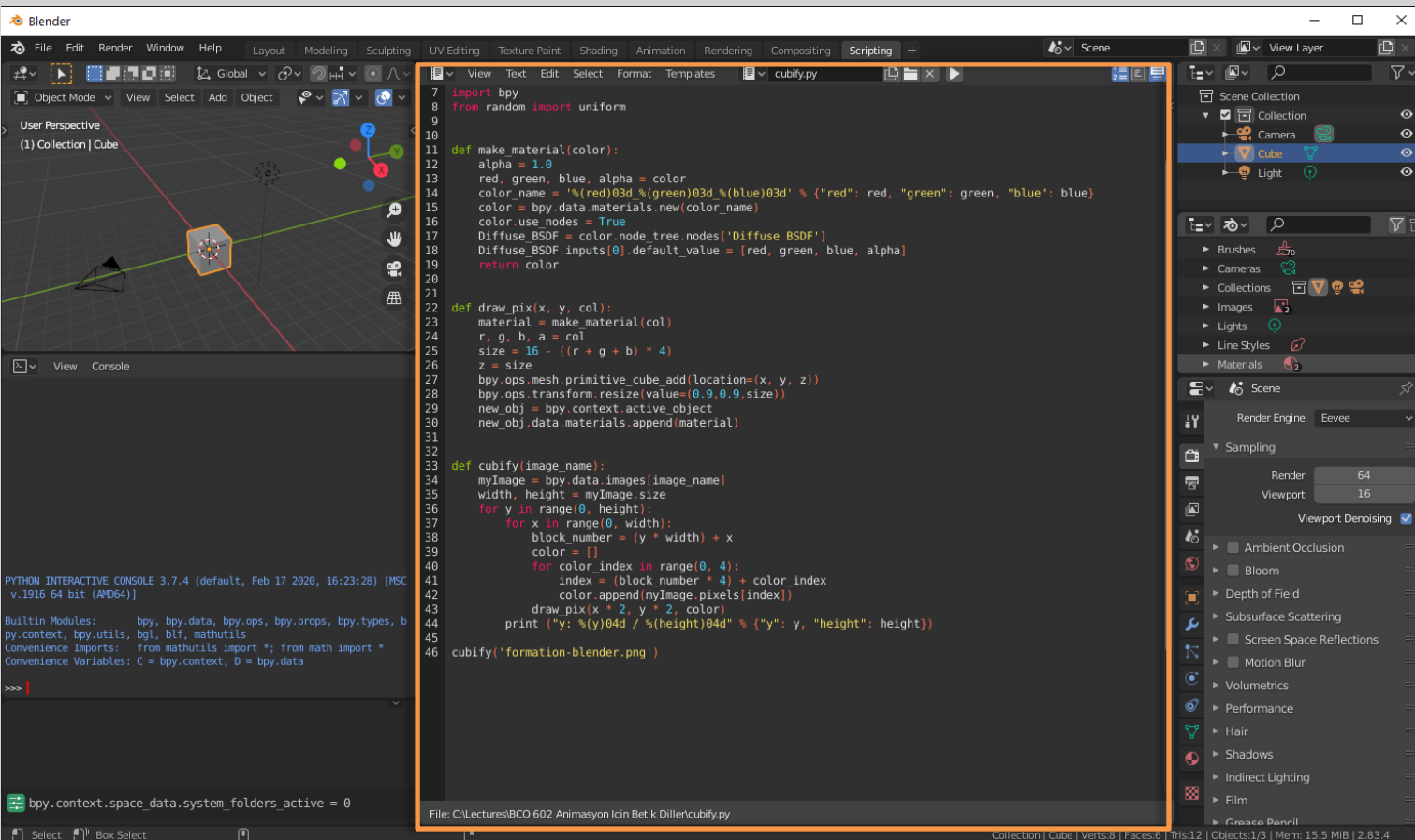


SCRIPT LANGUAGES FOR ANIMATION

Blender/Python API









SCRIPT LANGUAGES FOR ANIMATION

Blender/Python API

CTRL+ALT+SPACE

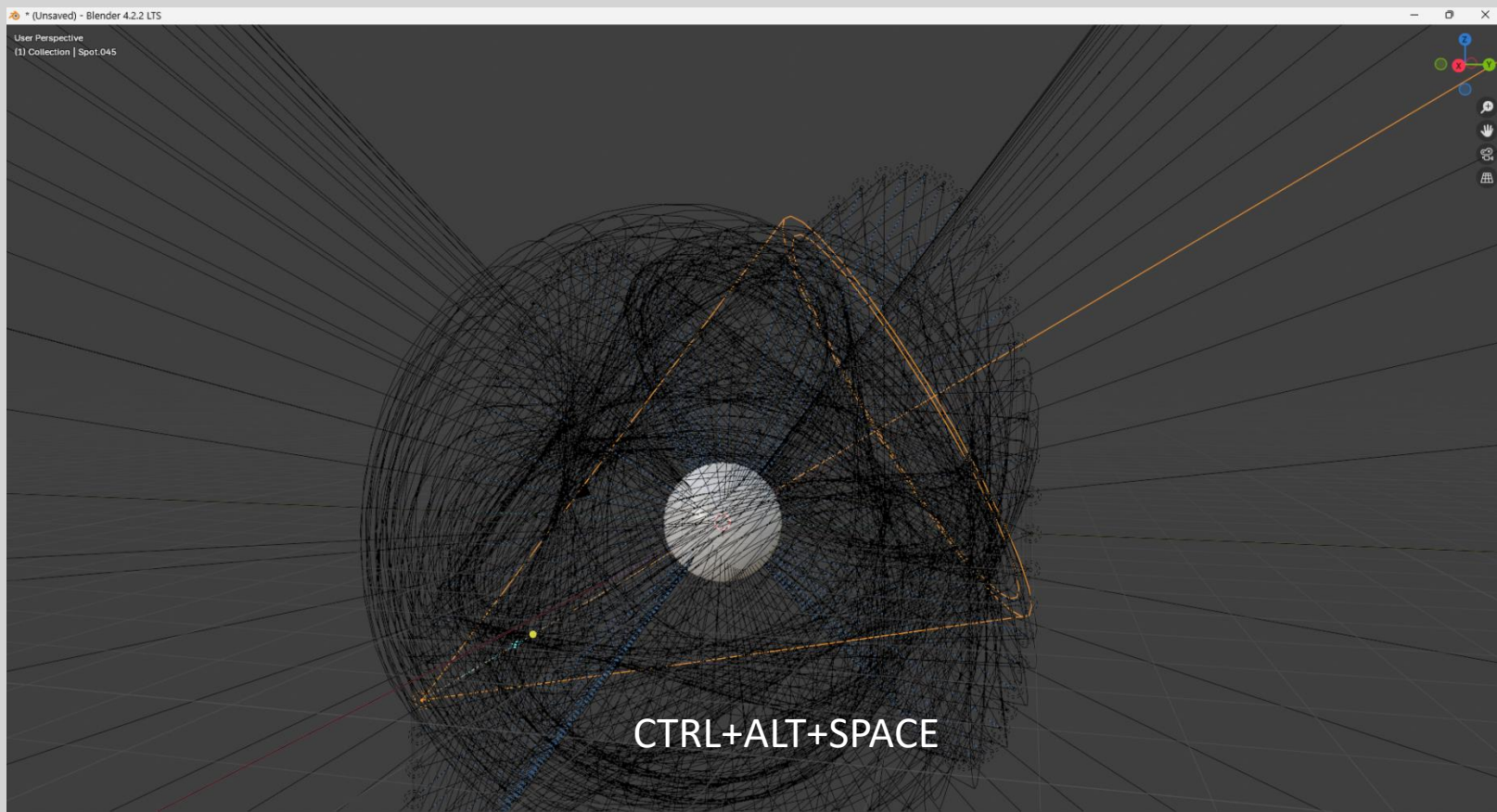
```
1 import bpy
2 from mathutils import Euler
3 from math import sin, cos, pi
4 import colorsys
5 import os, sys
6
7 tk = bpy.data.texts["bco602tk.py"].as_module()
8
9 def colorfulLights(trg=None, r=5, n=100, freq=2, e=130):
10
11     for i in range(n):
12         t = float(i)/float(n)
13         pos = (r*sin(2*pi*t), r*cos(2*pi*t), r*sin(freq*2*pi*t))
14
15         # Set HSV color and lamp energy
16         # Apply gamma correction for Blender
17         myColor = tuple(pow(c, 2.2) for c in colorsys.hsv_to_rgb(t, 0.6, 1))
18         tk.create.light(pos, target=trg, lamptype='SPOT', energy=e, color=myColor)
19
20
21 if __name__ == '__main__':
22
23     if(bpy.context.space_data == None):
24         cwd = os.path.dirname(os.path.abspath(__file__))
25     else:
26         cwd = os.path.dirname(bpy.context.space_data.text.filepath)
27     # Get folder of script and add current working directory to path
28     sys.path.append(cwd)
29     # Remove all elements
30     tk.mode("OBJECT")
31     tk.delete_all()
32     # Define material
33
34     white = tk.makeMaterial('White', (1, 1, 1, 1), 0.2, 0.8, 0.2)
35     # Create object and its material
36     mySphere = tk.create_sphere('WhiteSphere')
37     mySphere.data.materials.append(white)
38     tk.setSmooth(mySphere, level = 1, smooth = True)
39
40     # Create camera
41     myCamera = tk.create_camera((0, -3.5, 0), target=mySphere, lens=35, clip_start=0.1, clip_end=200, type='PERS')
42     myCamera.rotation_euler = Euler((pi/2, 0, 0), 'XYZ')
43     # Make this the current camera
44     bpy.context.scene.camera = myCamera
45
46     # Create lamps
47     colorfulLights(trg=mySphere)
48
49
50 # Specify folder to save rendering
51 tk.renderToFolder(renderFolder = os.path.join(cwd, 'Sphere_rendering'), \
52                 renderName = 'Lecture_sphere.png', \
53                 resX=800, resY=600, resPercentage=100, \
54                 animation=False, frame_end = 0)
55
56
```

PYTHON INTERACTIVE CONSOLE 3.11.7 (main, Feb 5 2024, 18:45:06) [MSC v.1928 64 bit (AMD64)]

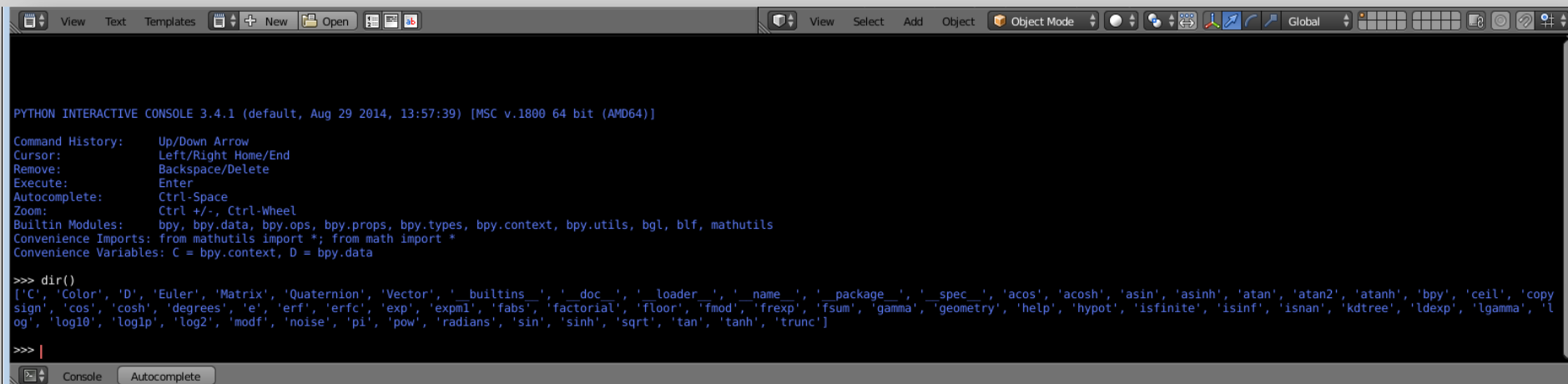
Builtin Modules: bpy, bpy.data, bpy.ops, bpy.props, bpy.types, bpy.context, bpy.utils, bgl, gpu, btf, mathutils
Convenience Imports: from mathutils import *; from math import *
Convenience Variables: C = bpy.context, D = bpy.data

```
>>> |
✓ bpy.ops.object.light_add(type='SPOT', align='WORLD', location=(-0.313953, 4.99813, -0.626666), scale=(1, 1, 1))
✓ bpy.ops.text.run_script()
✗ Python: Traceback (most recent call last):
  File "C:\colorful_sphere_target.py", line 52, in <module>
```

File: C:\Lectures\BCO 602 Animasyon İçin Betik Dilleri\Hafta_07\colorful_sphere_target.py



To check what symbols are loaded into the default environment, type **dir()** and press **<ENTER>**



```
PYTHON INTERACTIVE CONSOLE 3.4.1 (default, Aug 29 2014, 13:57:39) [MSC v.1800 64 bit (AMD64)]

Command History: Up/Down Arrow
Cursor: Left/Right Home/End
Remove: Backspace/Delete
Execute: Enter
Autocomplete: Ctrl-Space
Zoom: Ctrl +/-, Ctrl-Wheel
Builtin Modules: bpy, bpy.data, bpy.ops, bpy.props, bpy.types, bpy.context, bpy.utils, bgl, blf, mathutils
Convenience Imports: from mathutils import *; from math import *
Convenience Variables: C = bpy.context, D = bpy.data

>>> dir()
['C', 'Color', 'D', 'Euler', 'Matrix', 'Quaternion', 'Vector', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'bpy', 'ceil', 'copy', 'sign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'geometry', 'help', 'hypot', 'isfinite', 'isinf', 'isnan', 'kdtree', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'noise', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']

>>> |
```



bpy. and then press <TAB> and you will see the Console auto-complete

```
Remove:      Backspace/Delete
Execute:     Enter
Autocomplete: Ctrl-Space
Zoom:       Ctrl +/-, Ctrl-Wheel
Builtin Modules: bpy, bpy.data, bpy.ops, bpy.props, bpy.types, bpy.context, bpy.utils, bgl, blf, mathutils
Convenience Imports: from mathutils import *; from math import *
Convenience Variables: C = bpy.context, D = bpy.data

>>> dir()
['C', 'Color', 'D', 'Euler', 'Matrix', 'Quaternion', 'Vector', '_builtins_', '_doc_', '_loader_', '_name_', '_package_', '_spec_', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'bpy', 'ceil', 'copy', 'sign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'geometry', 'help', 'hypot', 'isfinite', 'isinf', 'isnan', 'kdtree', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'noise', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']

>>> bpy.
bpy
  app
  context
  data
  ops
  path
  props
  types
  utils

>>> bpy.
```



- The interactive console is great for testing one-liners, It also has auto completion so you can inspect the api quickly.
- Button tool tips show **Python** attributes and operator names. Right clicking on buttons and menu items directly links to **API** documentation.
- For more examples, the text menu has a templates section where some example operators can be found to examine further scripts distributed with Blender, see `~/.blender/scripts/startup/bl_ui` for the user interface and `~/.blender/scripts/startup/bl_operators` for operators.

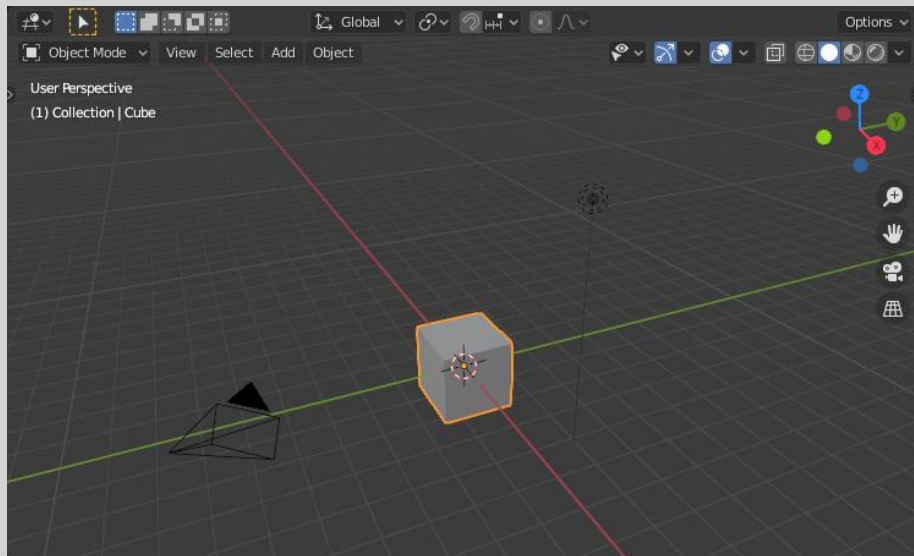
`C:\Program Files\Blender Foundation\Blender 4.2\4.2\scripts\startup`



```
>>> bpy.app.binary_path  
'C:\\Program Files\\Blender Foundation\\Blender 4.2\\blender.exe'  
  
>>> bpy.app.version <TAB>  
  
_char  
_cycle  
_string  
  
>>> bpy.app.version  
(4, 2, 2)  
  
>>> bpy.app.version_string  
'4.2.2 LTS'
```


Blender/Python API

Contextual access of information in a Blender file/scene



```
>>> bpy.context.mode
```

```
'OBJECT'
```

```
# Will print the current 3D View mode  
# (Object, Edit, Sculpt etc.,)
```

```
>>> bpy.context.object
```

```
bpy.data.objects['Cube']
```

```
>>> C.mode
```

```
'OBJECT'
```

```
>>> C.object
```

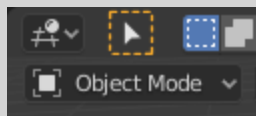
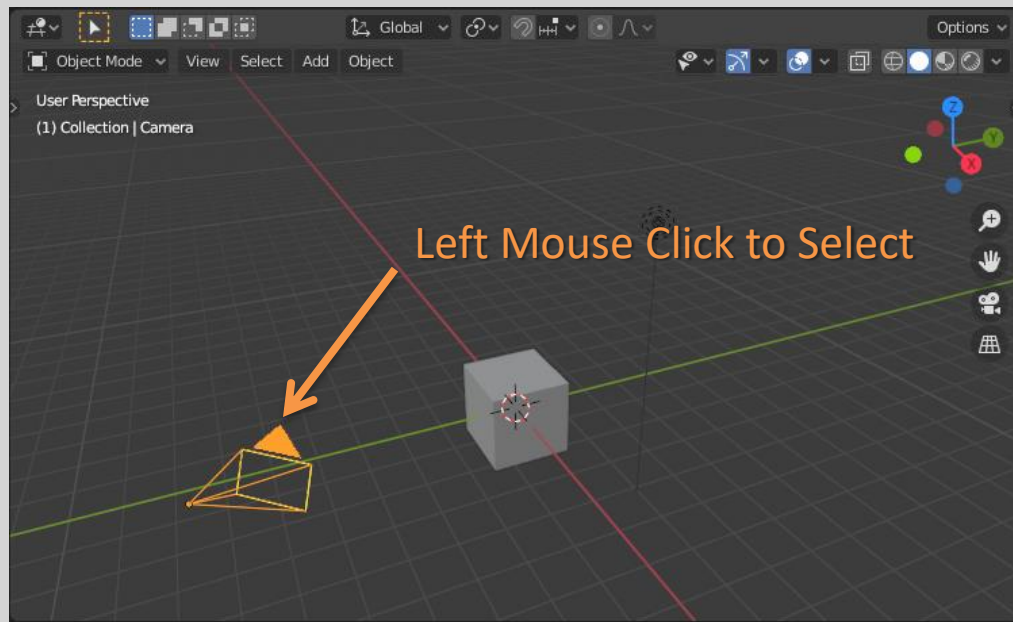
```
bpy.data.objects['Cube']
```

Convenience Variables: `C = bpy.context`, `D = bpy.data`



Blender/Python API

Contextual access of information in a Blender file/scene



```
>> bpy.context.mode  
'OBJECT'
```

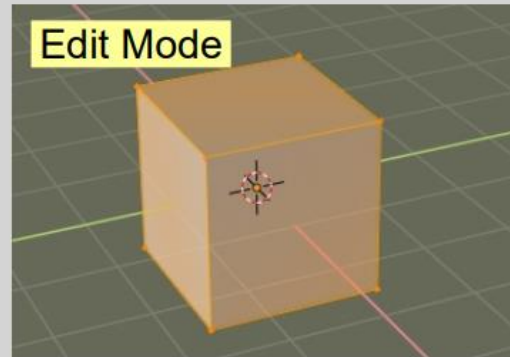
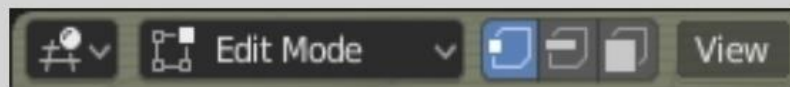
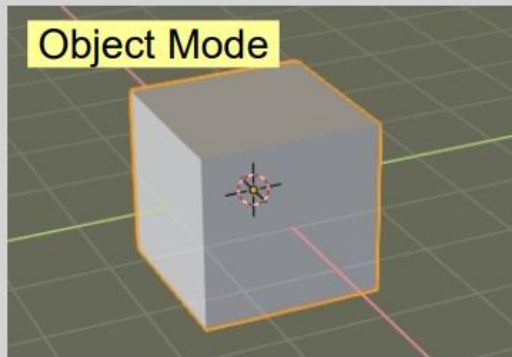
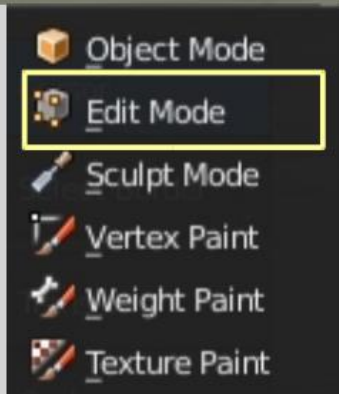
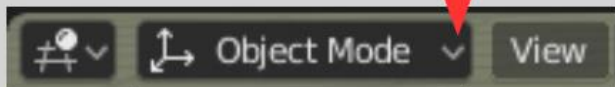
```
>>> bpy.context.object  
bpy.data.objects['Cube']
```

```
>>> bpy.context.object  
bpy.data.objects['Camera']
```




Object Mode and Edit Mode

Switching Modes: Click to display the menu and select **Edit Mode** in the 3D Viewport Header.

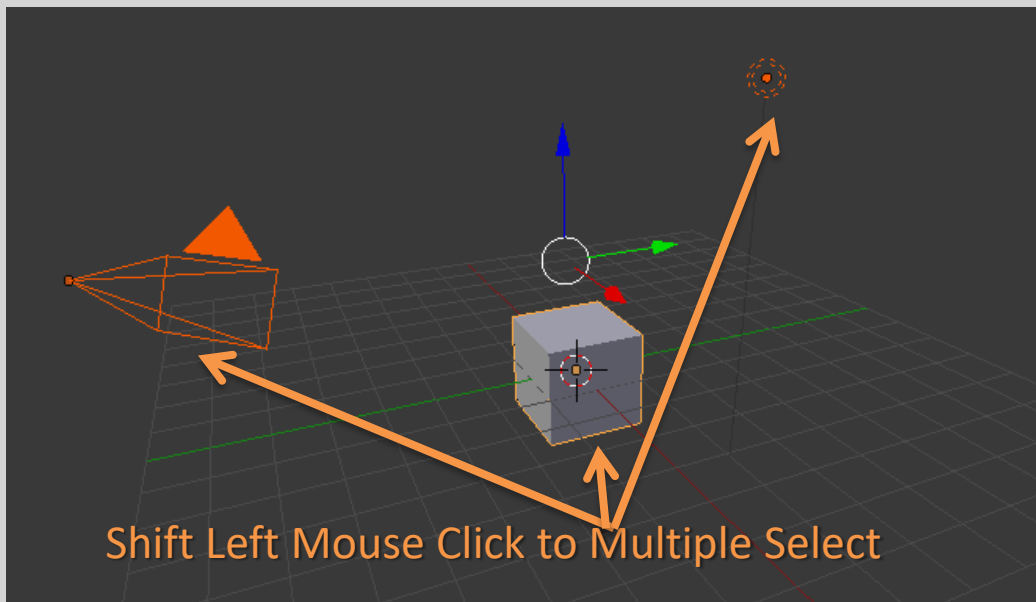


Alternative: Press the **Tab Key** on the Keyboard to toggle between modes.



Blender/Python API

Contextual access of information in a Blender file/scene



```
>>> bpy.context.selected_objects  
[bpy.data.objects['Cube'], bpy.data.objects['Light'],  
bpy.data.objects['Camera']]
```

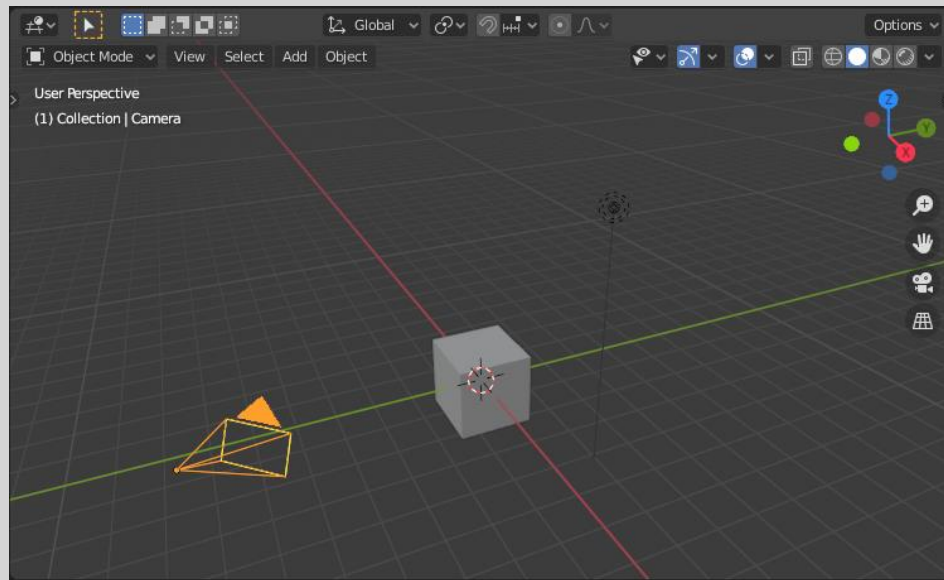
Blender/Python API

Data access of information in a Blender file/scene

```
>>> for object in bpy.data.scenes['Scene'].objects:  
...     print(object.name)  
...  
Camera  
Cube  
Lamp
```

You can access following data in the current Blender file

objects, meshes, materials, textures,
scenes, screens, sounds, scripts,
texts, cameras, curves, lamps,
brushes, armatures, images, lattices,
libraries, worlds, groups, metaballs,
particles, node_groups

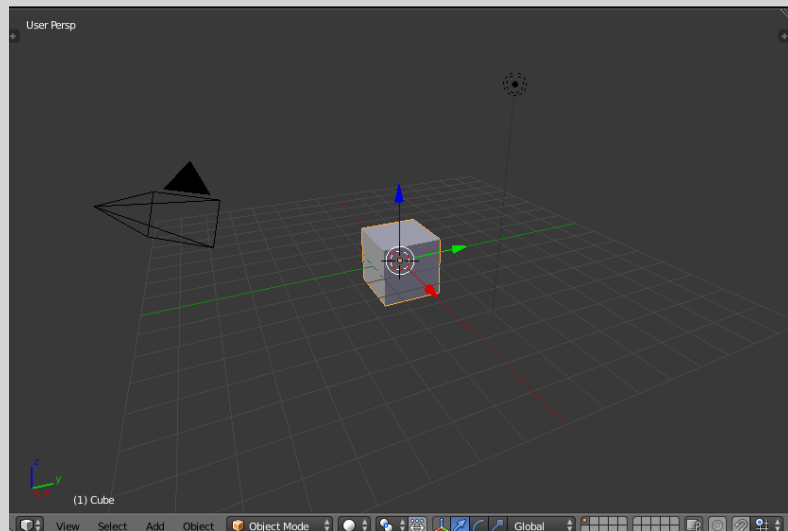




Blender/Python API

Data access of information in a Blender file/scene

```
>>> for object in bpy.data.objects:  
...     print(object.name + ' is at location' + str(object.location))  
...  
Camera is at location<Vector (7.4811, -6.5076, 5.3437)>  
Cube is at location<Vector (0.0000, 0.2765, -0.2134)>  
Lamp is at location<Vector (4.0762, 1.0055, 5.9039)>
```

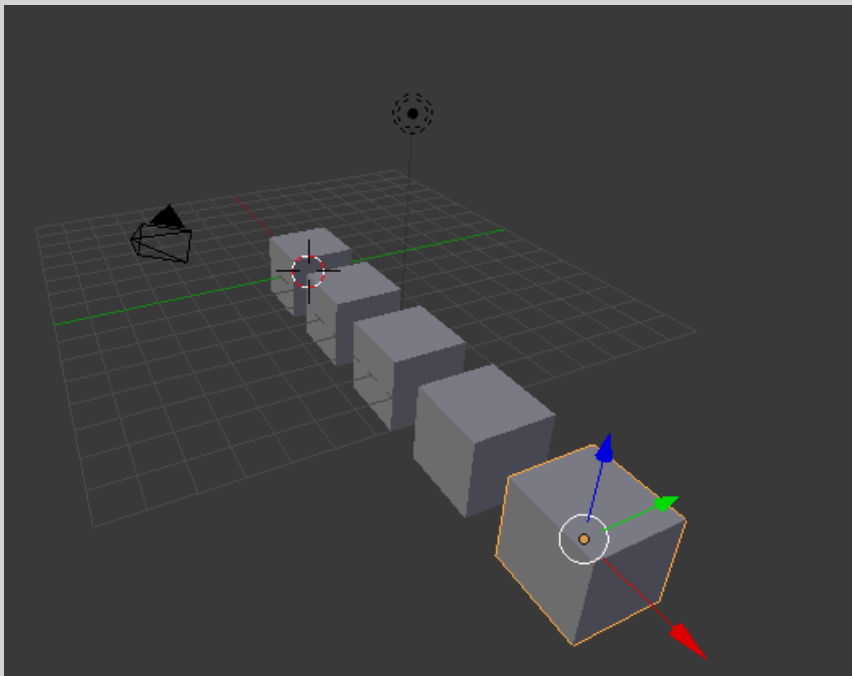




Blender/Python API

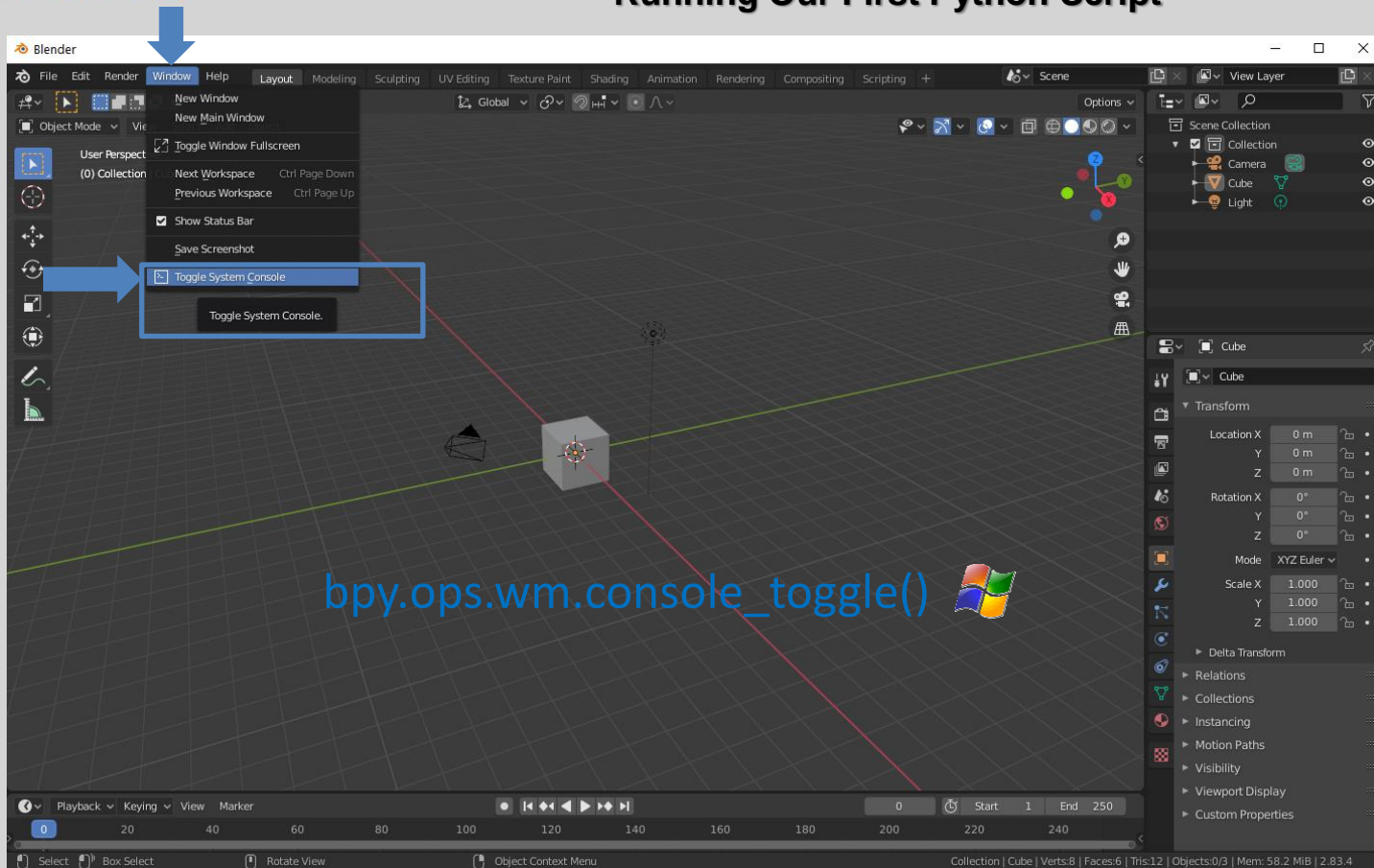
Python access to calling operators in Blender file/scene

```
>>> add_cube = bpy.ops.mesh.primitive_cube_add
>>> for index in range(5):
...     add_cube(location=(index*3, 0, 0))
...
{'FINISHED'}
{'FINISHED'}
{'FINISHED'}
{'FINISHED'}
{'FINISHED'}
```



Blender/Python API

Running Our First Python Script





Import Libraries

Importing the Blender Python API is the first step for any Blender script.

```
import bpy          # Imports the Blender Python API
import mathutils    # Imports Blender vector math utilities
import math         # Imports the standard Python math library
```

Set up some variables

```
a = 'hello'
b = ' world!'
```

Print the addition to the system console

```
print (a + b)
```

open the system console

```
bpy.ops.wm.console_toggle() 
```



Blender/Python API

Running Our First Python Script

```
1 import bpy          # Imports the Blender Python API
2 import mathutils    # Imports Blender vector math utilities
3 import math          # Imports the standard Python math library
4
5 # Set up some variables
6 a = 'hello'
7 b = ' world!'
8
9 # Print the addition to the system console
10 print(a + b)
11
12 # open the system console
13 bpy.ops.wm.console_toggle()
14 |
```

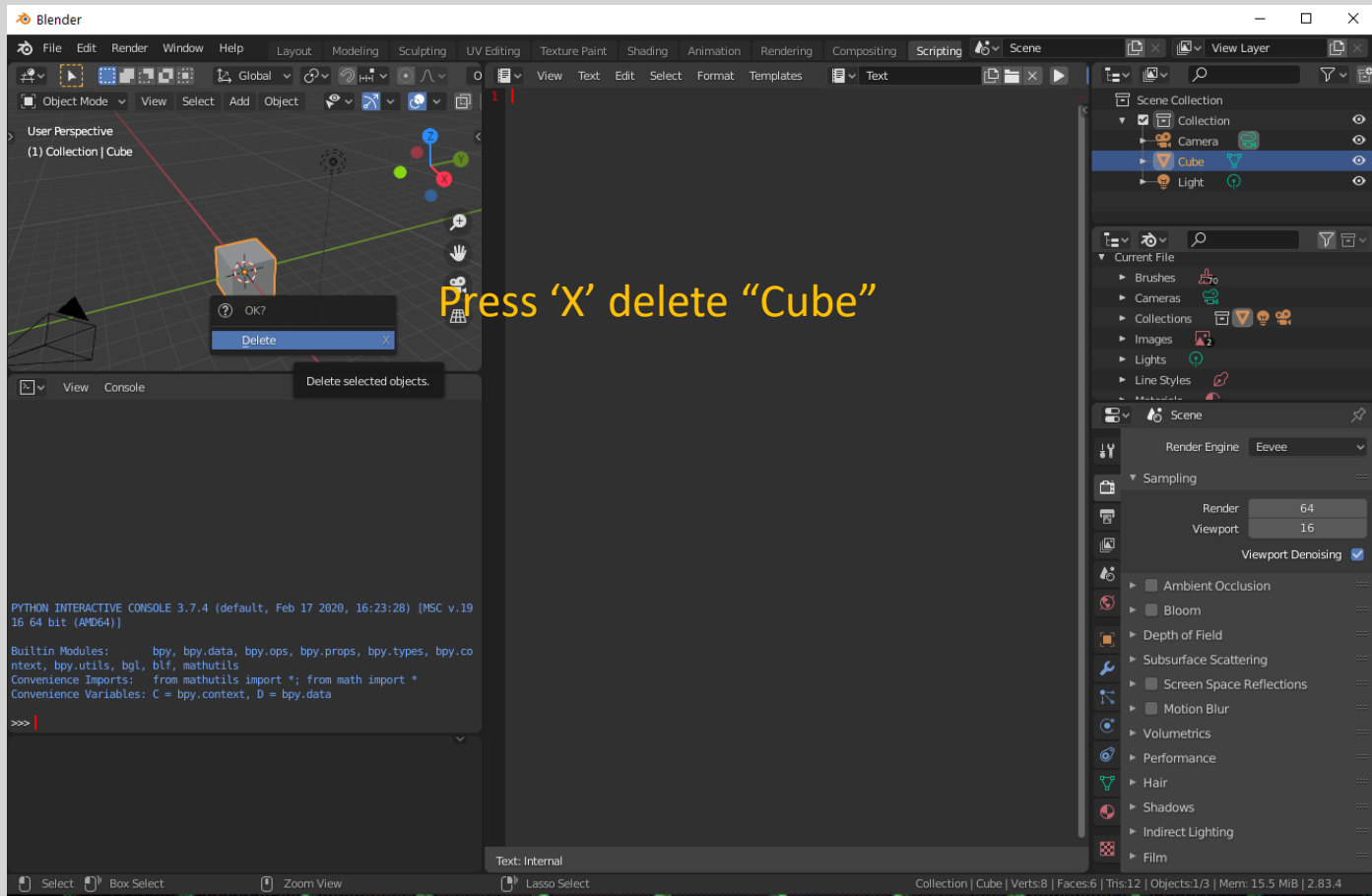
blender

Read prefs: C:\Users\SA-Lenovo\AppData\Roaming\Blender Foundation\Blender\2.93\config\userpref.blend
hello world!

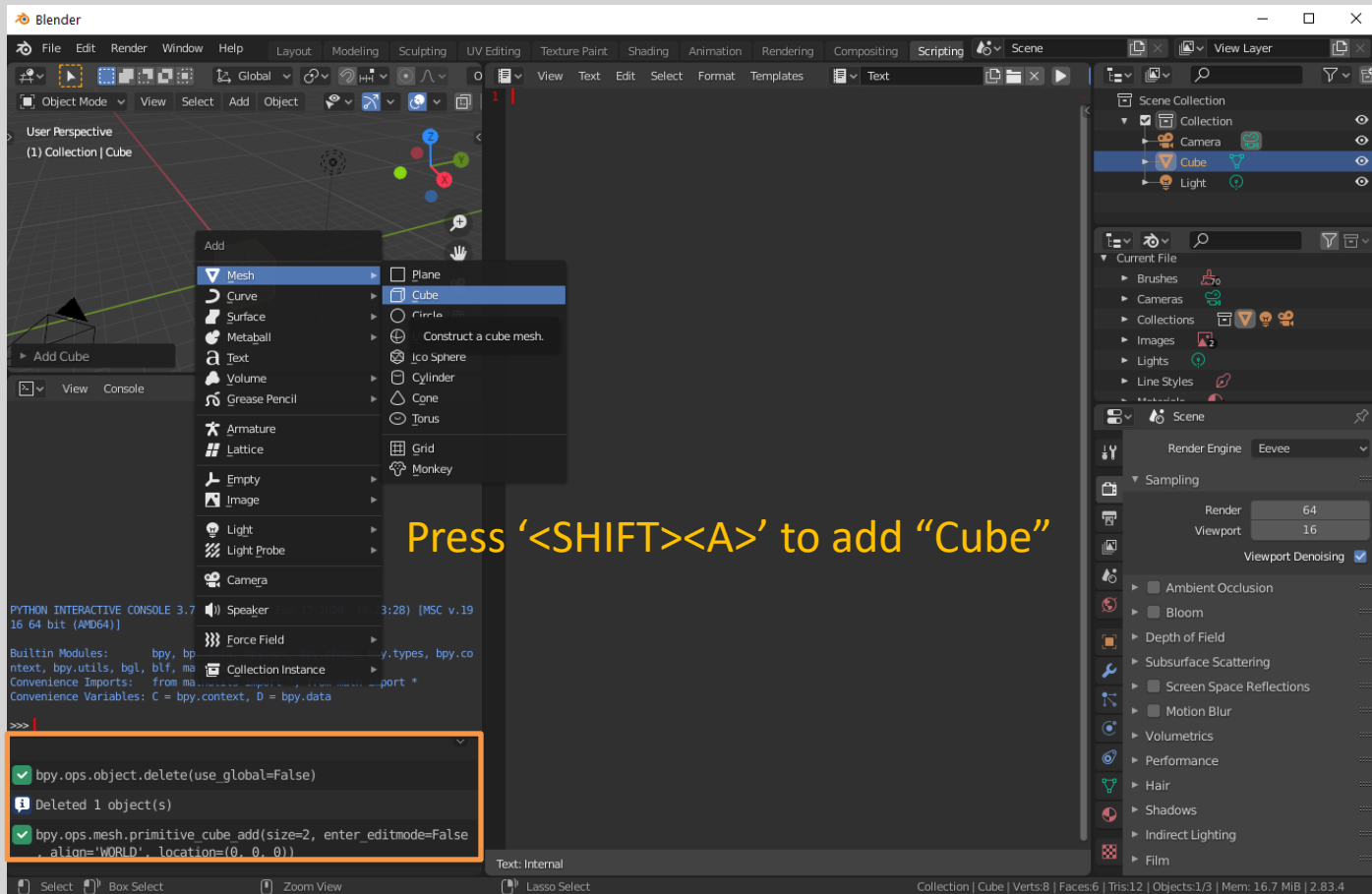
open the system console
bpy.ops.wm.console_toggle()

Blender/Python API

Running Our First Python Script



Blender/Python API Running Our First Python Script



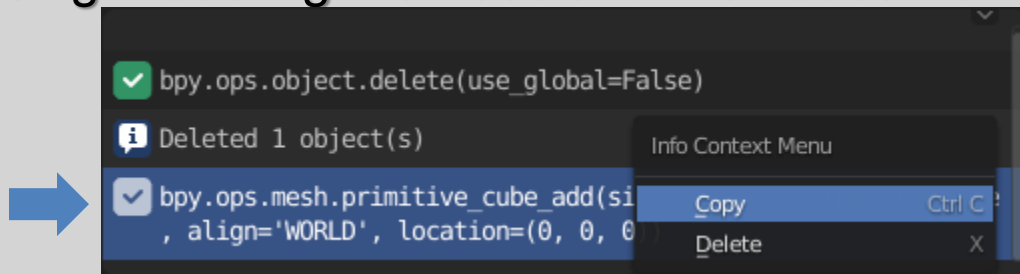
Press '**<SHIFT><A>**' to add "Cube"

```
PYTHON INTERACTIVE CONSOLE 3.7  
16 64 bit (AMD64)  
  
Builtin Modules: bpy, bpy.types, bpy.co  
ntext, bpy.utils, bgl, bli, ma  
Convenience Imports: from ma  
Convenience Variables: C = bpy.context, D = bpy.data  
  
>>>  
✓ bpy.ops.object.delete(use_global=False)  
! Deleted 1 object(s)  
✓ bpy.ops.mesh.primitive_cube_add(size=2, enter_editmode=False  
  , align='WORLD', location=(0, 0, 0))
```

Text: Internal

Collection | Cube | Vets:8 | Faces:6 | Tris:12 | Objects:1/3 | Mem: 16.7 MiB | 2.83.4

Blender's **'Info'** view port shows you all recent Blender activity as executable Python commands. This is very handy for prototyping a process using modeling methods and then assembling them into a script



The selected text from above can be copied and pasted into the Text Editor. We can delete any unneeded options from the command. This script will create a single mesh cube in the scene.

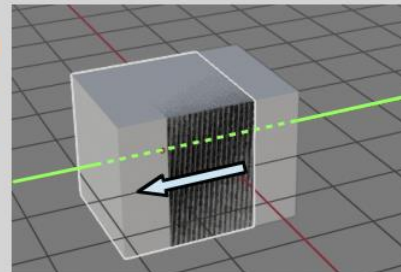
```
import bpy                                #Imports the Blender Python API
# Create a mesh cube in the scene
bpy.ops.mesh.primitive_cube_add(location=(5, 5, 5))
```

Blender/Python API

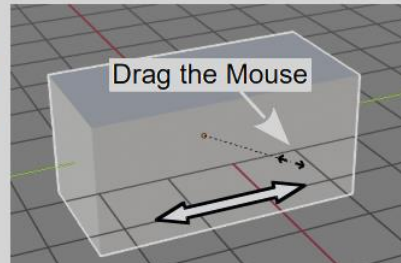
Object Mode Manipulation

The three basic manipulation controls are: **Translate, Rotate and Scale.**

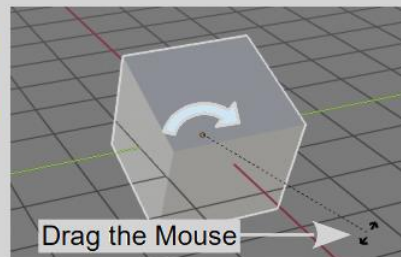
Translate: To move an Object freely in the plane of the view, press the **G Key** (Grab Mode) with the object selected and drag the Mouse. In Grab mode the outline turns white. To lock the movement to a particular axis, press the G Key + X, Y, or Z. G key + Y restricts the movement to the Y (green) Axis

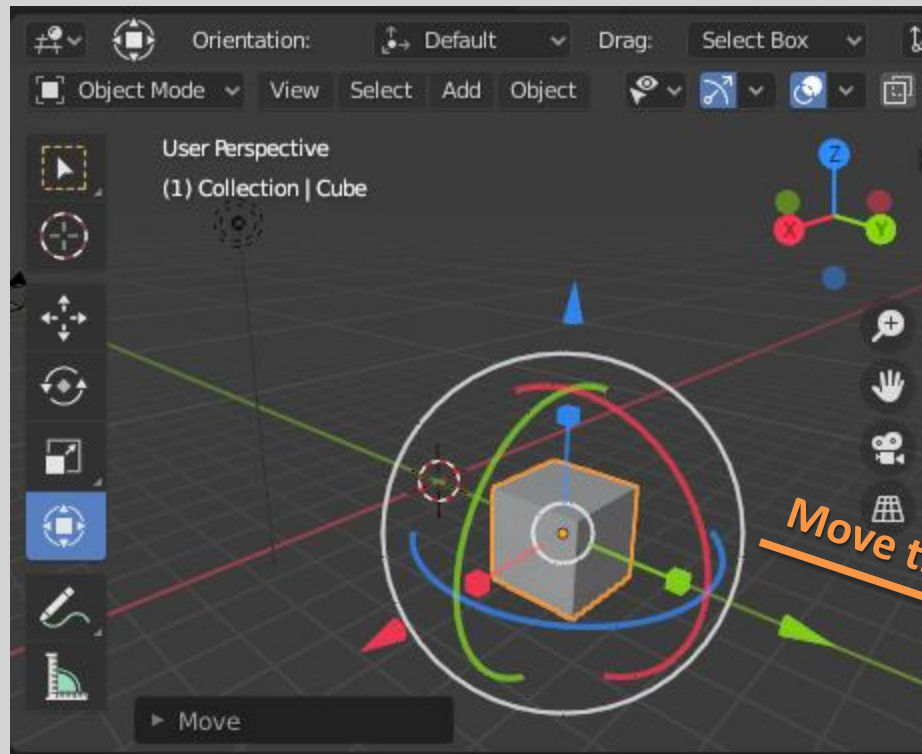


Scaling: To scale an Object (make larger or smaller), press the **S Key** and drag the mouse. To lock the scale to a particular axis, press S Key + X, Y, or Z. To scale by a specific value press S Key + Number Key (S + 2 + Y = Scale twice on the Y Axis



Rotating: To rotate an Object, press the **R Key** and move the mouse about the **Object's center**. To lock the rotation to an axis, press the R Key + X, Y, or Z. To rotate a set number of degrees, press R + the number of degrees of rotation and press Enter. R+30 rotates the Object 30 degrees. R + Y + 30 rotates the Object 30 degrees about the Y-Axis







Place the Mouse Cursor in proximity to the Widget to display the circle. The Mouse Cursor changes to a white **+**, click, hold and move the Mouse to rotate the view.

Circle displays on Mouse-over

Click, hold and move the mouse to Rotate the view

Colored Circles Confine Manipulation to a Plane

Click, hold and move the mouse to pan the view

Click on the Camera to toggle Camera View On/Off



The Circle represents a Sphere

Click to display Properties Panel

XYZ Axis Manipulation

Click and hold and move the Mouse up/down to zoom the view

Click the Grid to toggle User Perspective/User Orthographic views

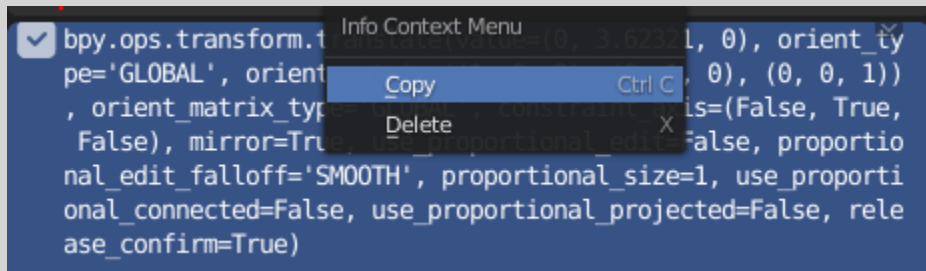
Note: The Red, Green and Blue circles on the rotation sphere align with the Red, Green and Blue Axis of the 3D Scene.



<SHIFT>Left Click to Select



Ctrl+C to Copy

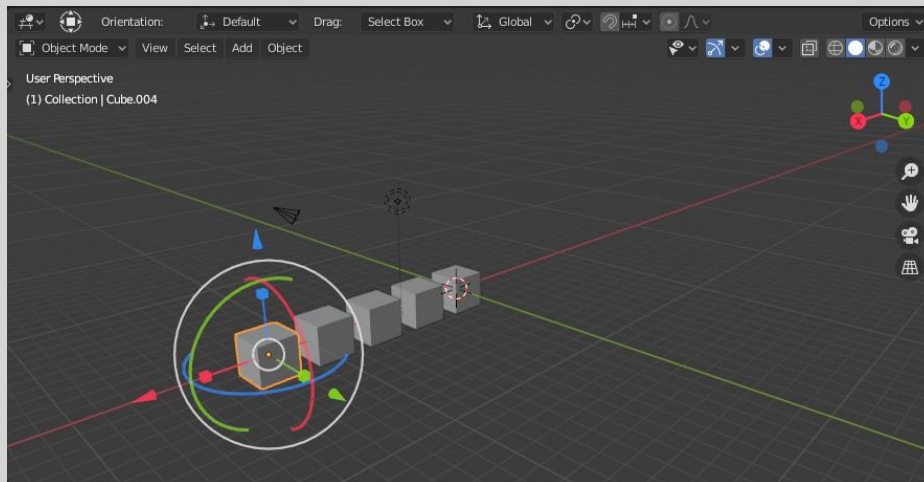


```
bpy.ops.transform.translate(value=(3.05332, 0, 0), constraint_axis=(True, False,  
False), constraint_orientation='GLOBAL', mirror=False,  
proportional='DISABLED', proportional_edit_falloff='SMOOTH',  
proportional_size=1, release_confirm=True)
```

```
bpy.ops.transform.translate(value=(3.0, 0, 0))
```



```
import bpy
for i in range (5):
    x = i * 3
    y = 0
    z = 0
    # Create a mesh cube in the scene
    bpy.ops.mesh.primitive_cube_add(location=(x, y, z))
```





The For Loop

for loop: Repeats a set of statements over a group of values.

– **Syntax:**

```
for variableName in groupOfValues:  
    indent➡statements
```

We indent the statements to be repeated with tabs or spaces.

`variableName` gives a name to each value, so you can refer to it in the statements.

`groupOfValues` can be a range of integers, specified with the `range` function.



– Example:

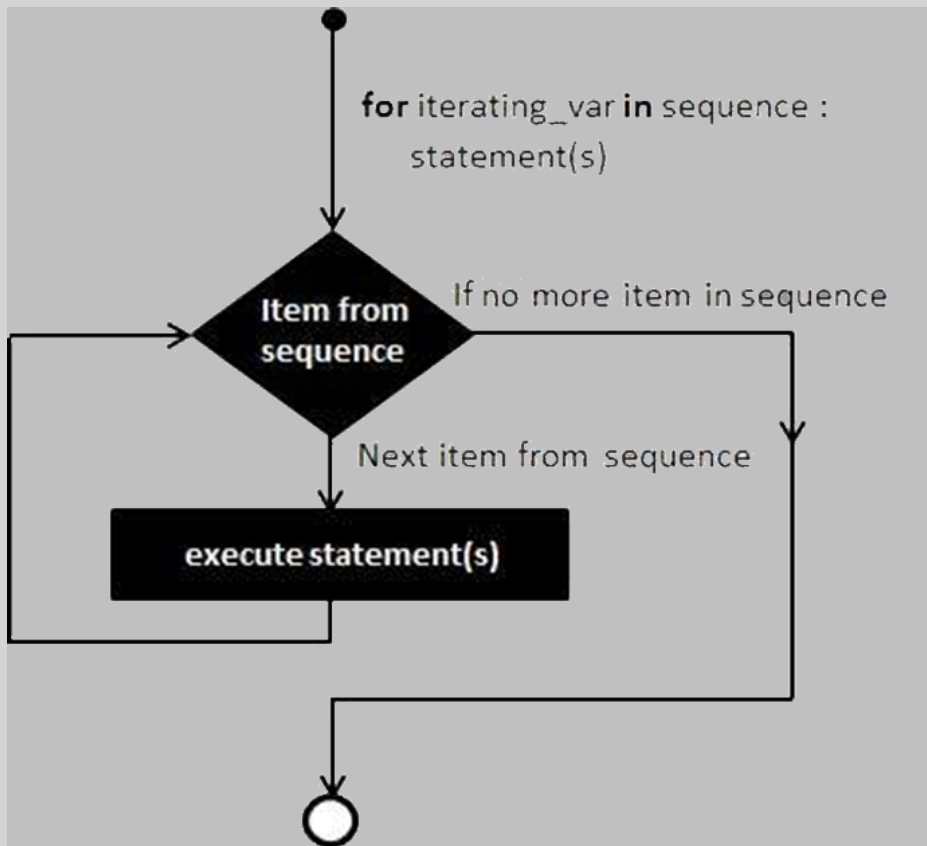
```
for x in range(6):  
    print (x)
```

Output:

0
1
2
3
4
5

! There is not number 6

The For Loop





The For Loop

– Example:

```
for harfler in 'Serdar':  
    print('Harf : ', harfler)
```



range() ?

```
Harf : S  
Harf : e  
Harf : r  
Harf : d  
Harf : a  
Harf : r
```



– Example:

```
for x in range(1, 6):  
    print (x, "squared is", x**2)
```

Output:

```
1 squared is 1  
2 squared is 4  
3 squared is 9  
4 squared is 16  
5 squared is 25
```

The For Loop

range

`range(start, stop)` - the integers between `start` (inclusive) and `stop` (exclusive)
It can also accept a third value specifying the change between values.

- `range(start, stop, step)` - the integers between start (**inclusive**) and stop (**exclusive**) by step

Example:

```
for x in range(5, 0, -1):  
    print(x)  
print ("Blastoff!")
```

Output:

```
5  
4  
3  
2  
1  
Blastoff!
```



The For Loop range

Example:

```
>>> for i in range(1,7):  
        print (i, i**2, i**3, i**4)  
  
1 1 1 1  
2 4 8 16  
3 9 27 81  
4 16 64 256  
5 25 125 625  
6 36 216 1296  
  
>>> for x in range(0, 5):  
        print('hello %s' % x)  
  
hello 0  
hello 1  
hello 2  
hello 3  
hello 4
```

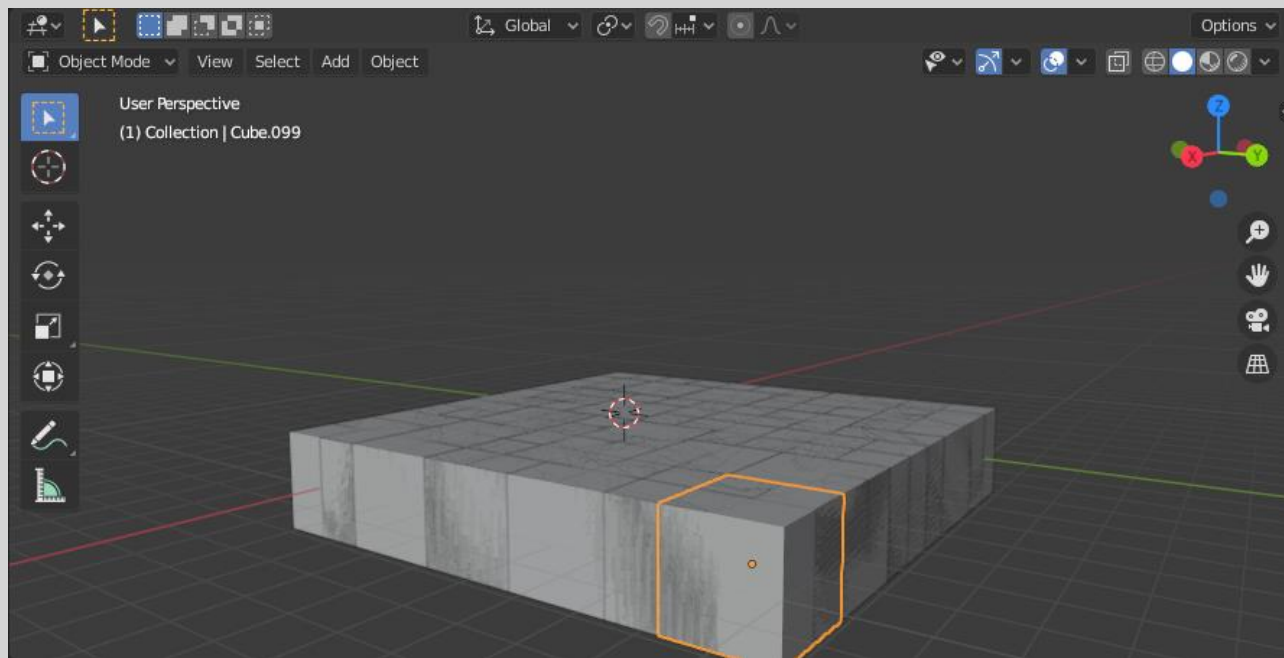


```
import bpy
```

```
for x in range(10):
```

```
    for y in range(10):
```

```
        bpy.ops.mesh.primitive_cube_add(location=(x, y, 0))
```

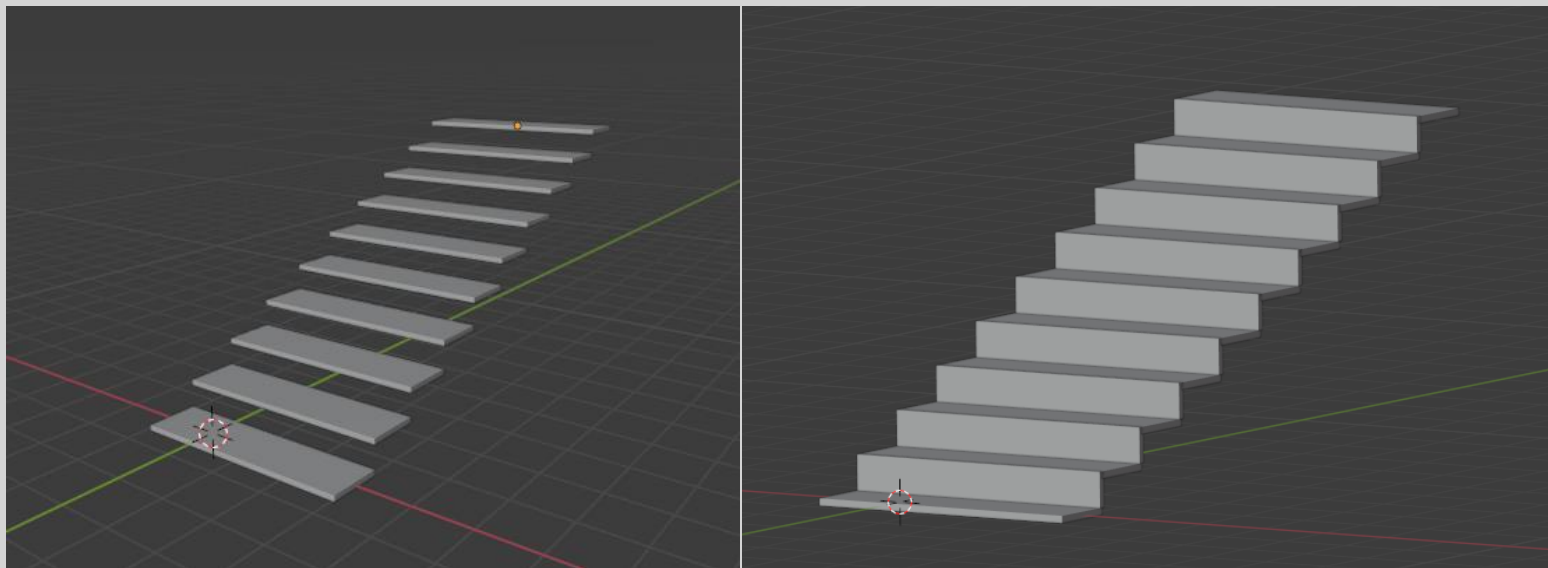




Blender/Python API

Classwork

10 Minutes Break to Write a Script

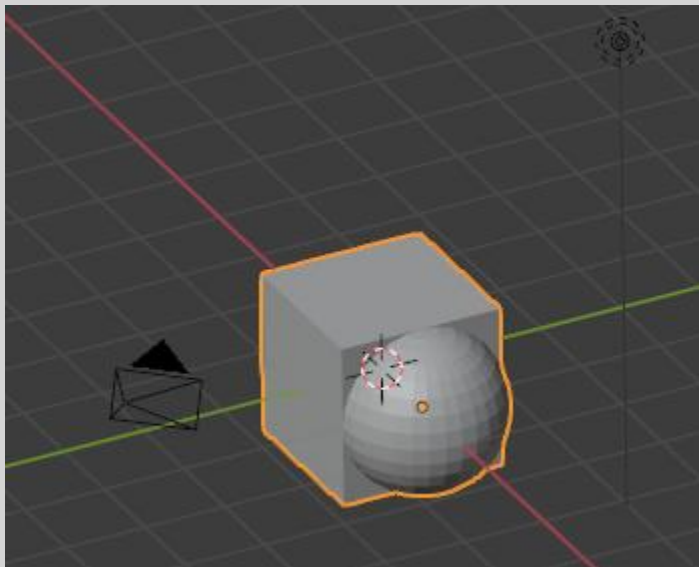


```
bpy.ops.mesh.primitive_cube_add  
bpy.ops.transform.resize  
bpy.ops.transform.translate
```


Blender/Python API

Classwork

10 Minutes Break to Write a Script

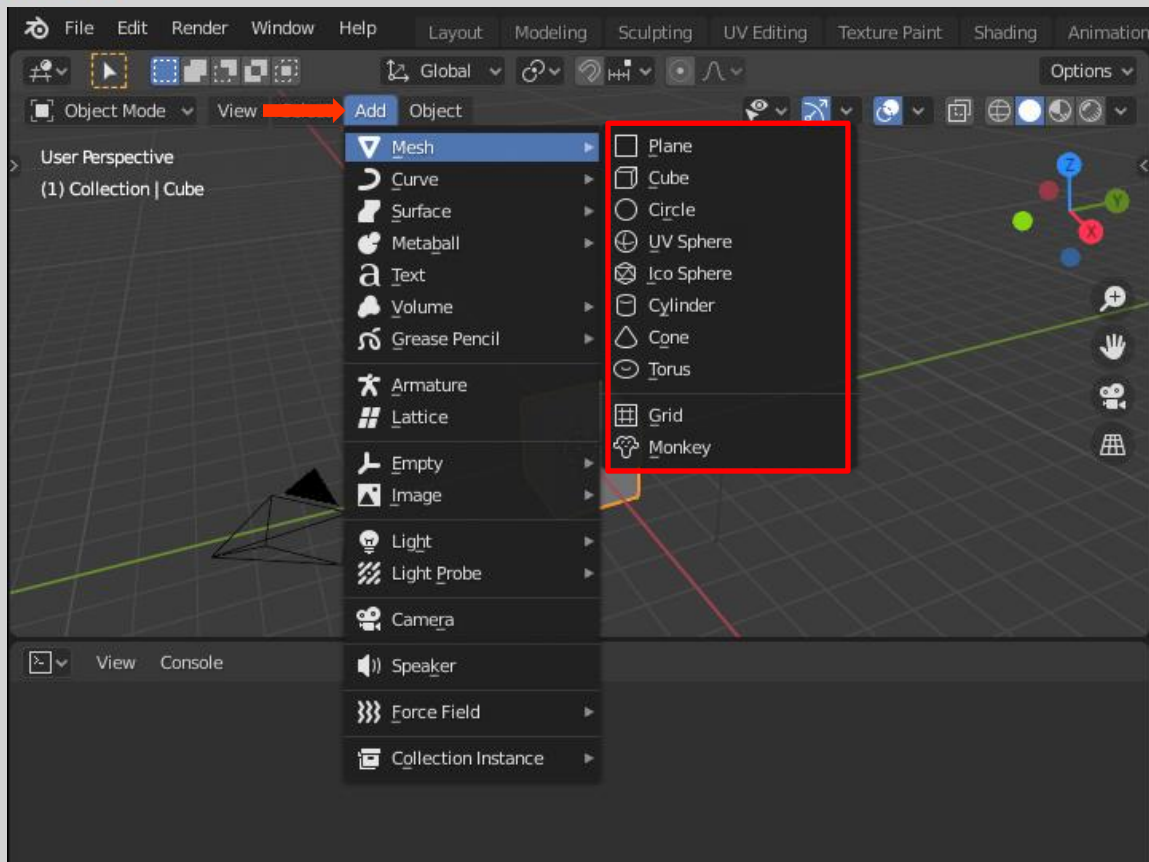


```
import bpy
```

```
bpy.ops.mesh.primitive_cube_add(size=2, location=(0, 0, 0))  
bpy.ops.mesh.primitive_uv_sphere_add(radius=1, location=(1, 0, 0))  
bpy.data.objects['Cube'].select_set(True)  
bpy.data.objects['Sphere'].select_set(True)  
bpy.ops.object.join()
```

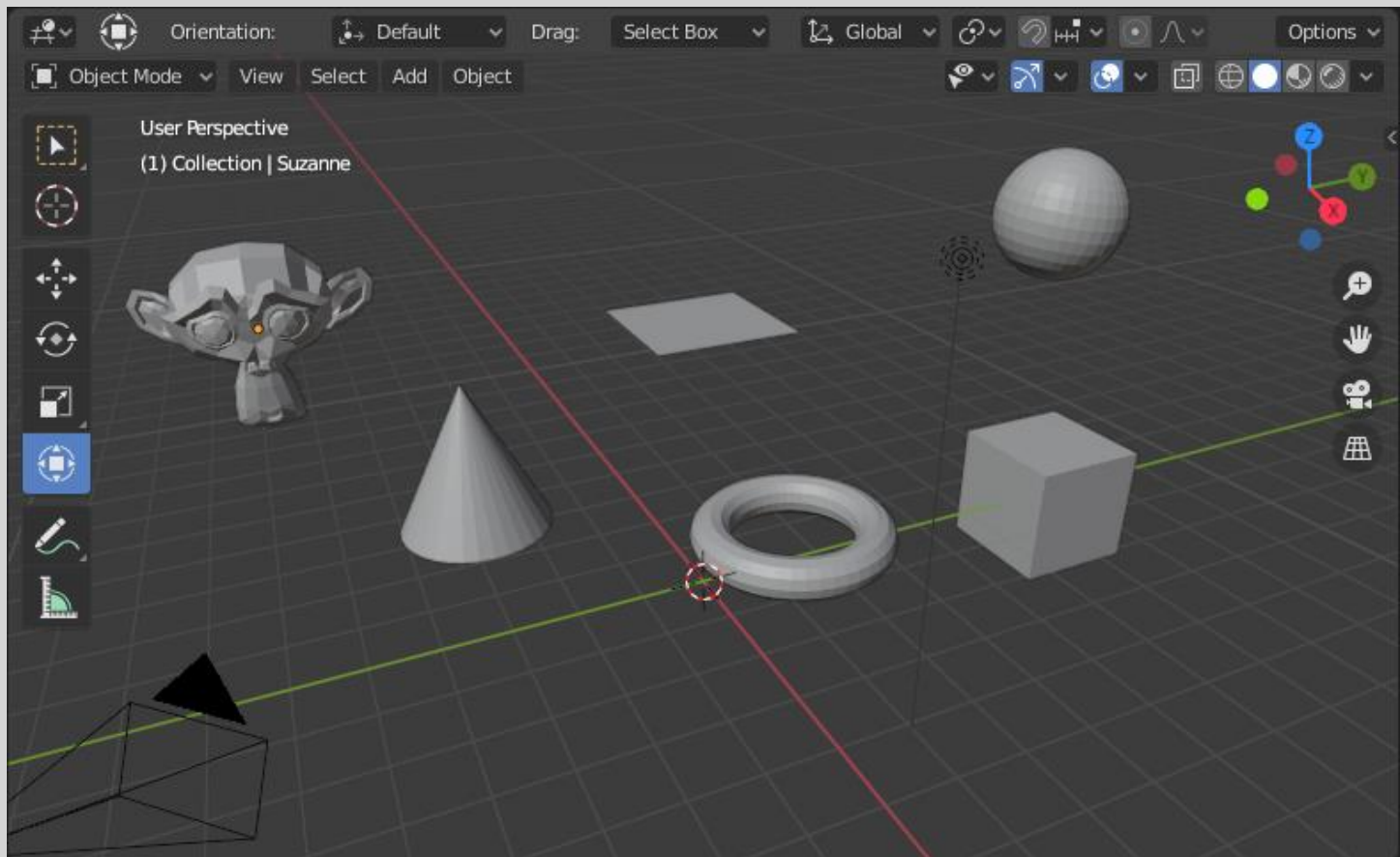


The bpy Module



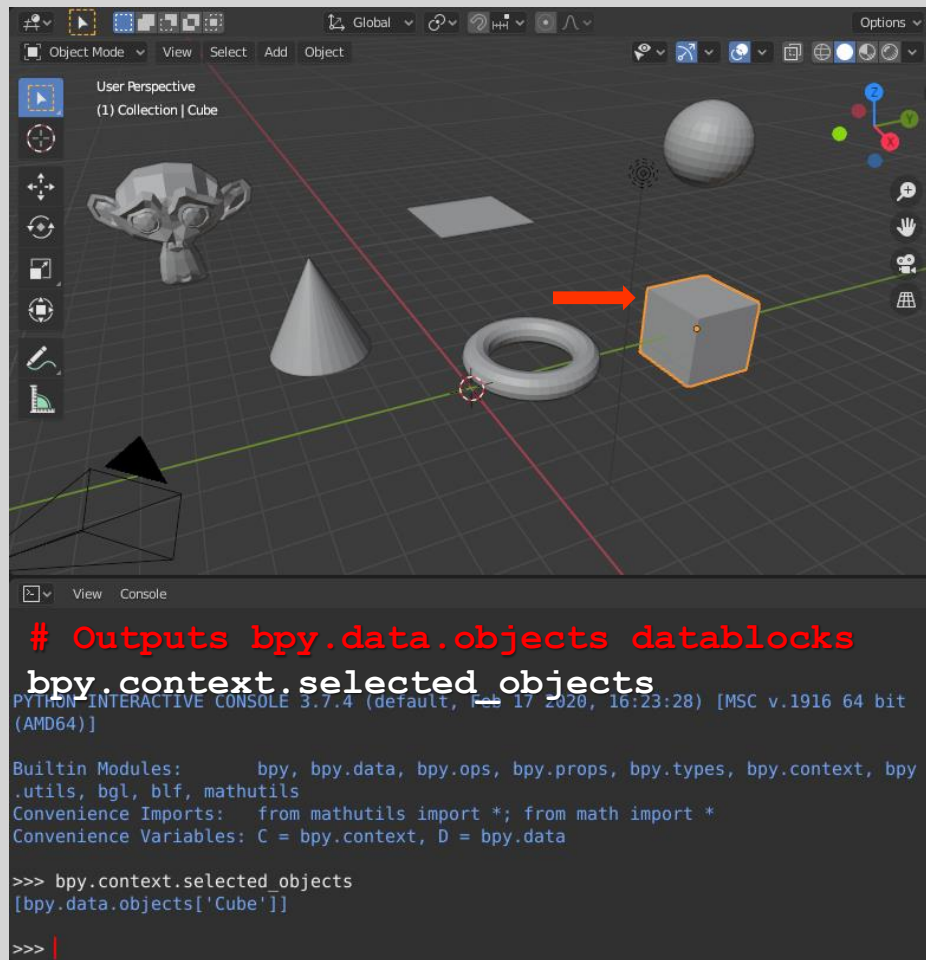


The bpy Module



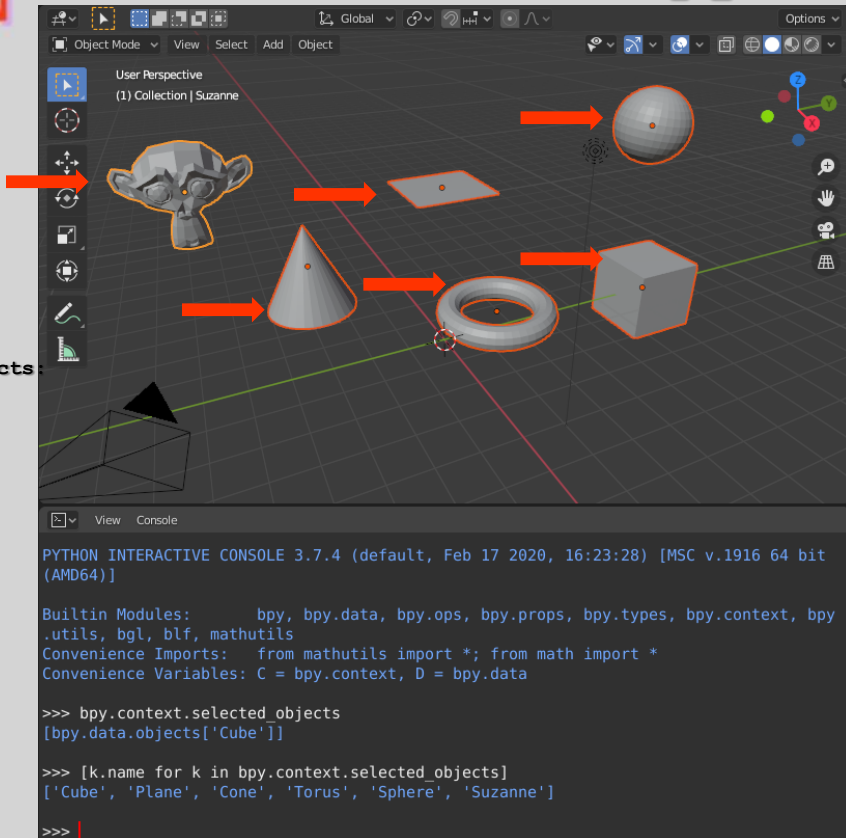
the **bpy.context** submodule is great for fetching lists of objects based on their state within Blender

the **bpy.context** submodule is used to **access objects** and areas of Blender by various status criteria.





The bpy Module



One of the language's most distinctive features is the **list comprehension**, which you can use to create powerful functionality within a single line of code.

Return the names of selected objects

```
[k.name for k in bpy.context.selected_objects]
```

```
['Sphere', 'Plane', 'Suzanne', 'Torus', 'Cone', 'Cube']
```



Using List Comprehensions

```
new_list = [expression for member in iterable]
```

```
squares = [i * i for i in range(10)]
```

Every list comprehension in Python includes **three** elements:

- **expression** is the member itself, a call to a method, or any other valid expression that returns a value. In the example above, the expression `i * i` is the square of the member value.
- **member** is the object or value in the list or iterable. In the example above, the member value is `i`.
- **iterable** is a list, set, sequence, generator, or any other object that can return its elements one at a time. In the example above, the iterable is `range(10)`.



Using List Comprehensions

Using Conditional Logic

```
new_list = [expression for member in iterable (if conditional)]  
  
>>> sentence = 'Blender is the best'  
>>> vowels = [i for i in sentence if i in 'aeiou']  
>>> vowels  
['e', 'e', 'i', 'e', 'e']
```

```
new_list = [expression (if conditional) for member in iterable]  
  
>>> marks = [1.25, -9.45, 10.22, 3.78, -5.92, 1.16]  
>>> positive_marks = [i if i > 0 else 0 for i in marks]  
>>> positive_marks  
[1.25, 0, 10.22, 3.78, 0, 1.16]
```



```
new_list = [expression for member in iterable]
```

```
# Return the names of selected objects
```

```
>>> [k.name for k in bpy.context.selected_objects]
```

```
['Cube', 'Plane', 'Cone', 'Torus', 'Sphere', 'Suzanne']
```

```
# Return the locations of selected objects
```

```
# (location of origin assuming no pending transformations)
```

```
>>>[k.location for k in bpy.context.selected_objects]
```

```
[Vector((0.0, 6.6893310546875, 0.0)), Vector((0.0, 0.0,  
4.201729774475098)), Vector((0.0, -3.2858810424804688,  
2.869234085083008)), Vector((2.727579355239868, 0.0,  
2.3433666229248047)), Vector((0.0, 6.11886739730835,  
5.0066142082214355)), Vector((-0.4047573506832123, -  
5.412367820739746, 4.79141902923584))]
```



Return the name and locations of selected objects

```
>>>[(k.name, k.location) for k in bpy.context.selected_objects]
```

```
[('Cube', Vector((0.0, 6.6893310546875, 0.0))), ('Plane', Vector((0.0, 0.0, 4.201729774475098))), ('Cone', Vector((0.0, -3.2858810424804688, 2.869234085083008))), ('Torus', Vector((2.727579355239868, 0.0, 2.3433666229248047))), ('Sphere', Vector((0.0, 6.11886739730835, 5.0066142082214355))), ('Suzanne', Vector((-0.4047573506832123, -5.412367820739746, 4.79141902923584)))]
```

? a, b



Like a string, a list is a sequence of values. In a string, the values are characters; in a list, they can be **any type**. The values in a list are called **elements** or sometimes items.

Basic properties:

Lists are contained in square brackets **[]**

Lists can contain numbers, strings, nested sublists, or nothing

Examples:

```
L1 = [0,1,2,3]
```

```
L2 = ['zero', 'one']
```

```
L3 = [0,1,[2,3], 'three', ['four', 'one']]
```

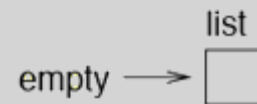
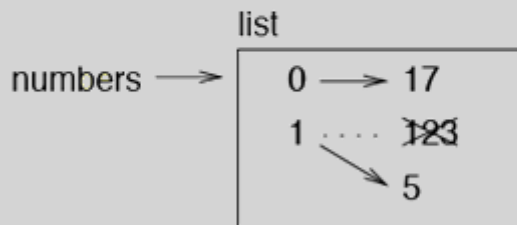
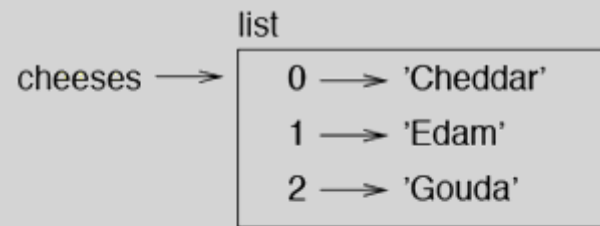
```
L4 = []
```



Lists

Lists Are Mutable : Unlike strings, lists are mutable. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print (cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
>>> numbers[1] = 5
>>> print (numbers)
[17, 5]
```





```
>>> L = [1, 2, 3]
>>> for element in L:
...     print(element)
...
1
2
3
>>> L.append("I am a string.")
>>> for element in L:
...     print(element)
...
1
2
3
I am a string.
```



Some basic operations on lists:

Indexing: `L1[i]`, `L2[i][j]` **Slicing:** `L3[i:j]`

Concatenation:

```
>>> L1 = [0,1,2]; L2 = [3,4,5]
>>> L1+L2
[0,1,2,3,4,5]
```

Repetition:

```
>>> L1*3
[0,1,2,0,1,2,0,1,2]
```

Appending:

```
>>> L1.append(3)
[0,1,2,3]
```

Sorting:

```
>>> L3 = [2, 1, 4, 3]
>>> L3.sort()
[1,2,3,4]
```




Reversal:

```
>>> L4 = [4,3,2,1]
>>> L4.reverse()
>>> L4
[1,2,3,4]
```

Append, Pop and Insert:

```
>>> L4.append(5)           # [0,1,2,3,4,5]
>>> L4.pop()              # [0,1,2,3,4]
>>> L4.insert(0, 42)       # [42,0,1,2,3,4]
>>> L4.pop(0)              # [0,1,2,3,4]
```



```
names = ['a', 'a', 'b', 'c', 'a']  
# Count the letter a.  
value = names.count('a')  
print(value)
```

```
# Input list.  
values = ["uno", "dos", "tres", "cuatro"]
```

```
# Locate string.  
n = values.index("dos")  
print(n, values[n])
```

```
# Locate another string.  
n = values.index("tres")  
print(n, values[n])
```

Module Overview

- **bpy.ops**

This submodule contains operators. These are primarily functions for manipulating objects, similarly to the way Blender artists manipulate objects in the default interface. The submodule can also manipulate the 3D Viewport, renderings, text, and much more.

For manipulating 3D objects, the two most important classes are `bpy.ops.object` and `bpy.ops.mesh`. The object class contains functions for manipulating multiple selected objects at the same time as well as many general utilities. The mesh class contains functions for manipulating vertices, edges, and faces of objects one at a time, typically in Edit Mode. There are currently **71** classes in the `bpy.ops` submodule, all fairly well-named and well-organized.

- **bpy.context**

The **bpy.context** submodule is used to access objects and areas of Blender by various status criteria. The primary function of this submodule is to give Python developers a means of accessing the current data that a user is working with. If we create a button that permutes all of the selected objects, we can allow the user to select the objects of his choice, then permute all objects in **bpy.context.select_objects**.

- **bpy.data**

This submodule is used to access Blender's internal data. The **bpy.data.objects** class contains all of the data determining an object's shape and position.



- **bpy.app**

This submodule is not entirely documented, but the information we are confident about thus far can be used to great effect in scripting and add-on development. The sub-submodule **bpy.app.handlers** is the only one we will concern ourselves with in this course. The handlers submodule contains special functions for triggering custom functions in response to events in Blender.

- **bpy.types**, **bpy.utils**, and **bpy.props**

These modules will be discussed in detail on add-on development.

- **bpy.path**

This submodule is essentially the same as the **os.path** submodule that ships natively with Python.



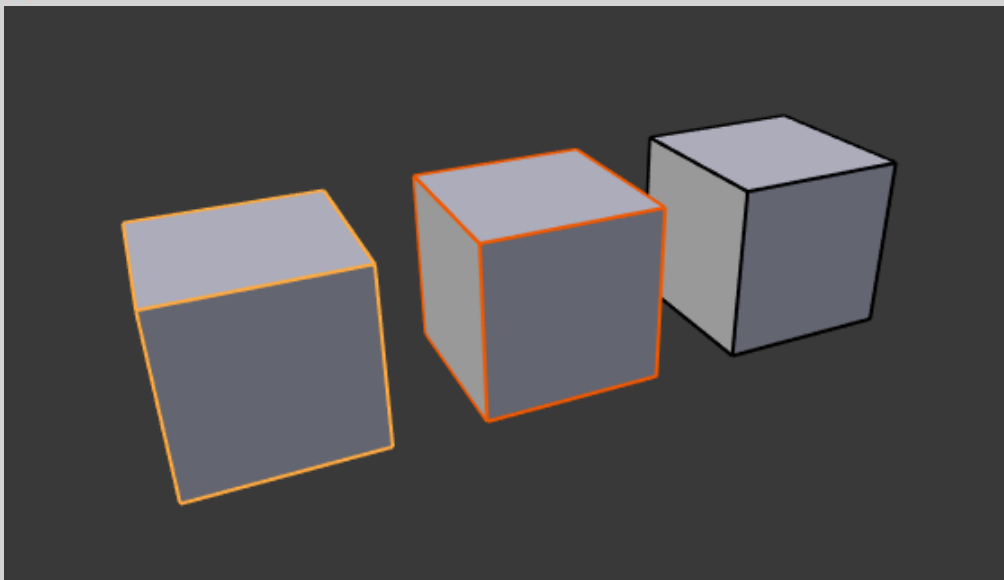
Selection, Activation, and Specification

The Blender interface was designed to be intuitive while also providing complex functionality.

- **Selection:** One, many, or zero objects can be selected at once.
- **Activation:** Only a single object can be active at any given time.
- **Specification:** (Python only) Python scripts can access objects by their names and write directly to their data blocks. While an operation that manipulates selected objects is typically a differential action like translate, rotate, or scale, writing data to specific objects is typically a declarative action like position, orientation, or size.



The bpy Module



Active object in **yellow**, selected object in **orange**, and unselected object in **black**. In Object Mode the **last (de)selected** item is called the “**Active Object**” and is outlined in yellow (the others are orange). There is at most **one active object** at any time.



Programmatically Selecting Objects

```
import bpy
```

```
def mySelector(objName, additive= False):
```

```
    # By default, clear other selections
```

```
    if not additive:
```

```
        bpy.ops.object.select_all(action='DESELECT')
```

```
    # Set the 'select' property of the datablock to True
```

```
    bpy.data.objects[objName].select_set(True)
```

```
# Select only 'Cube'
```

```
mySelector('Cube')
```

```
# Select 'Sphere', keeping other selections
```

```
mySelector('Sphere', additive= True)
```

```
# Translate selected objects 1 unit along the x-axis
```

```
bpy.ops.transform.translate(value=(1, 0, 0))
```



Programmatically Selecting Objects

```
import bpy
```

```
def select_one_object(objName):
```

```
    bpy.ops.object.select_all(action='DESELECT')
```

```
    bpy.context.view_layer.objects.active = bpy.data.objects[objName]
```

```
    objName.select_set(True)
```

```
select_one_object(cylinder) # This will select the cylinder and set it as active
```

```
select_one_object(cube)    # This will select the cube and set it as active
```

```
bpy.context.view_layer.objects.active = None # After that you won't have an  
active object in the scene. Note it doesn't deselect the active object though.
```



```
# Programmatically Activating an Object
```

```
import bpy
```

```
def myActivator(objName):
```

```
    # Pass bpy.data.objects datablock to scene class
```

```
    bpy.context.view_layer.objects.active = bpy.data.objects[objName]
```

```
# Activate the object named 'Sphere'
```

```
myActivator('Sphere')
```

```
# Verify the 'Sphere' was activated
```

```
print("Active object:", bpy.context.object.name)
```

```
# Selected objects were unaffected
```

```
print("Selected objects:", bpy.context.selected_objects)
```



```
# how to programmatically select all objects of a certain
collection
import bpy

col = bpy.data.collections.get("Collection")
if col:
    for obj in col.objects:
        obj.select_set(True)

# Selected objects
print("Selected objects:", bpy.context.selected_objects)
```



Programmatically Accessing an Object by Specification

```
import bpy
```

```
def mySpecifier(objName):
```

```
    # Return the datablock
```

```
    return bpy.data.objects[objName]
```

Store a reference to the datablock

```
myCube = mySpecifier('Cube')
```

Output the location of the origin

```
print(myCube.location)
```

Works exactly the same as above

```
myCube = bpy.data.objects['Cube']
```

```
print(myCube.location)
```



SCRIPT LANGUAGES FOR ANIMATION

The bpy Module

The image shows the Blender 2.80.0 interface. The 3D viewport on the left displays a scene with a cube and a sphere. The right side of the interface shows a text editor with Python code for selecting and moving objects. The code defines functions for selecting objects, activating objects, and specifying objects. The console at the bottom shows the execution of the code.

```
1 # Programmatically Selecting Objects
2 import bpy
3
4 def mySelector(objName, additive= False):
5     # By default, clear other selections
6     if not additive:
7         bpy.ops.object.select_all(action='DESELECT')
8     # Set the 'select' property of the datablock to True
9     bpy.data.objects[objName].select_set(True)
10
11 def myActivator(objName):
12     # Pass bpy.data.objects datablock to scene class
13     bpy.context.view_layer.objects.active = bpy.data.objects[objName]
14
15 def mySpecifier(objName):
16     # Return the datablock
17     return bpy.data.objects[objName]
18
19 |
20 # Select only 'Cube'
21 mySelector('Cube')
22 # Select 'Sphere', keeping other selections
23 mySelector('Sphere', additive= True)
24 # Translate selected objects 1 unit along the x-axis
25 bpy.ops.transform.translate(value=(1, 0, 0))
26 mySelector('Cube')
27 # Translate selected objects 1 unit along the x-axis
28 bpy.ops.transform.translate(value=(-2, 0, 0))
29
30 # Activate the object named 'Sphere'
31 myActivator('Sphere')
32 # Verify the 'Sphere' was activated
33 print("Active object:", bpy.context.object.name)
34 # Selected objects were unaffected
35 print("Selected objects:", bpy.context.selected_objects)
36
37 # Store a reference to the datablock
38 myCube = mySpecifier('Cube')
39
40 # Output the location of the origin
41 print(myCube.location)
42
43 # Works exactly the same as above
44 myCube = bpy.data.objects['Cube']
45 print(myCube.location)
46
```

PYTHON INTERACTIVE CONSOLE 3.9.2 (default, Mar 1 2021, 08:18:55) [MSC v.1916 64 bit (AMD64)]

Built-in Modules: bpy, bpy.data, bpy.ops, bpy.props, bpy.types, bpy.context, bpy.utils, bgl, bif, mathutils
Convenience Imports: from mathutils import *; from math import *
Convenience Variables: C = bpy.context, D = bpy.data

```
>>>
bpy.context.space_data.recent_folders_active = 0
Saved text "C:\Lectures\BCO 602 Animasyon Icin Betik Diller\HafTa 02\sel_act_spec.py"
bpy.ops.mesh.primitive_uv_sphere_add(radius=1, enter_editmode=False, align='WORLD', location=(0, 0, 0), scale=(1, 1, 1))
bpy.ops.transform.translate(value=(0, 4.33348, 0), orient_type='GLOBAL', orient_matrix=((1, 0, 0), (0, 1, 0), (0, 0, 1)), orient_matrix_type='GLOBAL', constraint_axis=(False, True, False), mirror=True, use_proportional_edit=False, proportional_edit_falloff='SMOOTH', proportional_size=1, use_proportional_connected=False)
```

Select Report | Box Select | Pan View | Info Context Menu

File: "C:\Lectures\BCO 602 Animasyon Icin Betik Diller\HafTa 02\sel_act_spec.py" (unsaved)



SCRIPT LANGUAGES FOR ANIMATION

The bpy Module

The image shows the Blender 2.80.0 interface. The 3D viewport on the left displays a scene with a cube and a sphere. The right side of the interface shows a text editor with Python code that demonstrates the use of the bpy module for selecting and manipulating objects.

```
1 # Programmatically Selecting Objects
2 import bpy
3
4 def mySelector(objName, additive= False):
5     # By default, clear other selections
6     if not additive:
7         bpy.ops.object.select_all(action='DESELECT')
8     # Set the 'select' property of the datablock to True
9     bpy.data.objects[objName].select_set(True)
10
11 def myActivator(objName):
12     # Pass bpy.data.objects datablock to scene class
13     bpy.context.view_layer.objects.active = bpy.data.objects[objName]
14
15
16 def mySpecifier(objName):
17     # Return the datablock
18     return bpy.data.objects[objName]
19
20 # Select only 'Cube'
21 mySelector('Cube')
22 # Select 'Sphere', keeping other selections
23 mySelector('Sphere', additive= True)
24 # Translate selected objects 1 unit along the x-axis
25 bpy.ops.transform.translate(value=(1, 0, 0))
26 mySelector('Cube')
27 # Translate selected objects -2 unit along the x-axis
28 bpy.ops.transform.translate(value=(-2, 0, 0))
29
30 # Activate the object named 'Sphere'
31 myActivator('Sphere')
32 # Verify the 'Sphere' was activated
33 print("Active object:", bpy.context.object.name)
34 # Selected objects were unaffected
35 print("Selected objects:", bpy.context.selected_objects)
36
37 # Store a reference to the datablock
38 myCube = mySpecifier('Cube')
39
40 # Output the location of the origin
41 print(myCube.location)
42
43 # Works exactly the same as above
44 myCube = bpy.data.objects['Cube']
45 print(myCube.location)
46
```

The Python Interactive Console at the bottom shows the execution of the code:

```
PYTHON INTERACTIVE CONSOLE 3.9.2 (default, Mar 1 2021, 08:18:55) [MSC v.1916 64 bit (AMD64)]
Built-in Modules:  bpy, bpy.data, bpy.ops, bpy.props, bpy.types, bpy.context, bpy.utils, bgl, bif, mathutils
Convenience Imports:  from mathutils import *; from math import *
Convenience Variables:  C = bpy.context, D = bpy.data
>>>
bpy.ops.object.select_all(action='DESELECT')
bpy.ops.transform.translate(value=(-2, 0, 0), orient_type='GLOBAL', orient_matrix=((1, 0, 0), (0, 1, 0), (0, 0, 1)), orient_matrix_type='GLOBAL', constraint_axis=(True, True, True), mirror=True, use_proportional_edit=False, proportional_edit_falloff='SMOOTH', proportional_size=1, use_proportional_connected=False, use_proportional_projected=False)
bpy.ops.text.run_script()
```




SCRIPT LANGUAGES FOR ANIMATION

The bpy Module

The image shows the Blender 2.93.2 interface. The 3D Viewport on the left displays a scene with a cube and a sphere. The right side of the interface is split into two panels. The top panel is the 'Scripting' editor, showing a Python script for programmatically selecting objects. The bottom panel is the 'Console' window, displaying the output of the script execution.

```
1 # Programmatically Selecting Objects
2 import bpy
3
4 def mySelector(objName, additive= False):
5     # By default, clear other selections
6     if not additive:
7         bpy.ops.object.select_all(action='DESELECT')
8     # Set the 'select' property of the datablock to True
9     bpy.data.objects[objName].select_set(True)
10
11 def myActivator(objName):
12     # Pass bpy.data.objects datablock to scene class
13     bpy.context.view_layer.objects.active = bpy.data.objects[objName]
```

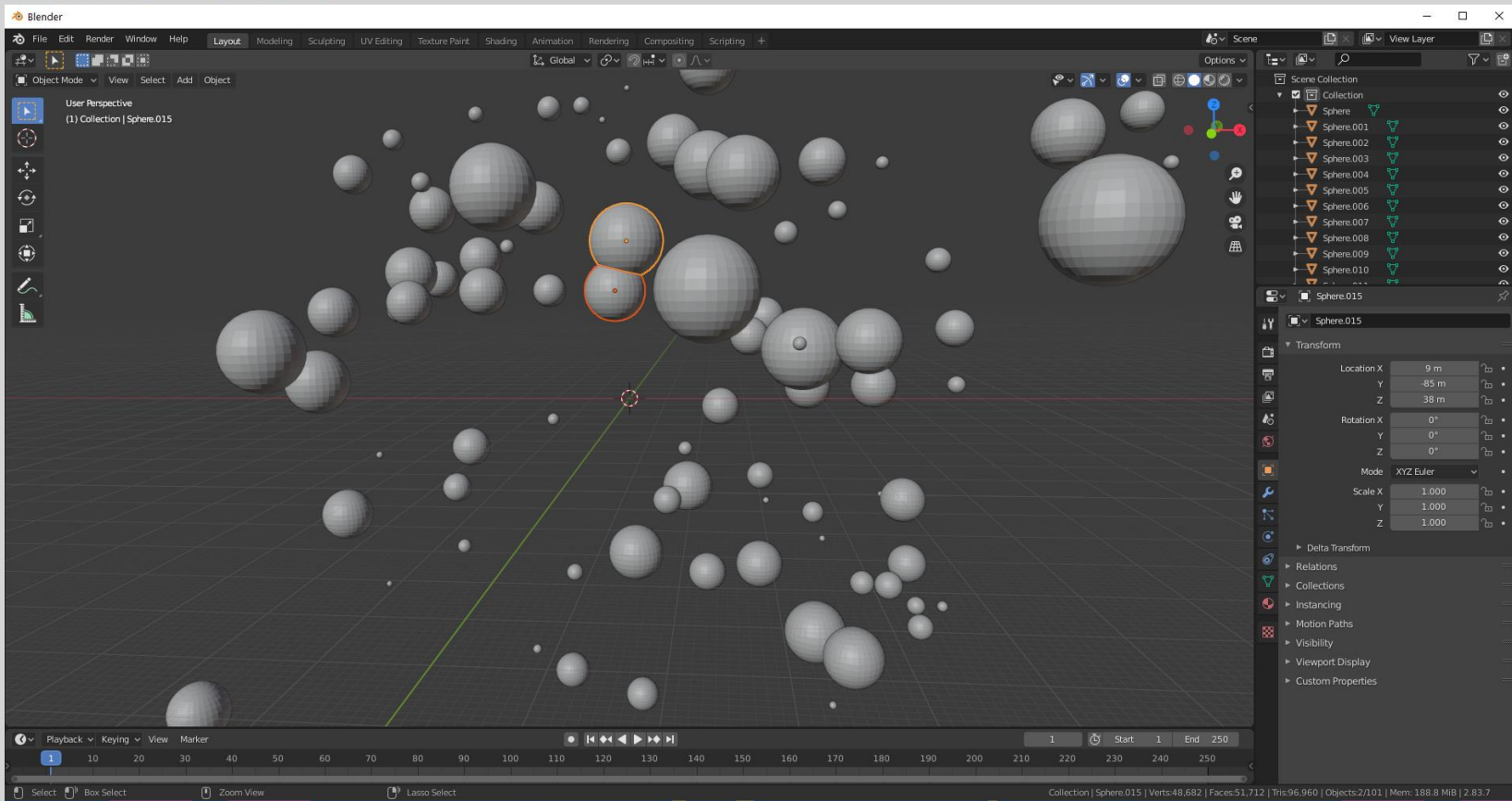
blender
Read prefs: C:\Users\SA-Lenovo\AppData\Roaming\Blender Foundation\Blender\2.93\config\userpref.blend
Info: Saved text "C:\Lectures\BCO 602 Animasyon Icin Betik Diller\Hafta_02\sel_act_spec.py"

Active object: Sphere
Selected objects: [bpy.data.objects['Cube']]
<Vector (-1.0000, 0.0000, 0.0000)>
<Vector (-1.0000, 0.0000, 0.0000)>
Info: Saved text "C:\Lectures\BCO 602 Animasyon Icin Betik Diller\Hafta_02\sel_act_spec.py"

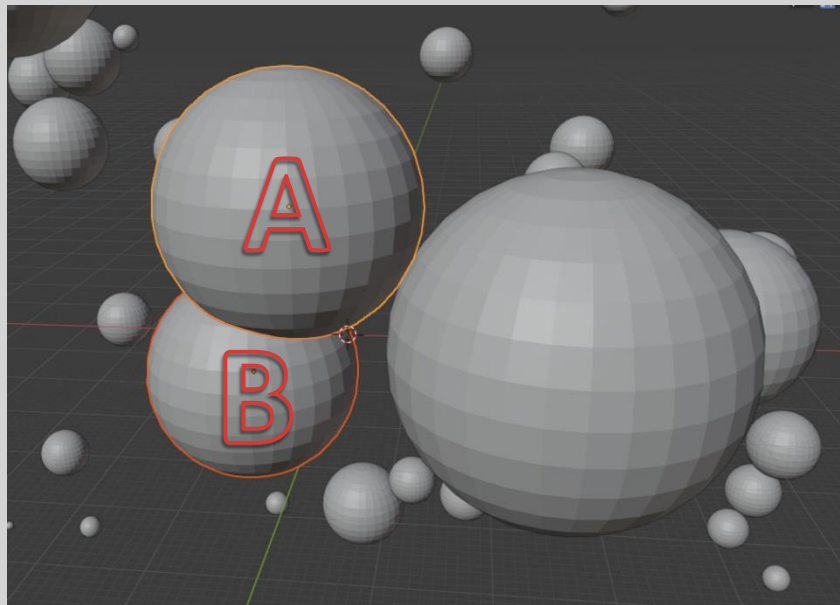
PYTHON INTERACTIVE CONSOLE 3.9.2 (default, Mar 1 2021, 08:18:55) [MSC v.1916 64 bit (AMD64)]
Builtin Modules: bpy, bpy.data, bpy.ops, bpy.props, bpy.types, bpy.context, bpy.utils, bgl, blf, mathutils
Convenience Imports: from mathutils import *; from math import *
Convenience Variables: C = bpy.context, D = bpy.data

```
>>> bpy.ops.object.select_all(action='DESELECT')
>>> bpy.ops.transform.translate(value=(-2, 0, 0), orient_type='GLOBAL', orient_matrix=((1, 0, 0), (0, 1, 0), (0, 0, 1)), orient_matrix_type='GLOBAL', constraint_axis=(True, True, True), mirror=True, use_proportional_edit=False, proportional_edit_falloff='SMOOTH', proportional_size=1, use_proportional_connected=False, use_proportional_projected=False)
>>> bpy.ops.text.run_script()
```

File: C:\Lectures\BCO 602 Animasyon Icin Betik Diller\Hafta_02\sel_act_spec.py



Homework



Write a Blender Script generates 100 random spheres that do **not collide** each other in the definite volume.

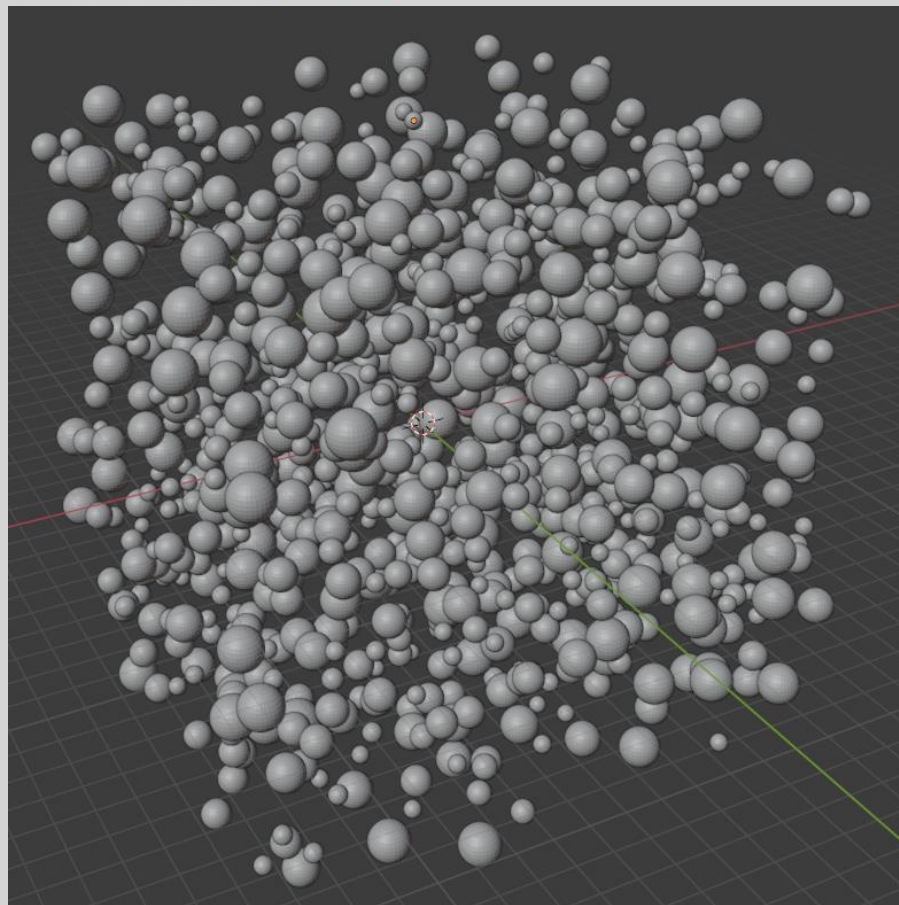
Some of the functions you may need

```
from random import randrange  
from math import sqrt
```

$$Collision = \sqrt{(A_x - B_x)^2 + (A_y - B_y)^2 + (A_z - B_z)^2} \leq A_{radius} + B_{radius}$$



Homework



Correct Example: 1000 spheres with diameter of between 6 to 16. Be careful not radius!

